

**Rechnerarchitekturgroßpraktikum**

Entwicklung eines RISC-V-Prozessors

# Dokumentation

Dominik Fuchsgruber  
Charlie Groh  
Franz Rieger  
Jan Schuchardt

12. Februar 2017

## **Zusammenfassung**

Die folgende Arbeit beschreibt die Struktur und den Entwicklungsprozess einer Implementierung eines Prozessors auf Basis der *RISC-V-ISA* auf einem FPGA. Dabei wurden die logischen Bausteine in der Hardwarebeschreibungssprache VHDL implementiert. Die Projektdauer beläuft sich auf zwei Semester, in denen die vier Gruppenmitglieder simultan an den einzelnen Komponenten gearbeitet haben. Der Arbeit liegen der entstandene VHDL-Code sowie einige demonstrative Programme, die vom Prozessor korrekt ausgeführt werden können, bei. Des Weiteren sind auch ein Assembler für das implementierte Instruction-Set sowie ein Simulator, der auf die spezielle Umgebung der VHDL-Implementierung angepasst ist, enthalten.

# Inhaltsverzeichnis

1	Projektablauf . . . . .	4
1.1	Aufgabe . . . . .	4
1.2	Basisziele . . . . .	4
1.3	Erweiterungsziele . . . . .	4
1.4	Projektablauf . . . . .	5
1.5	Verwendete Tools . . . . .	5
2	Das Leitwerk . . . . .	7
2.1	Überblick . . . . .	7
2.2	Integer Rechenbefehle . . . . .	7
2.3	LUI und AUPIC . . . . .	8
2.4	Bedingte Sprünge . . . . .	8
2.5	Unbedingte Sprünge . . . . .	9
2.6	LOAD . . . . .	9
2.7	STORE . . . . .	9
2.8	Timer und Counter . . . . .	10
3	Die arithmetisch-logische Einheit . . . . .	10
3.1	Überblick . . . . .	10
3.2	Das Interface . . . . .	11
3.3	Interne Befehle . . . . .	11
3.4	Befehlsausführung . . . . .	12
3.5	Die Register . . . . .	13
3.6	Reset . . . . .	14
4	Die MMU . . . . .	14
4.1	Überblick . . . . .	14
4.2	Interface . . . . .	15
4.3	Aufbau des Speichers . . . . .	16
4.4	Memory-Mapped I/O . . . . .	17
4.5	Implementierung als State Machine . . . . .	18
4.6	Zugriffssdauer . . . . .	20
5	Die ASCII-Unit . . . . .	20
5.1	Überblick . . . . .	21
5.2	Interface . . . . .	22
5.3	Funktionsweise . . . . .	22
6	Benutzerschnittstelle . . . . .	23
6.1	Initialisierung . . . . .	23
6.2	Der Reset . . . . .	24
6.3	Graphische Oberflächen . . . . .	24
6.4	Benutzereingabe . . . . .	25

7	Entwicklung eines Demo-Programms . . . . .	25
7.1	Struktur und Funktion des Demo-Programms . . . . .	25
7.2	Assemblierer und Simulator . . . . .	26
1	Anhang . . . . .	28
1.1	Übersicht über die zentralen Einheiten des Prozessors . . . . .	28
1.2	Known Bugs . . . . .	29
1.3	Ausführungszeit der Befehle . . . . .	30
1.4	1. Pflichtenheft vom 02.06.2016 . . . . .	31
1.5	2. Pflichtenheft vom 19.11.2016 . . . . .	32

# Tabellenverzeichnis

1	Übersicht über den Projektablauf . . . . .	5
2	Aufbau des Kommunikationssignals zwischen Leitwerk und MMU . . . . .	16
3	Aufbau des Adressvektors . . . . .	16
4	Aufbau des Speichers als Blöcke . . . . .	17
5	Memory-Mapped I/O im Detail . . . . .	17
6	Zugriffsdauer einzelner Speicherzugriffe . . . . .	20
7	Befehlsübersicht des Simulators . . . . .	27
8	Übersicht über die Ausführungszeit der Befehle . . . . .	30

# Abbildungsverzeichnis

1	Das verwendete Entwicklungsboard von oben, das FPGA liegt zentral . . . . .	4
2	Der Xilinx <i>ISE Project Navigator</i> . . . . .	6
3	Zustandsübergangsdiagramm der Befehle ADD[I], SUB, SLT[I][U], AND[I], OR[I], XOR[I], SLL[I], SRL[I], SRA[I], MUL und MULH[[S]U]. . . . .	7
4	Zustandsübergangsdiagramm der Befehle DIV[U] und REM[U]. . . . .	7
5	Zustandsübergangsdiagramm der Befehle LUI und AUPIC. . . . .	8
6	Zustandsübergangsdiagramm der Befehle BEQ und BGE[U]. . . . .	8
7	Zustandsübergangsdiagramm der Befehle BNE und BLT[U]. . . . .	8
8	Zustandsübergangsdiagramm der Befehle JAL und JALR. . . . .	9
9	Zustandsübergangsdiagramm der Befehle LB[U], LH[U] und LW. . . . .	9
10	Zustandsübergangsdiagramm des Befehle SB, SH und SW. . . . .	10
11	Zustandsübergangsdiagramm der Befehle RDINSTRET[H], RDCYCLE[H] und RDTIME[H]. . . . .	10

12	Schnittstelle der ALU-Einheit . . . . .	11
13	Zustandsübergangsdiagramm der Befehlsausführung innerhalb der ALU . . . . .	12
14	Schnittstelle der MMU-Einheit . . . . .	15
15	Verteilung der I/O-Signale . . . . .	18
16	Zustandsübergangsdiagramm der MMU. . . . .	18
17	Beispiel für die Textausgabe . . . . .	20
18	Veranschaulichung der Adressberechnung der ASCII-Unit . . . . .	21
19	Interface der ASCII-Unit . . . . .	22
20	Übersicht über die ASCII-Unit . . . . .	23
21	VGA-Ausgabe im Debugging-Modus . . . . .	24
22	Bildschirmausgabe während der Ausführung des Demo-Programms . . . . .	25
23	Übersicht über die zentralen Einheiten des Prozessors . . . . .	28
24	Pflichtenheft vom 02.06.2016 . . . . .	31
25	Pflichtenheft vom 19.11.2016 . . . . .	32

# 1 Projektablauf

Zu Beginn des Projekts wurde eine Unterteilung der Anforderungen in ein Basisangebot sowie ein daran anknüpfendes Erweiterungsangebot vollzogen. Während innerhalb des ersten Semesters vor allem der Eingewöhnung in die Entwicklungsumgebung und Realisierung der Basisziele diente, sah die Planung für das zweite Semester eine Erweiterung des dann bereits lauffähigen Prozessors um zusätzliche Module vor.

## 1.1 Aufgabe

Die Hauptaufgabe bestand darin, eine Implementierung eines funktionsfähigen Prozessors auf Basis der *RISC-V Instruction Set Architecture*<sup>1</sup> anzufertigen, welche auf dem zur Verfügung gestellten Entwicklungsboard (*Spartan-3A FPGA Starter Kit*) lauffähig sein sollte.



Abbildung 1: Das verwendete Entwicklungsboard von oben, das FPGA liegt zentral

## 1.2 Basisziele

Als Mindestanforderung sollte dabei die Kompatibilität zur *RV32I Base Integer Instruction Set* gewährleistet werden. Ausgenommen davon waren die speziell für Aspekte des Multithreadings auf Mehrkernprozessoren enthaltenen Befehle **FENCE**, **FENCE.I**, **SCALL** und **SBREAK**, da die geforderte Implementierung lediglich einen Rechenkern besitzen sollte.

Um die Funktionsfähigkeit des Prozessors auch nach außen hin sichtbar zu machen und damit einhergehend keine Black-Box Komponente zu entwickeln, sollte eine Möglichkeit zur Benutzerinteraktion über geeignete, auf dem Entwicklungsboard vorhandene Bausteine wie etwa Schalter bestehen. Insbesondere zu Zwecken des Debuggings wurde auch eine grafische Ausgabe der internen Register über die VGA-Schnittstelle zu einem angeschlossenen Monitor inkludiert.<sup>2</sup>

## 1.3 Erweiterungsziele

Die Modularität der RISC-V ISA legt einige sinnvolle Erweiterungen nahe, darunter auch die *RV32M Standard Extension for Integer Multiplication and Division*, welche Multiplikations- und Divisionsbefehle beinhaltet. Zudem sollte die rudimentäre Ausgabe der internen Register um einen Textmodus erweitert werden, sodass mittels Memory-Mapping

<sup>1</sup><https://riscv.org/specifications/>

<sup>2</sup>Siehe dazu auch Pflichtenheft vom 02.06.16 bzw. 19.11.16

auch ASCII-Zeichen auf dem Monitor ausgegeben werden können. Weiterhin wurde, um eine bessere Interaktion mit dem Programmierer zu ermöglichen, auch die Umsetzung einer seriellen Schnittstelle in den geplanten Erweiterungsrahmen miteinbezogen. Zuletzt stellt auch das Entwerfen eines demonstrativen Programms, genauer eines einfachen Spiels, einen maßgeblichen Bestandteil des Erweiterungsangebots dar, mit dem Ziel die endgültige Implementierung adäquat präsentieren zu können.<sup>3</sup>

## 1.4 Projektablauf

### Chronologischer Verlauf

Wie eingangs erwähnt umfasste die Dauer des Praktikums zwei Semester, in denen im Allgemeinen drei verschiedene Implementierungsversionen angefertigt wurden.

Version	Zeitraum	Ziele	davon nicht erreicht
1	April 2016 - Juni 2016	<ul style="list-style-type: none"> <li>• Prüfung der Strukturierung des Prozessors in ALU, Leitwerk und MMU</li> <li>• Prüfung der Arbeitsaufteilung</li> <li>• Verständnis der Tools</li> <li>• Ausführung einiger einfacher Befehle</li> </ul>	
2	Juni 2016 - September 2016	<ul style="list-style-type: none"> <li>• Implementierung der RV32I-Spezifikation</li> <li>• Lese- und Schreibzugriff auf den DDR2-RAM</li> <li>• Debugging-Ausgabe</li> </ul>	Lese- und Schreibzugriff auf den DDR2-RAM
3	September 2016 - Januar 2017	<ul style="list-style-type: none"> <li>• Implementierung der RV32M-Spezifikation</li> <li>• Lese- und Schreibzugriff auf den DDR2-RAM</li> <li>• ASCII-Ausgabe</li> <li>• Zugriff auf Buttons, LEDs etc. des Boards durch Memory-Mapped-I/O</li> <li>• serielle Schnittstelle (UART)</li> </ul>	serielle Schnittstelle (nur teilweise)

Tabelle 1: Übersicht über den Projektablauf

### Arbeitsteilung und Entscheidungsprozesse

Gemäß den vier Prinzipien der Von-Neumann-Architektur wurde die Implementierung grob in die Hauptkomponenten Leit-, Rechen-, Ein-/Ausgabewerk und Speicher gegliedert, was sich konkret in den drei Subkomponenten CU, ALU und MMU widerspiegelt. Für jeden dieser Bausteine war durchgängig eine unabhängige Kleingruppe verantwortlich, wobei es trotzdem für sinnvoll erachtet wurde, wöchentliche Treffen zu Zwecken der Planung und Integration der Komponenten sowie zum Testen, zu veranschlagen. Entscheidungen von größerer Tragweite, besonders im Hinblick auf generelle Designentscheidungen wurden meist im Plenum besprochen.

## 1.5 Verwendete Tools

Das Projekt wurde in der Hardwarebeschreibungssprache VHDL implementiert, was sich darin begründet, dass dieser Themenkomplex Teil der zugrundeliegenden Lehrveranstaltung war und die Gruppenmitglieder daher auf einem

---

<sup>3</sup>Siehe Fußnote 1

einigermaßen gleichwertigen Kenntnisstand waren.

Zur Entwicklung wurde hauptsächlich die Entwicklungsumgebung Xilinx' *ISE Project Navigator* in Version 14.7 verwendet. Diese ermöglichte einerseits das Editieren des VHDL-Codes und beinhaltete andererseits auch eine integrierte Toolchain, um den VHDL-Code in ein Programming-File zu übersetzen. Dieses wurde dann benutzt, um das Board zu programmieren. Der zusätzlich enthaltene *Core Generator* wurde dabei verwendet, um einzelne Subkomponenten wie beispielsweise eine Divisionseinheit zu generieren.

