Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät Informatik und Medien

Masterarbeit

Memory Tagging als Sicherheitsmechanismus für RISC-V am Beispiel des SEC-V-Prozessors

Leipzig, Februar 2024

vorgelegt von
Till Mahlburg
Studiengang Informatik

Betreuender Hochschullehrer:
Prof. Dr.-Ing. Jean-Alexander Müller
Fakultät Informatik und Medien

Externe Betreuung:
Dr. Jörn Hoffmann
bitaggregat GmbH

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Jahr 2019 hielt Matt Miller vom Microsoft Security Response Center einen Vortrag, in dem er u. a. zeigte, dass etwa 70 % der behobenen Schwachstellen in Software bei Microsoft auf Probleme mit der Speichersicherheit zurückgehen [16][S. 10]. Andere Berichte, etwa von Google über Android [27], gehen in eine ähnliche Richtung. Es ist also keine Überraschung, dass eine neue Sicherheitstechnik zunehmend Forschung und Verbreitung erfährt, die genau solche Speicherfehler zu erkennen, und damit ihre Nutzung als Sicherheitslücken zu verhindern verspricht. Diese Technik nennt sich Memory Tagging.

Auch die Verwendung mehrerer Threads zur Bewältigung der stetig zunehmenden Anforderungen an die Rechenleistung von Mikrocontrollern ist eine für diesen Teilbereich immer noch herausfordernde und an Relevanz gewinnende Entwicklung, auch wenn sie in größeren Prozessoren längst weit verbreitet ist. Dabei ist *Interleaved Multithreading* ein besonders leichtgewichtiger Ansatz, um die Ressourcenauslastung eines Prozessors bei geringen zusätzlichen Hardwarekosten zu optimieren. Gerade bei Mikrocontrollern bietet sich dieses Verfahren daher an. Unter anderem deshalb soll auch der *SEC-V*-Prozessor, der bei der Leipziger Firma *bitaggregat* entwickelt wird, Interleaved Multithreading einsetzen. Dieser stellt eine Neuentwicklung auf Basis der offenen Befehlssatzarchitektur RISC-V dar und dient aktuell vor allem als Forschungsplattform, soll aber zukünftig Ausgangspunkt für den produktiven Einsatz werden. Ein besonderer Fokus liegt dabei u. a. auf Modularität und Erweiterbarkeit, vor allem aber auf Sicherheitstechniken. Daher besteht ein Interesse an Techniken, mit denen die einzelnen Threads des Prozessors voneinander isoliert werden können und sie nur die Daten untereinander teilen zu lassen, die auch tatsächlich geteilt werden sollen.

In dieser Arbeit wird untersucht, wie sich beides, also Memory Tagging und Interleaved Multithreading, verbinden lässt, um sowohl vor typischen Speicherfehlern innerhalb eines, als auch zwischen verschiedenen Threads zu schützen. Dafür wird der SEC-V um die nötigen Module erweitert, die Memory Tagging möglich machen. Die Funktionsweise wird außerdem anhand von Testprogrammen überprüft, die typische Speicherfehler simulieren und zeigen, wie der angepasste Prozessor diese erkennt und idealerweise deren Ausnutzung verhindern kann.

Aufgeteilt ist die Arbeit dabei in sieben Kapitel, bei denen in Kapitel 2 die *Grundlagen* des Memory Taggings und dann in Kapitel 3 einige aktuelle Umsetzungen als *Stand der Technik* Stand der Technik vorgestellt werden. Es folgt Kapitel 4 mit der Beschreibung des *Entwurfs* sowie der eigentlichen *Implementierung* in Kapitel 5. In Kapitel 6 wird dann die *Auswertung* der Umsetzung vorgenommen. Am Schluss steht das Kapitel 7 mit *Zusammenfassung und Ausblick*.

2 Grundlagen

Um den Entwurf der zu entwickelnden Erweiterung vorzubereiten, wird hier die theoretische Basis der zugrunde liegenden Technologie erklärt. Dabei geht es in erster Linie um die Grundlagen der Speichersicherheit und die typische Umsetzung des Memory Taggings, um diese zu gewährleisten. Darüber hinaus wird der Fokus der Arbeit gegenüber verwandten Ansätzen abgegrenzt. Konkrete Umsetzungen werden dagegen im Kapitel *Stand der Technik* vorgestellt.

2.1 Memory Tagging

Memory Tagging bezeichnet als Oberbegriff verschiedene Techniken, bei denen Speicherblöcke mit Metadaten versehen werden. Dabei sind die Metadaten technisch unabhängig vom eigentlichen Speicherinhalt und können passend zur Problemstellung gewählt werden [12][S.124:2]. Konkret werden Speicherbereiche (*Granule*) einer festen Länge (*tag granularity*) mit einem *Tag* ebenfalls fester Größe (*tag size*) assoziiert [12][S.124:2-3].

So können etwa wie im Beispiel in Tabelle 2.1 je 16 Adressen des Speichers mit einem vier Bit langen Tag versehen werden. Diese Zuordnung wird dann selbst entweder in der bestehenden oder in einer isolierten Speicherhierarchie abgelegt [12][S. 124:13-14]. Welcher Tag konkret welchem Granule zugeordnet wird, ist abhängig von der jeweiligen Implementierung. Die Tags werden dem Prozessor von der ausgeführten Anwendung mitgeteilt oder teil- oder vollautomatisch generiert. Der Tag wird dann üblicherweise in den oberen Bits von Adresszeigern kodiert, die je nach System häufig ohnehin keine Verwendung haben, da die unterstützte Anzahl von Speicheradressen oft kleiner ist, als es die tatsächliche Länge eines Zeigers ermöglichen würde (vgl. z.B. [6][S.3]). Bei der Allokation muss dann dieser Tag dem entsprechenden Speicherbereich zugeordnet werden. Beim späteren Zugriff auf den Speicherinhalt muss dann wieder ein Tag im Zeiger eingebettet sein. Dieser Anteil wird bei der Auflösung der Adresse ignoriert und stattdessen genutzt, um in der Zuordnung zu überprüfen, ob der gegebene mit dem erwarteten Tag übereinstimmt. Ist das der Fall, so wird ganz normal der entsprechende Speicherinhalt gelesen. Sollte es keine

Tag	Speicheradresse	Granule
0110	m_0 m_1 ... m_{15}	g_1
1011	m_{16} ... m_{31}	g_2
0010	m_{32} ... m_{47}	g_3
1001	m_{48} ... m_{63}	g_4

Tabelle 2.1: Beispielhafte Darstellung eines *tagged memory* mit einem vier Bit langen Tag für Granules je 16 Adressen bzw. Bytes.

Übereinstimmung geben, kann der Prozessor auf verschiedene Weise reagieren; von einfachen Warnmeldungen bis zum Abbruch des Programms.

Während frühe Architekturen Memory Tagging als Mechanismus zum Debugging oder zur Auszeichnung des Datentyps eines Speicherworts verwendet haben (vgl. z.B. [9][S. 644]), wird es in den letzten Jahren besonders als Sicherheitstechnik verstanden [17][S. 202].

2.2 Speicherfehler

Den Ausführungen in [34] folgend, können in C und anderen Programmiersprachen mit vergleichbar freiem Zugriff auf den Speicher verschiedene Typen von Fehlern beim Speicherzugriff auftreten. Grundsätzlich spricht man von einem solchen Speicherfehler, wenn ein Zeiger auf ein anderes Speicherelement verweist, als von der Programmiererin oder dem Programmierer vorgesehen bzw. erwartet. Das eigentlich gemeinte Speicherelement wird dabei auch als *Referenz* bezeichnet. Dabei können mögliche Fehler in zwei Kategorien klassifiziert werden: räumliche und temporale [34][S.1].

Ein **räumlicher Speicherfehler** tritt dann auf, wenn ein Zeiger auf ein Element verweist, das außerhalb der räumlichen Grenzen der eigentlichen Referenz liegt. Ein Beispiel dafür wäre der Versuch, auf das sechste Element eines fünf-elementigen Arrays zuzugreifen. Typische Fälle sind etwa das Überlaufen

von Puffern, deren Größe bei ihrer Befüllung nicht korrekt überprüft wird, also *buffer overflows*. Andere Beispiele sind fehlerhafte Zeigerarithmetik und die Verwendung von uninitialisierten oder NULL-Zeigern [34][S. 1].

Demgegenüber passiert ein **temporaler Speicherfehler**, wenn versucht wird, auf ein Speicherelement zuzugreifen, das nicht mehr oder noch nicht existiert. Das ist etwa der Fall, wenn ein Zeiger verwendet wird, nachdem auf das referenzierte Speicherelement bereits `free` angewendet wurde, es also deallokiert wurde [34][S.1]. Man spricht auch von *use-after-free* [29][S. 3-4]. Ein Sonderfall ist dabei die erneute Anwendung von `free` mithilfe des verbliebenen Zeigers. Dies wird als *double-free* bezeichnet [29][S. 4].

Beide Arten von Speicherfehlern führen unter Umständen nicht nur zu unerwünschtem Verhalten eines Programms, sondern können auch Angreiferinnen und Angreifern die Möglichkeit geben, den Speicher auf unvorhergesehene Art und Weise zu lesen oder sogar zu verändern [29][S. 4]. Sie stellen auf diese Weise eine Einstiegsmöglichkeit für komplexere Angriffe dar [6][S. 1].

