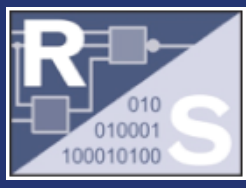
Implementierung eines Spill-and-Fill-fähigen Framestacks für einen AMIDAR basierten Java-Prozessor

Seminararbeit vorgelegt von (Robert Wiesner)
Betreuer: Dipl.-Ing. (Changgong Li)
Beginn: (Beginndatum) | Abgabe: 27.02.2017
Institut für Datentechnik | Fachgebiet Rechnersysteme | Prof. Dr.-Ing. Christian Hochberger

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Inhaltsverzeichnis

1 Aufgabenstellung

Entwicklung eines Spill and Fill Mechanismus um den Framestack des AMIDAR Prozessors auf einen externen DDR3 Speicher auszulagern und so die maximal mögliche Stackgröße deutlich zu erhöhen. Dazu gehört das Einrichten verschiebbarer "Windows", die jeweils einen Ausschnitt des Stacks beinhalten. Bei einem Methodenaufruf, dessen Stackframe den aktuellen Ausschnitt überschreiten würde, soll ein Teil der im Window enthaltenen Daten auf den externen Speicher ausgelagert werden, um Platz für den neuen Stackframe zu schaffen. Bei einem Methodenrücksprung sollen die Daten zurück kopiert werden, um den alten Stackframe wiederherzustellen. Es sollen 4 Windows angelegt werden, um die Stacks von bis zu 4 Threads vorzuhalten. Es soll die Möglichkeit einer Threadverdrängung realisiert werden, damit mehr als 4 Threads ausgeführt werden können.

2 Grundlagen

2.1 AMIDAR

Bei der Klasse der AMIDAR Prozessoren handelt es sich um ein konfigurierbares System, bestehend aus Function Units (FU). Komplexe Instruktionen werden in mehrere Arbeitsschritte unterteilt, die als Token an die einzelnen FUs verteilt werden. Dafür wird das Token Distribution Network genutzt. Für die Kommunikation zwischen den FUs, gibt es auch noch einen Datenbus. Ein Vorteil der AMIDAR Architektur ist, dass leicht FUs eingefügt werden können, um den Prozessor an die Aufgaben anzupassen. Ein gutes Beispiel dafür sind die Hardwarebeschleuniger. [4][3]

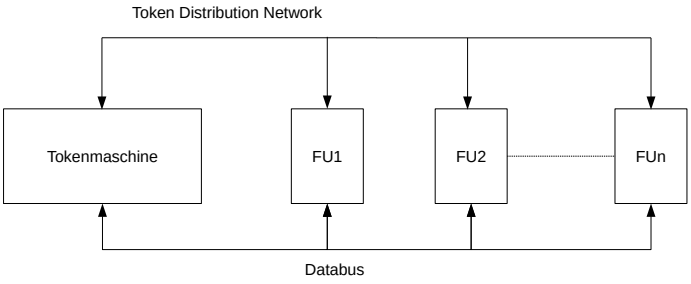


Abbildung 2.1: AMIDAR

2.1.1 Lokale Variablen und Operanden in Java

Java ist ein Objektorientierte Programmiersprache. Programme die in Java geschrieben worden sind, werden üblicherweise in einer virtuellen Maschine ausgeführt. Dabei werden Frames genutzt, die für die Ausführung einer Methode wichtige Daten beinhalten. Ein Frame wird bei einen Methodenaufruf erzeugt und bei einen Methodenrücksprung zerstört. Ein Frame enhält jeweils den Speicher für die lokalen Variablen, wie auch den Operandstack. Die Java Virtual Machine arbeitet Stackbasiert, was bedeutet, das Operanden nicht, wie in einer Registermaschine, in Registern gespeichert werden, sondern auf den Operandstack gelegt werden. Rechenoperationen und Funktionsaufrufe nehmen jeweils die obersten Werte des Operandstacks als Parameter.

2.1.2 Framestack

Die AMIDAR Implementierung, die am Fachgebiet Rechnersysteme entwickelt wird, arbeitet mit Java Bytecode. Dabei wird der Stack für Lokale Variablen und der Operandenstack in der Framestack FU zusammengefasst. Durch das Zusammenfassen beider Stacks, werden Kopiervorgänge über den AMIDAR

Bus eingespart. Zum einen müssten lokale Variablen bei jeder Verwendung auf den Operandstack kopiert werden, zum anderen müssten Parameter bei jeden Funktionsaufruf in den lokalen Variablenspeicher kopiert werden. Der AMIDAR Framestack arbeitet bei der Stackframe Verarbeitung mit drei grundsätzlichen Zeigern. Der Stackpointer gibt die nächste freie Speicheradresse über den aktuellen Stackframe an. Der Locals Pointer gibt die unterste Lokale Variable und damit auch das untere Ende des Stackframes an. Der dritte wichtige Zeiger ist der Callercontext Pointer, der auf die unterste Adresse des Callercontext deutet. [1]

Stackframe

Ein Stackframe beginnt mit den Parametern und den Lokalen Variablen. Darüber kommt der Callercontext mit den alten Localspointer, Stackpointer und Callercontextpointer. Das obere Ende des Stackframes besteht aus dem Callercontext. [1]

Funktionsaufrufe

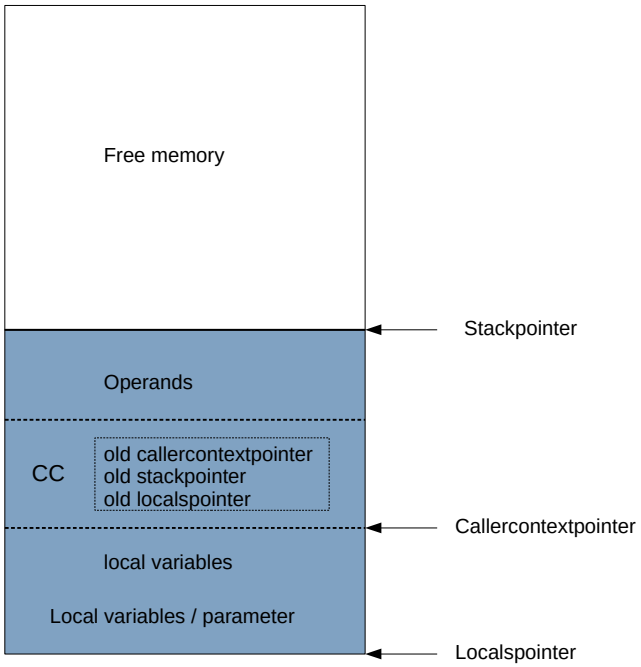


Abbildung 2.2: Stackframe vor einen Funktionsaufruf

Eine für dieses Projekt wichtige Funktion des Framestacks sind Funktionsaufrufe. Bei einen Funktionsaufruf werden im Framestack die 3 wichtigen Pointer neu gesetzt, wobei die alten Werte im Callercontext gesichert werden. Der neue Localspointer wird berechnet, in dem vom aktuellen Stackpointer die Anzahl der Parameter, der aufgerufenen Methode, abgezogen wird. Der neue Callercontextpointer wird berechnet, indem auf dem neuen Localspointer, die mit den Token übergebene, Anzahl lokaler Variablen addiert wird. Der neue Stackpointer wird berechnet, in dem auf dem Callercontextpointer die größe des Callercontext addiert wird. In den nächten Takten werden die alten Localspointer, Stackpointer und Callercontextpointer in den Callercontext geschrieben. [1]