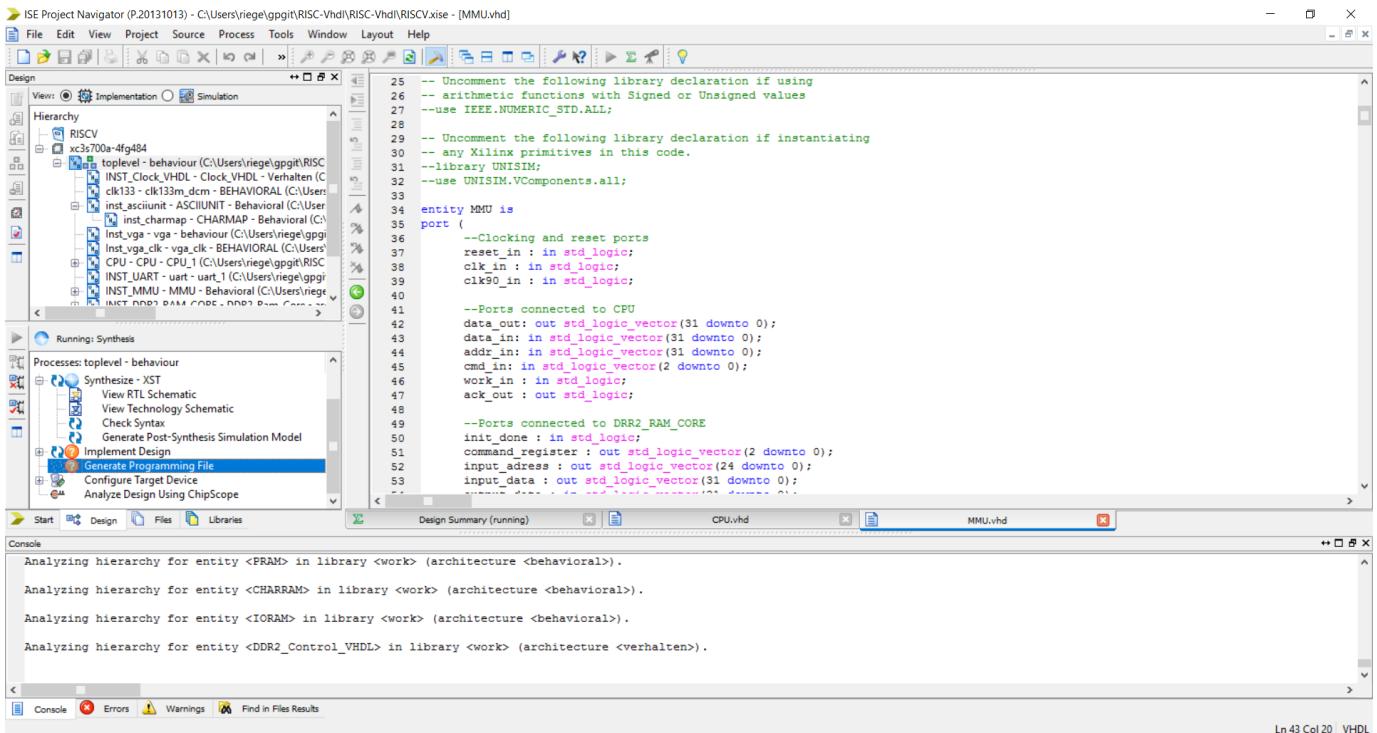


Abbildung 2: Der Xilinx *ISE Project Navigator*

Mittels des Tools *impact*, welches ebenfalls vom Anbieter der Entwicklungsumgebung Xilinx stammt, konnte in Kombination mit einem *Cable-Server* das Board über USB gemäß den erstellten Programming-Files beschrieben werden. Zur Versionsverwaltung wurde dabei auf ein *Git-Repository* gesetzt.

Um die doch recht umfassende Arbeit des Assemblierens eines Programms nicht händisch erledigen zu müssen, wurde, wie später genauer erläutert wird, ein automatischer Assemblierer in der Skriptsprache Python entwickelt. Da die Generierung eines Programming-Files und das anschließende Beschreiben des FPGAs nichtsdestotrotz jedes Mal mehrere Minuten in Anspruch nahm, wurden entsprechende Simulatoren verwendet. Zur Verifikation des VHDL-Codes wurde beispielsweise das Tool *GHDL* in Kombination mit *GTKWave* benutzt. Auch zum Testen der Assemblerprogramme wurde ein Simulator auf Python-Basis, der die Umgebung des Boards emuliert, entworfen und in den Arbeitsprozess mit eingebunden.

## 2 Das Leitwerk

### 2.1 Überblick

Das Leitwerk *CU* stellt die zentrale Steuereinheit des Prozessors dar. Es interpretiert die Befehle und überwacht ihre Ausführung durch ALU und MMU. Der Fokus lag dabei mehr auf Robustheit und weniger auf Aspekten der Performanz. Außerdem verwaltet das Leitwerk den Program-Counter (PC) und das Instruction-Register (IR).

Es besitzt für jeden Befehl eine eigene Zustandsmaschine, deren Verhalten durch den Inhalt des Instruction-Register gesteuert wird. Daher muss jeder Befehl als letztes das IR mit dem Opcode des folgenden Befehl laden. Wann dies geschieht kann in jedem Befehl einzeln optimiert werden.

Um den Implementierungsaufwand bei Änderungen von Befehlen zu minimieren wurde ein Compiler-Skript erstellt das mehrere Makros bereitstellt, aus denen dann die Befehle zusammengebaut werden können. Das Skript kompiliert dann eine Eingabe aus diesen Makros in VHDL-Code.

#### Legende

Da jeder Befehl eine eigene Zustandsmaschine besitzt, wird hier für jeden Befehl ein eigenes Zustandsübergangsdiagramm gezeigt. Die Präfixe "MMU:" und "ALU:" werden genutzt, um anzusehen, dass eine Operation an die jeweiligen Einheit delegiert wird und das Leitwerk lediglich eine Anfrage an die entsprechende Komponente sendet. Obwohl in der ISA regelmäßig verlangt wird, dass Operanden sign-extended werden, wurde dies in den Zustandsübergangsdiagrammen der Übersicht halber weggelassen. Im Prozessor ist es selbstverständlich wie in der ISA beschrieben implementiert.

### 2.2 Integer Rechenbefehle

Der Prozessor wurde auf die in *RV32I* und *RV32M* definierten Rechenbefehle optimiert. Dadurch können diese RISC-typischen Befehle sehr schnell ausgeführt werden.

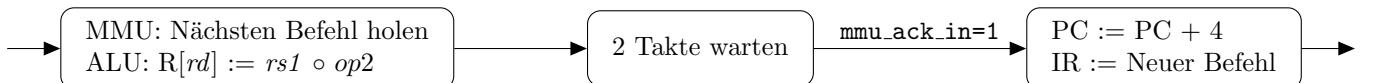


Abbildung 3: Zustandsübergangsdiagramm der Befehle **ADD[I]**, **SUB**, **SLT[I][U]**, **AND[I]**, **OR[I]**, **XOR[I]**, **SLL[I]**, **SRL[I]**, **SRA[I]**, **MUL** und **MULH[[S]U]**.  $op2$  ist entweder das Register, das mit  $rs2$  angegeben ist, oder eine Immediate ( $imm$ ). "○" repräsentiert die jeweilige Operation (+, -, ...).

#### Division und Modulo

Da bei der Division unmöglich zu garantieren ist, dass diese immer innerhalb von drei Takten erfolgreich beendet wird, muss das Leitwerk hier auf eine Bestätigung der ALU3 warten. Diese sieht vor, dass das Rechenwerk die Leitungen *alu\_data\_in* auf 0 setzt.

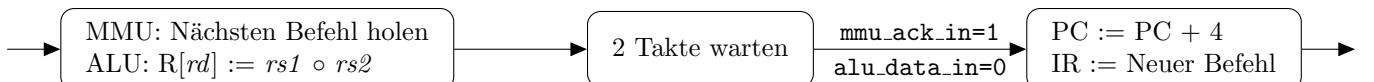


Abbildung 4: Zustandsübergangsdiagramm der Befehle **DIV[U]** und **REM[U]**. "○" repräsentiert die jeweilige Operation (/, mod, ...).

## 2.3 LUI und AUPIC

Da durch die Integer Rechenbefehle keine 32-Bit immediates direkt geladen werden können, definiert die RISC-V-ISA die Befehle **LUI** und **AUPIC**, die diesen Mangel beheben.



Abbildung 5: Zustandsübergangsdiagramm der Befehle **LUI** und **AUPIC**.  $op2$  ist entweder 0 (bei **LUI**) oder der aktuelle Program-Counter (bei **AUPIC**).

## 2.4 Bedingte Sprünge

Da bei Speicherzugriffen auf den DDR2-SDRAM Block der MMU4 nicht garantiert werden kann, dass ein Speicherzugriff ohne Zeit- und Datenverlust abgebrochen werden kann, wurde auf Branch-Prediction gänzlich verzichtet. Dadurch gehören die bedingten Sprünge im Hinblick auf Rechenzeit zu den teuersten Befehlen des Prozessors.

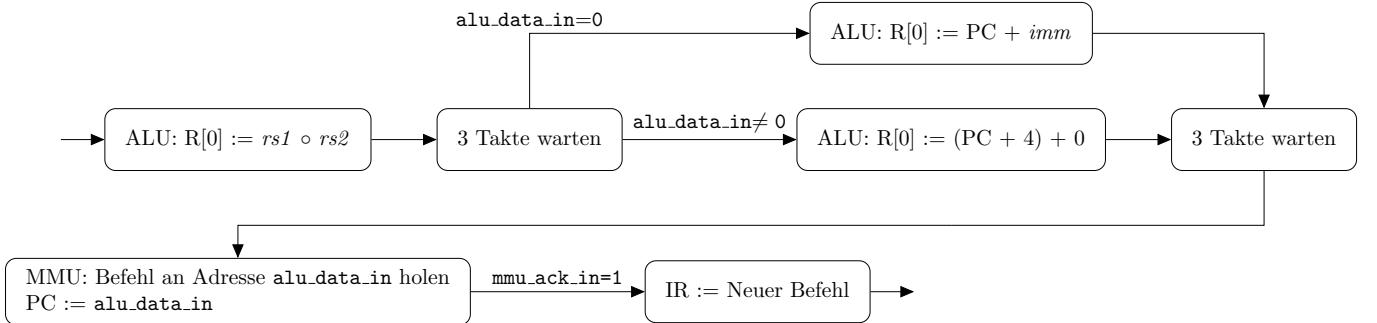


Abbildung 6: Zustandsübergangsdiagramm der Befehle **BEQ** und **BGE[U]**. “ $\circ$ ” ist bei **BEQ** “ $-$ ”, bei **BGE** die SLT-Operation und bei **BGEU** die SLTU-Operation.

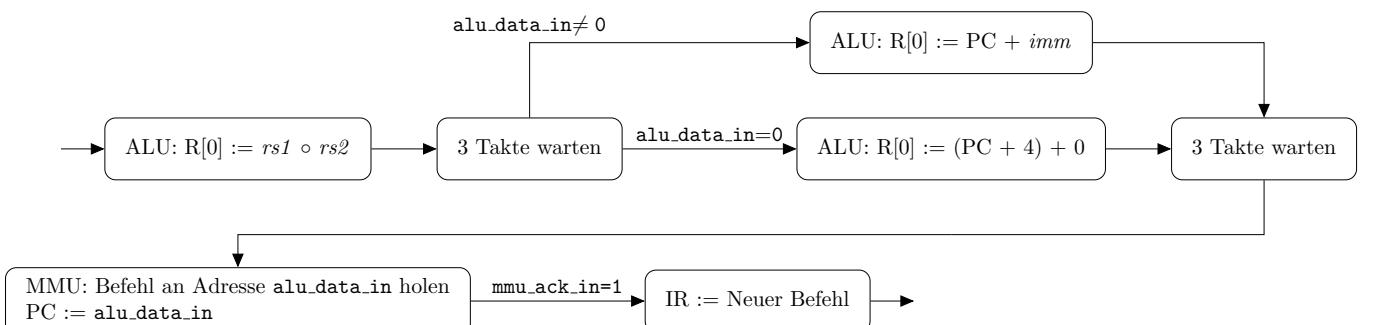


Abbildung 7: Zustandsübergangsdiagramm der Befehle **BNE** und **BLT[U]**. “ $\circ$ ” ist bei **BNE** “ $-$ ”, bei **BLT** die SLT-Operation und bei **BLTU** die SLTU-Operation.

## 2.5 Unbedingte Sprünge

Anders als bei bedingten Sprüngen, kann bei unbedingten Sprüngen das Sprungziel immer vorhergesagt werden. Dadurch kann das Schreiben der Return-Adresse und das Holen des nächsten Befehls parallelisiert werden, was zu einer merklichen Geschwindigkeitssteigerung führt.

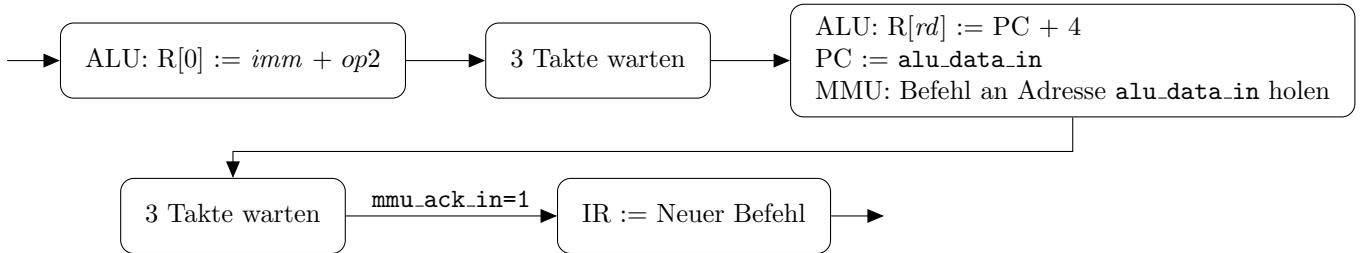


Abbildung 8: Zustandsübergangsdiagramm der Befehle **JAL** und **JALR**.  $op2$  ist bei **JAL** der Program-Counter und bei **JALR** das Register, welches durch  $rs1$  adressiert wird.

## 2.6 LOAD

Der LOAD-Befehl lädt aus dem Speicher immer einen 32-Bit Wert, den das Leitwerk dann zuschneidet. Dies sollte ursprünglich die Implementierung der MMU vereinfachen und aligned-Speicherzugriffe beschleunigen. Es hat sich allerdings herausgestellt, dass diese Entscheidung derzeit nur Nachteile mit sich bringt. Die Ausführungszeit des Befehls wird in der Praxis hauptsächlich von der adressierten Speicherkomponente bestimmt.