2.3 Memory Tagging als Sicherheitsmechanismus

Ein Ansatz, die eben beschriebenen Speicherfehler zu verhindern oder zumindest zu bemerken, ist die Verwendung von Memory Tagging. Die grundlegendste Art der Umsetzung wird dabei z. T. auch als *Memory Coloring* [17][S. 201] oder *lock and key memory access* [6][S. 2] bezeichnet.

Der Tag eines Granules kann hierbei als Schlüssel oder *Color* bzw. Farbe des entsprechenden Speicherbereichs angesehen werden [17][S. 201]. Dieser wird oft zufällig bei der Allokation gewählt, andere Auswahlansätze sind jedoch möglich [20][S. 3-6]. Versucht man wieder auf den Speicher zuzugreifen, muss man den korrekten Tag als Schlüssel mit angeben. Ist dieser verkehrt bzw. stimmen die Farben nicht überein, wird der unerlaubte Zugriff verweigert oder zumindest protokolliert [25][S. 3]. Auf diese Weise kann der versehentliche oder gezielte Zugriff auf unerlaubte Speicherbereiche deutlich erschwert werden.

An Tabelle 2.2 wird deutlich, wie mit diesem Verfahren **räumliche Speicherfehler** unterbunden werden können. In g_1 und g_2 könnte hier ein Array gespeichert sein, das aus 32 Bytes (0-31) besteht. Beim Zugriff auf das Byte an der Stelle m_{31} muss die Anwendung den korrekten Tag (0110) mit der Speicheradresse zusammen angeben, damit der Inhalt ausgelesen werden kann. Versucht die Anwendung nun aber die nächste Speicherstelle (m_{32}) zu lesen, etwa weil die Größe des Arrays in einer Schleife nicht korrekt geprüft wird, ist der mitgegebene Tag wie zuvor noch 0110. Da Adresse m_{32} aber dem Tag 0010 zugeordnet

Tag	Speicheradresse	Granule
0110	m_0 m_1 ... m_{15}	g_1
0110	m_{16} ... m_{31}	g_2
0010	m_{32} ... m_{47}	g_3
1001	m_{48} ... m_{63}	g_4

Tabelle 2.2: Beispiel für einen Speicher mit Memory Coloring mit einem 32 Adressen langen zusammenhängenden Speicherbereich und zwei jeweils 16 Adressen langen Speicherbereichen. Die verschiedenen Farben entsprechen jeweils gleichen Tags.

ist, die Tags also nicht übereinstimmen, bemerkt die Hardware, dass der Zugriff nicht erlaubt sein sollte. Daher verhindert sie je nach Implementierung z. B. den Zugriff oder reagiert durch eine Warnmeldung.

Um **temporale Speicherfehler** mithilfe von Memory Tagging zu verhindern, ist auch die richtige Implementierung in der Software wichtig. So muss bei der De- oder Reallokation von Speicher dem kompletten Speicherbereich ein neuer Tag zugeordnet werden. Dadurch wird ein use-after-free unterbunden, da ein Zeiger nach einer `free`-Operation nicht mehr den korrekten Tag für den Zugriff auf den deallokierten Speicherbereich angeben kann. Um ein double-free zu verhindern, kann die `free`-Funktion überprüfen, ob der Tag des gegebenen Zeigers mit dem des zu deallokierenden Speicherbereiches übereinstimmt. Das ist nur dann der Fall, wenn der Speicherbereich nicht bereits deallokiert und dadurch auch mit neuen Tags versehen wurde [20][S. 1].

Memory Coloring ist also in der Lage, mit sehr begrenzter Softwareunterstützung Speicherfehler auf Hardwareebene zu verhindern. Zu beachten ist allerdings, dass es sich hierbei um ein probabilistisches Verfahren handelt; wurde zufällig derselbe Tag für zwei Granules gewählt, ist für diese kein Schutz mehr garantiert. In Tabelle 2.3 sieht man eine mögliche Auswirkung: Bei der zufälligen Auswahl der Tags ist g_3 derselbe zugeordnet worden wie g_1 und g_2, obwohl sie nicht gemeinsam allokiert wurden. Wenn nun also ein eigentlich ungültiger Zugriff auf m_{32} mit einem dem Array in g_1 und g_2 zugeordneten Zeiger versucht

Tag	Speicheradresse	Granule
0110	m_0 m_1 ... m_{15}	g_1
0110	m_{16} ... m_{31}	g_2
0110	m_{32} ... m_{47}	g_3
1001	m_{48} ... m_{63}	g_4

Tabelle 2.3: Beispiel für einen Speicher mit Memory Coloring mit einem 32 Adressen langen zusammenhängenden Speicherbereich und zwei jeweils 16 Adressen langen Speicherbereichen. Hier ist allerdings zufällig derselbe Tag für g_1, g_2 und g_3 gewählt, obwohl die drei Bereiche nicht logisch zusammengehören.

wird, kann von Hardwareseite kein Problem festgestellt werden. Mögliche Lösungsansätze hierfür sind die Erhöhung der Taglänge, um die Wahrscheinlichkeit von zufällig gleichen Tags zu verringern, sowie eine klügere Auswahl von Tags, so dass sie nicht nur zufällig gewählt werden, sondern bspw. auch immer unterschiedlich zu benachbarten Tags sein müssen. Bei letzterer Variante ist zu bedenken, dass auf diese Weise ausschließlich räumliche Speicherfehler in der Art eines Buffer-Overflows effektiver verhindert werden, temporale Fehler bleiben gleich wahrscheinlich.

2.4 Umsetzungsansätze

Neben dem Memory Coloring ist auch die Umsetzung weiterer Sicherheitsmechanismen mithilfe von Memory Tagging möglich. In [12][S. 124:5-6] werden verschiedene Umsetzungen in fünf verschiedenen Kategorien eingeordnet:

memory safety Prozessoren dieser Kategorie setzen das oben beschriebene Memory Coloring oder ähnliche Verfahren um, sind also in erster Linie für die Verhinderung der Ausnutzung von Speicherfehlern entworfen.

information-flow control (IFC) Umsetzungen mit information-flow control sind besonders darauf bedacht, zu verhindern, dass unerlaubter Zugriff auf vertrauliche Informationen möglich ist. Dafür werden zum Beispiel Konzepte von privilegiertem und nicht-privilegiertem Zugriff durch Memory Tagging abgesichert.

dynamic information-flow tracking (DIFT) DIFT ist eine Unterkategorie von IFC, wobei ein besonderes Augenmerk auf Nutzereingaben und deren möglichem Einfluss auf die Daten und den Kontrollfluss des Programms gelegt wird.

capabilities Hier ist die Idee, Daten bestimmte *capabilities* bzw. Fähigkeiten zu erlauben, deren Einhaltung dann mithilfe von Memory Tagging überprüft und durchgesetzt wird.

programmable Einige Architekturen überlassen es der Programmiererin bzw. dem Programmierer, auf welche Weise die Tags genutzt werden. Dadurch sind sie besonders flexibel und ermöglichen die Umsetzung sehr verschiedener Aufgaben, gleichzeitig erfordern sie aber gerade deswegen auch erheblichen Konfigurations- bzw. Programmieraufwand.

Kommerziell breit verfügbar, besonders als physische Umsetzungen, die nicht auf FPGAs angewiesen sind, waren in den letzten Jahren jedoch nur *SPARC Application Data Integrity (ADI)* [4] und *ARM Memory Tagging Extension (MTE)* [6]. Beide werden in [12][S. 124:12] der Kategorie *memory safety* zugeordnet und werden in dieser Arbeit im Kapitel *Stand der Technik* genauer vorgestellt. Auf dieser Kategorie liegt hier auch der Fokus, da ihr am ehesten die zu entwickelnde Umsetzung zugeordnet werden kann.

2.5 Stärken, Schwächen und Herausforderungen

Die Vorteile, die Memory Tagging gegenüber Architekturen ohne Tags bietet, wurden in den vorherigen Absätzen deutlich. Die Metadaten, die in den Tags gespeichert werden, können für verschiedene Zwecke nutzbar gemacht werden. Mithilfe einer konzeptionell vergleichsweise einfachen Technik können verschiedene Formen von Sicherheitsmechanismen umgesetzt werden.

2.5.1 Laufzeitkosten

Gerade die zusätzlichen Laufzeitkosten sind dabei in vielen Fällen überschaubarer als bei Softwarelösungen mit ähnlichen Zielen. In [25][S. 6-7] wird bspw. der

Einfluss von SPARC ADI auf die Zunahme der Laufzeit verschiedener Benchmarks um durchschnittlich nur etwa 4 % im sogenannten *imprecise mode* gezeigt, bei dem Speicherfehler nicht direkt, sondern mit einer leichten Verzögerung gemeldet werden [25][S. 3]. Im Vergleich dazu werden die Laufzeitkosten bei einer vergleichbaren Softwarelösung (AdressSanitizer) von den Autoren derselben Veröffentlichung auf das 1,5- bis sogar 3-fache geschätzt [25][S. 2].