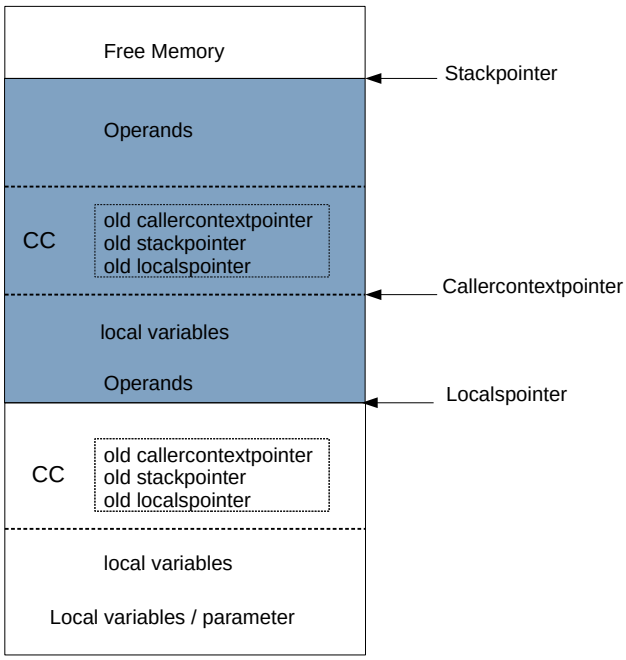


Abbildung 2.3: Stackframe nach einen Funktionsaufruf

Funktionsrücksprung

Bei einem Funktionsrücksprung werden im AMIDAR Framestack die Locals- Stack- und Callercontext-pointer aktualisiert. Der Vorgang beginnt, wenn eines der entsprechenden Token über den AMDIAR Bus gesendet wird. Es gibt 3 Varianten des Tokens. Eine liefert keinen Rückgabewert, die anderen geben jeweils 32 oder 64 Bit zurück. Wenn der Token abgearbeitet wird, werden nacheinander die drei Pointer ausgelesen und wiederhergestellt. Anschließend wird gegebenenfalls der Rückgabewerte gesichert, dieser steht in den top of Stack bzw. next of stack Registern. Der Rückgabewert wird an Stelle des obersten, im Falle eines Return64 der oberen beiden, Operanden des Ursprünglichen Funktionsaufruf gespeichert. [1]

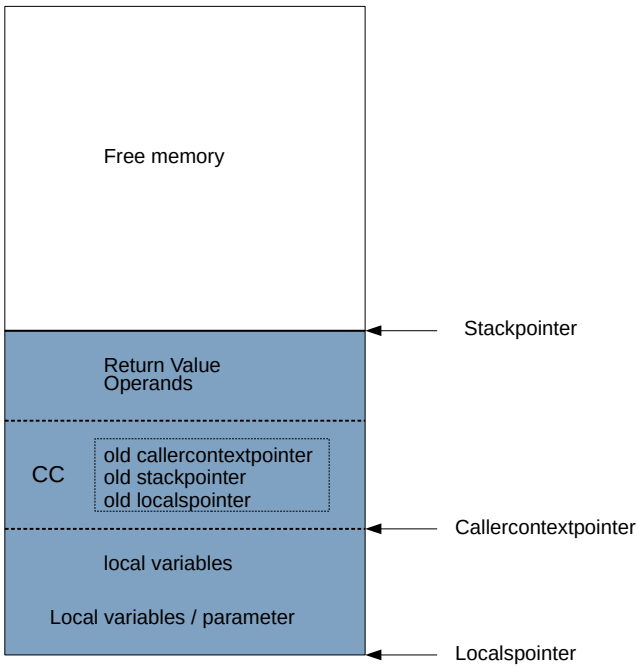


Abbildung 2.4: Stackframe nach einen Funktionsrücksprung

2.2 Spill and Fill

Spill and Fill beschreibt eine Strategie, durch die der Zugriff auf einen Stapelspeicher deutlich beschleunigt werden kann. Ein kleinerer, schnellerer Speicher hält einen Ausschnitt des Hauptspeichers vor, der je nach Bedarf hoch oder runter geschoben werden kann, sodass immer das obere Ende des Stapelspeichers im Window Ausschnitt liegt. Spill beschreibt dabei den Vorgang des nach oben Schiebens des Speicherausschnitts, bei den der untere Abschnitt des Ausschnitts in den Hauptspeicher geschrieben wird. Wird der Stack weit genug abgebaut, wird ein Fill durchgeführt, bei dem die ausgelagerten Teile zurück kopiert werden.

