

Rechnerarchitekturgroßpraktikum

Entwicklung eines RISC-V-Prozessors

Dokumentation

Dominik Fuchsgruber

Charlie Groh

Franz Rieger

Jan Schuchardt

6. Februar 2017

Inhaltsverzeichnis

1	Projektmanagement	3
1.1	Aufgabe	3
1.2	Basisziele (Pflichtangebot)	3
1.3	Erweiterungsziele (Erweiterungsangebot)	3
1.4	Verwendete Tools	3
2	Das Leitwerk	5
2.1	Überblick	5
2.1.1	Legende	5
2.2	Integer Rechenbefehle	5
2.2.1	Division und Modulo	7
2.3	LUI und AUPIC	9
2.4	Bedingte Sprünge	11
2.5	Unbedingte Sprünge	14
2.6	LOAD	16
2.7	STORE	18
2.8	Timer und Counter	20
2.9	Minimale Taktanzahl	22
3	Die Arithmetische Logische Einheit	23
3.1	Überblick	23
3.2	Das Interface	23
3.3	Interne Befehle	23
3.4	Befehlsausführung	24
3.4.1	State 1 - Selektion der Operanden	24
3.4.2	State 2 - Operatonsausführung	24
3.4.3	State 3 - Write-Back	25
3.4.4	State 4 - Division Unsigned	25
3.4.5	State 5 - Division Signed	25
3.5	Die Register	25
3.6	Reset	26

4	Die MMU	27
4.1	Überblick	27
4.2	Interface	27
4.2.1	Kommunikation mit dem Leitwerk	27
4.2.2	Durch die MMU geleitete Signale an andere Komponenten	28
4.3	Aufbau des Speichers	28
4.4	Memory-Mapped I/O	29
4.5	Implementierung als State Machine	30
4.5.1	MMU-IDLE	30
4.5.2	MMU-WAITING	30
4.5.3	MMU-DATA-VALID	30
4.5.4	MMU-READ-NEXT	30
4.5.5	MMU-READ-DONE	31
4.5.6	MMU-WRITE-NEXT	31
4.5.7	MMU-WRITE-DONE	31
4.5.8	MMU-RESET	31
4.6	Zugrifssdauer	31
5	Die ASCII-Unit	32
5.1	berblick	32
5.2	Interface	32
5.3	Funktionsweise	33

Kapitel 1

Projektmanagement

1.1 Aufgabe

Die Aufgabe bestand darin, einen funktionsfhigen Prozessor auf Basis der RISC-V Instruction Set Architecture zu entwickeln, der auf dem FPGA des gegebenen Entwicklungsboards (Spartan-3A FPGA Starter Kit) luft.
(Bild vom Board (evtl. mitsamt Monitor))

1.2 Basisziele (Pflichtangebot)

Als Mindestanforderung sollte ein Prozessor mit Kompatibilitt zur RV32I Base Integer Instruction Set implementiert werden. Ausgenommen davon sind die Befehle FENCE, FENCE.I, SCALL und SBREAK, da keine Hardwareuntersttzung fr den Multitaskingbetrieb bentigt wird. Um die Funktionsfhigkeit des Prozessors auch nach auen sichtbar zu machen und keine Black-Box zu erstellen soll die Mglichkeit der Interaktion ber die Schnittstellen und Pins des Boards bestehen. Insbesondere soll zum Debugging eine grafische Registerausgabe ber den VGA-Port an einem Monitor mglich sein.

Siehe Anlage (Abgegebenes)

1.3 Erweiterungsziele (Erweitwerungsangebot)

Aufgrund der Modularitt von RISC-V bietet es sich an mindestens eine Erweiterung zu implementieren, nmlich die RV32M Standard Extension for Integer Multiplication and Division, die Multiplikations- und Divisionsbefehle beinhaltet. Zudem soll die rudimentre Ausgabe um einen Textmodus erweitert werden, sodass mittels Memory-Mapping ASCII-Zeichen auf dem Monitor ausgegeben werden knnen. Zum Demonstrieren der Funktionalitt soll auerdem ein auf dem Prozessor lauffhiges Spiel entwickelt werden. Um auch Datenbertragung mit der Auenwelt zu ermöglichen soll die serielle Schnittstelle genutzt werden.