Der Laufzeitoverhead von Memory Tagging kann aber auch größer ausfallen, je nachdem wie viele zusätzliche Instruktionen für das Taggen an sich und die Generierung der Tags notwendig sind. Hier spielt auch die Softwareseite eine Rolle, also wie genau die Speicherallokation jeweils abläuft und mit Tags unterstützt wird. Darüber hinaus spielt die Größe der Granules eine Rolle: Je mehr Granules bei einer Speicherallokation oder -deallokation mit neuen Tags versehen werden müssen, desto aufwendiger ist die entsprechende Operation. Größere Granules führen also zu schnelleren Speicheroperationen, kleinere zu langsameren.

2.5.2 Speicherkosten

Höhere Kosten können je nach Umsetzung auf der Speicherseite entstehen. Dabei ist zuerst potenziell verlorener Platz durch **zu große Granules** zu beachten. Wählt man bspw. Granules der Größe 64 Byte, passen in diese 16 Integer der Länge 32 Bit.

$$\frac{64\text{B} \cdot 8}{32\text{bit}} = 16$$

Das bedeutet, dass z. B. bei Integer-Arrays mit weniger als 16 Elementen Speicherplatz ungenutzt bleibt. Ist das Array bspw. nur zehn Elemente groß, würde man 24 Byte an Speicher unbenutzt lassen müssen. Das sind 37,5 % des Granules.

$$64\text{B} - \frac{(10 \cdot 32\text{bit})}{8} = 24\text{B}$$

Würde man das Granule mit anderen Variablen auffüllen, gibt man gleichzeitig die Vorteile der Tags wieder auf. Dieses Problem tritt in verschiedener Intensität immer dann auf, wenn die Größe eines zusammengehörigen Speicherbereiches kein ganzzahliges Vielfaches eines Granules ist. Daraus folgt aber auch, dass kleinere Granules weniger Speicher auf diese Art ungenutzt lassen; je kleiner die einzelnen Abschnitte, desto weniger Fragmentierung kann durch diese entstehen. Speichert man etwa nur sechs Byte in einem Granule mit 64 Byte, hat man einen Überschuss von 58 Byte, oder gut 90 %. Dasselbe Szenario mit einem nur acht Byte großem Granule lässt nur zwei Byte, oder 25 % ungenutzt.

Größe eines Granules in Byte	4	8	16	32	64
benötigter Speicher in MiB	8192	4096	2048	1024	512
Verhältnis zum Hauptspeicher in %	12,5	6,25	ca. 3,1	ca. 1,5	ca. 0,7

Tabelle 2.4: Speicherverbrauch von vier Bit langen Tags für einen 64 GiB großen Speicher.

Größe eines Tags in Bit	1	2	4	6	8
benötigter Speicher in MiB	1024	2048	4096	6144	8192
Verhältnis zum Hauptspeicher in %	ca. 1,5	ca. 3,1	6,25	ca. 9,4	12,5

Tabelle 2.5: Speicherverbrauch der Tags für einen 64 GiB großen Speicher mit 16 Byte großen Granules.

Zu kleine Granules bringen allerdings an anderer Stelle Probleme. Die Zuordnung von Tags zu Granules muss selbst auch irgendwo gespeichert werden. Es gilt also: Je kleiner die Granules, desto größer der notwendige Speicher, um die Tags zu speichern. In der Tabelle 2.4 ist das für einen 64 GiB großen Hauptspeicher und vier Bit lange Tags für verschiedene Granulegrößen mithilfe folgender Formel beispielhaft vorgerechnet:

$$\frac{M}{G} \cdot T = m \tag{2.1}$$

Dabei ist M die Größe des zu taggenden Hauptspeichers, G die Größe eines Granules, T die eines Tags und m bezeichnet den benötigten Speicherplatz, um M zu taggen. Es ist also zwischen dem nötigen Speicherverbrauch der Tags für kleine Granules und dem möglichen unbenutzbaren Speicheranteil bei der Verwendung großer Granules abzuwägen.

Die Formel 2.1 zeigt, dass die auch **Größe eines Tags** den Speicherverbrauch beeinflusst: Je großer die Tags, desto höher der Speicherverbrauch. Das zeigt sich auch in den Beispielrechnungen in Tabelle 2.5. Darüber hinaus ist bei der Wahl der Taglänge im Bezug auf den Speicher ein weiterer Punkt zu beachten: Da der Tag in die oberen Bits der Adressen eingebettet wird, hängt die Anzahl der möglichen Speicheradressen direkt von der Größe der Tags ab. Tabelle 2.6 zeigt für Byte-adressierbare Architektur wie RISC-V [31][S. 24], bei denen je eine Adresse einem Byte Speicher entspricht, dass das gerade bei 32-Bit-Prozessoren schnell zu großen Einschränkungen des maximal adressierbaren Speichers führen kann. So sind bei vier Bit langen Tags nur noch 256 MiB Speicher adressierbar. Für solche Architekturen ist also nur ein sehr begrenzter Einsatz für Anwendungen mit geringem Speicherbedarf bspw. im Bereich eingebetteter Systeme denkbar.

Taglänge	Anzahl Tags	max. Speichergr. (64b)	max. Speichergr. (32b)
1 b	2	8 EiB	2 GiB
2 b	4	4 EiB	1 GiB
3 b	8	2 EiB	512 MiB
4 b	16	1 EiB	256 MiB
5 b	32	512 PiB	128 MiB
6 b	64	256 PiB	64 MiB
7 b	128	128 PiB	32 MiB
8 b	256	64 PiB	16 MiB
12 b	4096	4 PiB	1 MiB
16 b	65536	256 TiB	64 KiB

Tabelle 2.6: Einfluss der Taglänge auf die Anzahl der verschiedenen möglichen Tags und den maximal adressierbaren Speicher.

2.5.3 Hardwarekosten

Neben Speicher- und Laufzeitkosten sind auch noch die Kosten der zusätzlich benötigten physischen Hardware relevant. Das drückt sich entweder in zusätzlich genutzten FPGA-Ressourcen bei Softcores oder in größerer Chipfläche für ICs aus. Die genauen Kosten sind allerdings abhängig von der jeweiligen konkreten Implementierung der Technologie, sodass grundsätzliche Aussagen dazu nur sehr begrenzt möglich sind.

2.5.4 Software

Eine weitere Herausforderung ist die Softwareunterstützung. Das betrifft zuerst die Instruktionsebene: Die Befehlssatzarchitektur muss die Möglichkeit bieten, eine gewisse Anzahl von Bits in Adressen bei der Auflösung dieser zu ignorieren und dafür als Tag zu interpretieren. Ist das nicht möglich, muss die Architektur dahingehend erweitert werden.

Auf der Programmebene gibt es beim Memory Coloring für C und C-ähnliche Sprachen zwei Aspekte zu beachten: die Umsetzung für den Heap und die für den Stack. Bei der Speicherallokation und -deallokation auf dem Heap ist die Anpassung der C-Bibliotheksfunktionen `malloc` und `free` sowie `calloc` und `realloc` notwendig [6][S. 7]. Auf dynamisch gelinkten Systemen kann so Memory Coloring für Heap-Elemente implementiert werden, ohne dass bereits übersetzte Programme neu kompiliert werden müssen; der dynamische Linker kann einfach die angepasste C-Standardbibliothek bereitstellen [6][S. 6].

Für den Stack ist es zum Beispiel möglich, jedem Stackframe jeweils einen Tag zuzuordnen. Dafür müssen allerdings der Compiler und je nach Konfiguration auch das Betriebssystem die entsprechende Unterstützung bereitstellen. Bereits kompilierte Programme müssen neu übersetzt werden, um die Vorteile nutzen zu können [6][S. 7].

Verschiedene Typen des Memory Tagging erfordern unterschiedliche Formen der Softwareunterstützung, bspw. durch das Betriebssystem. Die Beschreibung der Anforderungen beim Memory Coloring kann hier als ein Beispiel verstanden werden.

2.5.5 Sicherheit

Auch die tatsächliche Sicherheit der Umsetzung hängt von verschiedenen Entscheidungen bei der Implementierung ab und kann daher eine Herausforderung darstellen. Ein Punkt, der die Sicherheit aller Formen des Memory Taggings beeinflusst, ist die Taglänge: Je länger die Tags, desto unwahrscheinlicher ist es, dass durch Zufall ein Fehler nicht vom Prozessor erkannt wird, weil die Tags ungewollt übereinstimmen. Zwei Bit lange Tags bieten bspw. nur vier unterschiedliche Tags; die Wahrscheinlichkeit, zufällig den richtigen zu wählen, liegt also bei 25 %. Sind die Tags vier Bit lang, liegt die Wahrscheinlichkeit noch bei 6,25 %. Demgegenüber sinkt die Wahrscheinlichkeit bei acht Bit langen Tags auf unter 0,4 %. Mit Memory Tagging können Speicherfehler also immer nur mit einer gewissen Wahrscheinlichkeit erkannt werden, wobei diese u. a. von der Menge der verfügbaren Tags und damit von der Taglänge abhängt. Hier ist also eine Abwägung zwischen Sicherheit und Speichereffizienz bzw. maximaler Speichergröße zu treffen.