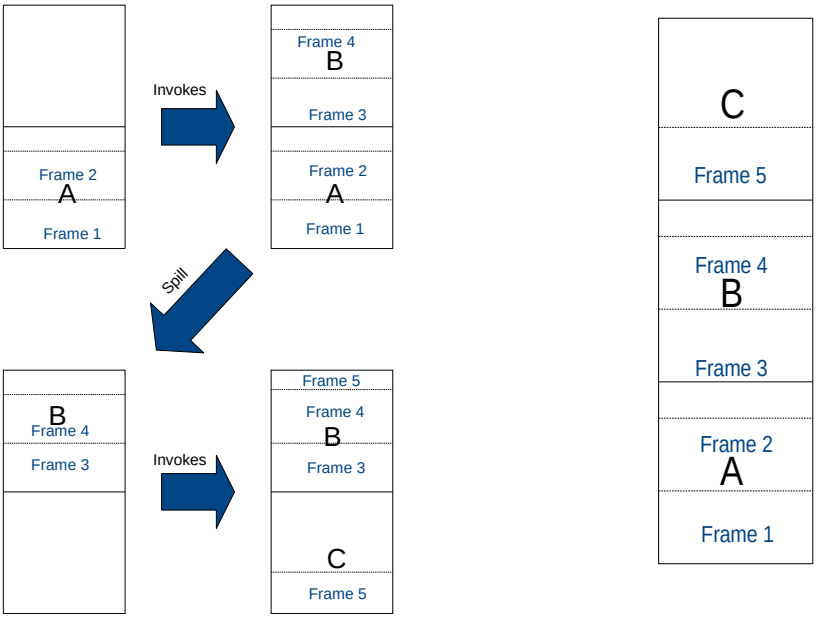


Abbildung 2.5: Spill: Verschieben des Windows

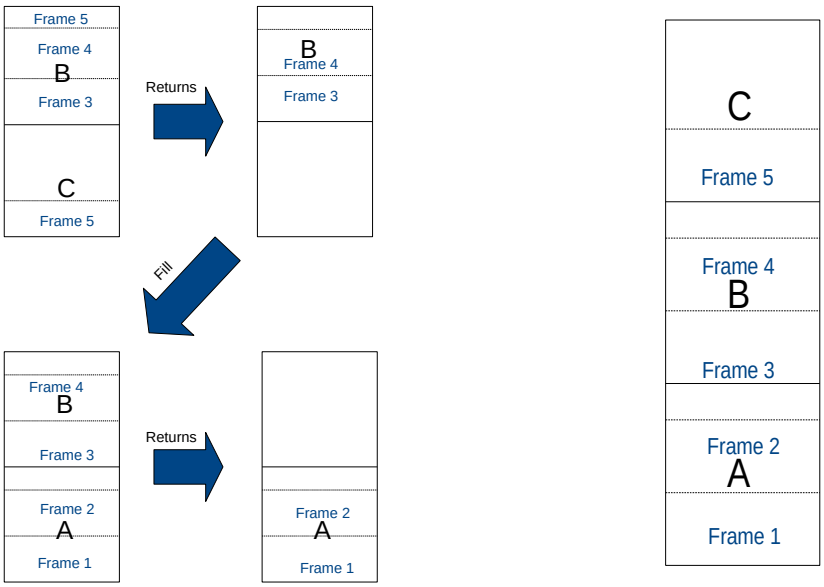


Abbildung 2.6: Fill: Verschieben des Windows

2.3 Verwendung eines Spill and Fill Mechanismus in anderen Prozessoren

2.3.1 Sun Sparc Prozessoren

Die Sun Sparc Prozessoren V8/V9 verfügen über ein sliding register Window mit jeweils 16 64 Bit großen Registern in 7 Registersätzen. Sliding Window bezeichnet ein Verfahren, bei dem die Registersätze im Falle eines Funktionsaufruf nicht auf dem Stack gesichert werden müssen, stattdessen wird auf den nächsten Registersatz gewechselt. Dabei wird meistens auch die Parameterübergabe realisiert. Dabei sind die Input Register in dem Registerwindow des Callers identisch mit dem Output Registern, in dem Registerwindow des Callee. Mit speziellen Bytecode Instruktionen werden im Spill an Fill Verfahren Registersätze auf einen externen Speicher gesichert. Da das Spill and Fill durch spezielle Bytecodes ausgelöst wird muss der Compiler vorher festlegen an welchen Stellen im Programm das Spill and Fill stattfinden soll. [2]

3 Implementierung

3.1 Übersicht über den AMIDAR FRAMESTACK

Für diese Arbeit wurde die Spill and Fill Windows und das überliegende Modul, das die Adressen bei Zugriffen auf die Windows umrechnet, neu geschrieben. Außerdem neu geschrieben wurde die Ansteuerung der AXI Verbindung und die dafür nötige Umrechnung von Adressen und Übertragungslängen. Der Zustandsautomat des Framestacks wurde grundlegend übernommen, jedoch stark erweitert und geändert, um Spill, Fill, Rootset Auslesen und externe Wishbone Lesezugriffe zu ermöglichen. Das Wishbone DataIf Modul wurde leicht erweitert, um mit den Zustandsautomaten interagieren zu können. Ohne Änderungen übernommen wurde das Threadtable Modul.

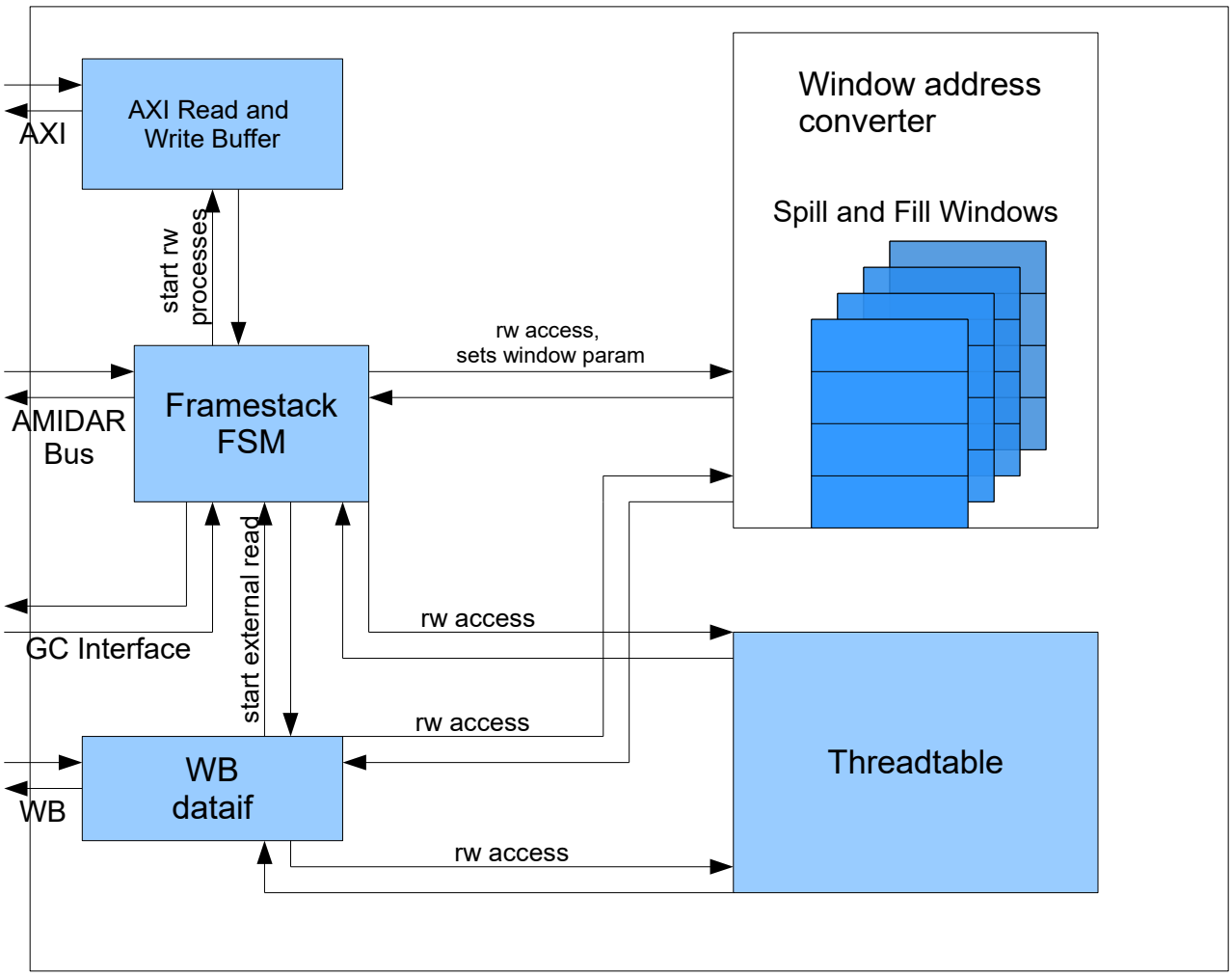


Abbildung 3.1: Übersicht über den AMIDAR FRAMESTACK

3.2 DRAM Anbindung über AXI

AMIDAR ist über AXI mit 512 MB DDR Ram verbunden, der auf 800 MHz getaktet.

3.2.1 Addressbereich des Framestacks im Hauptspeicher

Eine Schwierigkeit bei der Auslagerung des Framestacks an den Hauptspeicher ist die Unterschiedliche Wortbreite. Für den Garbarbagecollector werden zu jedem 32 Bit Datenwort noch 2 Statusbit gespeichert, in denen festgehalten wird, ob es sich bei den Daten um Metadaten, Referenzen oder Werte handelt. Damit beim Auslagern der Daten auf dem externen Speicher diese Informationen nicht verloren gehen, werden die Schreibvorgänge immer in Blöcken von 16 Wörtern durchgeführt, wobei die Statusbits der 16 Wörter als weiteres 32 Bit Wort in den Speicher geschrieben werden. Dies ist kein Problem, da die Größe der beim Spill and Fill Vorgang übertragenen Bereiche einen Vielfachen von 16 Wörtern entspricht.