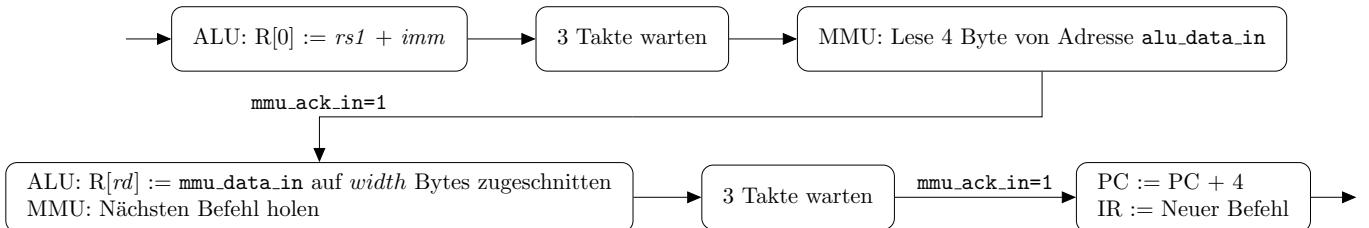


Abbildung 9: Zustandsübergangsdiagramm der Befehle **LB[U]**, **LH[U]** und **LW**.  $width$  ist je nach Befehl 1, 2 oder 4.

## 2.7 STORE

Der Store-Befehl ist mit nur einem Rechen- und Speicherwerk nicht zu parallelisieren, was dazu führt, dass er den langsamsten Befehl des Leitwerks darstellt. Da Schreibzugriffe jedoch generell auf RISC-Architekturen sehr hohe Kosten hinsichtlich der benötigten Rechenzeit aufweisen, sind Compiler und Programmierer ohnehin dazu angehalten, schreibende Speicherzugriffe möglichst zu vermeiden.

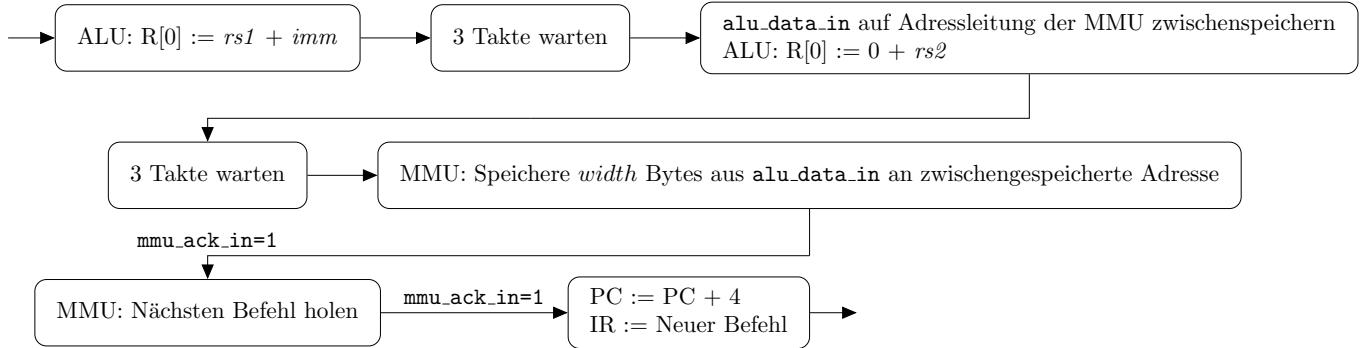


Abbildung 10: Zustandsübergangsdiagramm des Befehle **SB**, **SH** und **SW**. *width* ist je nach Befehl 1, 2 oder 4.

## 2.8 Timer und Counter

Wie in der RISC-V-ISA gefordert existiert des Weiteren ein Counter, der die Anzahl der bisher ausgeführten Befehle zählt. Außerdem gibt es einen Timer, der analog dazu die Anzahl der bereits vergangenen Takte speichert. Da das Entwicklungsboard allerdings keine Echtzeituhr bereitstellt, greift die Implementierung dieser Funktionalität auf den Taktzähler zurück.

Ausgelesen werden können diese Counter durch die Befehle **RDINSTRET[H]**, **RDCYCLE[H]** und **RDTIME[H]**.

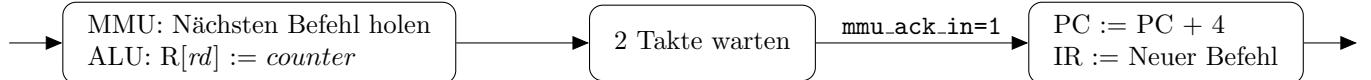


Abbildung 11: Zustandsübergangsdiagramm der Befehle **RDINSTRET[H]**, **RDCYCLE[H]** und **RDTIME[H]**. *counter* sind die oberen bzw. unteren 32 Bit des jeweiligen 64 Bit Zählers.

## 3 Die arithmetisch-logische Einheit

Die arithmetisch-logische Einheit *ALU* ist verantwortlich für die Durchführung aller für die Befehlausführung durch das Leitwerk relevanten Rechenoperationen. Zusätzlich verwaltet sie die Register des Prozessors.

### 3.1 Überblick

Zentrale Design-Idee hinter der ALU ist es, mehr als einen reinen Multiplexer für Befehle zu entwickeln. Stattdessen bietet sie dem Leitwerk ein Interface, über das Operationen auf Registern oder Immmediates angefragt werden können. Hierzu wurde eine Menge von Prozessor-internen Opcodes definiert.

Da die ALU auch zur Ausführung von nicht-arithmetischen Maschinenbefehlen, beispielsweise bei Sprüngen, oft beansprucht wird, sollte der Synchronisations- und Kommunikationsaufwand zwischen ALU und Leitwerk möglichst reduziert werden. Aus diesem Grund werden grundsätzlich alle Befehle (exklusive der Division) mittels einer Zustandsmaschine innerhalb von drei Takten ausgeführt, wodurch kein Synchronisations-Protokoll zwischen ALU und Leitwerk von Nöten ist.

Um die Komplexität des Leitwerks zu reduzieren, wurde die Low-Level-Verwaltung der Register in die ALU ausgelagert. Diese wurden als Dual-Port-Blockram realisiert, um die Anzahl an belegten Slices auf dem FPGA zu reduzieren.

## 3.2 Das Interface

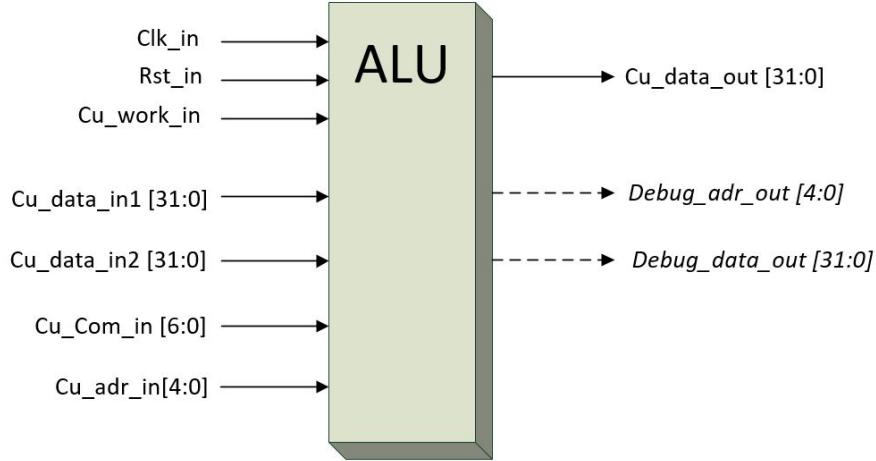


Abbildung 12: Schnittstelle der ALU mit Portnamen und -breiten

Neben zwei 32-Bit-Eingängen für Operanden, gibt es einen dedizierten Eingang für Registeradressen sowie einen zur Auswahl der gewünschten Operation. Der Adresseingang dient hierbei lediglich zur Adressierung des Zielregisters. Da alle Befehle nur aus zwei Operanden bestehen, werden die Operanden-Eingänge entweder zur Übermittlung von Immmediates, oder zur Adressierung der Operanden-Register verwendet, was die Anzahl der Eingänge reduziert. Aktiviert wird die ALU vom Leitwerk über `cu_work_in`, innerhalb von drei Takten liegt dann auf `cu_data_out` das entsprechende Ergebnis an.

## 3.3 Interne Befehle

Zur Auswahl der gewünschten Operation bietet die ALU einen 7 Bit breiten Befehls-Eingang an. Jeder Befehl besteht dabei aus drei Sektionen:

### Reg1 — Reg2 — OpC

Das oberste Bit *Reg1* entscheidet darüber, ob der erste Operand als Immediate vom Dateneingang `cu_data_in1`, oder aus dem durch `cu_data_in1` adressierten Register genommen wird. Analog wird anhand *Reg2* determiniert, ob auf `cu_data_in2` eine Adresse oder ein Immediate anliegt. Darauf folgt der fünf Bit umfassende interne OpCode. Zur Verfügung stehen OpCodes für:

- Addition, Subtraktion
- Logische Shifts
- Arithmetische Shifts
- Set Less Than Immediate Signed und Unsigned (SLT,SLTU)
- Multiply Lower auf Signed und Unsigned-Operanden
- Multiply Upper auf Signed und Unsigned Operanden
- Division Signed und Unsigned

- Modulo-Rechnung Signed und Unsigned

### 3.4 Befehlsausführung

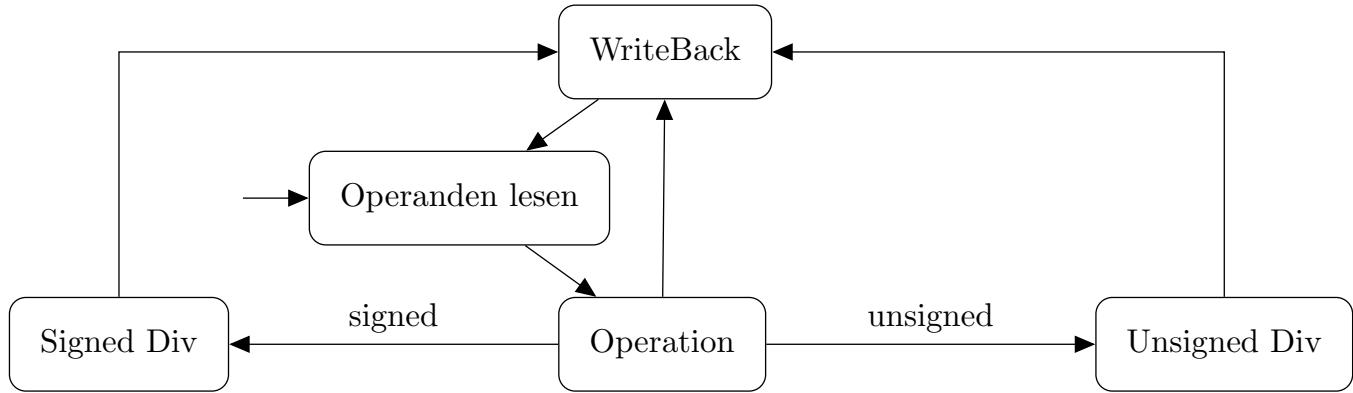


Abbildung 13: Zustandsübergangsdiagramm der Befehlsausführung innerhalb der ALU

Die Ausführung von Befehlen ist über eine State-Machine mit drei zentralen Zuständen, sowie zwei Zusatzzuständen für Signed- und Unsigned-Division bzw. Modulo-Operationen realisiert. In jedem der drei zentralen Zustände wird dabei nur für einen Takt verblieben.

#### State 1 - Selektion der Operanden

Die ALU wartet in State 1, bis vom Leitwerk das entsprechende Work-Signal gesendet wird. Anschließend werden auf Grundlage des oben beschriebenen Befehls zwei Operanden-Signale `s_op1` und `s_op2` mit einer Immediate vom Daten-Eingang belegt, oder aber es wird ein Register-Lesezugriff angestoßen, dessen Ergebnis dann im nächsten Takt zur Verfügung steht.

#### State 2 - Operationsausführung

State 2 dient der eigentlichen Befehlsausführung. Realisiert ist er als Case-Statement über den internen OpCode. Innerhalb jedes Cases wird zwischen den vier möglichen Kombinationen auf Immediate- und Register-Operanden unterschieden. Dies ist nötig, da das Ergebnis eines möglichen Register-Zugriffs erst in diesem Takt anliegt und deshalb nicht einfach die `s_op1`- und `s_op2`-Signale für Immediate-Operationen überschrieben werden können. Die Operation wird auf den jeweiligen Operanden durchgeführt und in einem Akkumulator gespeichert.

Ein Großteil der Operationen ist über die `IEEE.NUMERIC_STD.ALL` Operatoren realisiert und kann innerhalb dieses einen Taktes vollständig abgeschlossen werden.

Da die Standard-Multiplikation von 32-Bit-Werten zu einem 64 Bit langen Ergebnis führt, werden Multiplikationsergebnisse nicht im normalen Akkumulator-Signal `acc`, sondern im Zusatzsignal `mult-result` gespeichert. Der VHDL-Compiler erlaubte es nicht, unmittelbar auf das Ergebnis zuzugreifen und entweder die oberen oder unteren 32 Bit in `acc` zu speichern.

Da der Standard-Operator `sar` nicht durch die gegebene Entwicklungsumgebung synthetisierbar ist, wird bei einem arithmetischen Rechtsshift um  $n$  Stellen zusätzlich das oberste Bit des ersten Operanden zwischengespeichert, um im nachfolgenden State wenn nötig die oberen  $n$  Bits auf 1 zu setzen.

Im Falle einer Modulo- oder Unsigned-Division wird in State 2 der Übergang in State 4 bzw. 5 eingeleitet. Ansonsten erfolgt ein direkter Übergang in State 3.

### **State 3 - Write-Back**

In State 3 werden die akkumulierten Ergebnisse in das von der ALU adressierte Register geschrieben. Bei den meisten der Operationen kann dies unmittelbar erfolgen. Bei Multiplikationsoperationen allerdings muss jedoch zuerst an Hand des OpCodes entschieden werden, ob die oberen oder unteren 32 Bit des Ergebnisses gespeichert werden sollen. Abhängig vom Status-Bit für arithmetische Shifts um n Stellen werden, wie oben beschrieben, gegebenenfalls die oberen n Bits des Ergebnisses auf 1 gesetzt.