Ebenfalls abhängig ist die Sicherheit von der Art, wie die Tags ausgewählt werden. In [20][S. 3-6] wurden verschiedene Softwareimplementierungen für ARM MTE (also eine Form des Memory Coloring) untersucht, die aktuell in Anwendungen verwendet werden. Darin wurde deutlich, dass die Entscheidungen, die bei der Auswahl von Tags bei der Allokation und Deallokation von Speicher getroffen werden, direkten Einfluss auf die Sicherheit eines Systems haben können. So ist zum Beispiel eine Auswahl von Tags in Abhängigkeit der umliegenden Tags sowie von dem vorherigen Tag gegenüber einem rein zufälligen Ansatz sicherer, da so auf jeden Fall garantiert werden kann, dass nebeneinanderliegende Granules von unterschiedlichen Speicherbereichen auch verschiedene Tags haben. Außerdem wird verhindert, dass bei einer Deallokation der freigegebene Speicher durch Zufall erneut mit demselben Tag wie zuvor versehen wird.

2.6 Abgrenzung

Zu unterscheiden von dem hier vorgestellten Thema sind rein **software-basierte Ansätze** wie ASAN [24], bei dem durch Anpassung des Compilers und der Verwendung eines *shadow memory* versucht wird, dieselben Speicherfehlerklassen zu verhindern, wie beim Memory Tagging. Durch die reine Umsetzung in Software entsteht ein enormer Overhead, durch den die Verwendung abseits der Entwicklung und des Testens nicht möglich ist. Daher werden solche Umsetzungen im Gegensatz zum Memory Tagging nicht zur Absicherung produktiv genutzter Software verwendet, sondern nur, um vor der Veröffentlichung Fehler zu entdecken.

In der Problemstellung dem Tagging ähnlich sind auch **Memory Protection Units** (MPUs). Diese werden genutzt, um den Speicherzugriff bei Mikrocontrollern ohne MMU abzusichern. ARM bietet bspw. optional eine MPU für Cortex-M3- und Cortex-M4-Mikrocontroller an [7][S. 951]. Sie sichert hier eine begrenzte Anzahl an Speicherbereichen ab, so dass nur berechtigte Anwendungen Zugriff auf diese haben. Im Vergleich zum Memory Tagging ist die Absicherung hier deutlich gröber, da nur eine begrenzte Anzahl verschiedener Zugriffslevel ermöglicht werden, nicht aber die Verhinderung üblicher Speicherfehler.

Wieder einen anderen Ansatz stellt **Memory Encryption** dar. Ein Beispiel dafür ist *AMD Secure Memory Encryption* (SME) [13]. Dieses soll im Gegensatz zum Memory Tagging Angriffe auf den physischen Speicher verhindern, in dem üblicherweise alle Daten in Klartext gespeichert werden. Memory Encryption nutzt Funktionen des Prozessors, um den Arbeitsspeicher zu verschlüsseln und so zu verhindern, dass sensible Daten aus diesem von außerhalb des Systems ausgelesen werden können. Mit CrypTag [17] gibt es aber auch Ansätze, beides zu kombinieren und zu zeigen, dass der Overhead von Memory Tagging in Systemen mit Speicherverschlüsselung vergleichsweise sehr gering ist.

3 Stand der Technik

Wie in [12][S. 124:12] dargestellt, gibt es eine Vielzahl verschiedener Prozessordesigns, die Memory Tagging implementieren. Nur zwei davon wurden allerdings in den letzten Jahren tatsächlich als ICs gefertigt, die anderen zielen auf den Einsatz in FPGAs oder Simulationen. In diesem Kapitel werden deshalb in erster Linie diese beiden Architekturen im Hinblick auf Memory Tagging vorgestellt. Darüber hinaus gibt es einen Überblick zu Designs, die wie der SEC-V auf der RISC-V-Befehlssatzarchitektur aufbauen.

3.1 SPARC Application Data Integrity

Mit dem SPARC M7 hat Oracle 2015 erstmals die sogenannte *Application Data Integrity* eingeführt [4][S. 40-41]. Dabei handelt es sich um eine Umsetzung des Memory Colorings [1][S. 2], nutzt also Memory Tagging zur Verbesserung der Speichersicherheit. Oracle setzt auf vier Bit lange Tags [4][S. 40] und Granules der Länge einer Cacheline [3], was hier 64 Byte entspricht [25][S. 4]. Die Tags werden in den oberen vier ungenutzten Bits der 64-Bit-Speicheradressen kodiert [4][S.40].

Die Technologie ist zwar standardmäßig für alle Threads aktiv, wenn es sich um 64-Bit-Software handelt, allerdings müssen die jeweiligen Speicherbereiche erst durch die Anwendung für ADI aktiviert werden. 32-Bit-Software wird nicht von ADI unterstützt. Darüber hinaus können auch kompatible Programme die Technik einfach ignorieren oder z. B. nur für bestimmte Threads oder Speicherbereiche verwenden [3].

In Oracles eigenem Betriebssystem Solaris wird die Aktivierung und Konfiguration von ADI über System Calls gesteuert [2]. Ist das geschehen, werden dann den Beschreibungen in [25][S. 4] und der Dokumentation des Linux-Kernels zu ADI [5] folgend die einzelnen Granules mithilfe der `stxa`-Instruktion mit ihrem jeweiligen Tag versehen. Ist dann bei einer zukünftigen Operation der angegebene Tag von dem gespeicherten verschieden, wird eine *version mismatch exception* ausgelöst, die zu einer *Trap* führt [3]. SPARC unterscheidet dabei verschiedene Formen von Traps und kann je nach Konfiguration unterschiedliche Typen

nutzen, um einen falschen Tag zu melden. Bei einer fehlschlagenden `load`-Operation kommt es immer zu einer *precise trap* [3], die geschieht, bevor sich irgendetwas an dem Zustand des gerade ausgeführten Programms ändern kann [18][S. 462]. Der Traptyp bei fehlerhaften `store`-Operationen kann konfiguriert werden. Entweder wird wie beim `load` eine *precise trap* genutzt, standardmäßig jedoch aus Gründen der Performance eine *disrupting trap* [3]. Diese muss im Gegensatz zur *precise trap* nicht sofort ausgeführt werden, sondern kann auch erst später auftreten [18][S. 447-448].

Die *version mismatch exception* wird dann über das Betriebssystem mit einem `SIGSEV`-Signal mit einigen Debugging-Informationen an den entsprechenden Prozess zurückgemeldet [3]. Die jeweilige Software hat dann die Möglichkeit, auf das Problem zu reagieren.

Eine Schwäche dieser Umsetzung ist einerseits die geringe Taglänge, bei der nur 16 verschiedene Farben möglich sind, wodurch Angriffe erleichtert werden. Andererseits führt die hohe Granulegröße dazu, dass die Einsatzgebiete beschränkt bleiben, da es nur bei großen zusammenhängenden Speicherbereichen nicht zu maßgeblicher Fragmentierung kommt. Gerade im Bereich von eingebetteten System mit begrenztem Speicherangebot ist das ein Nachteil. Gleichzeitig ist die Allokation solcher Bereiche aus demselben Grund performant möglich. Dazu passt die flexible Tagverwaltung, die eine Nutzung nur für ausgewählte Speicherbereiche oder Threads ermöglicht. Diese Entscheidungen führen auf Kosten einiger Abstriche bei der Sicherheit zu einer vergleichsweise alltagstauglichen Lösung, die eher auf die Absicherung ausgewählter Teile einer Anwendung, als auf den ganzheitlichen Gebrauch ausgelegt ist.

3.2 ARM Memory Tagging Extension

Auch ARM hat ähnlich wie Oracle eine Architektur mit Unterstützung für Memory Coloring: Seit 2019 existiert mit der Befehlssatzarchitektur Armv8.5-A auch die *Memory Tagging Extension* [6]. Im Unterschied zu SPARC ADI handelt es sich allerdings in erster Linie um eine Hardwarelösung. Die weitgehende Integration in eine C-Bibliothek und ein Betriebssystem wie bei Oracles Solaris ist in Arbeit, sowohl von ARM [6][S. 7] als auch von Entwicklern anderer Software, die bspw. ihre eigenen Speicherallokatoren nutzen.

MTE baut auf der Funktion *Top Byte Ignore* (TBI) von Armv8-A auf. Diese erlaubt es Prozessoren, das obere Byte einer Adresse bei der Auflösung derselben zu ignorieren. Der dadurch entstandene Platz wird bei MTE genutzt, um Tags zu

kodieren. Wie ADI nutzt MTE allerdings nur die oberen vier Bit als Tag, die Größe eines Granules ist dagegen mit 16 Byte deutlich kleiner [6][S. 3].

ARM fügt eine Reihe neuer Instruktionen für die verschiedenen Aufgaben im Zusammenhang mit Memory Coloring hinzu [6][S. 4-5]. Dazu gehören Instruktionen, mit denen Tags auf verschiedene Arten festgelegt (`STG`, `STZG`, `ST2G`, `STZ2G`, `STGP`) oder ausgelesen (`LDG`) werden können. Außerdem erlaubt MTE die Generierung von zufälligen oder pseudo-zufälligen Tags mithilfe der `IRG`-Instruktion, die einen so erstellten Tag in ein Register schreibt. `GMI` ermöglicht es außerdem, die Tagmenge zu verwalten, etwa um bestimmte Werte aus der normalen Generierung auszuschließen, um sie anderweitig zu verwenden. Weiterhin gibt es Instruktionen für Zeigerarithmetik, bei der kodierte Tags im oberen Teil der jeweiligen Zeiger berücksichtigt werden (`ADDG`, `SUBG`, `SUBP`, `SUBPS`). Die Instruktionen `LDGM`, `STGM` und `STGZM` sind dagegen vorgesehen, um Software auf der Systemebene Werkzeuge zur massenhaften Verwaltung von Tags zu geben, etwa bei der Initialisierung von Speicher. Auf die Instruktionen zur Cache-Verwaltung, bei der Tags berücksichtigt werden müssen, soll hier nicht weiter eingegangen werden.