1.4 Verwendete Tools

Das Projekt wurde in VHDL implementiert, da smtliche Programmierer ausschlielich in dieser Hardwareprogrammierungssprache gute Kenntnisse hatten. Zur Entwicklung wurden hauptschlich Xilinx ISE Project Navigator verwendet.

Dieser diente zugleich als Editor für den VHDL-Code und auch als Werkzeug um daraus die Programming-Files, mit denen das FPGA beschrieben wird zu generieren. Der integrierte Core Generator wurde benutzt um einzelne Module zu erstellen. Mittels Xilinx impact in Kombination mit dem Cable-Server wurde das Board über USB mit den Programming-Files beschrieben. Zur Versionsverwaltung wurde auf ein Github Repository gesetzt. Um nicht jedes zu testende Programm per Hand assemblieren zu müssen wurde ein Assemblierer auf Python-Basis erstellt. Da die Generierung eines Programming-Files mit anschließendem Beschreiben des FPGAs mehrere Minuten in Anspruch nimmt wurden zum Testen Simulatoren verwendet. Um den VHDL-Code zu verifizieren wurde so auf GHDL in Kombination mit GTKWave gesetzt. Auch zur Assemblerprogrammierung wurde ein Simulator mitsamt Textausgabe entwickelt um schneller debuggen zu können.

Kapitel 2

Das Leitwerk

Das Leitwerk ist die zentrale Steuereinheit des Prozessors. Es interpretiert die Befehle und überwacht ihre Ausführung durch ALU und MMU. Dazu verwaltet das Leitwerk den Program-Counter (PC) und das Instruction-Register (IR).

2.1 Überblick

Im Prozessor wurden die durch das *RV32I Base Integer Instruction Set* und die *RV32M Standard Extension for Integer Multiplication and Division* definierten Befehle implementiert. Eine Ausnahme bilden dabei alle Befehle, die Multitasking ermöglichen sollen, also **SCALL**, **SBREAK**, **FENCE** und **FENCE.I**. Besondere Aufmerksamkeit wurde dabei mehr auf Robustheit und weniger auf maximale Geschwindigkeit gelegt.

Das Leitwerk besitzt für jeden Befehl eine eigene Zustandsmaschine. Welche ausgeführt wird hängt allein vom Inhalt des Instruction-Register ab. Daher muss jeder Befehl als letztes das IR mit dem folgenden Befehl belegen. Wann dieser Befehl geladen wird kann nun 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.

2.1.1 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 anzuzeigen, dass eine Aktion von der jeweiligen Einheit ausgeführt wird und das Leitwerk lediglich eine Anweisung gibt.

In der ISA wird regelmäßig verlangt, dass Operanden sign-extended werden. Dies wird 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.

img/integer_rechenbefehle.pdf

Abbildung 2.1: Zustandsübergangsdiagramm der Befehle **ADD[I]**, **SUB**, **SLT[I][U]**, **AND[I]**, **OR[I]**, **XOR[I]**, **SLL[I]**, **SRL[I]**, **SRA[I]**, **MUL[W]** und **MULH[[S]U]**. *op2* ist entweder das Register, das mit *rs2* angegeben ist, oder eine Immediate (*imm*). “o” repräsentiert die jeweilige Operation (+, −, ...).

2.2.1 Division und Modulo

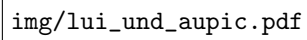
Da bei der Division unmöglich zu garantieren ist, dass diese immer nach drei Takten beendet ist, muss das Leitwerk hier auf eine Bestätigung der ALU warten. Diese sieht vor, dass das Rechenwerk die Leitungen `alu_data_in` auf 0 setzt.

img/division.pdf

Abbildung 2.2: Zustandsübergangsdiagramm der Befehle **DIV**[U][W] und **REM**[U][W]. “o” repräsentiert die jeweilige Operation (/ , mod , ...).