Neben dem Speichern des Ergebnisses werden zwei zusätzliche Ausgänge belegt:

Auf `cu_data_out` wird das Ergebnis angelegt. Einzige Ausnahme stellt die Division bzw. Modulorechnung dar, bei der als Synchronisationssignal der Daten-Ausgang mit 0 belegt wird. Das ist nötig, da die Division vom normalen 3-Takte-Schema abweicht. Auch die Debug-Schnittstelle erhält über `debug_signal` den entsprechenden Wert sowie das relevante Register über `debug_adr_signal`.

### **State 4 - Division Unsigned**

Um die Anzahl an nötigen Takten zu reduzieren, wird bei der Umsetzung der Unsigned Division eine durch den Xilinx-Core-Generator erstellte Divisionseinheit genutzt, welche eine gepipelinte Variante der SRT-Division durchführt. Obwohl der Prozessor keinen unmittelbaren Nutzen aus dem Pipelining zieht, kann durch die effiziente Implementierung die Anzahl an nötigen Takten reduziert werden. Zusätzlich liefert die Einheit sowohl den Rest, als auch das Divisionsergebnis, weshalb die beiden Operationen gleich behandelt werden können.

Hierzu werden bei Betreten des States die Operanden angelegt. Danach wird mit einem Zähler gewartet, bis das Ergebnis der Operation anliegt. Anschließend werden je nach Anfrage entweder der Rest oder das Ergebnis in den Akkumulator eingelesen und in State 3 übergegangen.

### **State 5 - Division Signed**

Die Implementierung verläuft analog zu State 4, abgesehen davon, dass eine Einheit zur Durchführung von Signed-Divisionen verwendet wird.

## **3.5 Die Register**

Insgesamt stellt die ALU 32 Register der Breite 32 Bit bereit, wobei Register x0 konstant mit dem Wert 0 belegt und nicht überschreibbar ist. Dies ist zum Beispiel nützlich beim unveränderten Laden eines Registers.

Implementiert wurden die Register als Dual-Port-Blockram. Dieses Vorgehen ermöglicht den zeitgleichen Zugriff auf zwei verschiedene Speicherinhalte, was gerade bei Register-Register-Operationen von Vorteil ist. Zusätzlich wird dadurch die Anzahl an verwendeten Slices reduziert, da dedizierte Block-Ram-Bausteine logisch zum Registersatz zusammengefügt werden.

Die Umsetzung erfolgte dabei nicht über eine Core-Generierte Variante, sondern durch die Einhaltung eines speziellen Verwendungsprotokolls, sodass der Compiler das definierte `std_logic_vector`-Array automatisch in einen Blockram umsetzt.

Voraussetzung dafür ist, dass nicht direkt auf Register-Inhalten operiert wird, sondern sie zuerst in einem Signal zwischengespeichert und im nächsten Takt verwendet werden. Dies ist aber ohnehin durch die 3-stufige State-Machine der ALU garantiert.

### 3.6 Reset

Die ALU verfügt zudem über einen Reset-Eingang, welcher direkt von der Toplevel-Komponente weitergeleitet wird. Bei einem Reset wird das Rechenwerk in den State 0 überführt, um nach Ende des Resets wieder Befehle des Leitwerks entgegennehmen zu können.

Zusätzlich wird der Divisions-Flankenzähler auf Null gesetzt, um keine Fehler bei nachfolgenden Operationen zu versursachen. Das Reset-Signal wird auch an den Reset (SCLR)-Eingang der Divisionseinheit weitergeleitet. Ein Reset der Register erfolgt nicht, sodass abgesehen vom Register x0 für den Entwickler eines Nutzerprogramms keine Aussage über die initialen Werte der Register möglich ist.

## 4 Die MMU

Die MMU (Memory Management Unit) verwaltet den in Blöcke gegliederten Speicherbereich und die darauf erfolgenden Zugriffe. Die Einheit bietet dabei eine Schnittstelle für lesende und schreibende Speicheranfragen, welche in unterschiedlichen Zeitintervallen bearbeitet werden.

### 4.1 Überblick

Der adressierbare Speicher innerhalb des Prozessors ist blockweise organisiert. Die MMU verwaltet einerseits die einzelnen Controller für die jeweiligen RAM-Blöcke und taktet andererseits die angefragten Zugriffe auf diese.

Sie ist aufgrund der überwiegend sehr ähnlichen Adressierungsprozeduren intern durch eine Statemachine realisiert, welche anhand einer Speicheradresse die jeweiligen Speicheranfragen an den dem RAM-Block entsprechenden Controller weiterleitet.

Um eine reibungslose Kommunikation mit diesen Controllern zu gewährleisten, ist die MMU mit einer vom restlichen Prozessor unterschiedlichen Frequenz, 133 MHz, getaktet. Dies ist vor allem im Bezug auf den integrierten DDR2-SDRAM<sup>4</sup>, welcher mit eben diesem Takt versorgt werden muss, um Daten halten zu können, begründet: Die Tatsache, dass die Integration dieser Komponente besonders zeitaufwändig verlief, rechtfertigt diese Designentscheidung. Daraus resultieren zusätzlich benötigte Synchronisations- und Kommunikationsmechanismen mit dem übergeordneten Leitwerk.

---

<sup>4</sup>Genauer handelt es sich um einen Micron Technology DDR2-SDRAM (MT47H32M1)

## 4.2 Interface

Das Interface der MMU untergliedert sich hauptsächlich in drei verschiedene Komponenten: Einerseits Signale zur Kommunikation mit dem Leitwerk, andererseits durch das Toplevel-Modul nach oben geleitete Signale zur Adressierung des DDR2-SDRAMs, welche vom in der MMU verwalteten Controller generiert werden, und zuletzt nach oben geleitete Daten- und Adressleitungen, die die ASCII-Einheit konstant mit Daten versorgt.

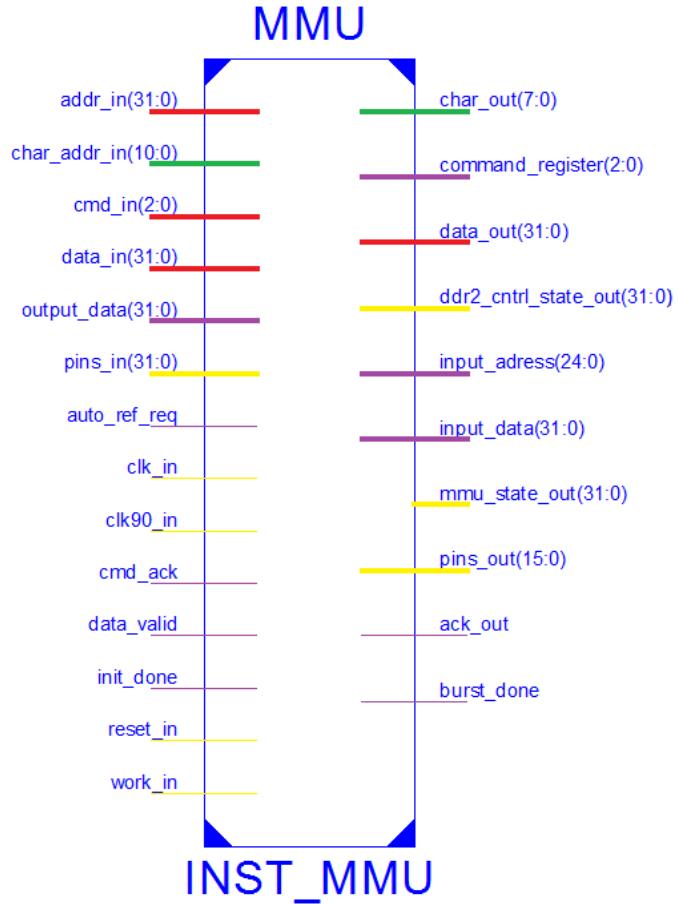


Abbildung 14: Schnittstelle der MMU-Einheit, Signale farblich gruppiert: *Rot* = Kommunikation mit Leitwerk, *Grün* = Character-Ausgabe an ASCII-Einheit, *Violett* = Verbindung mit DDR2-SDRAM, *Gelb* = Sonstige

### Kommunikation mit dem Leitwerk

Die Kommunikation mit dem Leitwerk lässt sich im Wesentlichen in Daten-, Adress- und Synchronisationsleitungen untergliedern. Dabei sendet das Leitwerk über das Signal `cmd_in` die angefragte Operation. Die MMU reagiert allerdings erst auf ein Setzen des `work_in` Signals mit der Bearbeitung der Anfrage. Der 3-Bit-Vektor `cmd_in` ist wie folgt aufgebaut:

Als Ausgabe liefert die MMU dem Leitwerk einerseits ein Acknowledgement `ack_out`, welches signalisiert, dass die MMU bereit ist, eine neue Anfrage zu bearbeiten und indirekt damit auch Auskunft darüber gibt, ob die bereits gesendete

Bit 2	Bit 1 - 0
0 := Read, 1 := Write	0 := 8 Bit, 1 := 16 Bit, 3 := 32 Bit

Tabelle 2: Aufbau des Kommunikationssignals zwischen Leitwerk und MMU

Anfrage erfolgreich bearbeitet wurde. Im Falle eines lesenden Speicherzugriffs wird, sofern das Acknowledgement-Signal den Wert 1 angenommen hat, gewährleistet, dass der Datenausgang `data_out` korrekt belegt wurde. Es sei angemerkt, dass während im Zuge der Entwicklung und der stark überwiegenden Zahl der 32-Bit Lesezugriffe (vor allem im Zuge des Ladens eines Befehls) beschlossen wurde, dass jeder lesende Speicherzugriff ungeachtet der im `cmd_in` definierten Datenbreite 32-Bit liest. Theoretisch bietet die MMU allerdings auch die Möglichkeit, 8-Bit oder 16-Bit Lesezugriffe durchzuführen.

#### Durch die MMU geleitete Signale an andere Komponenten

Wie eingangs erwähnt lassen sich die übrigen Signale dem Weiterleiten von Signalen aus einerseits dem Controller der DDR2-SDRAM-Komponente sowie den kontinuierlichen lesenden Anfragen der ASCII-Einheit an den entsprechenden CHARRAM-Block (die parallel und unabhängig von der Funktionalität der MMU laufen) zuordnen und werden hier nicht weiter vertieft.

### 4.3 Aufbau des Speichers

Wie bereits geschildert wird der Speicher in verschiedene Bereiche unterschiedlicher Größe untergliedert. Jeder dieser Bereiche wird von einem Controller verwaltet, welcher bei einer eingehenden Anfrage durch die MMU angesprochen wird. Dabei hängt die Bearbeitungsdauer maßgeblich vom adressierten Speicherblock ab.

Aus der durch den Prozessor implementierten Wortgröße von 32 Bit ergibt sich ein Adressraum, der potenziell  $2^{32}$  Speicherzellen mit einer Größe von je 8 Bit umfasst. Dass nicht jeder dadurch zur Verfügung stehende Bereich auch tatsächlich auch nutzbar ist, lässt sich auf die vom FPGA zur Verfügung gestellten Speicherressourcen zurückführen. Stattdessen wird die Adresse in ein Präfix, welches den adressierten Speicherbereich bestimmt, und ein Offset innerhalb dieses Speicherblocks wie folgt unterteilt:

Bit 31 - 28	Bit 27 - 0
Präfix	Offset

Tabelle 3: Aufbau des Adressvektors

Insgesamt existieren fünf zulässige Werte für das 4-Bit Präfix, wobei Zugriffe auf nicht gültige Speicherpräfixe nicht verarbeitet werden. Zudem unterscheiden sich die Größen der jeweiligen Speicherblöcke von dem potenziell 28-Bit großen Raum innerhalb eines Blocks. Aufgrund der Tatsache aber, dass diese sich stets als natürliche Potenz von 2 darstellen lassen, kann durch Spiegelung des tatsächlich nutzbaren Speicherraums der gesamte vom Offset darstellbare Bereich adressiert werden. Im Endeffekt wird der Offset also lediglich in seiner wirksamen Größe entsprechend des Speicherblocks beschnitten. Die folgende Tabelle zeigt die implementierten Speicherblöcke sowie deren nutzbare Größe.

Präfix	Kürzel	Größe in Bytes	Kurzbeschreibung
0x0	BIOS	$2^{11}$	Programmeinsprungspunkt
0x1	SDRAM	$2^{16}$	DDR2-SDRAM <sup>5</sup>
0x2	CHARRAM	$2^{11}$	Character-Anzeige
0x3	IORM	$2^3$	Memory-Mapped I/O
0x4	SERIALRAM	$2^{11}$	Serielle Schnittstelle

Tabelle 4: Aufbau des Speichers als Blöcke

Dabei sind alle Blöcke, ausgenommen der DDR2-SDRAM-Block, durch auf dem FPGA verfügbaren Dual-Port-Speicher realisiert, sodass die implementierten Controller im Groben gleich sind. Angemerkt sei an dieser Stelle, dass - in Absprache mit dem Betreuer - der Controller für den DDR2-SDRAM eine Implementierung von Opencores<sup>6</sup> verwendet und entsprechend den Anforderungen abgeändert wurde.

Jeder Speicherbereich wird von der MMU im Little-Endian-Format adressiert, was insbesondere bei 16-Bit beziehungsweise 32-Bit Zugriffen berücksichtigt werden muss.