Stimmen die Tags in Adresse und Speicher nicht miteinander überein, kann ein Prozessor mit MTE entweder direkt eine Exception auslösen oder das Problem asynchron melden. Die Details dieser Meldung werden in einem Systemregister gesammelt und gespeichert. Das Betriebssystem kann dieses auslesen und dann mithilfe der Informationen entscheiden, wie mit dem Problem umgegangen werden soll [6][S. 3-4]. Der Vorteil dabei ist die höhere Ausführungsperformance [6][S. 5]. Allerdings kann im Gegensatz zur Exception nicht die genaue `load`- oder `store`-Operation festgestellt werden, die das Problem ausgelöst hat, was zu einer erschwerten Fehlersuche führen kann [6][S. 3-4]. ARM empfiehlt, die verschiedenen Konfigurationen bspw. für unterschiedliche Stadien der Entwicklung bzw. der Auslieferung einer Software zu nutzen. So können in der Entwicklungs- und Testphase einer Software die genauen Exceptions bei der Fehlersuche helfen. Nach der Veröffentlichung einer Software kann dann entsprechend der jeweiligen Sicherheitsanforderung auch die performantere asynchrone Behandlung von Speicherfehlern aktiv sein [6][S. 5].

MTE ist auf der Hardwareseite ADI sehr ähnlich, weswegen auch ähnliche Vor- und Nachteile gelten. Die geringe Größe eines Granules führt jedoch zu einer besseren Eignung für Mikrocontroller. Auf der anderen Seite ist die Software noch in einer frühen Phase der Entwicklung und kann nicht auf ein eigenes Betriebssystem für die Umsetzung zurückgreifen, wie es bei SPARC und Solaris der Fall ist.

3.3 Memory Tagging für RISC-V

Aktuell sind Memory-Tagging-Instruktionen nicht Teil der RISC-V-Befehlssatzarchitektur (vgl. [31]). Es gibt allerdings einige meist akademische Projekte, die sich im Kontext von RISC-V-Prozessoren mit Memory Tagging beschäftigen. Außerdem gibt es Bestrebungen, Unterstützung für Memory Tagging offiziell in RISC-V aufzunehmen. Dazu gehört der Vorschlag für eine Pointer-Masking-Funktionalität, die ähnlich wie ARMs TBI einen ersten Schritt in Richtung Memory Tagging gehen soll. Im Unterschied zu ARMs Lösung wäre hier allerdings konfigurierbar, wie viele der oberen Bits einer Adresse bei der Auflösung ignoriert werden. Der Vorschlag zielt explizit auf die Vorbereitung von Memory Tagging ab, ist aber aktuell nur als Entwurf und Teil eines noch zu ratifizierenden J-Moduls für RISC-V ausgearbeitet. [35]

Unter den darüber hinausgehenden Arbeiten zu RISC-V ist nur bei wenigen Memory Coloring die implementierte Anwendung [12][S. 124:12]. Die folgenden drei Ansätze wurden ausgewählt, da sie sich auf sehr unterschiedliche Weise mit Memory Coloring oder Mikrocontrollern beschäftigen und damit einen beispielhaften Überblick über die aktuelle Forschung in diesem Bereich geben können. Neben den vorgestellten Ansätzen gibt es selbstverständlich noch andere, [12][S. 124:12] listet 2022 noch mindestens sechs weitere Designs auf ([8], [10], [19], [14], [26] und [28]).

3.3.1 Shakti-T

Bei Shakti-T handelt es sich um einen Prozessor mit einem Speichersicherheitsansatz, hier unter dem Begriff *fat-pointer* gefasst [15][S. 2]. Das steht für die Verwendung der oberen Bits eines Zeigers, um die Tags zu speichern. Eine Besonderheit des Ansatzes ist die Umsetzung der Tags: Statt eines klassisches Coloring-Prinzips, wie im Kapitel *Grundlagen* vorgestellt, wird hier jedem Zeiger eine ID und ein Speicherbereich als Beginn- und Endadresse zugeordnet. Letzterer legt den Gültigkeitsbereich des Zeigers fest. Das alles wird in einem separaten Teil des normalen Hauptspeichers abgelegt. Die Zeiger-ID ist so gewählt, dass mit ihr und der Startadresse dieses Hauptspeicherbereiches die Adresse der Zugriffsinformationen berechnet werden kann. Wird der Zeiger nun verwendet, um auf den Speicher zuzugreifen, kann hier also gewissermaßen nachgeschlagen werden, ob der Zugriff erlaubt sein sollte, oder nicht [15][S. 2]. Das bietet den Vorteil, weniger Speicher zu benötigen, wenn das Programm häufig große Speicherbereiche allokiert, da nur drei Speicherworte je Allokation verwendet werden (ID, Start- und Endadresse) [15][S. 7-8]. Dadurch, dass die Zeiger-ID nicht zufällig, sondern in die Berechnung von Speicheradressen eingebunden

ist, gibt es allerdings auch eine potenzielle Angriffsfläche, um das Tagging zu umgehen.

3.3.2 CrypTag

Hierbei handelt es sich um eine Prozessorerweiterung, mit der gezeigt werden soll, dass Memory Tagging auf Systemen, die bereits Memory Encryption unterstützen, nur sehr geringe zusätzliche Hardware- und Laufzeitkosten erfordert [17][S. 200]. Dafür wurde ein RISC-V-Prozessor mit Memory Encryption um ein klassisches Coloring-Verfahren erweitert, das denen von ARM und SPARC ähnelt, außerdem wurde die dazugehörige Compiler-Toolchain angepasst [17][S. 201]. Dabei wird die bereits bestehende Memory Encryption Unit genutzt, um die Mechanismen des Taggings auszuführen. Die Kosten dafür sind den eigenen Benchmarks zufolge gering; weniger als 1 % zusätzliche Hardware und ein durchschnittlicher Laufzeitoverhead von 1,5 - 6,1 % [17][S. 200, 206-208]. Aufgrund dessen implementiert CrypTag besonders lange 16-Bit-Tags, was zu höherer Sicherheit führen kann, wie im Abschnitt *Sicherheit* des Kapitels *Grundlagen* bereits dargestellt wurde. Hier zeigt sich, dass passende Sicherheitsmechanismen verknüpft werden können, ohne dass die Kosten dafür unüberschaubar werden; es ist teilweise sogar die gemeinsame Nutzung einzelner Elemente möglich.

3.3.3 TIMBER-V

TIMBER-V verfolgt keine Memory-Safety-Strategie im klassischen Sinne, sondern implementiert einen Ansatz, bei dem innerhalb eines Prozesses mithilfe von Memory Tagging feingliedrige und dynamische Daten- und Codeisolation erreicht wird, während eine MPU darüber die einzelnen Prozesse voneinander isoliert [33][S. 3]. Das Konzept richtet sich dabei explizit an kleine eingebettete Systeme [33][S. 1]. Die Arbeit nutzt den Begriff der *Enklave*, um die einzelnen isolierten Bereiche zu beschreiben, und setzt auf Betriebssystemerweiterungen, um *Trusted OS Services* zu ermöglichen, die dann diese Enklaven verwalten können [33][S. 5]. Hier wird also ein eher privilegienbasiertes Memory Tagging mit einer Prozessisolation durch eine MPU verknüpft; ein weiteres Beispiel für die Verknüpfung verschiedener Sicherheitsmechanismen. Gegenüber einem Ansatz der Hardware-Threads voneinander isoliert, ist hier u. a. mehr Betriebssystemunterstützung und die Unterscheidung verschiedener Privilegienlevel auf Hardwareseite nötig.

4 Entwurf

Die Neuerung dieser Arbeit gegenüber den vorgestellten und anderen bestehenden Ansätzen stellt die Implementierung von Memory Coloring für einen Prozessor dar, der Interleaved Multithreading (*IMT*) verwenden wird, um mehrere Threads in Hardware mit einem Prozessorkern bereitzustellen. Die Erwartung ist, eine leichtgewichtige Isolation von Threaddaten im Speicher zu realisieren, ohne dafür ein Betriebssystem zu benötigen. Auch die sonst üblichen Vorteile des Memory Colorings sollen dabei möglichst erhalten bleiben. Besonders beachtet sollen dabei die zusätzlich benötigten Hardwareressourcen und der Softwareaufwand werden. Das übergeordnete Ziel ist die Erkundung der Möglichkeiten von Memory Tagging in kleinen Mikrocontrollern mit IMT, gerade auch solchen ohne Cache.