Tabelle1

Framestack data format

Address	Framestack Entry	
	34 Bit	
	2 Bit	32Bit
0	Type 0	Data 0
1	Type 1	Data 1
2	Type 2	Data 2
3	Type 3	Data 3
4	Type 4	Data 4
5	Type 5	Data 5
6	Type 6	Data 6
7	Type 7	Data 7
8	Type 8	Data 8
9	Type 9	Data 9
10	Type 10	Data 10
11	Type 11	Data 11
12	Type 12	Data 12
13	Type 13	Data 13
14	Type 14	Data 14
15	Type 15	Data 15
16	Type 16	Data 16
17	Type 17	Data 17
18	Type 18	Data 18

Framestack Data in external RAM

Address *	Data 32 Bit
0	Data 0
1	Data 1
2	Data 2
3	Data 3
4	Data 4
5	Data 5
6	Data 6
7	Data 7
8	Data 8
9	Data 9
10	Data 10
11	Data 11
12	Data 12
13	Data 13
14	Data 14
15	Data 15
16	Type 0-15
17	Data 16
18	Data 17
19	Data 18

*Nicht die eigentliche Adresse im RAM, sondern der Offset im Speicherbereich des Threads

Seite 1

Abbildung 3.2: Zuordnung der Framestack Adressen im Hauptspeicher

Der Ausgelagerter Framestack wird in das obere Ende des 512MB großen DDR3 Speichers gelegt und belegt:
$(numbermaxthreads * wordsperthread) * (1 + 1/16)$ Wörter.
Als Ausgangspunkt wird die höchste Speicheradresse des DDR Speichers verwendet und 2 mal nach links geshiftet, da der externe Speicher Byte adressiert, der Framestack jedoch Wort adressiert ist. Davon wird die maximale Größe des Framestacks abgezogen. Von da an aufwärts werden die Daten der einzelnen Threads gespeichert.

3.2.2 Schreiben von Framestackinhalten in den Hauptspeicher

Für das Zwischenspeichern und Packen der Blöcke wird ein eigenes Modul verwendet. Vom Framestack aus kann auf dieses Modul geschrieben werden. Dabei wird die Startadresse des Blocks, die Länge des AXI Bursts und das erste Datenwort angelegt. Zum Zwischenspeichern dieser Daten werden jeweils FIFOs verwendet. Wobei die FIFOs für Burstlänge und Startadresse einen Sechzehntel der Größe des DatenFIFOs haben. Für jedes übertragene Datenwort werden 2 Statusbits in ein 32 Bit Register geschrieben. Nach dem 16 Datenworte in den Datenfifo geschrieben wurden, wird der Inhalt dieses Registers in den Fifo geschrieben. Das Modul, indem die eigentliche AXI Übertragung verarbeitet wird, wartet darauf das der Adress- und der BurstlängenFIFO nicht leer sind. Wenn dies der Fall ist, wird die Adresse und Burstlänge ausgelesen und wie folgt umgerechnet:

$AXI\_address := (RAM\_MaxAddress - MaxThreads * WordsPThread + address + (address >> 4)) << 2$
$AXI\_burst := burst\_length + (burst\_length >> 4)$

Für die Adressenberechnung wird die maximale Anzahl an Wörtern, die in den DRAM gespeichert werden kann, als Basis genommen davon wird die maximale Größe des Framestacks im Hauptspeicher abgezogen um die Basisadresse des Framestacks zu bekommen. Auf diese wird die Framestack Adresse, mit eingeplanten Plätzen für die Statusworte, drauf gerechnet. Die übergebene Burstlänge wird ebenfalls umgerechnet um den Platz für die zusammengefassten Statusbits zu berücksichtigen. Anschließend werden die Daten aus dem Fifo übertragen, bis die neu berechnete Burstlänge erreicht wurde. Solange noch Einträge in den Fifos für Burslänge und Adresse liegen wird, eine neue Übertragung gestartet.

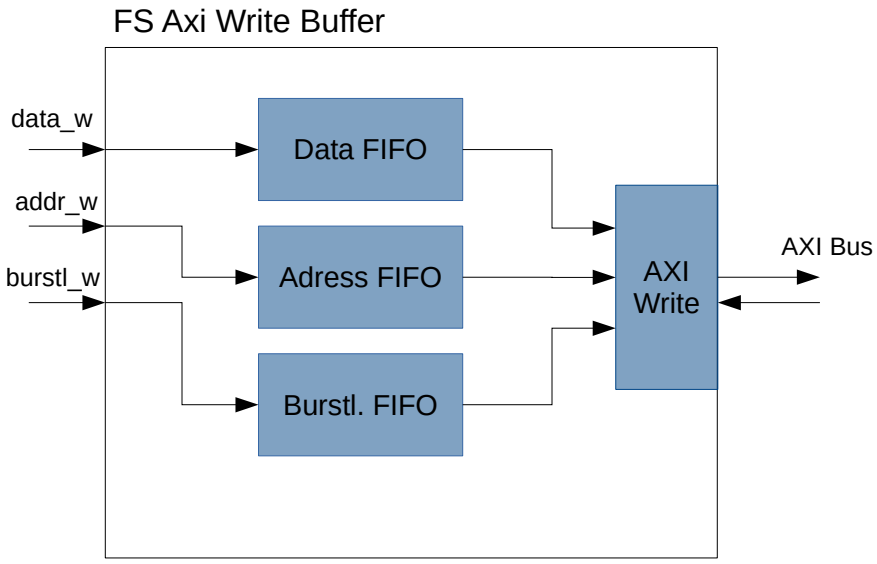


Abbildung 3.3: Schema des AXI Schreib Puffers

3.2.3 Lesen von Framestackinhalten aus den Hauptspeicher

Wenn Daten aus den externen Speicher ausgelesen werden sollen, wird das selbe Modul verwendet. Es wird die Startadresse des auszulesenden Blocks und die Burstlänge übergeben. Diese wird für den DDR Speicher umgerechnet und der Lesevorgang über AXI gestartet. Es wird Das erfolgreiche Einlesen wird dem Framestack signalisiert, woraufhin die Daten des Blocks ausgelesen werden. Für das Zwischenspeichern der ausgelesenen Daten, werden zwei FIFOs verwendet. Ein FIFO wird für die gelesenen Daten verwendet und speichert 32 Wörter mit 32 Bit Wortbreite. In einen weiteren FIFO, mit ebenfalls 32 Bit Wortbreite, werden die kumulierten Statusbits gespeichert. Nachdem die ersten 16 Datenworte in den FIFO gespeichert wurden, enthält das nächste Wort die Statusbits, dieses wird in den entsprechenden FIFO gespeichert. Sobald das FIFO für die Statusbits nicht leer ist, kann auf Seiten des Framestacks mit dem Auslesen begonnen werden. Bei jeden takt in dem eine 1 an readenable angelegt ist, wird ein Datenwort aus dem Fifo und 2 Statusbits übertragen. Nach dem 16 Wörter aus den Datenfifo gelesen wurden wird wenn vorhanden das nächste Statuswort aus den StatusFIFO gelesen.