#### 4.4 Memory-Mapped I/O

Einer der geschilderten Speicherblöcke, genauer der IORM, stellt die Schnittstelle zwischen Benutzer und Programmcode dar. Dabei sind einige der auf dem FPGA verfügbaren Ein- und Ausgabemöglichkeiten direkt auf einzelne Bits innerhalb der Speicherzellen des IORMs gemappt. Aus den acht verfügbaren Speicherzellen sind folgende sechs nutzbar:

Zelle	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x0	Read-Only	BTN 0	-	-	-	-	-	-	-
0x1	Read-Only	SW 3	SW 2	SW 1	SW 0	BTN 4	BTN 3	BTN 2	BTN 1
0x2	Read/Write	LED 7	LED 6	LED 5	LED 4	LED 3	LED 2	LED 1	LED 0
0x3	Read/Write	-	-	-	-	-	-	-	-
0x4	Read-Only	UART 7	UART 6	UART 5	UART 4	UART 3	UART 2	UART 1	UART 0
0x5	Read-Only	-	-	-	-	-	-	UART VALID	UART ERR
0x6	Read/Write	-	-	-	-	-	-	-	-
0x7	Read/Write	-	-	-	-	-	-	-	-

Tabelle 5: Memory-Mapped I/O im Detail

Dabei steht BTN jeweils für entsprechende Buttons auf dem Board, SW entspricht einem Schalter und LED den Ausgabe-LEDs. Außerdem sind die Eingabedaten der seriellen Schnittstelle in Form eines 8-Bit Vektors sowie einem Bestätigungssignal, dass dieser vollständig übertragen wurde und einem Fehlersignal, das ebenfalls von der seriellen Schnittstelle ausgeht, ebenfalls auf den IORM gemappt. Angemerkt sei an dieser Stelle aber, dass der Prozessor nicht schnell genug taktet, um diese Funktionalität wirklich sinnvoll zu nutzen, weswegen zur Initialisierung des Programm speichers auch eine andere Methode verwendet wird. Bits, die nicht genutzt und in der Tabelle mit ‘-’ vermerkt sind, entsprechen stets dem konstanten Wert 0 und bieten daher auch keinerlei Speicherfähigkeit.

Die nachfolgende Abbildung 4.4 zeigt, wo sich welches Ein-/Ausgabesignal auf der Hardware wiederfindet. Dabei entsprechen die Bezeichner denen aus der zuvor abgebildeten Tabelle.

<sup>5</sup>Von 512 MBit verfügbaren Speicherplatz macht der genutzte Controller nur  $2^{16}$  Bytes zugänglich

<sup>6</sup>[http://opencores.org/project,ddr2\\_sdram](http://opencores.org/project,ddr2_sdram)



Abbildung 15: Verteilung der I/O-Signale

#### 4.5 Implementierung als State Machine

Aufgrund der identisch aufgebauten Controller für die meisten Speicherblöcke ist die MMU durch eine State Machine realisiert. Dabei die MMU zwar im State MMU-IDLE initialisiert, wechselt bei einem Reset den Zustand aber sofort zu MMU-RESET.

Das nachfolgende Zustandsübergangsdiagramm veranschaulicht die durch einen Lese- oder Schreibzugriff entstehende Prozedur.

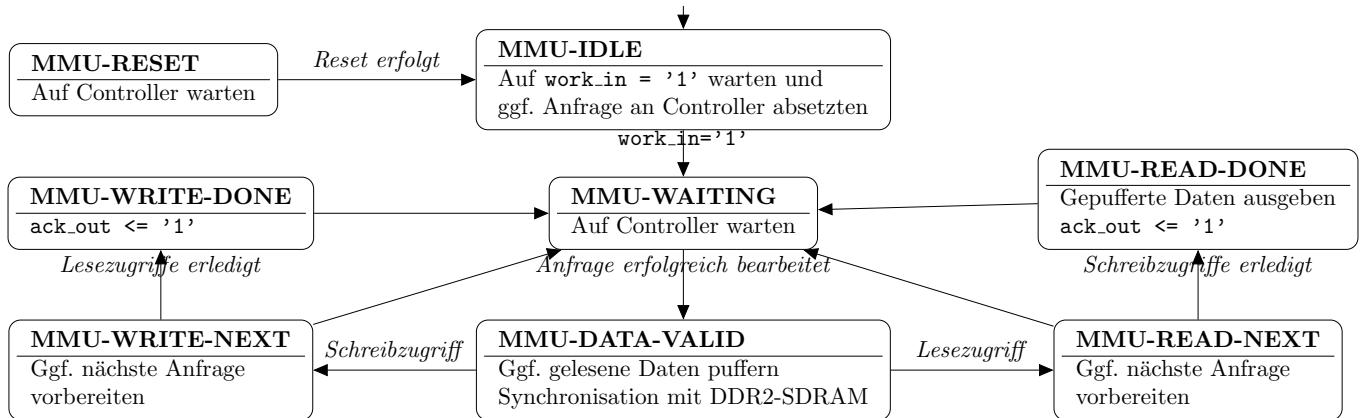


Abbildung 16: Zustandsübergangsdiagramm der MMU. Übergänge zum Zustand MMU-RESET, die von jedem anderen Zustand durch `rst_in = '1'` ausgelöst werden, sind der Übersicht halber ausgelassen.

## **MMU-IDLE**

Im Zustand MMU-IDLE wartet die Einheit auf das eingehen eines Befehls über das `cmd_in` Signal. Solange dies nicht erfolgt, wird das Synchronisationssignal `ack_out` mit dem Wert 1 belegt. Andernfalls wird die Eingabeadresse ausgewertet, die durchzuführende Operation an den entsprechenden RAM-Controller weitergeleitet und die Einheit in den Zustand MMU-WAITING überführt.

## **MMU-WAITING**

Durch diesen Zustand wird die Ausführung eines Lese- oder Schreibzugriffs solange verzögert, bis der entsprechende RAM-Controller die 8-Bit Anfrage erfolgreich bearbeitet hat. Im Fall des DDR2-SDRAM-Controllers erfolgt dies über ein Synchronisationssignal, anderenfalls kann mit einer festen Wartedauer von einem Takt gerechnet werden. Während des Wartevorgangs werden die Befehle, die am entsprechenden RAM-Controller anliegen, entfernt.

Sollte der Controller bereit sein, eine neue 8-Bit Anfrage zu verarbeiten, wird die MMU in den Zustand MMU-DATA-VALID überführt.

## **MMU-DATA-VALID**

In diesem Zustand werden, sofern es sich bei der zuletzt durchgeföhrten Operation um einen Lesezugriff gehandelt hat, die gelesenen Daten mittels eines Buffer-Signals zwischengespeichert. Außerdem wird die Einheit je nach Zugriffsmodus in den Zustand MMU-READ-NEXT beziehungsweise MMU-WRITE-NEXT überführt.

## **MMU-READ-NEXT**

Sofern noch weitere 8-Bit Zellen gelesen werden müssen, wird eine neue lesende Anfrage an den entsprechenden RAM-Controller weitergeleitet und die Einheit wieder in den Zustand MMU-WAITING überführt. Andernfalls wechselt die MMU in den Zustand MMU-READ-DONE.

## **MMU-READ-DONE**

Alle gelesenen und zwischengespeicherten Daten werden gemäß dem Little-Endian-Encoding zusammengesetzt und auf der Datenausgabeleitung an das Leitwerk übergeben. Damit einher geht das Setzen des Acknowledgements-Signals `ack_out` auf den Wert 1 sowie der Zustandswechsel nach MMU-IDLE.

## **MMU-WRITE-NEXT**

Sollte der vom Leitwerk geforderte Schreibzugriff weitere schreibende 8-Bit Zugriffe erfordern, wird in diesem Zustand der der Adresse entsprechende RAM-Controller mit neuen Daten und einer inkrementierten Adresse angesprochen und die MMU-Einheit in den Zustand MMU-WAITING überführt. Andernfalls wechselt die MMU in den Zustand MMU-WRITE-DONE.

## **MMU-WRITE-DONE**

Da nach einem erfolgreichen Schreibzugriff keinerlei Ausgabedaten übermittelt werden müssen, wird in diesem Zustand lediglich das Acknowledgement-Signals `ack_out` auf den Wert 1 gesetzt sowie die Einheit zurück in den Zustand MMU-IDLE überführt.

## **MMU-RESET**

Bei einem Reset wechselt die MMU ungeachtet ihres derzeitigen Zustands in den MMU-RESET Zustand und unterbricht alle derzeitigen Anfragen ausnahmslos. Da die Einheit das eingehende Reset-Signal an alle RAM-Controller

weitergeleitet, wird in diesem Zustand lediglich auf die Beendigung der Resets jedes einzelnen Controllers gewartet. Sollten diese wieder bereit für neue Anfragen sein, wechselt die MMU wieder in den MMU-IDLE Zustand.

## 4.6 Zugriffsdauer

Aus dem geschilderten detaillierten Ablauf eines Zugriffs innerhalb der MMU lassen sich nun für die einzelnen Speicherblöcke die exakten Zugriffsdauern beziehungsweise im Fall des DDR2-SDRAMs, welcher unter Umständen durch einen periodisch auftretenden Auto-Refresh eine erhöhte Zugriffszeit benötigt, eine Mindestzugriffsdauer errechnen.

Datengröße	(minimale) Zugriffsdauer
8-Bit	4 Takte
16-Bit	7 Takte
32-Bit	13 Takte

Tabelle 6: Zugriffsdauer einzelner Speicherzugriffe

Aufgrund der identischen Struktur der RAM-Controller für als Dual-Port-Blockram realisierte Speicherbereiche ergeben sich für jene Speicherblöcke identische Zugriffszeiten sowohl für lesende und schreibende Zugriffe, wobei eine 8-Bit Anfrage immer genau einen Takt kostet. Da der DDR2-SDRAM für seine Zugriffsdauer im Bezug auf lesende und schreibende Operationen nicht nach oben hin abgeschätzt werden kann, jedoch keinesfalls weniger als einen Takt brauchen wird, entsprechen die Mindestzugriffszeiten ebenfalls der oben dargestellten Tabelle.

## 5 Die ASCII-Unit

Die ASCII-Unit ist für die Textausgabe auf dem Monitor zuständig.

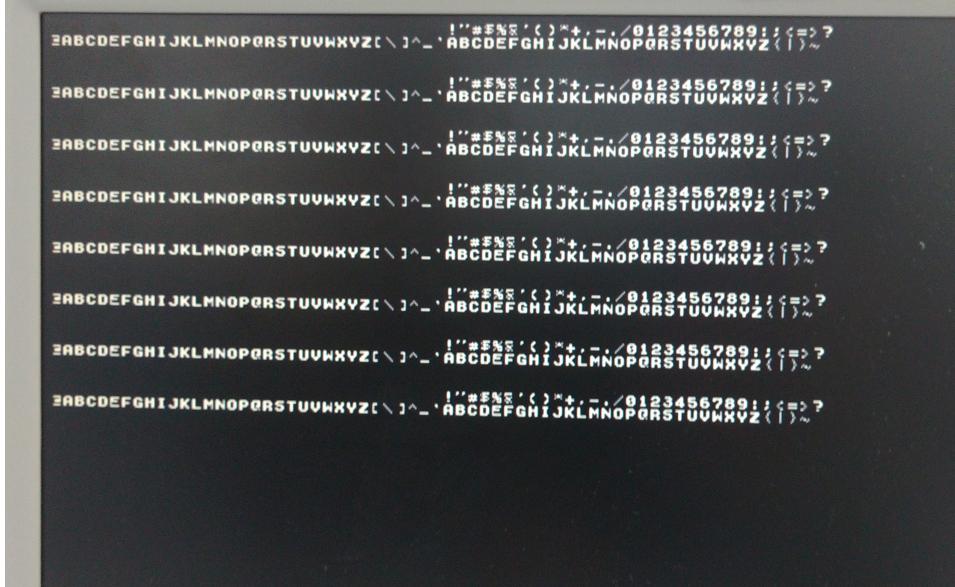


Abbildung 17: Beispiel für die Textausgabe: jedes darstellbare Zeichen wird abgebildet

## 5.1 Überblick

Die ASCII-Unit gibt auf dem Monitor mittels Memory-Mapping den Inhalt des CHARRAMs gemäß ASCII-Kodierung aus. Dazu wird den Zeichen-Stellen auf dem Monitor jeweils eine Adresse zugeordnet: Die Stelle im Eck oben links erhält die 0, nach rechts wird bis zum Zeilenende durchinkrementiert, dann wird jeweils in der nächsten Zeile fortgefahren, sodass sich insgesamt 32 Zeilen à 64 Zeichen ergeben und die letzte Stelle rechts unten die Adresse 2047 erhält. Diese Adressen beziehen sich auf die 2048 Byte des CHARRAMs in der MMU, die eben jeweils genau ein Zeichen repräsentieren.

Um jedem Zeichen neben seiner ASCII-Nummer auch das entsprechende Aussehen zuordnen zu können, wurde auf eine CHARMAP gesetzt, die zu jedem der 256 ASCII Zeichen einen 64-Bit-Vektor abgespeichert hat, der das 8\*8 Pixelfeld eines jeden Zeichens repräsentiert, wobei die Zeilen eines jeden Zeichens dazu konkateniert wurden. Von diesen 8\*8 Bit sind jeweils nur 6\*6 für das Zeichen reserviert, die Restlichen sind immer ungesetzt, sodass auf dem Monitor ein Abstand von 2 freien Pixeln zwischen zwei nebeneinanderliegenden Zeichen bleibt. Diese Entscheidung auf Kosten der Speichervergeudung ermöglichte eine einfachere Implementierung, da für die Berechnungen ASCII-Unit Divisionen und Multiplikationen notwendig sind und diese mit Zweierpotenzen (hier 8 bzw. 64) erheblich einfacher zu realisieren sind.