4.1 SEC-V

Beim SEC-V handelt es sich um einen RISC-V-Mikrocontroller, der für sicherheitskritische Anwendung im kommerziellen und regulierten Umfeld eingesetzt werden soll. Er wird zukünftig als programmierbare *Dataplane Engine* in sicheren Netzwerkcontrollern, als Krypto-Coprozessor in Hardware-Security-Modulen und eingebetteten Anwendungen im KRITIS oder Secure-IoT-Bereich verwendet werden. Der Prozessor verbindet dabei wissenschaftliche Ansätze mit Funktionen, die aus kommerziellen Prozessoren bekannt sind [11]. Er befindet sich in aktiver Entwicklung, weswegen das sich Design noch regelmäßig ändert. Die Beschreibung hier bezieht sich auf den Stand im Januar 2024, auf dem auch der praktische Teil dieser Arbeit basiert. Der Prozessor unterstützt eine Daten- und Adressbreite von 64 Bit, nutzt allerdings trotzdem 32-Bit-Instruktionen, wie von der RV64-ISA vorgesehen. Die kürzeren Instruktionen führen einerseits zu einem geringeren Programmspeicherbedarf, andererseits ermöglicht es die weite Adressbreite, zusätzliche Informationen in den ungenutzten Bits in Adressen zu speichern, bspw. Tags. Der unterstützte Befehlssatz ist *RV64I*, wobei auch schon Teile der *Zicsr*-Erweiterung implementiert sind.

Der SEC-V nutzt eine vierstufige, nicht-überlappende Pipeline. Die Stufen sind das Abrufen der Instruktion aus dem Instruktionsspeicher (*fetch*), das Dekodieren der Instruktion und das Bereitstellen der Operanden, ggf. auch aus dem Register (*decode*), das Ausführen der gewünschten Operation (*execute*) und schließlich das Zurückschreiben des Ergebnisses ins Register (*write-back*). Es ist immer nur eine dieser Stufen aktiv, so dass zu einem gegebenen Zeitpunkt nur eine Instruktion verarbeitet wird. Darüber hinaus, nicht in die Pipeline eingebettet, findet die Berechnung der Adresse der nächsten Instruktion statt, hierfür existiert eine *branch decision unit*. Eine vereinfachte Übersicht über den Aufbau des Prozessors findet sich als Abbildung 4.1.

4.1.1 Fetch

In der *fetch*-Stufe wird mithilfe des *program counters* (*PC*) die nächste Instruktion aus dem Speicher geladen. Besonders beim SEC-V ist hier, dass es sich dabei um einen vom eigentlichen Arbeitsspeicher unabhängigen Instruktionsspeicher handelt. Dieser ist außerdem nur lesbar; ein einmal geladenes Programm kann nicht durch den Prozessor selbst verändert werden. Instruktions- und Datenspeicher werden unabhängig voneinander adressiert, die *fetch*-Stufe kann also nie vom Datenspeicher lesen, da sie keine Verbindung zu diesem hat. Beide Speicher werden über je einen *Wishbone*-Bus angesteuert, verfügen also über ein einheitliches Interface.

4.1.2 Decode

Die *decode*-Stufe ist selbst zweistufig. Zuerst wird der *opcode* bestimmt, und damit entschieden, welche Operanden verwendet werden (z. B. *immediates* oder Registerinhalte). Außerdem wird festgelegt, welche Funktionseinheit in der *execute*-Stufe für die Verarbeitung zuständig sein wird. Darauf folgt die zweite Stufe, in der je nach Funktionseinheit vom *opcode* und den *funct3*- und ggf. auch *funct7*-Teilen der Instruktion ausgehend die tatsächlich auszuführende Operation bestimmt wird. Eine Ausnahme ist aktuell noch die CSR-Funktionseinheit, bei der diese Stufe erst in der *execute*-Phase passiert.

4.1.3 Execute

In dieser Stufe findet die eigentliche Ausführung der jeweiligen Operation statt. Zu diesem Zweck stehen drei Funktionseinheiten zur Verfügung, die für die

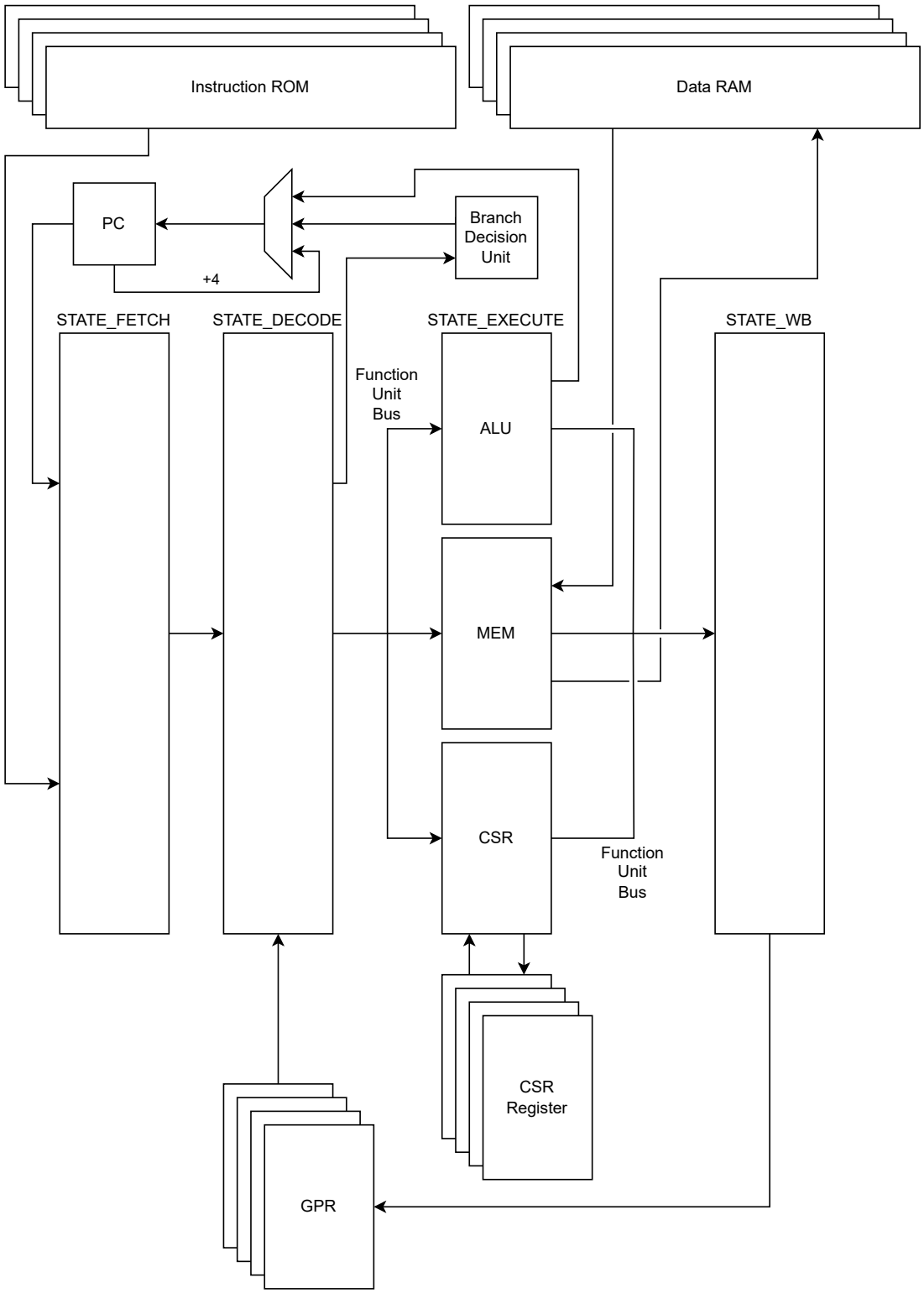

Abbildung 4.1: Übersicht zum Pipeline-Aufbau des SEC-V, inklusive der zugehörigen Steuer- und Datensignale

Typ	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	funct7							rs2					rs1					funct3			rd					opcode						
I	imm[11:0]												rs1					funct3			rd					opcode						
S	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
B	imm[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				imm[11]	opcode						
U	imm[31:12]																				rd					opcode						
J	imm[20]	imm[10:1]										imm[11]	imm[19:12]								rd					opcode						

Tabelle 4.1: Übersicht über RISC-V-Instruktionsformate [21][S. 15]

verschiedenen möglichen Operationen zuständig sind und je nach Ergebnis der *decode*-Stufe ausgewählt werden. Es ist immer nur maximal eine Funktionseinheit zu einem gegebenen Zeitpunkt aktiv, die jeweils anderen werden nicht verwendet. Die notwendigen Steuer- und Datensignale für die Einheiten werden von einem *function unit input bus* übermittelt, der die von der *decode*-Stufe angegebene Einheit aktiviert und mit der auszuführenden Operation und den beiden Operanden versorgt. Auf der anderen Seite werden auch die Ergebnisse und eventuelle Fehler von einem vergleichbaren *function unit output bus* abgerufen, wenn die Funktionseinheit ihre Aufgaben abgearbeitet hat, und an die nächste Stufe weitergegeben.