2.3 LUI und AUIPC

Da durch die Integer Rechenbefehle keine 32-Bit Immediates direkt geladen werden können, definiert die RISC-V-ISA die Befehle **LUI** und **AUIPC**, die diesen Mangel beheben. Auch diese Befehle wurden auf Geschwindigkeit optimiert.



img/lui_und_aupic.pdf

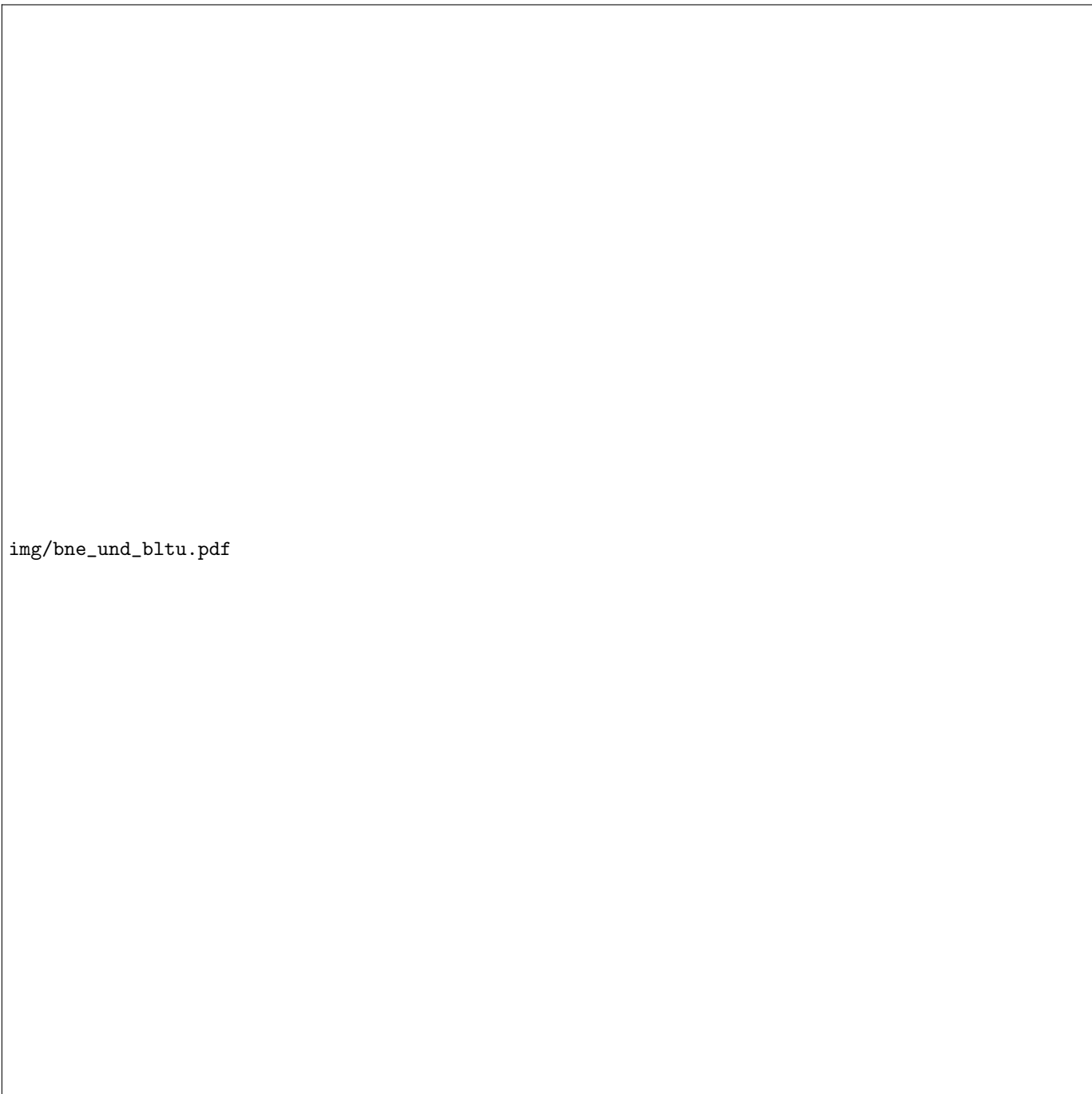
Abbildung 2.3: Zustandsübergangsdiagramm der Befehle **LUI** und **AUPIC**. *op2* ist entweder 0 (bei **LUI**) oder der aktuelle Program-Counter (bei **AUPIC**).