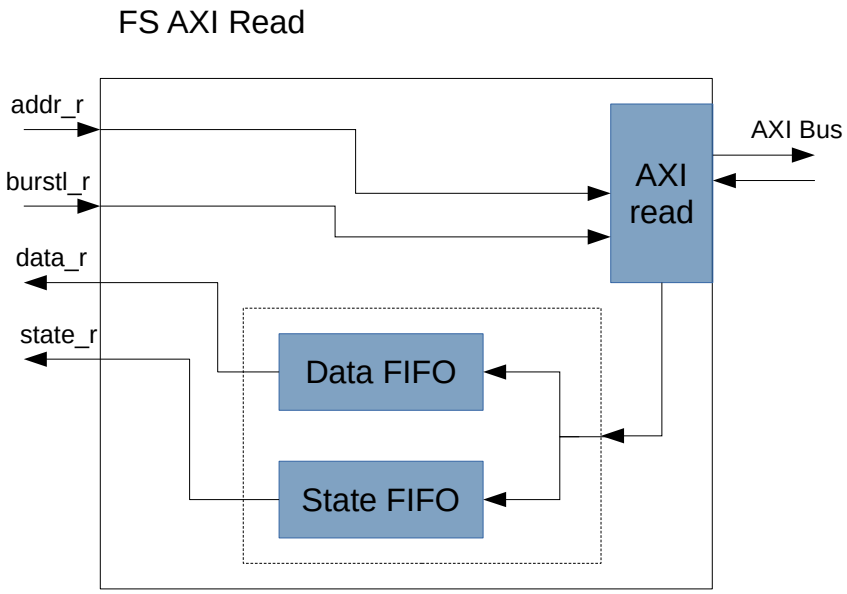


Abbildung 3.4: Schema des AXI Lese Puffers

3.3 Spill and Fill Windows

Bei den Spill and Fill Prozess werden Teile des aktuellen Threads und evtl die anderer Threads in sogenanntes Windows vorgehalten. Die Windows werden als Blockrams realisiert. Die Anzahl und Größe dieser Windows kann über Parameter eingestellt werden. Für jedes Window werden die Framestack Adressen der oberen und unteren Grenze, des Adressbereichs gespeichert, der in dem Window vorgehalten wird.Das Window selbst wird dabei als Ringspeicher organisiert. Dazu wird jeweils die Basis Adresse im Window gespeichert, die angibt an welcher Stelle innerhalb des Windows die nach auÃŸen angezeigt untere Grenze zeigt. Mit diesen drei Adresse kann jeweils der Speichertort der gewünschten Daten berechnet werden. Für jedes Window wird in einen Register gespeichert welcher Thread in diesen momentan vorgehalten wird.

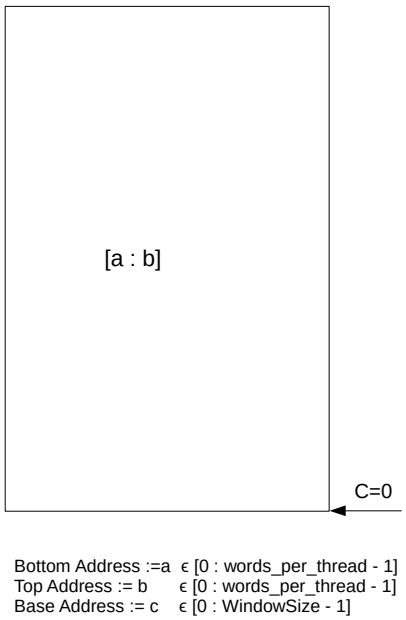


Abbildung 3.5: Schema des Spill and Fill Windows im Ursprungszustand

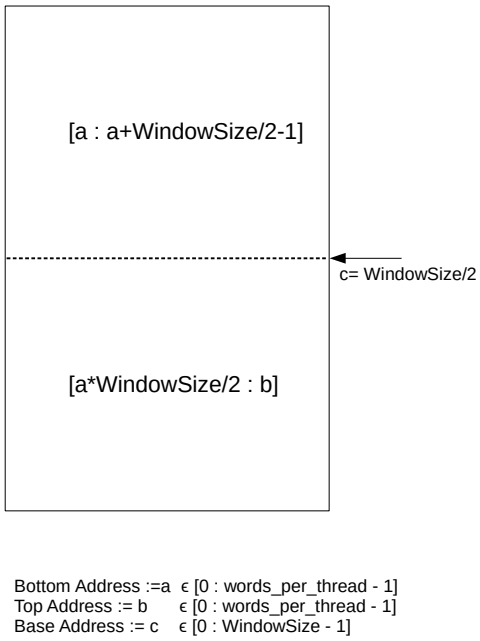


Abbildung 3.6: Schema des Spill and Fill Windows nach einen Spill

3.4 Framestackteile des aktiven Threads auslagern (Spill)

Stackteile des aktiven Threads auf den externen Speicher auslagern ist eines der zwei Hauptbestandteile des Spill and Fill Mechanismus. Nämlich Spill. Bei dieser Implementierung wird der Spill Mechanismus ausgelöst, wenn beim Funktionsaufruf festgestellt wird, dass der Stack das aktuelle Window überschreiten würde.

3.4.1 Überprüfen des freien Speichers im Window

Bei jeden Funktionsaufruf wird der Platzbedarf im Stack abgeschätzt und geprüft ob noch Platz dafür im aktuellen Window ist.

Ein Stackframe in AMIDAR beginnt mit den Argumenten und lokalen Variablen, darüber kommt der Callercontext bestehend aus: Localspointer, Callercontextpointer und Stackpointer. Zum Bestimmen ob der im Window vorhandene Platz noch ausreicht wird vom aktuellen Stackpointer die Anzahl der Parameter der aufgerufen nen Funktion abgezogen und Anzahl lokaler Variablen die grösse des Callercontexts darauf addiert und eine Kostante für die Anzahl möglicher Parameter drauf addiert. Wenn das Ergebnis noch im aktuellen Addressbereichs des Windows liegt, wird ein normaler Invoke durchgeführt, andernfalls wird ein Spill durchgeführt.

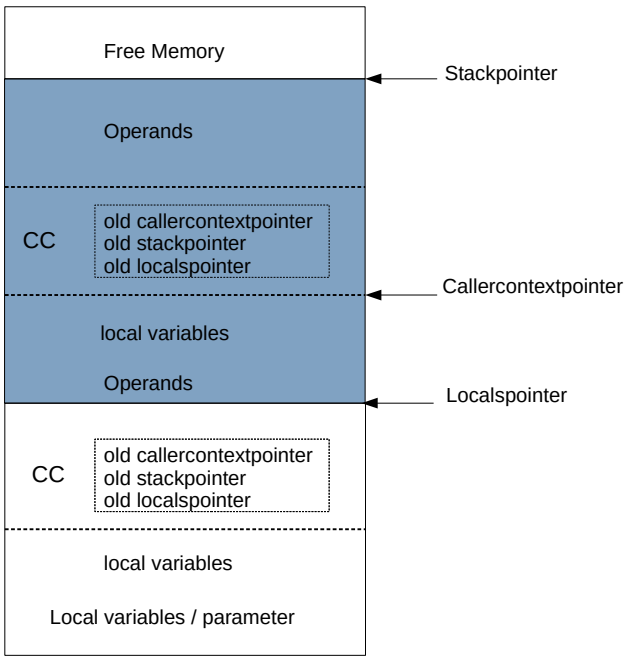


Abbildung 3.7: Stackframe nach einen Funktionsaufruf

3.4.2 Spill

Zu beginn des Spill Vorgangs wird die Burstlänge für die AXI Übertragung auf 128 gesetzt und die Bottom Adresse des unteren Windows wird in ein Register zwischengespeichert, das als Laufvariable für den folgenden Prozess verwendet wird und in ein Register kopiert die Basis des aktuellen 16er Block

angibt. Anschließend wird darauf gewartet, dass das AXI Modul geschrieben werden kann, ist dies der Fall wird der Speicher im Window mit den Wert im Address Register adressiert. Nachdem die Daten aus dem Blockram gelesen wurden wird ein in das oben beschriebene AXI Modul geschrieben. Dafür wird die Threadadresse zur Framestackadresse umgerechnet. Gleichzeitig wird der Wert im Register mit der Laufvariable um eins erhöht. In den folgenden Takten wird dieses Vorgang wiederholt, bis die Laufvariable um 15 über der Basisadresse des aktuellen Blocks liegt. Ist dies der Fall wird geprüft, ob das Ende des zu übertragenen Window Parts erreicht ist. Ansonsten wird der Vorgang mit geänderter Adresse neu gestartet. Am Ende des Spill Vorgangs werden die Register für die bottom und top Adresse aktualisiert. Im darauf folgenden Takt wird die Ausführung der Invoke Instruktion neu gestartet.