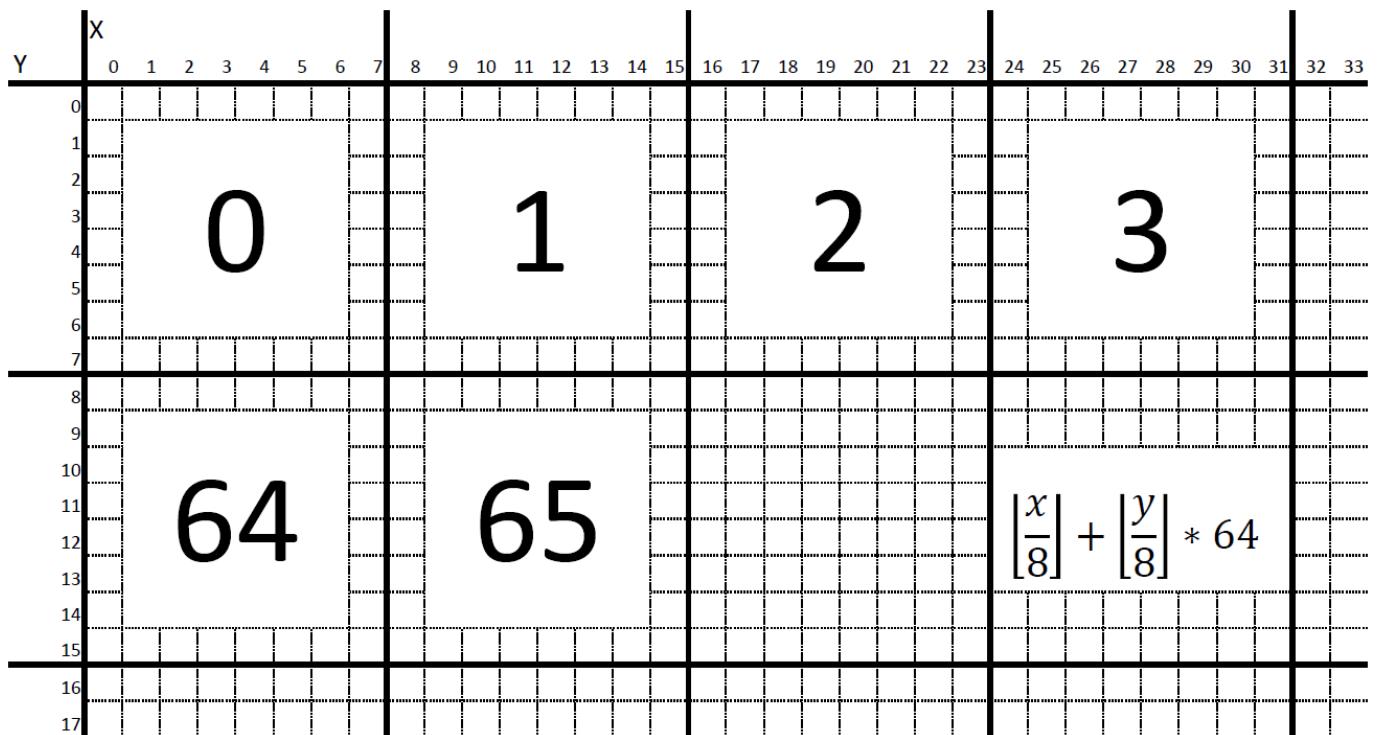


Abbildung 18: Veranschaulichung der Adressberechnung anhand des Pixelfelds des Monitors: Links oben wird von 0 ab nach rechts zeilenweise hochgezählt. Die Adresse des Zeichenfeldes zu dem ein Pixel mit Position (x,y) gehört wird wie folgt berechnet:  $\lfloor \frac{x}{8} \rfloor + \lfloor \frac{y}{8} \rfloor * 64$ . Aus dem 64-Bit-Vektor berechnet man das Bit für das aktuelle Pixel so:  $(x \bmod 8) + (y \bmod 8) * 8$ .

Da eine Darstellung von Kleinbuchstaben auf einem 6\*6 Bitfeld kaum sinnvoll zu realisieren ist, eine Unterscheidung ohnehin schwierig wäre und der Fokus nicht auf Ästhetik lag wurden diese durch Großbuchstaben ersetzt. Nicht zeichenbare ASCII-Zeichen (Zeilenumbrüche, Tabulator etc.) werden als leere Zeichen dargestellt, sodass bei der Berechnung der Adresse nicht auf vorangegangene Sonderzeichen geachtet werden muss, was die Implementierung erleichtert. Dadurch muss allerdings die Software die Verwaltung des CHARRAMs übernehmen, um diese Zeichen korrekt darzu-

stellen.

Die ASCII-Unit ist wie die VGA-Unit auf 25 MHz getaktet. Denn die VGA-Unit sendet in jedem ihrer Takte die Informationen zu einem Pixel zum Monitor und die ASCII-Unit berechnet in jedem Takt genau die Information, ob das jeweilige Pixel gesetzt oder frei, sprich Weiß oder Schwarz sein soll. Die Berechnung des Pixels in der ASCII-Unit erfolgt dabei über mehrere Takte hinweg stufenweise und für jedes Pixel in jedem Frame neu, sodass gleichzeitig für verschiedene Pixel in verschiedenen Stufen eine Berechnung stattfindet.

## 5.2 Interface

Neben dem bereits erwähnten Takteingang gibt es jeweils einen Eingang für die x- und y-Koordinate des aktuellen Pixels (bzw. Position des Fadenstrahls) aus der VGA-Unit, welche mit jedem Takt aktualisiert werden und ein Ausgangssignal an die VGA-Einheit, welches angibt, ob das aktuelle Pixel gesetzt werden soll oder nicht.

Außerdem gibt es zur Kommunikation mit dem CHARRAM in der MMU einen Ausgang, welcher die Adresse des Zeichenplatzes des aktuell zu berechnenden Pixels angibt sowie einen Eingang, der im darauffolgenden Takt die ASCII-Nummer des zugehörigen Zeichens erhält.

Zur CHARMAP, die als ROM fungiert, wird der Takt durchgeleitet und ebenso die aus dem CHARRAM kommende ASCII-Nummer des aktuellen Zeichens, die hierbei als Adresse fungiert. Aus der CHARMAP kommt im darauffolgendem Takt der oben erwähnte 64-Bit-Vektor der das jeweilige Zeichen repräsentiert.

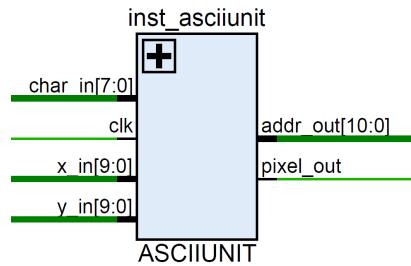


Abbildung 19: Das Interface der ASCII-Unit

## 5.3 Funktionsweise

Die stufenweise Berechnung für jedes Pixel beginnt mit der Verrechnung der x- und y-Koordinate aus der VGA-Einheit. Dazu wird zuerst die x-Koordinate um 2 erhöht und in ein internes Signal gespeichert, sodass quasi im Vorraus berechnet wird, ob ein Pixel gesetzt werden wird oder nicht.

Im nächsten Takt wird daraus die Adresse des aktuellen Zeichen-Platzes berechnet, welche dann durch die MMU in den CHARRAM geleitet wird.

Von dort wird im anschließenden Takt die ASCII-Nummer des aktuellen Zeichens geliefert, welche direkt in die CHARMAP weitergeleitet wird. Dies ist auch problemlos bei gleichzeitigem Zugriff der MMU auf den CHARRAM möglich, da dieser als Dual-Port-Blockram realisiert wird. Auch die schnellere Taktung der MMU stellt keine Herausforderung dar: Der jeweilige Lesezugriff erfolgt einfach mehrmals hintereinander. Lediglich falls zwischen den insgesamt 64 Zugriffen pro Frame der ASCII-Unit auf eine Adresse im CHARRAM eben diese Zelle von der MMU überschrieben wird kann es zu einer kurzzeitigen Störung kommen: Dieses Zeichen würde dann möglicherweise in diesem Frame falsch dargestellt, was sich im Betrieb allerdings kaum bemerkbar macht.

Die CHARMAP liefert im folgenden Takt wiederum den aktuellen 64-Bitvektor aus dem mittels der aktuellen x- und y-Koordinate berechnet wird, ob das Pixel zu setzen ist oder nicht.

Diese Information wird dann im letzten Takt an die VGA-Unit zurückgesandt.

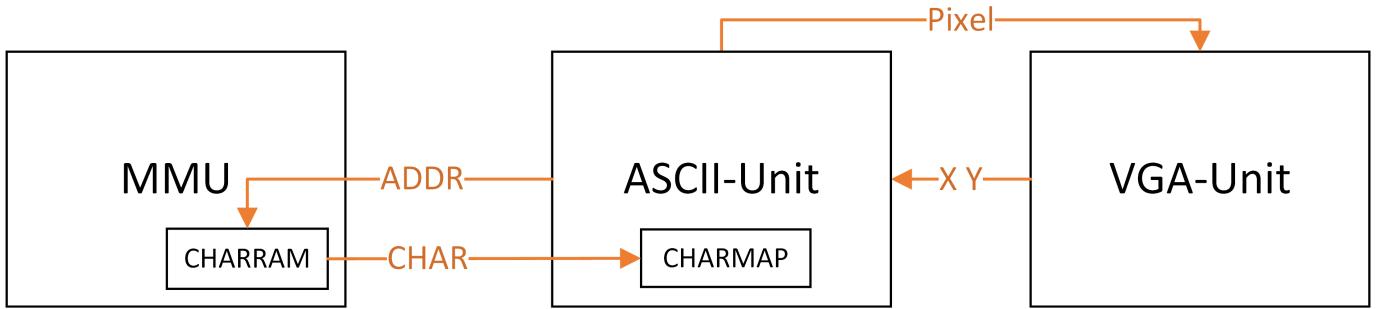


Abbildung 20: Übersicht: Beginn bei x- bzw. y-Koordinate; Verrechnung dieser zur Adresse für CHARRAM; dann Durchleiten in CHARMAP; anschließend Berechnung des Pixels; letztendlich Zurücksenden an VGA-Unit

## 6 Benutzerschnittstelle

Im Folgenden wird die Benutzung des auf dem FPGA realisierten RISC-V Prozessors geschildert, wobei im Besonderen auf die Nutzerschnittstelle eingegangen wird. Um das FPGA in Betrieb zu nehmen, muss dieses logischerweise an eine entsprechende Stromquelle angeschlossen sein.

### 6.1 Initialisierung

Im Allgemeinen kann ein Assemblerprogramm, das aus den implementierten Instruktionen zusammengesetzt ist, auf zwei Weisen in den Programmspeicher geladen und dann auf dem Prozessor ausgeführt werden.

#### Initialisierung durch BIOS und serielle Schnittstelle

Die Programm-Einsprungsadresse, bei welcher die CPU die Ausführung nach Beendigung eines Resets beginnt, Befehle abzuarbeiten, ist die erste Adresse des BIOS-RAM-Blocks *0x00000000*. Dieser RAM-Block wird provisorisch mit einem einfachen BIOS-Programmcode initialisiert:

```

lui x1, 0x40000
jalr x0, x1, 0

```

Dieser Programmcode führt die Ausführung in einem anderen RAM-Block fort, dem SERIALRAM, welcher durch das Präfix *0x4* auf die Speicheradresse *0x40000000* gemappt ist. Der Programmierer muss zuvor also gewährleisten, dass dieser Speicherbereich, welcher durch die serielle Schnittstelle UART gefüllt wird, mit ausführbarem Programmcode initialisiert wurde.

Indem der Programmierer nach einem Reset den Programmstart bei oben erwähnter Einsprungsadresse durch das Halten des entsprechenden Blockadesignals auf Switch 4 (siehe Abbildung 4.4) verzögert, bleibt Zeit währenddessen über die UART Schnittstelle den SERIALRAM-Speicherblock zu initialisieren. Indem das Signal auf Switch 4 wieder entfernt wird, beginnt der Programmablauf wie eingangs erwähnt im BIOS-RAM. Da die serielle Schnittstelle nur für begrenzte Datenmengen verlässlich und fehlerfrei funktioniert, können keine größeren Programme auf diese Weise ausgeführt werden.

#### Initialisierung durch initiales Beschreiben des BIOS-RAM Blocks

Alternativ besteht auch die Möglichkeit, den BIOS-RAM Block direkt beim Beschreiben des FPGAs mit dem auszuführenden Programmcode zu initialisieren. Da der BIOS-RAM-Block als Dual-Port-Blockram realisiert wurde, kann dies ohne zeitlichen Aufwand bei einem Reset erfolgen.

Das BIOS wird auf diese Weise zweckentfremdet und der SERIALRAM Block bleibt ungenutzt. Auch muss das FPGA zur Ausführung eines anderen Programms jedes Mal neu beschrieben werden. Trotz der genannten Nachteile aber bleibt diese Methode vorerst die einzige verlässliche Vorgehensweise, auch größere Datenmengen auf das Board zu übertragen. Sie findet beispielsweise im Fall des später beschriebenen Demoprogramms Anwendung.

## 6.2 Der Reset

Nachdem das FPGA mit entsprechender Methodik initialisiert wurde oder aber nachdem eine Programmausführung terminiert hat, muss der Prozessor zurückgesetzt werden. Dies geschieht durch das kurzzeitige Halten des Reset-Signals, welches auf den Hardware-Switch *SW 0* (siehe Abbildung 4.4) gemappt ist.

Zu beachten ist, dass der Reset nur im verlangsamten Ausführungsmodus (5 Hz) verlässlich funktioniert. Ob ein Reset im schnellen Ausführungsmodus (50 MHz) erfolgreich verläuft, ist nicht deterministisch und daher auch nicht zu empfehlen.