2.4 Bedingte Sprünge

Da bei dem DDR2-Speicher des benutzten Boards nicht garantiert werden konnte, dass ein Speicherzugriff ohne Zeit- und Datenverlust abgebrochen werden kann, wurde auf eine einfache Branch-Prediction gänzlich verzichtet. Dadurch sind die bedingten Sprünge unter den teuersten Befehlen des Prozessors.

img/beq_und_bgeu.pdf

Abbildung 2.4: Zustandsübergangsdiagramm der Befehle **BEQ** und **BGE[U]**. “o” ist bei **BEQ** “–”, bei **BGE** die SLT-Operation und bei **BGEU** die SLTU-Operation.



img/bne_und_bltu.pdf

Abbildung 2.5: Zustandsübergangsdiagramm der Befehle **BNE** und **BLT[U]**. “o” 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, wodurch das schreiben der Return-Adresse und das holen des nächsten Befehls parallelisiert werden kann, was zu einer merklichen Geschwindigkeitssteigerung führt.

img/unbedingte_spruenge.pdf

Abbildung 2.6: Zustandsübergangsdiagramm der Befehle **JAL** und **JALR**.

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 Ausführung des Befehls ist durchschnittlich und wird in der Praxis hauptsächlich von der Speichergeschwindigkeit bestimmt.

img/load.pdf

Abbildung 2.7: 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 einer ALU und nur einer MMU nicht zu parallelisieren, was dazu führt, dass er der langsamste Befehl des Prozessors ist. Da dieser Befehl jedoch generell auf RISC-Architekturen sehr langsam ausgeführt wird, versuchen Compiler und Programmierer ohnehin schreibende Speicherzugriffe zu vermeiden.

img/store.pdf

Abbildung 2.8: 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 gibt es einen Counter, der die Anzahl der bisher ausgeführten Befehle speichert. Außerdem gibt es einen Timer, der die Anzahl der vergangenen Takte speichert. Da das FPGA keine Echtzeituhr bereitstellt, wurde auch hierfür der Taktzähler verwendet.

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

img/timer_und_counter.pdf

Abbildung 2.9: Zustandsübergangsdiagramm der Befehle **RDINSTRET[H]**, **RDCYCLE[H]** und **RDTIME[H]**. *counter* sind die oberen bzw. unteren 32 Bit des jeweiligen 64 Bit Zählers.

2.9 Minimale Taktanzahl

Die hier aufgeführten Taktzahlen, die der Prozessor zur Ausführung eines Befehls benötigt, sind Mindestangaben, dass der Speicherzugriff auf den nächsten Befehl hinreichend schnell erfolgt. (siehe dazu auch MMU)

Kapitel 3

Die Arithmetische Logische Einheit

Die Arithmetisch Logische Einheit ist verantwortlich für die Durchführung aller für die Befehlsausführung 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 Immediates angefragt werden können, wozu eine Menge von Prozessor-internen OPCODEs definiert wurden (SIEHE).

Da die ALU auch in nicht-arithmetischen Maschinenbefehlen, wie Sprüngen, oft gebraucht wird, sollte der Synchronisations- und Kommunikationsaufwand zwischen ALU und Leitwerk möglichst reduziert werden. Aus diesem Grund werden grundstzlich alle Befehle (exklusive Division) mittels einer State-Machine innerhalb von drei Takten ausgeführt (SIEHE). Aus diesem Grund ist kein Synchronisations-Protokoll zwischen ALU und Leitwerk von Nöten.

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

3.2 Das Interface

HIER KOMMT EIN BILD VOM INTERFACE ODER SÖ

Neben zwei 32-Bit-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-Eingnge entweder zur Übermittlung von Immediates, oder zur Adressierung der Operanden-Register verwendet, um die Anzahl der Eingnge zu reduzieren.