3.5 Ausgelagerte Framestackteile des aktiven Threads wiederherstellen (Fill)

Wenn der Stack durch Rücksprünge schrumpft und nahe dem unteren Ende des vorgehaltenen Adressbereichs kommt muss der ausgelagerte Framestack wiederhergestellt werden.

3.5.1 Ablauf eines return Vorgangs mit fill

Im ersten Takt wird der aktuelle callercontext pointer im Register old_callercontext_pointer zwischengespeichert und es wird der Callercontext des wiederherzustellenden Stackframes ausgelesen beginnend mit den localspointer . Anschließend wird der ausgelesene localspointer gespeichert und der Stackpointer zum Auslesen adressiert. Der lokalspointer stellt das untere Ende des neuen Stackframes da. Nach dem der Stackpointer wiederhergestellt wurde wird überpüft ob, der localspointer kleiner, als die unterste Adresse im Window ist, ist dies der Fall muss der Fill Vorgang gestartet werden. Ansonsten geht der return Vorgang normal weiter mit der Wiederherstellung des Callercontextpointers, der Rückgabe des neuen Programmcounters über den AMIDAR Bus und dem aktualisieren der Top_of_Stack und Next_of_Stack Register.

3.5.2 Ablauf des Fill Vorgangs

Bevor das kopieren der Stackdaten passieren kann werden müssen die Bottom- Top- und Baseadressregister des aktuellen Windows angepasst werden. Im nächsten Takt muss der vorher ausgelesene Programmcounter in einen Register zwischengespeichert werden, damit dieser später zurück gegeben werden kann. Anschließend wird die neue untere Adresse des Windows umgerechnet und der Lesevorgang im AXI Modul gestartet. In dem nächsten Takt wird gewartet bis der Lesebuffer gefüllt ist. Wenn dies der Fall ist wird mit jeden weiteren Takt ein Datenwort und die dazu gehörigen Status Bits ausgelesen und von unten beginnend in das Window gespeichert. Nach jeden gespeicherten 16 Bit Block wird überpüft ob ein ganzes Segment übertragen wurde oder der ganze Burst ausgelesen wurde. Wenn ersteres der Fall ist wird der Vorgang beendet. Falls der Burst komplett übertragen wurde, das aktuelle Segment jedoch nicht vollständig übertragen ist, wird ein neuer Leseburst mit der aktuellen Adresse gestartet.

3.6 Multithreading mit Verdrängung

Es kann eine konstante Anzahl an Threads in den Windows vorgehalten werden. Um die Multithreading Funktionalität mit Threads weiterhin zu gewärleisten wurde eine Threadverdrängung implementiert durch die Stacks von in den Hauptspeicher ausgelagert und wiederhergestellt werden können.

3.6.1 Zuordnung von Threads zu den Windows

Es gibt für jedes Window ein Register in dem die zugeordnete ThreadID gespeichert wird. Dazu kommt 1 Status Bit, mit dem angegeben wird, ob dem Window bereits eine ThreadID zugeordnet ist und 4 Bits, die für einen Least Recently Used counter genutzt werden. Nach einen Reset des Framestack Moduls werden alle Bits der Windows 1-n bis auf das erste Statusbit auf 0 gesetzt. Für Window 0, in dem der Stack des Thread 0 liegt, wird das erste Bit auf 0 gesetzt, damit es ohne explizite Threaderzeugung vom Bootloader genutzt werden kann.

3.6.2 Änderungen am der Threaderzeugung

Bisher wurden Threads im Framestack angelegt in dem über das Wishbone Interface erst der Threadtable initialisiert wird und anschließsend der Stackframe durch Schreiben der Rücksprungadresse und eines leeren Callercontext in den Blockram. Das Problem dabei ist durch die Auslagerung der Threads in den Framestack die Daten in den Hauptspeicher geschrieben werden müsste, was aufwendig zu implementieren wäre und sehr viel Zeit bei der Ausfürung benötigen würde. Es ist außerdem nicht möglich neue Threads direkt im Window zu initialisieren, da es möglich ist das mehr Threads initialisiert werden, als Windows verfügbar sind, bevor die Ausführung der Threads zum ersten mal gestartet wird. Stattdessen wurde für die Threaderzeugung der bisher ungenutzte Framestack Opcode OP_FRAMESTACK_NEWTHREAD genutzt um den Stackframe der Threads zu initialisieren. Der Threadtable wird weiterhin über Wishbone initialisiert. Sobald der NEWTHREAD Opcode abgearbeitet wird geprüft ob schon ein Thread mit der zu über Wishbone übergebenen ID in einen der Windows vorhanden ist. Ist dies der Fall wird der als Parameter mit den Opcode übergebene Threadhandle an Adresse 0 des entsprechenden Windows geschrieben. Darüber wird der Callercontext initialisiert. Wenn die ThreadID noch keinen der Windows zugewiesen ist, muss der Stackframe des Threads im externen Speicher initialisiert werden. Dafür wird auch das vorher beschriebene AXI Modul verwendet. Es wird der Threadhandle und der leere Callercontext übertragen. Außerdem müssen noch 12 weitere leere Wörter übertragen werden um einen 16 Wort Block übertragen zu können.

3.6.3 Threadwechsel

Für einen Threadwechsel bei einen AMIDAR Prozessor werden im Framestack die Werte der Register Stackpointer, Localspointer und Callercontext aus dem Threadtable übernommen. In der neuen Implementierung wird zusätzlich geprüft ob, der Thread auf den gewechselt wird bereits in einen der Windows vorgehalten wird. Ist dies der Fall kann einfach auf das entsprechende Window gewechselt werden um den Thread weiter auszuführen. Wenn der Stack des Threads noch nicht in einen der Windows sein sollte wird dieser aus dem Speicher geladen. Davor wird gegebenenfalls noch der Stack eines zu verdrängender Threads in den externen Speicher vorschoben.