Die drei Einheiten sind ALU, Speichereinheit und CSR-Einheit. Die **ALU** führt ihrem Namen entsprechend die arithmetisch-logischen Funktionen aus, die der Prozessor unterstützt. In der **Speichereinheit** werden die Operationen ausgeführt, die auf den Datenspeicher lesend oder schreibend zugreifen. Wie schon im Abschnitt über die *fetch*-Phase ausgeführt, verwendet der SEC-V eine Architektur mit geteilten Speichern. So kann die Speichereinheit ausschließlich auf den Datenspeicher zugreifen, zum Instruktionsspeicher besteht keine direkte Anbindung. Es handelt sich also um eine echte Harvard-Architektur. Die **CSR-Einheit** ermöglicht den Zugriff auf die *control and status registers* (*CSR*s). In diesen sind verschiedene Informationen und Einstellungen zum Zustand des Prozessors gespeichert, etwa der aktuell aktive Hardware-Thread, aber auch Informationen zu Interrupts bzw. Traps. Diese Einheit ist bisher nur in Ansätzen entwickelt; in Zukunft sollen hier etwa Exceptions behandelt oder die Privilegienstufe des aktuellen Threads überprüft und festgelegt werden können. Aktuell unterstützt der Prozessor aber nur ein Privilegienlevel (*machine mode*) und einen Thread. Exceptions werden bereits korrekt übertragen, allerdings noch nicht behandelt.

4.1.4 Write-back

Die letzte Stufe der Pipeline nimmt das Ergebnis der *execute*-Phase und schreibt es in das Register, das in der *decode*-Stufe als Zielregister festgelegt wurde. Damit

ist die Instruktion vollständig abgearbeitet und die Bearbeitung der nächsten kann beginnen.

4.1.5 Interleaved Multithreading

Wie die Behandlung von Exceptions ist auch das *interleaved multithreading* (*IMT*) noch nicht im SEC-V implementiert. In der Literatur ist es auch als *fine-grained multithreading* bekannt. Hierbei handelt es sich um ein hardwareseitiges Multithreading-Verfahren, bei dem die Pipelinestruktur des Prozessors ausgenutzt wird, um mehrere Hardware-Threads überlappend auszuführen. Jede Pipelinestufe bearbeitet dann zu jedem Zeitpunkt je einen eigenen Thread [22][S. 534-535]. Tabelle 4.2 zeigt ein Beispiel dazu, in dem T_iI_j die Instruktion j des Threads mit der ID i bezeichnet. Es zeigt sich, dass je Thread jede Instruktion alle Stufen durchläuft, bis die Arbeit an der nächsten begonnen wird. Die Ausführungszeit eines einzelnen Threads wird also nicht verkürzt, dafür aber die Abarbeitung mehrerer Threads zur selben Zeit ermöglicht.

Im Gegensatz zu anderen Formen der überlappenden Pipelineausführung kann es hier nicht zu Problemen mit falschen Annahmen über Programmsprünge (also *control hazards* [22][S. 291-294]) kommen, da gar keine Annahmen getroffen werden müssen. Zum Ausführungszeitpunkt der nächsten Instruktion eines Threads ist die vorherige völlig abgearbeitet, weswegen auch die definitive Adresse der nächsten bekannt ist. Dadurch ist keine zusätzliche Hardware zur Sprungvorhersage o. Ä. nötig, weswegen sich dieser Ansatz besonders für einfache Prozessoren eignet, die parallelisierbare Aufgaben zu berechnen haben.

Obwohl diese Funktionalität geplant und im Design auch an einigen Stellen schon mitgedacht ist, kam es nicht mehr zur rechtzeitigen Implementierung der Technik, um in dieser Arbeit angewendet zu werden. Die Module für das Memory Tagging sind aber so entworfen, dass sie bereits eine frei definierbare Anzahl von Hardware-Threads unterstützen, beim SEC-V gibt es allerdings bislang nur einen.

Zukünftig soll dieser eine neue Art des IMT nutzen: *Scalable Interleaved Multithreading* (*SIMT*). Dieses Verfahren unterstützt eine beliebig konfigurierbare Anzahl von Hardware-Threads und ist darin nur durch die Größe des FPGAs bzw. die Chipressourcen begrenzt. Das Verfahren ist insbesondere für Echtzeit-Anwendungen wie die Netzwerkdatenverarbeitung gedacht.

	Ausführungsphase			
Takt	Fetch	Decode	Execute	Write-Back
1	T_0I_0	-	-	-
2	T_1I_0	T_0I_0	-	-
3	T_2I_0	T_1I_0	T_0I_0	-
4	T_3I_0	T_2I_0	T_1I_0	T_0I_0
6	T_0I_1	T_3I_0	T_2I_0	T_1I_0
7	T_1I_1	T_0I_1	T_3I_0	T_2I_0
8	T_2I_1	T_1I_1	T_0I_1	T_3I_0
9	T_3I_1	T_2I_1	T_1I_1	T_0I_1
10	T_0I_2	T_3I_1	T_2I_1	T_1I_1
...				

Tabelle 4.2: Beispielhafte Instruktionsabarbeitung mit Interleaved Multithreading

4.2 Aufbau

Um Memory Tagging zu realisieren, muss der Prozessor um folgende Funktionen erweitert werden:

- Zuordnung von Tags zu Speicheradressen
- Erkennung von abweichenden Tags beim Zugriff auf Speicheradressen

Aus diesen beiden Anforderungen folgen dann verschiedene Konsequenzen. Die Zuordnung erfordert es, dass die Nutzerin bzw. der Nutzer Verknüpfungen von Adressen und Tags vornehmen und ändern kann. Dafür müssen neue Instruktionen geschaffen werden, die vom Prozessor dann dekodiert und bearbeitet werden können. Die Tagerkennung wiederum muss bei jedem Zugriff auf den Speicher laufen und dann entsprechend reagieren. Daraus ergeben sich zwei verschiedene Aufgabenfelder, die in diesem Entwurf zwei verschiedene Einheiten übernehmen. Beide werden hier kurz in ihren Aufgaben und ihrem grundlegenden Aufbau beschrieben. Der Rest des Kapitels geht dann genauer auf ausgewählte Designentscheidungen ein.

4.2.1 Tagging Unit

Die Tagging Unit ist für die Zuordnung der Tags zu den Speicheradressen zuständig. Sie agiert als zusätzliche Funktionseinheit und bearbeitet die Aufgaben, die von den neuen Instruktionen benötigt werden. Hierfür muss auch der De-

coder im SEC-V erweitert werden, um die zusätzlichen Instruktionen richtig zu entschlüsseln. In der Tagging Unit selbst wird der zuzuordnende Tag je nach gewünschter Operation dekodiert, die der Hauptspeicheradresse zugehörige Adresse des Tagspeichers berechnet und schließlich der Tag an dieser berechneten Adresse dort gespeichert. Die Einheit vermittelt also jede direkte Interaktion der Software mit dem Tagspeicher.

4.2.2 Tag Checking Unit

Die Tag Checking Unit erweitert die Memory-Funktionseinheit und arbeitet parallel zu jedem Speicherzugriff, der darüber läuft. Sie berechnet die der aktuellen Speicheradresse zugehörige Adresse im Tagspeicher und liest den dort liegenden Tag aus. Außerdem dekodiert sie den in der Speicheradresse eingebetteten Tag und vergleicht diesen dann anschließend mit dem ausgelesenen. Dazu kommt die Überprüfung der Rechte des zugreifenden Hardware-Thread anhand des entsprechenden Taganteils. Sind die Tags nicht gleich oder ist der Thread nicht berechtigt, löst die Einheit eine Exception aus. Falls sie übereinstimmen und der Zugriff erlaubt ist, läuft das Programm regulär weiter. Für die Software ist diese Einheit bis auf die Kodierung der Tags im Kopf der Speicheradressen transparent und fällt nur im Falle von Fehlern in der Software auf.

4.3 Interleaved Multithreading und Memory Tagging

Besonders zu bedenken beim Entwurf der Erweiterung ist die Auswirkung der Existenz mehrerer Threads auf das Memory Tagging. Zuerst ist dabei zu klären, wie diese Technik dabei helfen kann, Threads zu isolieren, gleichzeitig aber auch das Teilen von Daten zwischen Threads erlaubt.

In RISC-V werden Hardware-Threads *Harts* genannt [31][S. 3]. Diese sind nummeriert und es gibt immer einen Hart, der die *ID* 0 hat. Eine Programmiererin oder ein Programmierer kann die jeweilige ID mit einer Anfrage an die CSRs (*mhartid*) abfragen [32][S. 20].

Der grundsätzliche Ansatz zur Verknüpfung von IMT mit Memory Tagging ist es, den Tag in mehrere Bereiche aufzuteilen: einmal die klassische *Color* oder Farbe im Sinne des Memory Coloring und dazu einen Bereich, der die Zugriffsrechte von Threads regelt. Hat man hier dann bspw. eine 14-Bit-Farbe und zwei

Bits, in denen die Rechte kodiert sind, käme man auf 16 Bit lange Tags. Beim Zugriff auf den Speicher muss dann aber nur die Farbe in der entsprechenden Speicheradresse kodiert werden, da die Threadberechtigung intern über die Hart-ID kontrolliert werden kann, ohne dass dafür Informationen von der Anwendung notwendig sind. Dadurch beeinträchtigt der Rechteanteil des Tags den maximal möglichen Adressraum nicht, da er nur im Tagspeicher vorhanden sein muss.