## 6.3 Graphische Oberflächen

Im Allgemeinen existieren zwei Methoden die Programmausführung graphisch darzustellen: Einerseits existiert der Debugging-Modus, in welchem die Werte aller verfügbaren Register sowie des Programmzählers und des Instruktionsregisters ausgegeben werden. Andererseits wird gleichzeitig der Speicherbereich des CHARRAM-Blocks auf eine ASCII-Darstellung gemappt. Mittels des Hardware-Switches *SW 2* (siehe Abbildung 4.4) wird bestimmt, welcher der beiden Modi auf dem VGA-Ausgang ausgegeben wird. Dabei bedeutet ein Halten des Switches die Darstellung über ASCII und ein Loslassen die direkte Ausgabe der Register innerhalb der CPU.

### Debugging-Modus

Der Debugging-Modus zeichnet die verfügbaren 32 Register x0-x31 bitweise auf das VGA-Ausgabegerät. Dabei werden die Register farblich voneinander getrennt und zeilenweise fortlaufend durchnummeriert (pro Zeile vier Register). Gesetzte Balken entsprechen einem gesetzten Bit innerhalb des Registerwerts, wobei das *least significant Bit* links, und das *most significant Bit* rechts dargestellt wird. Die beiden separat dargestellten Register sind der Programmzähler (links) sowie das Instruktionsregister (rechts), ebenfalls bitweise abgebildet.



Abbildung 21: VGA-Ausgabe im Debugging-Modus: PC und IR liegen zentral

Da die VGA-Ausgabe mit 25 MHz getaktet wird, der Prozessor im schnellen Ausführungsmodus dagegen den dreifachen Takt erhält, wechseln die Registerwerte unter Umständen schneller, als sie dargestellt werden können, wodurch es zu erheblichen Darstellungsfehlern kommen kann. Der Debugging-Modus ist dementsprechend, wie der Name bereits suggeriert lediglich für Debugging-Zwecke im langsamen Ausführungsmodus sinnvoll nutzbar.

## ASCII-Modus

Der ASCII-Modus stellt zeilenweise fortlaufend durchnummiert die Werte der Speicherzellen innerhalb des CHAR-RAM-Blocks als ASCII-Zeichen dar. Die genaue Abbildungsmethodik wird im entsprechenden Kapitel 5 näher erläutert. Um das Demo-Programm sinnvoll zu benutzen, wird dieser Ausführungsmodus empfohlen, da er einerseits auch im schnellen Ausführungsmodus aufgrund der Seltenheit von schreibenden Speicherzugriffen im entsprechenden CHAR-RAM-Block konsistenter Daten anzeigt, und andererseits das Programm so konzipiert ist, dass das Tic-Tac-Toe Spiel die ASCII-Schnittstelle zur Benutzerinteraktion vorsieht.

## 6.4 Benutzereingabe

Analog zur Ausgabe von Daten bietet der Prozessor auch eine Möglichkeit zur Dateneingabe durch den Benutzer. Wie im entsprechenden Kapitel 4.4 genauer erläutert wird, sind alle Buttons, Switches und LEDs auf Speicherzellen gemappt. Ein Halten oder Loslassen der entsprechenden Komponente führt zu einem Wechsel des Bits an entsprechender Speicheradresse.

Programmierer haben so die Möglichkeit Benutzereingaben abzufragen oder auf diese zu reagieren, müssen das aber innerhalb ihrer Implementierung selbst tun, da keinerlei Interrupts oder dergleichen bereitgestellt werden.

# 7 Entwicklung eines Demo-Programms

Das folgende Kapitel erläutert die Struktur, Motivation und Vorgehensweise hinter dem zu demonstrativen Zwecken entwickelten, auf dem Prozessor lauffähigen Tic-Tac-Toe Programms. Dieses vereint alle zentralen Funktionalitäten der Prozessor-Implementierung auf dem FPGA.

## 7.1 Struktur und Funktion des Demo-Programms

Wie eingangs geschildert, bietet das entwickelte Demo-Programm die Möglichkeit, über die Buttons des FPGAs gegen einander Tic-Tac-Toe zu spielen. Dabei wird mittels der Buttons BTN 0 bis BTN 3 (siehe Kapitel 4.4) das Steuerkreuz über die das Spielbrett navigiert. Dabei ist das Brett zyklisch angeordnet, sodass eine Linksbewegung des Steuerkreuzes an den linken Rand dieses an die rechte Brettseite manövriert. Mittels des Buttons BTN 4 (siehe Kapitel 4.4) setzt der Zugspieler seine Markierung an der zuvor ausgewählten Position, sofern dies regelkonform ist. Das Spiel enthält keinen internen Reset und muss daher über einen Hard-Reset durch das FPGA erfolgen.

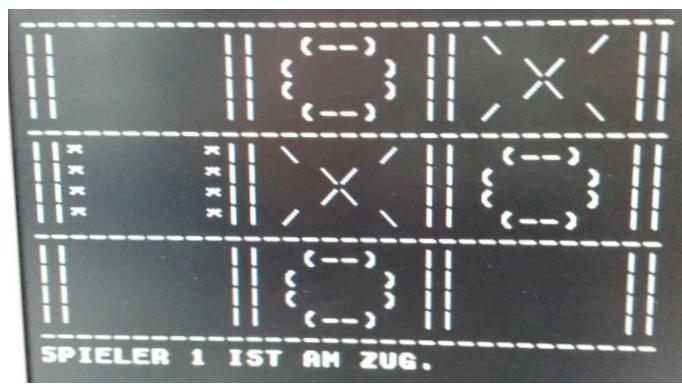


Abbildung 22: Bildschirmausgabe während der Ausführung des Demo-Programms

Um möglichst viel der implementierten Funktionalität abzudecken, wurde bei der Implementierung darauf geachtet, die meisten Komponenten zu beanspruchen. So wird bei Programmstart im DDR2-SDRAM-Block eine, als Array von

vorzeichenbehafteten 8-Bit Werten realisierte, Matrix  $M$  wie folgt initialisiert:

$$M = \begin{pmatrix} -4 & -4 & -4 \\ -4 & -4 & -4 \\ -4 & -4 & -4 \end{pmatrix} \quad m_{i,j} = \begin{cases} 0 & \text{wenn Markierung von Spieler 1} \\ 1 & \text{wenn Markierung von Spieler 2} \\ -4 & \text{sonst} \end{cases}$$

Wobei sich anhand dieser Definition durch Spalten-, Zeilen- und Diagonalsummen sofort der Zustand des Spiels ermitteln lässt.

Außerdem werden die Komponenten der grafischen Oberfläche mittels Schleifen und geeigneten Subroutinen im CHAR-RAM (siehe 4.3) in ihren initialen Zustand versetzt. Dabei werden zwangsläufig Sprungbefehle verwendet und auf diese Weise in die Demonstration miteinbezogen.

Da der Benutzer zwangsläufig mit dem Board interagieren muss, um ein Spiel zu bestreiten, nutzt das Demo-Programm auch die zuvor beschriebenen Memory-Mapped-I/O Funktionalitäten aus. Dazu wird, wie im folgenden Ausschnitt des Programmcodes gezeigt, in einer Endlosschleife nach neuen Eingaben auf den fünf verwendeten Buttons gesucht, indem durch Puffern des letzten Zustands und bitweise Verknüpfungen des gerade losgelassenen Buttons ermittelt wird und die Button-Nummer als Rückgabewert liefert. Nebeneffekte des Prellens werden hierbei allerdings nicht berücksichtigt.

```
.key_release
    lui x26, 0x30000 //ioram prefix
    lh x27, x26, 0
    srli x27, x27, 7
    andi x27, x27, 0x1F //isolating buttons
.key_release_wait
    //load new keys and check if any key was released
    lh x28, x26, 0
    srli x28, x28, 7
    andi x28, x28, 0x1F //new key input
    addi x30, x28, 0
    xor x28, x28, x27 //event vector (pressed and released keys)
    and x28, x28, x27 //filter released keys
    addi x27, x30, 0
    beq x28, x0, key_release_wait //keep waiting
    addi x18, x0, 0 //return value := which button was pressed
    addi x29, x0, 1
.key_release_shift_result
    beq x28, x29, end_key_release
    srli x28, x28, 1
    addi x18, x18, 1
    jal x0, key_release_shift_result
.end_key_release
jalr x0, x1, 0
```

Der gesamte Programmcode umfasst circa 400 Zeilen und liegt der VHDL-Implementierung bei, wurde aber aufgrund des unverhältnismäßigen Umfangs an dieser Stelle ausgelassen und durch den obigen Ausschnitt ersetzt.

## 7.2 Assemblierer und Simulator

Aufgrund der eben erwähnten Programmgröße, des reduzierten Instruction Sets und der projektspezifischen RAM-Struktur wurde der Beschluss gefasst, beim Assemblieren des Programmcodes auf einen im Rahmen des Projekts programmierten Assemblierer zurückzugreifen. Auch das Debugging erfolgte durch ein eigenes Tool, welches beispielsweise Aspekte des Memory-Mapped-I/O berücksichtigen und entsprechend simulieren kann. Die beiden Programme

Befehl	Beschreibung
n	Führt nächsten Befehl aus
s	Führt Subroutine aus, ohne in diese zu springen
c	Setzt Programmausführung fort
printchars	Zeigt die Ausgabe der ASCII-Unit
bp label/offset	Neuer Breakpoint
pin show	Zeigt alle I/O-Signale
pin set pinid 0/1 [-d duration]	Setzt ein I/O-Signal auf einen Wert (ggf. für ein Zeitintervall in s)
m offset cnt [chunksize]	Stellt die Speicherzellen eines bestimmten Offsets blockweise dar
sleep duration	Unterbricht die Programmausführung nach einem Zeitintervall in s

Tabelle 7: Befehlsübersicht des Simulators

wurden in der Skriptsprache Python entwickelt und genügen den Ansprüchen des Projekts insofern, als dass ein Assemblerprogramm entsprechend assembliert und simuliert werden kann, die Software allerdings keineswegs ausgiebig auf Stabilität oder dergleichen getestet wurde, zumal dies nicht als Schwerpunkt für das Projekt gesetzt war.

### Assemblierer

Der Assemblierer ist dabei in der Lage aus einem Eingabeprogrammcode, welches Sprünge, Labels und Kommentare enthalten kann, einerseits Bytecode sowie eine Symboltabelle zu erstellen. Des Weiteren wird der assemblierte Programmcode auch in korrekter VHDL-Syntax ausgegeben, sodass dieser zur Initialisierung einer Dual-Port-Blockram Komponente direkt eingebunden werden kann. Dabei wird der Assemblierer als Python-Script mit den Eingabewerten als Parametern aufgerufen:

```
$ python riscv_as.py -i {input} -o {output(vhdl)} -s {symbols} file -b {binary}
```

Die Symboltabelle gibt dabei Auskunft über die Speicheradresse eines im Programmcode definierten Labels. Aufgrund des modularen Aufbaus des Assembliers können Erweiterungen des Befehlssatzes sowie andere syntaktische Feinheiten dem Parsing- und Compilingprozess problemlos hinzugefügt werden.

### Simulator

Der Simulator ist als Backend-Erweiterung zum Assemblierer gedacht, da dieser mithilfe der Ausgabedateien den Programmablauf simulieren kann. Analog zu bekannten Debugging-Umgebungen unterstützt der Simulator das Setzen von Breakpoints sowie das Schrittweise Ausführen eines Befehls, während die verfügbaren Register direkt im Blick behalten werden können. Ferner ist es aber auch möglich, den Programmablauf nach einem bestimmten Zeitintervall automatisch zu unterbrechen oder aber bestimmte Eingabesignale zu stimulieren (auch hier kann mit einem beliebigen Zeitintervall gearbeitet werden). Das Python-Script des Simulators wird in Verbindung mit dem Assemblierer wie folgt aufgerufen:

```
$ python riscv_as.py -i {input} -o {output(vhdl)} -s {symbols} file -b {binary}
$ python riscv_simulation.py -s {symbols} -b {binary}
```

Dabei startet der Simulator an der Programmeinsprungadresse und unterstützt die folgenden Befehle:

Dabei wird durch die eben beschriebenen Befehle durch die Programmausführung navigiert, um so potenzielle Fehlerquellen zu entdecken und berichtigen. Zur Entwicklung des Demo-Programms hat der Simulator eine maßgebliche Rolle gespielt: Nur dank dessen Funktionalitäten war es überhaupt möglich mit vertretbarem Aufwand ein vergleichsweise komplexes, funktionstüchtiges Programm zu entwerfen.