Ablauf des Threadwechsels

Der Threadwechsel beginnt, wenn der Opcode ÖP_FRAMESTACK_THREADSWITCHäbgearbeitet wird. Die ThreadID des neuen Threads wird über den AMIDAR-Bus übergeben und in einem Register gespeichert. Zu Beginn des Vorgangs wird gepüft ob die ThreadID schon im WINDOW_TO_THREAD Register eingetragen ist. Ist dies nicht der Fall wird geprüft ob es noch ein Window leer ist. Wenn kein Window verfügbar ist werden die LRU Bits überprüft und der am wenigsten verwendete Thread bestimmt. Wenn auf einen schon vorgehaltenen Thread gewechselt wird, wird der LFU Counter des Threads um eins erhöht. Dabei

wird überprüft ob der counter des aktuellen nächsten Threads gleich 15 ist. Ist dies der Fall wird der counter, aller Threads halbiert. In dem Fall, in dem ein Thread verdrängt werden muss wird der am wenigsten verwendete Thread in den externen Speicher transferiert. Dafür wird die passende Burstlänge anhand des Stackpointers und der unteren Windowadresse berechnet. Sie muss ein Vielfaches von 16 sein. Danach findet die eigentliche Übertragung des Stacks statt. Nachdem der Stackframe des zu verdrängenden Threads gesichert wurde kann das Window mit den Stack des neuen Threads überschrieben werden. Dafür wird die untere Startadresse der Übertragung anhand des Localspointer und der größe der Windowabschnitte sodaß der Locals und der Stackpointer im Window liegen. Daraufhin wird der zu übertragende Stackframe ausgelesen und in das Window kopiert.

3.7 Garbage Collector Interface

Der Framestack muss den Garabage Collector ein Interface zur Verfügung stellen über dass, das Rootset ausgelesen werden kann. Dafür werden für jedes Datenwort im Framestack zwei Statusbits angegeben, die den Entrytype beschreiben. Der Entrytype 2'b10 gibt dabei Referenzen an. Um das Rootset zu erzeugen muss der Stack jedes einzelnen Threads ausgelesen werden, dabei wird bei jeden Wort geprüft ob der EntryType 2'b10 hinterlegt wurde, in dem Fall wird das Word an an Garbage Collector übertragen. Da Teile des Framestacks im externen Speicher liegen müssen diese ausgelesen werden. Wenn der Eingang gc_request_rootset gesetzt ist beginnt der Auslese Vorgang. Der Stack wird Thread für Thread ausgelesen. Nachdem ein Thread zum auslesen ausgewählt wurde wird geprüft, ob dieser im Window liegt. Ist das der Fall wird geprüft, ob der komplette Stack im Window liegt. Wenn Teile des Stacks ausgelagert wurden, werden diese in 16er Blöcken ausgelesen und geprüft. Sobald die untere Grenze des Windows dabei erreicht ist, kann der Rest aus dem Window gelesen werden. Sobald alle Threads nach diesen Verfahren abgearbeitet wurden ist das Rootset komplett übertragen.

3.8 Lesezugriff Wishbone

Lesezugriff über Wishbone auf dem Framestack musste auch umgeschrieben werden. Bisher liefen die Zugriffe direkt über den 2. Port des Blockram in dem der Framestack gespeichert wurde. Wegen des ausgelagerten Threads funktioniert das nur wenn die auszulesenden Daten im Window liegen. Wenn das nicht der Fall ist, müssen diese jeweils aus dem RAM geladen werden, was von der FramestackFSM abgearbeitet wird. Währenddessen muss allerdings das Wishbone Acknowledge Signal zurück gehalten werden, wodurch der Wishbone-Bus eine Weile blockiert wird. Da dieser Lesezugriff nur für den Debugger genutzt wird, ist die Performance bei diesen Vorgang jedoch vernachlässigbar.

4 Evaluation

Um die korrekte Funktionsweise und die Performance des geänderten Framestacks zu überprüfen wurde eine modifizierte AMIDAR Testbench verwendet. Vorallem im Fokus stand die korrekte Ausführung der SSpill and FillÄbläufe und die Threadwechsel.

4.1 Testumgebung

Für die Tests wurde die AMIDAR Testbench modifiziert um Funktionen mit einen groÃŸen Framestack oder Threadwechseln zu testen. Außerdem wurde die Hardware des Framestacks erweitert um die benötigte Anzahl an Takten für einen bestimmten Vorgang aufzuzeichnen.

4.1.1 Hardwareänderungen zur Performance Evaluation

Um die Performance evaluieren zu können ein zusätzlicher Blockram angelegt in dem jeweils die Dauer eines bestimmten Vorgangs protokolliert wird. Es kann entweder die Dauer der Spill, der Fill oder der Threadswitch Vorgänge protokolliert werden. Um auf diese Daten zuzugreifen wurden dem Framestack die Wishbone Register 15, 16, 17 und 18 hinzugefüfgt. Mit den Register 18 wird dabei angegeben ob und welcher Vorgang protokolliert werden soll. Wird eine 0 in das Register geschrieben wird die Protokollfunktion deaktiviert. Mit Register 15 Wird angegeben aus welcher Adresse des Speichers gelesen werden soll. Die ausgelesenen Daten selber stehen in Register 18. Das Register 17 gibt die Anzahl der Einträge in dem Speicher an. Zur Protokollierung eines Vorgans werden die Takte gezählt sobald der Startzustand eines Vorgangs erreicht ist und endet wenn der Vorgang beendet wurde und neue Tokens abgearbeitet werden können. Zu beachten dabei ist, das bei jeden Vorgang nur die Anzahl der Takte gemessen wird bei denen die Framestack FSM beschäftig ist. Beim Spill zum Beispiel wird das zählen beendet sobald die letzten Stackdaten in den Schreibfifo geschrieben wurde. Die Zeit, in der der AXI Bus belegt ist, ist etwas länger, da der FIFO noch abgearbeitet werden muss. Der Framestack selber kann währenddessen schon weitere Token abarbeiten. Alle eben beschriebenen Änderungen lassen sich wenn nicht benötigt durch eine Präprozessordefinition deaktivieren.

4.1.2 angepasste Testbench

Um die weiterhin korrekte Funktionsweise der Grundlegenden Framestack Funktionalitäten zu testen wurde die AMIDAR Testbench genutzt. Erweitert wurde diese dabei zum einen um Tests, die den Spill and Fill Vorgang testen. Dafür wurde eine Funktion benötigt, die einen möglichst großen Stack erzeugt und dennoch eine einigermaßen kurze Laufzeit hat. Die Funktion zur Rekursiven Berechnung der Fibonaci Folge erzeugt zwar schnell einen sehr groÃŸen Stack, allerdings ist die Laufzeit dieser deutlich zu lang um als Test in Frage zu kommen. Zum Testen der Spill and Fill Funktionen wurde eine vereinfachte Ackermann Funktion genutzt, die mit 2 Parametern arbeitet.

Tabelle 4.1: Ackermanntests

Ackermann (3,3):	
	Rekursionstiefe = 63
	Maximale Stackgröße = 450 Wörter
	Rückgabewert = 9
Ackermann (3,4)	
	Rekursionstiefe = 127
	Maximale Stackgröße = 890 Wörter
	Rückgabewert = 61
Ackermann (3,5):	
	Rekursionstiefe = 255
	Maximale Stackgröße = 1785 Wörter
	Rückgabewert = 253

```
private static int ackermann(int n, int m) {
   int i= 0;
      if(n==0 && m== 0) {
         i=1;
      }
      if (n == 0)
         return m + 1;
      else
      if (m == 0){
         return ackermann(n - 1, 1);
      }
      else
         return ackermann(n - 1, ackermann(n, m - 1));
   }
```

In dem neuen Testfall AckermannTest wird diese Funktion mit unterschiedlichen Parametern aufgerufen:

Weitere wichtige Testfälle waren die schon vorhandenen, jedoch nicht einkommentierten Multithreading Tests. Genutzt wurde der "BasicThreadTestünd der ÄdvancedThreadTest". Außerdem wurde eine Modifizierte Variante des BasicThreadTests genutzt, bei der der Ackermanntest in mehreren Threads ausgeführt wurde.