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 Befehle 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* genommen, oder aus dem durch *Cu_data_in1* adressierten Register. Analog entscheidet *Reg2* darüber, 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
- 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

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

3.4.1 State 1 - Selektion der Operanden

Die ALU verbleibt in State 1, bis vom Leitwerk das entsprechende Work-Signal gesendet wird. Anschließend werden auf Grundlage des IN SEKTION () BESCHRIEBENEN Befehls zwei Operanden-Signale *s_op1* und *s_op2* mit einem Immediate vom Daten-Eingang belegt, oder es wird ein Register-Lesezugriff angestoßen, dessen Ergebnis im nächsten Takt zur Verfügung steht.

3.4.2 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 anlegt 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 innerhalb dieses Taktes vollständig abgeschlossen. 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 erlaubt 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 gegebenen 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.

3.4.3 State 3 - Write-Back

In State 3 werden die akkumulierten Ergebnisse in das von der ALU adressierte Register geschrieben. Im Grotteil der Operationen erfolgt dies unmittelbar. Bei Multiplikationsoperationen 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, wenn nötig 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 genullt 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`.

3.4.4 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. Anschließend wird mit einem Zähler gewartet, bis das Ergebnis der Operation anliegt. Anschließend werden entweder der Rest oder das Ergebnis in den Akkumulator eingelesen und in State 3 übergegangen.

3.4.5 State 5 - Division Signed

Die Umsetzung ist 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 mit 32 Bit Breite bereit, wobei Register 0 konstant den Wert 0 liefert und nicht überschreibbar ist. Dies ist nützlich zum unveränderten Laden eines Registers.

Implementiert wurden die Register als Dual-Port-Block-Ram. Dies ermöglicht den zeitgleichen Zugriff auf zwei verschiedene Speicherinhalte, was bei Register-Register-Operationen vorkommt. Zusätzlich wird dadurch die Anzahl an verwendeten Slices reduziert, da dedizierte BlockRam-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 garantiert die 3-stufige State-Machine der ALU.

3.6 Reset

Die ALU verfügt über einen Reset-Eingang, welcher direkt vom CPU-Toplevel zu ihr durchgeleitet wird. Bei einem Reset wird sie in den State 0 überführt, um nach Ende des Resets Befehle des Leitwerks entgegennehmen zu können. Zusätzlich wird der Divisions-Flankenzähler genullt, um keine Fehler bei nachfolgenden Operationen zu verursachen. Das Reset-Signal wird auch an den Reset (SCLR)-Eingang der Divisionseinheit durchgeleitet. Ein Reset der Register erfolgt nicht, abgesehen vom Register 0 ist für den Entwickler eines Nutzerprogramms keine Aussage über die enthaltenen Werte der Register möglich.

Kapitel 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 State-Machine 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-SDRAM¹, 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.

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.

TODO: Einfügen des Schaubilds zum Interface??

4.2.1 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

¹Genauer handelt es sich um einen Micron Technology DDR2-SDRAM (MT47H32M1)

erst auf ein Setzen des work_in Signals mit der Bearbeitung der Anfrage. Der 3-Bit Vektor cmd_in ist wie folgt aufgebaut:

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

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 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 liegt. Theoretisch bietet die MMU allerdings auch die Möglichkeit, 8-Bit oder 16-Bit Lesezugriffe durchzuführen.

4.2.2 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, potenziell 2^{32} potenzielle 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

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 groen 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	2 ¹⁶²	DDR2-SDRAM
0x2	CHARRAM	2 ¹¹	Character-Anzeige
0x3	IORAM	2 ³	Memory-Mapped I/O
0x4	SERIALRAM	2 ¹¹	Serielle Schnittstelle

Dabei sind alle Blöcke, ausgenommen der DDR2-SDRAM-Block, durch auf dem FPGA verfügbaren Dual-Port-Blockram 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 Opencores³ verwendet und entsprechend den Anforderungen abgeändert.