# 1 Anhang

## 1.1 Übersicht über die zentralen Einheiten des Prozessors

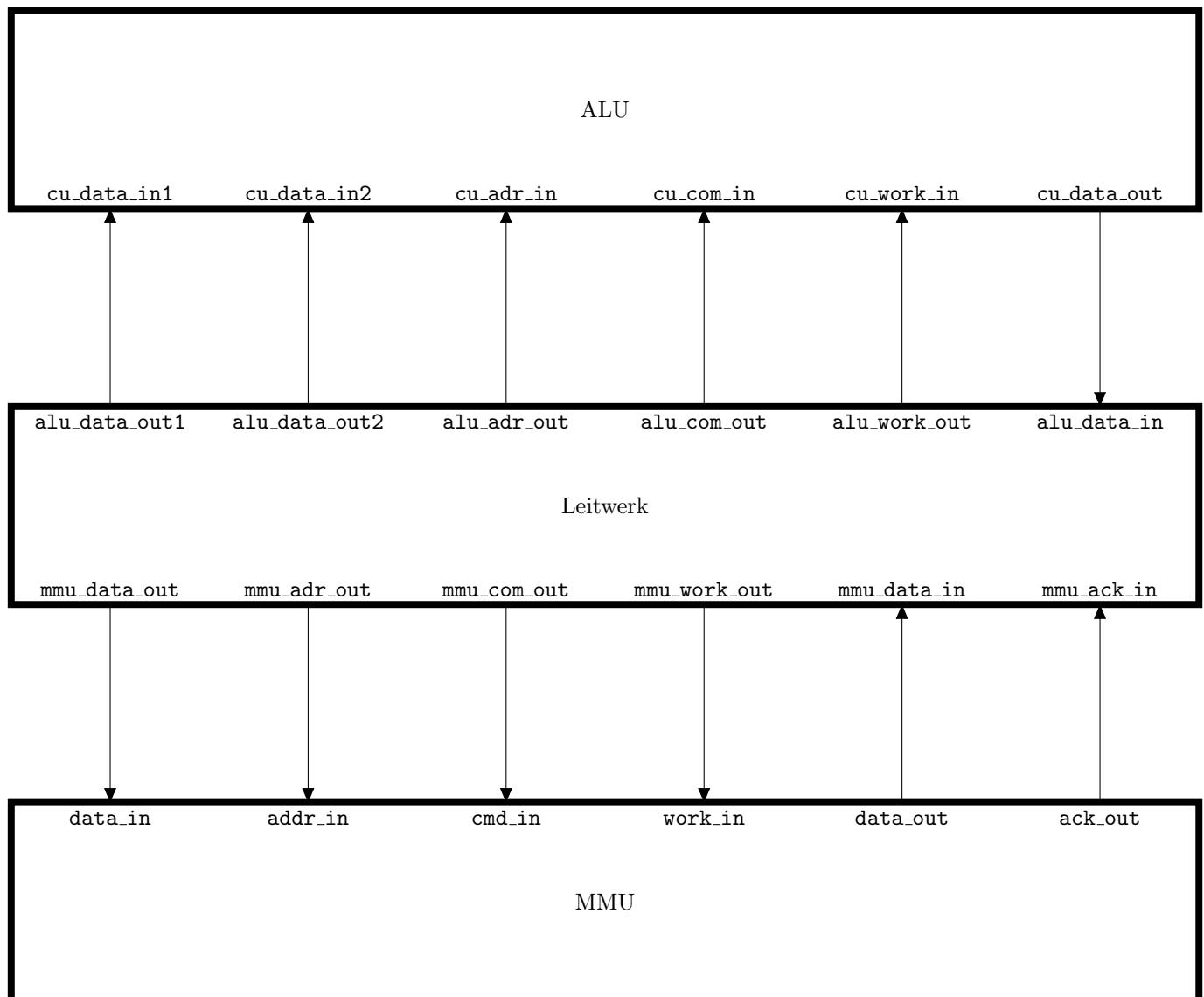


Abbildung 23: Übersicht über die zentralen Einheiten des Prozessors

## 1.2 Known Bugs

**ASCII-Ausgabe** Teile des Zeichens an Position 0 werden in den darunterliegenden Zeilen ebenfalls angezeigt.

**Leitwerk, MMU** Beim LOAD-Befehl sollte die MMU lediglich die benötigte Anzahl an Bits laden und nicht immer 32 Bits.

**Leitwerk, MMU** Der Prozessor stürzt, vermutlich wegen Synchronisationsfehlern zwischen Leitwerk und MMU, gelegentlich ab.

**ALU** Die ALU wartet wahrscheinlich länger auf das Ergebnis der Divisionseinheit, als nötig wäre.

**MMU** Durch eine Unterscheidung der Speicherarten bei Speicherzugriffen könnte man hier eine Geschwindigkeitssteigerung erzielen.

**Leitwerk** Das Leitwerk wartet am Ende einiger Befehle noch 3 Takte auf die ALU, ohne dass diese vorher gestartet wurde.

**MMU, allgemein** Die Buttons und Switches werden nicht entprellt.

**UART** Die UART-Einheit korrigiert keine Fehler in der Übertragung, weshalb nur eine sehr begrenzte Datenmenge fehlerfrei übertragen werden kann. Dadurch können bei einer Programmierung durch die serielle Schnittstelle nur sehr kleine Programme zum Einsatz kommen.

**UART** Die serielle Schnittstelle kann während der normalen Codeausführung des Prozessors nicht zuverlässig genutzt werden, da die Refresh-Zyklen des DDR2-RAMs die Befehlsausführung behindern und damit übertragene Bytes verpasst werden.

**allgemein** Reset ist nur im langsamen Modus zuverlässig möglich. Im schnellen Modus kann eine Reset zum Absturz des Prozessors führen.

### 1.3 Ausführungsduauer der Befehle

Befehl(e)	Ausführungsduauer (in Takten)		
	$s \leq 2$	$2 < s < 4$	$s \geq 4$
<b>LUI</b>	4	4	$s$
<b>AUPIC</b>	4	4	$s$
<b>JAL</b>	8	8	$4 + s$
<b>JALR</b>	8	8	$4 + s$
<b>BEQ</b>	12	12	$8 + s$
<b>BNE</b>	12	12	$8 + s$
<b>BLT</b>	12	12	$8 + s$
<b>BLTU</b>	12	12	$8 + s$
<b>BGE</b>	12	12	$8 + s$
<b>BGEU</b>	12	12	$8 + s$
<b>LB</b>	10	$8 + s$	$4 + 2 \cdot s$
<b>LBU</b>	10	$8 + s$	$4 + 2 \cdot s$
<b>LH</b>	10	$8 + s$	$4 + 2 \cdot s$
<b>LHU</b>	10	$8 + s$	$4 + 2 \cdot s$
<b>LW</b>	10	$8 + s$	$4 + 2 \cdot s$
<b>SB</b>	13	$11 + s$	$8 + 2 \cdot s$
<b>SH</b>	13	$11 + s$	$8 + 2 \cdot s$
<b>SW</b>	13	$11 + s$	$8 + 2 \cdot s$
<b>ADDI</b>	4	4	$s$
<b>SLTI</b>	4	4	$s$
<b>SLTIU</b>	4	4	$s$
<b>XORI</b>	4	4	$s$
<b>ORI</b>	4	4	$s$
<b>ANDI</b>	4	4	$s$
<b>SLLI</b>	4	4	$s$
<b>SRLI</b>	4	4	$s$
<b>SRAI</b>	4	4	$s$

Befehl(e)	Ausführungsduauer (in Takten)		
	$s \leq 2$	$2 < s < 4$	$s \geq 4$
<b>ADD</b>	4	4	$s$
<b>SUB</b>	4	4	$s$
<b>SLL</b>	4	4	$s$
<b>SLT</b>	4	4	$s$
<b>SLTU</b>	4	4	$s$
<b>XOR</b>	4	4	$s$
<b>SRL</b>	4	4	$s$
<b>SRA</b>	4	4	$s$
<b>OR</b>	4	4	$s$
<b>AND</b>	4	4	$s$
<b>FENCE</b>	-	-	-
<b>FENCE.I</b>	-	-	-
<b>SCALL</b>	-	-	-
<b>SBREAK</b>	-	-	-
<b>RDCYCLE</b>	4	4	$s$
<b>RDCYCLEH</b>	4	4	$s$
<b>RDTIME</b>	4	4	$s$
<b>RDTIMEH</b>	4	4	$s$
<b>RDINSTRET</b>	4	4	$s$
<b>RDINSTRETH</b>	4	4	$s$
<b>MUL</b>	4	4	$s$
<b>MULH</b>	4	4	$s$
<b>MULHSU</b>	4	4	$s$
<b>MULHU</b>	4	4	$s$
<b>DIV</b>	35	35	$\max(s, 35)$
<b>DIVU</b>	35	35	$\max(s, 35)$
<b>REM</b>	35	35	$\max(s, 35)$
<b>REMU</b>	35	35	$\max(s, 35)$

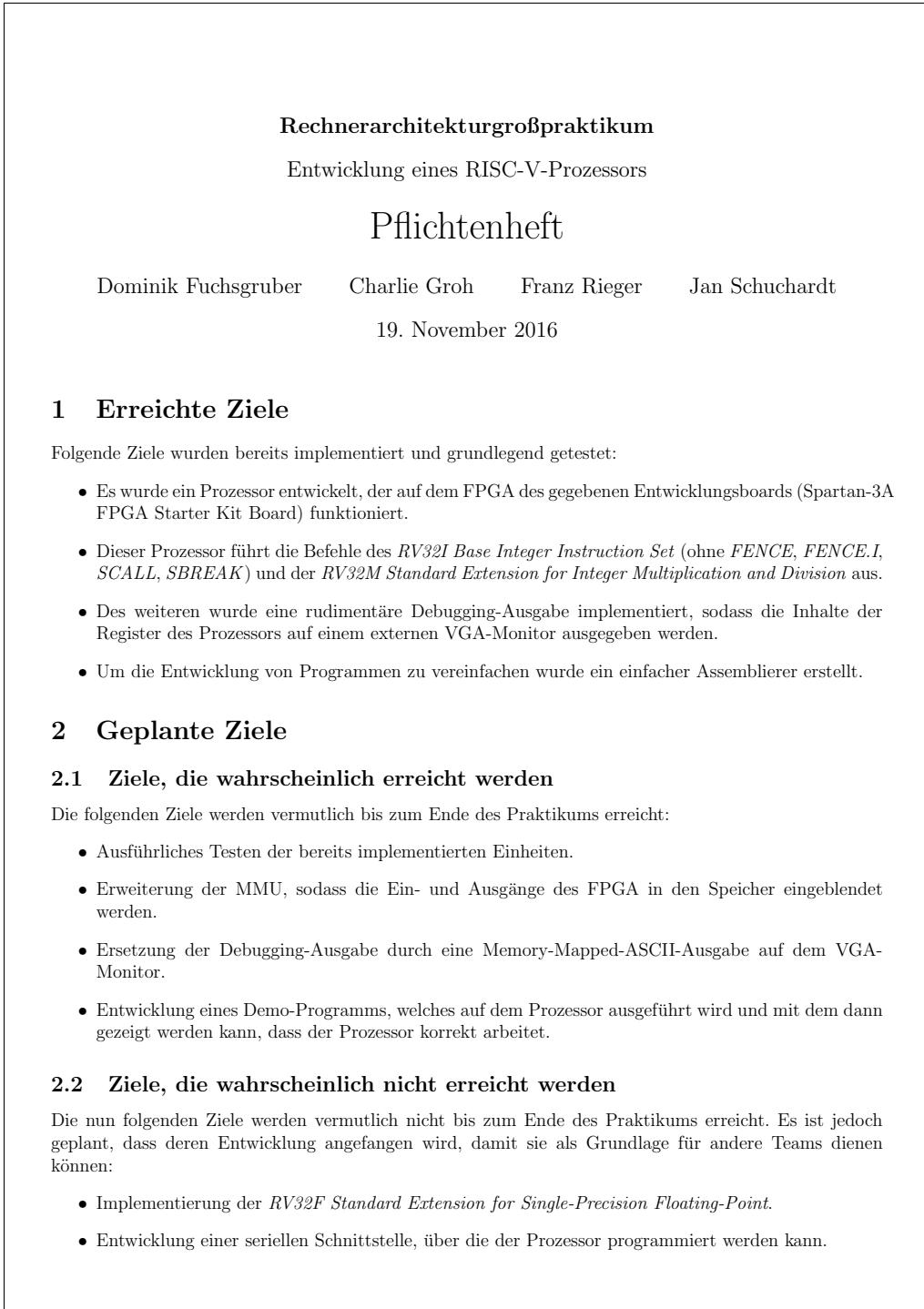
Tabelle 8: Übersicht über die Ausführungsduauer der Befehle. Da die Dauer eines Speicherzugriffs abhängig von dem angesprochenen Speichertyp und selbst dann nicht immer exakt vorhersagbar ist, werden hier für unterschiedliche Speicherzugriffsdauern  $s$  die jeweilige Ausführungsduauer der Befehle gelistet.

## 1.4 1. Pflichtenheft vom 02.06.2016



Abbildung 24: Pflichtenheft vom 02.06.2016

## 1.5 2. Pflichtenheft vom 19.11.2016



### 1 Erreichte Ziele

Folgende Ziele wurden bereits implementiert und grundlegend getestet:

- Es wurde ein Prozessor entwickelt, der auf dem FPGA des gegebenen Entwicklungsboards (Spartan-3A FPGA Starter Kit Board) funktioniert.
- Dieser Prozessor führt die Befehle des *RV32I Base Integer Instruction Set* (ohne *FENCE*, *FENCE.I*, *SCALL*, *SBREAK*) und der *RV32M Standard Extension for Integer Multiplication and Division* aus.
- Des weiteren wurde eine rudimentäre Debugging-Ausgabe implementiert, sodass die Inhalte der Register des Prozessors auf einem externen VGA-Monitor ausgegeben werden.
- Um die Entwicklung von Programmen zu vereinfachen wurde ein einfacher Assembler erstellt.

### 2 Geplante Ziele

#### 2.1 Ziele, die wahrscheinlich erreicht werden

Die folgenden Ziele werden vermutlich bis zum Ende des Praktikums erreicht:

- Ausführliches Testen der bereits implementierten Einheiten.
- Erweiterung der MMU, sodass die Ein- und Ausgänge des FPGA in den Speicher eingeblendet werden.
- Ersetzung der Debugging-Ausgabe durch eine Memory-Mapped-ASCII-Ausgabe auf dem VGA-Monitor.
- Entwicklung eines Demo-Programms, welches auf dem Prozessor ausgeführt wird und mit dem dann gezeigt werden kann, dass der Prozessor korrekt arbeitet.

#### 2.2 Ziele, die wahrscheinlich nicht erreicht werden

Die nun folgenden Ziele werden vermutlich nicht bis zum Ende des Praktikums erreicht. Es ist jedoch geplant, dass deren Entwicklung angefangen wird, damit sie als Grundlage für andere Teams dienen können:

- Implementierung der *RV32F Standard Extension for Single-Precision Floating-Point*.
- Entwicklung einer seriellen Schnittstelle, über die der Prozessor programmiert werden kann.

Abbildung 25: Pflichtenheft vom 19.11.2016