

Rechnerarchitekturgroßpraktikum

Entwicklung eines RISC-V-Prozessors

Dokumentation

Dominik Fuchsgruber

Charlie Groh

Franz Rieger

Jan Schuchardt

5. Februar 2017

Inhaltsverzeichnis

1	Das Leitwerk	2
1.1	Überblick	2
1.1.1	Legende	2
1.2	Integer Rechenbefehle	2
1.2.1	Division und Modulo	4
1.3	LUI und AUPIC	6
1.4	Bedingte Sprünge	8
1.5	Unbedingte Sprünge	11
1.6	LOAD	13
1.7	STORE	15
1.8	Timer und Counter	17
1.9	Minimale Taktanzahl	19
2	Die Arithmetische Logische Einheit	20
2.1	Überblick	20
2.2	Das Interface	20
2.3	Interne Befehle	20
2.4	Befehlsausführung	21
2.4.1	State 1 - Selektion der Operanden	21
2.4.2	State 2 - Operatonsausführung	21
2.4.3	State 3 - Write-Back	22
2.4.4	State 4 - Division Unsigned	22
2.4.5	State 5 - Division Signed	22
2.5	Die Register	22
2.6	Reset	23
3	Die MMU	24
3.1	Überblick	24
3.2	Aufbau des Speichers	24
3.3	Memory-Mapped I/O	25

Kapitel 1

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).

1.1 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.

1.1.1 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.

1.2 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 1.1: 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 (+, −, ...).

1.2.1 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 1.2: Zustandsübergangsdiagramm der Befehle **DIV**[U][W] und **REM**[U][W]. “o” repräsentiert die jeweilige Operation (/ , mod , ...).

1.3 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

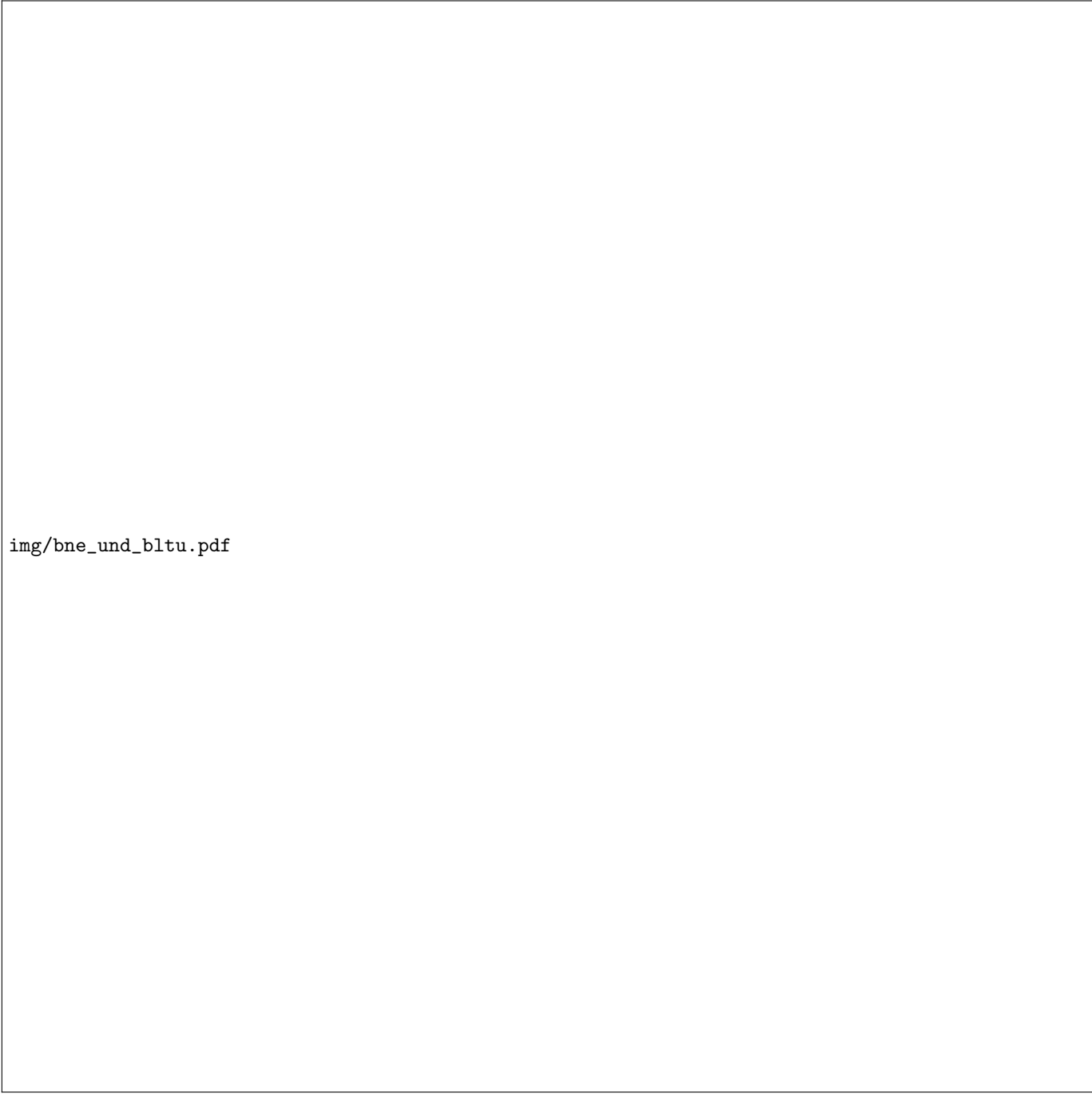
Abbildung 1.3: Zustandsübergangsdiagramm der Befehle **LUI** und **AUPIC**. *op2* ist entweder 0 (bei **LUI**) oder der aktuelle Program-Counter (bei **AUPIC**).

1.4 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 1.4: 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 1.5: Zustandsübergangsdiagramm der Befehle **BNE** und **BLT[U]**. “o” ist bei **BNE** “–”, bei **BLT** die SLT-Operation und bei **BLTU** die SLTU-Operation.

1.5 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 1.6: Zustandsübergangsdiagramm der Befehle **JAL** und **JALR**.

1.6 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 1.7: Zustandsübergangsdiagramm der Befehle **LB[U]**, **LH[U]** und **LW**. *width* ist je nach Befehl 1, 2 oder 4.

1.7 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 1.8: Zustandsübergangsdiagramm des Befehle **SB**, **SH** und **SW**. *width* ist je nach Befehl 1, 2 oder 4.

1.8 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 1.9: Zustandsübergangsdiagramm der Befehle **RDINSTRET[H]**, **RDCYCLE[H]** und **RDTIME[H]**. *counter* sind die oberen bzw. unteren 32 Bit des jeweiligen 64 Bit Zählers.

1.9 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 2

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.

2.1 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.

2.2 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.

2.3 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

2.4 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.

2.4.1 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.

2.4.2 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.

2.4.3 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`.

2.4.4 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.

2.4.5 State 5 - Division Signed

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

2.5 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.

2.6 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 3

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.

3.1 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. Daraus resultieren zusätzlich benötigte Synchronisations- und Kommunikationsmechanismen mit dem übergeordneten Leitwerk.

3.2 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^{16}^1	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 Opencores² verwendet und entsprechend den Anforderungen abgeändert.

3.3 Memory-Mapped I/O

¹Von 512 MBit verfügbaren Speicherplatz macht der genutzte Controller nur 2^{16} Bytes zugänglich

²http://opencores.org/project,ddr2_sdram