Eine weitere Anpassung der Testbench ist eine Methode die über Wishbone die im Framestackmodul gemessenen Performancedaten ausliest. Zuerst wird die Anzahl der erfassten Vorgänge ausgelesen. AnschlieÃŸend wird über den Speicher mit den Messdaten iteriert und der Inhalt ausgegeben, dabei wird das arithmetische Mittel der Messwerte berechnet und ebenfalls ausgegeben.

4.2 Performance Messungen

Bei den Performance Messungen wurde die Dauer bestimmter Vorgänge gemessen in dem die oben beschriebenen Tests ausgeführt und die Messdaten ausgelesen worden sind. Die Tests wurden mit Unterschiedlichen Konfigurationen den Spill and Fill Windows des Framestacks durchgeführt. Es wurde sowohl mit einer Window Größe von 512 Wörter als auch mit 1024 Wörtern getestet. 1024 Wörter entspricht dabei der Anzahl der Wörter die im ursprünglichen Framestack einen Thread jeweils zur

Tabelle 4.2: Spill and Fill Performance

Anzahl Segmente pro Window	2		4		8	
Fenstergröße	512	1024	512	1024	512	1024
Anzahl Takte Spill	288	576	144	288	72	144
Anzahl Takte Fill	363	726	183	363	111	182
Anzahl Spill and Fill Vorgänge	462	131	787	199	1435	329
Takte gesamt	300762	170562	257349	129549	262605	107254
Anteil an Ausführungszeit	1,676%	0,914%	1,448%	0,729%	1,467%	0,604%

Verfügung standen. Das Window wird dabei in 2 Segmente geteilt, die jeweils 256 oder 512 Wörter groß sind.

4.2.1 Spill and Fill

Bei der Variante mit 512 Wörtern werden für einen Spill Vorgang 288 Takte benötigt. Der Wert ist konstant da, Verzögerungen der AXI Verbindung durch den Schreibpuffer ausgeglichen werden. Ein Fill Vorgang dauerte in dem Fall mindestens 363 und maximal 370, im Durchschnitt 363 Takte Beim Durchlauf der Ackermanntests findet der Spill und der Fill Vorgang jeweils 462 mal statt. Ein Fill benötigt mehr Takte, da zum einen das adressieren des AXI Busses für den Auslese Vorgang Zeit benötigt. AuÃŸerdem müssen nach dem Ende der AXI Übertragung noch mindestens 16 Wörter aus dem Lesepuffer gelesen werden. Wenn ein Window 1024 Wörter umfasst werden für einen Spill 576 Takte gemessen. Für den Fill werden dabei zwischen 725 und 744 Takte gemessen. Im Durchschnitt werden 726 Takte benötigt. Es finden beim Durchlauf der Testbench jeweils 131 Spill and Fill Vorgänge statt.

Interpretation

Bei der Variante mit 512 Wörtern werden insgesamt 300762 Takte für Spill and Fill Operationen genutzt. Mit einen 1024 Wörter großen Window werden lediglich 162702 Takte durch Spill and Fill blockiert. Bei der Variante mit einen 1024 Wörter groÃŸen Window ist in diesen Testfällen die Anzahl der Takte die für Spill and Fill benötigt werden beinahe halbiert, was ein groÃŸen Performance Vorteil andeutet. Berücksichtigt man jedoch, das die Gesamtlaufzeit der Testfälle 17943867 bzw 17805807 Takte beträgt, ist Spill and Fill nur für 0,91376% bzw 0,1676% der Laufzeit der Tests verantwortlich. Gemessen wurde dabei wirklich nur die Laufzeit der Testfälle ohne Bootloader, static Initializer und uart Ausgaben. Bei realen Anwendungen dürften der Prozentsatz deutlich niedriger liegen als bei der Ackermannfunktion.

4.2.2 Threadswitch

Die Dauer eines Threadwechsels ist stark schwankend, je nachdem wie groÃŸ der Stack des nächsten Stacks ist. Wird auf einen Thread gewechselt, der bereits in einen der Windows liegt, werden lediglich 5 Takte für den Wechsel benötigt. Nach Ausführung der Multithreading Tests werden 43 Threadwechsel gemessen. Diese benötigten zwischen 5 und 245 Takten. Im Durchschnitt werden 99 Takte. Mehrere Windows bringen bei den Threadtests der AMIDAR Testbench jedoch keinen groÃŸen Vorteil, da alle 10 Threads nacheinander aufgerufen werden. Bei Anwendungen in denen wenige Threads oft, viele andere dagegen selten geweckt werden, treten Threadverdrängungen seltener auf, was zu zu deutlich schnellere Threadwechseln führen dürfte.

5 Fazit

Im Zusammenhang dieses Projektseminars wurde die AMIDAR Framestack FU um Spill and Fill erweitert. Was deutlich grössere Stackframes ermöglicht bei gleichzeitiger Reduzierung des benötigten Blockrams. Der Gesamteinfluss auf die Performance des Prozessors dabei ist minimal.

Um dies zu ermöglichen wurde eine Anbindung des Framestacks über AXI eingebaut. Dafür wurde ein Modul gebaut was neben der AXI Ansteuerung auch das konvertieren der Unterschiedlichen Adressbereiche von Framestack zu DDR RAM übernimmt sowie das Puffern der Daten beim Lesen und schreiben.

Der Ablauf der Spill and Fill Vorgänge wurde implementiert und neue Testfälle geschrieben um diese zu testen. Um weiterhin Multithreading ohne Einschränkungen zu erlauben ein System zur Verdrängung wenig genutzten Threads aus den Spill and Fill Windows implementiert.

Dazu kamen einige Probleme die durch den in den Hauptspeicher ausgelagerten Framestack auftraten, an die vorher nicht gedacht wurde. Es musste sichergestellt werden das der Framestack weiterhin über das Wishbone Interface ausgelesen werden kann um die Funktion des Debuggers weiterhin zu gewährleisten ohne das es zu Deadlocks mit anderen FUs kommt. Des weiteren musste die Threadinitialisierung geändert werden und das Garbage Collector Interface wurde weitgehend neu geschrieben, um auch das Rootset des ausgelagerten Teil des Framestacks zu erfassen.

5.1 Ausblick

Als Zukünftige Verbesserung kann eine dynamische Vergrößerung des Speicherbereichs implementiert werden. Dabei sollten Maßnahmen eingebaut werden um eine Überlappung von Heap und Framestack zu verhindern.

6 Literaturverzeichnis

[1] Illy Christian. Implementierung der Thread-Verwaltung in einem AMIDAR. Master's thesis, TU Darmstadt, Fachgebiet Rechnersysteme, Dezember 2015.

[2] Gove Darryl. Flush Register Windows. `https://blogs.oracle.com/d/entry/flush_register_windows`, März 2008. Accessed: 9.02.2016.

[3] Andresen Jan. Erweiterung des Bus Protokolls für AMIDAR-Prozessoren. Master's thesis, Tu Darmstadt, Fachgebiet Rechnersysteme, Juni 2016.

[4] Burkert Tim. Entwicklung eines Beschleuniger-Frameworks für den AMIDAR-Prozessor, Juni 2016.

Abbildungsverzeichnis