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 IORAM, 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 IORAMs gemappt. Aus den acht verfügbaren Speicherzellen sind folgende sechs nutzbar:

Zelle	Zugriff (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	-	-	-	-	-	-	-	-

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 IORAM 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 Programmspeichers 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 Speicherqualität.

Die nachfolgende Abbildung 4.1 zeigt, wo sich welches Ein-/Ausgabesignal auf der Hardware wiederfindet. Dabei entsprechen die Bezeichner denen aus der zuvor abgebildeten Tabelle.

²Von 512 MBit verfügbaren Speicherplatz macht der genutzte Controller nur 2¹⁶ Bytes zugänglich

³http://opencores.org/project,ddr2_sdram

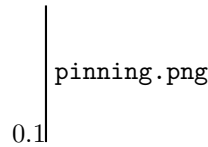


Abbildung 4.1: Verteilung der I/O-Signale

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.

TODO: Einfügen der State Machine.

4.5.1 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. Anderenfalls wird die Eingabeadresse ausgewertet, die durchzuführende Operation an den entsprechenden RAM-Controller weitergeleitet und die Einheit in den Zustand MMU-WAITING überführt.

4.5.2 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.

4.5.3 MMU-DATA-VALID

In diesem Zustand werden, sofern es sich bei der zuletzt durchgeführten 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.

4.5.4 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. Anderenfalls wechselt die MMU in den Zustand MMU-READ-DONE.

4.5.5 MMU-READ-DONE

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

4.5.6 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. Anderenfalls wechselt die MMU in den Zustand MMU-WRITE-DONE.

4.5.7 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.

4.5.8 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 Zugrifssdauer

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

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

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 Zugrifsdauer 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.

Kapitel 5

Die ASCII-Unit

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

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 \cdot 64 Zeichen ergeben und die letzte Stelle rechts unten die Adresse 2047 erhält. Diese Adressen beziehen sich genau 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 zuzuordnen zu können wurde auf eine CHARMAP gesetzt, die zu jedem der 256 ASCII Zeichen einen 64-Bit-Vektor eingespeichert hat, der eben das 8*8 Bitfeld eines jeden Zeichens repräsentiert. 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 8,32 bzw. 64) erheblich einfacher zu realisieren sind. 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 die 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 die Software die Verwaltung des CHARRAMs übernehmen, um diese Zeichen korrekt darzustellen. 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.

Schaubild ZeichenGitter+Adressen

5.2 Interface

Neben dem bereits erwähnten Takteingang gibt es jeweils einen Eingang für die x- und y- Koordinate des Fadenstrahls, sprich des aktuellen Pixels, aus der VGA-Unit, welcher sich mit jedem Takt ändert 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 aktuell zu berechnenden Pixels angibt sowie einen Eingang, der im darauffolgenden Takt die ASCII-Nummer des Zeichens erhält. Zur CHARMAP wird der Takt durchgeleitet und ebenso die aus dem CHARRAM kommende Adresse des aktuellen Zeichens. Aus der CHARMAP kommt im darauffolgenden Takt der oben erwähnte 64-Bit-Vektor, der das jeweilige Zeichen repräsentiert.

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 Voraus berechnet wird. Im nächsten Takt wird daraus die Adresse des aktuellen Zeichen-Platzes berechnet, welche durch die MMU in den CHARRAM geleitet wird. $\text{round_down}(y/8) * 64 + \text{round_down}(x_{mod}/8)$ Von dort wird im nächsten Takt die ASCII-Nummer des aktuellen Zeichens geliefert, welche direkt in die CHARMAP weitergeleitet wird. Dies ist auch problemlos bei gleichzeitiger Port-Blockramrealisierung möglich. Lediglich falls zwischen den insgesamt 64 Zugriffen pro Frame der ASCII-Unit auf eine Adresse dieses Zeichens dann möglicherweise in diesem Frame falsch dargestellt wird, was sich aber im Betrieb allerdings kaum bemerkbar macht. Der Bitvektor aus dem mittels der aktuellen x- und y-Koordinate berechnet wird, ob das Pixel zu setzen ist oder nicht. Dazu insgesamt benötigt die Berechnung also 4 Takte.