Die genaue Ausgestaltung der Kodierung der Zugriffsrechte ist wiederum auf verschiedene Arten möglich. Der einfachste Ansatz ist es, die Hart-ID des berechtigten Threads als Zahl zu kodieren und dann bei einem Zugriff direkt zu vergleichen. Die höchstmögliche Hart-ID bestimmt dabei die notwendige Länge des Taganteils: Gibt es nur drei Threads, reichen zwei Bit, gibt es sechs, sind drei Bit notwendig. Um Daten zwischen Harts austauschen zu können, muss es auch die Möglichkeit geben, mehreren oder allen Threads den Zugriff zu erlauben. Dafür kann dann z. B. die mit der gewählten Länge des Taganteils höchstmögliche Zahl verwendet werden, die dann allen Threads den Zugriff erlaubt. Bei vier Threads sind dann allerdings schon drei Bit notwendig, da fünf Varianten kodiert werden müssen: jeweils die vier Harts und alle gemeinsam. Insgesamt ist die benötigte Länge dann nicht nur von der höchsten Hart-ID abhängig, sondern auch von der Anzahl der zusätzlichen Sonderfälle, im Beispiel die Zugriffserlaubnis für alle Threads. Trotzdem ist der Ansatz noch vergleichsweise speichereffizient, da nur wenige Bits benötigt werden, um den Zugriff zu regeln. Dadurch wird weniger Platz im Tagspeicher verwendet.

Andererseits ist der Ansatz unflexibel: Nur einer oder alle Threads dürfen auf einen Speicherbereich zugreifen. Einigen, aber nicht allen Harts den Zugriff zu erlauben, ist nicht möglich. Auch ein (vorübergehendes) Zugriffsverbot für alle Threads ist hier nicht vorgesehen. Darüber hinaus ist die Implementierung trotz der konzeptionellen Einfachheit komplizierter als nötig, da der allgemein erlaubte Zugriff einen unabhängigen Sonderfall darstellt. Sollte man noch andere Sonderfälle implementieren wollen, z. B. die eben genannten, wird es schnell unübersichtlich. Alternativ wäre auch die Umsetzung der Fallbehandlung in Software möglich, was die Hardware auf Kosten der Laufzeit vereinfachen würde.

Aufgrund dieser Beschränkungen wird in dieser Arbeit ein anderer Ansatz verfolgt: Der Taganteil für die Berechtigungen ist ein Bitvektor, der genau so viele Stellen besitzt, wie der Prozessor Harts. Jede Stelle i entspricht dabei einem »Schalter« für das Recht auf den Zugriff des entsprechenden Harts mit der ID i auf den zugehörigen Speicherbereich. Würde auf einen Bereich etwa nur von dem Hart mit der ID 2 zugegriffen werden dürfen und es insgesamt vier Threads geben, lautete der Taganteil 0100. Wären Harts 0 und 1 berechtigt, wäre der Bit-

vektor entsprechend 0011. Allen oder keinem Thread Zugriffe zu erlauben, ist auf diese Weise kein Sonderfall, sondern eine ganz reguläre Zuordnung mit 1111 bzw. 0000. Dadurch wird auch die Überprüfung der Rechte noch einfacher und damit auch weniger fehleranfällig als im vorherigen Ansatz: Ob ein Thread mit der ID i berechtigt ist, lässt sich so einfach dadurch überprüfen, ob im Bitvektor an der entsprechenden Stelle i eine 1 steht. Die restlichen Stellen des Vektors sind dann irrelevant. Der Nachteil dieser Strategie ist, dass die Tags im Tagspeicher entweder länger sein müssen als bei dem ersten Ansatz, oder aber die Länge des Farbanteils verringert werden muss. Auch dieser Ansatz hat auf den maximal möglichen Adressraum des Prozessors keine zusätzlichen Auswirkungen.

4.4 Tagspeicher

Die Umsetzung des Tagspeichers ist eine der zentralen Problemstellungen im Zusammenhang mit Memory Tagging und es gibt sehr unterschiedliche Konzepte dazu. Da der Fokus dieser Arbeit besonders auf der Verknüpfung von Memory Coloring und Multithreading liegt, soll die Umsetzung des Speichers simpel gehalten sein, auch auf Kosten von Effizienz in Laufzeit oder Ressourcennutzung. Einfache Implementierungen bieten darüber hinaus durch den geringeren Codeumfang auch den Vorteil, dass die mögliche Angriffsfläche klein bleibt. Außerdem sind sie für Außenstehende, bspw. externe Prüfstellen, leichter nachvollziehbar, was sich positiv auf Kosten und Aufwand auswirkt.

4.4.1 Speicherort

Eine wesentliche Frage ist, wo genau die Tags gespeichert werden sollen. Dabei ist zu entscheiden, ob sie im normalen Arbeitsspeicher abgelegt werden oder in einem eigenen, zusätzlichen Speicher. Der Vorteil der Verwendung des Arbeitsspeichers ist, dass die Speicherarchitektur des Systems nicht verändert werden muss. Andererseits kann der so verwendete Bereich nicht mehr von der Software genutzt werden, es ist also potenziell insgesamt weniger Speicher verfügbar. Außerdem wird die Adressierung erschwert, da der Tagbereich logisch klar von dem restlichen Speicher abgegrenzt werden muss, um zu verhindern, dass die Tags versehentlich von gewöhnlichen Operationen überschrieben werden können. Ist der gleichzeitige Zugriff auf Tags und andere Daten notwendig, kann auch ein zusätzliches Speicherinterface nötig sein. Da die Tags im Allgemeinen kürzer sind als die Datenworte im Speicher, kann es außerdem zu brachliegenden Ressourcen kommen, etwa, wenn je adressierbarem Byte nur ein vier Bit

langer Tag gespeichert ist. Um das zu verhindern, wäre es entweder notwendig, Tags in der Länge eines Vielfachen der kleinsten adressierbaren Einheit des Speichers zu verwenden, oder die Art und Weise, wie ein Tag aus dem Speicher geladen wird, komplizierter zu gestalten.

Eine Möglichkeit, die auch dieses Problem löst, ist es, einen separaten Speicher für die Tags zu verwenden. So kann die Wortlänge auf exakt die Größe beschränkt sein, die ein einzelner Tag benötigt, wodurch der Speicher optimal ausgenutzt werden kann. Ein separater Speicher ist darüber hinaus physisch und logisch von dem Restspeicher entkoppelt und kann mit dem richtigen Design nicht durch normale Speicheroperationen versehentlich adressiert werden. Aus einer Sicherheitsperspektive ist das ein großer Vorteil, da Angriffe auf die Tags so deutlich erschwert werden. Außerdem ist der gleichzeitige Zugriff auf Daten und Tags ohne Probleme möglich, da der zusätzliche Speicher ohnehin ein eigenes Interface benötigt. Schließlich ist es möglich, den Speicher auf eine Weise zu entwerfen, dass möglichst wenig Ressourcen verwendet werden und nur Funktionen implementiert sind, die auch tatsächlich benötigt werden. Dieser Ansatz ist jedoch ein tieferer Eingriff in die Architektur des Systems, besonders dann, wenn der Speicher separate Hardware ist. Dazu kommt potenziell zusätzlicher Aufwand durch die Anbindung des neuen Speichers. Für diese Arbeit ist dieser Ansatz aufgrund seiner deutlichen Vorteile und der flexiblen Architektur des SEC-V der passendere.

4.4.2 Effizienz

Ein naiver Speicheransatz legt je Granule einfach einen Tag ab, wobei es eine direkte Übersetzung der Haupt- in die Tagspeicheradresse des entsprechenden Granules gibt. Dadurch sind Tagspeicherzugriffe mindestens vergleichbar schnell mit Hauptspeicherzugriffen, was die Handhabung deutlich erleichtert. Auch die Einfachheit der Adressübersetzung bietet Vorteile, bspw. für das Debugging von Designfehlern. Aufgrund dieser Vorteile nutzt die Implementierung genau so einen naiven Speicher, auch weil das Augenmerk der Arbeit in erster Linie auf der Sicherheit und weniger auf der Effizienz liegt.

Dieser Ansatz ist allerdings vergleichsweise ressourcenintensiv. Denkt man an typische Anwendungen, in denen große Speicherbereiche mit denselben Tags gekennzeichnet werden würden, müssen große Bereiche des Speichers mit demselben Tag gefüllt werden. Hier deutet sich Einsparpotenzial an. So gibt es bspw. Arbeiten, in denen der Tagspeicher selbst lauflängenkodiert ist [20]. In der zitierten Arbeit besteht ein gespeicherter Tag immer aus dem Tag selbst, sowie der Startadresse und der Anzahl der dazugehörigen Granules [20][S. 2]. Da-

durch wird allerdings die Überprüfung der Tags komplizierter als die einfache Umwandlung der Adresse. Da der Speicher nach der zu überprüfenden oder auch zu taggenden Adresse durchsucht werden muss, wird zusätzlich ein *BTree* gespeichert, der eine effiziente Suche nach dem korrekten Tag ermöglichen soll [20][S. 7]. Neben diesem Ansatz sind auch andere Speicherentwürfe möglich, in denen die direkte Verknüpfung von Speicheradressen und Tagspeicheradressen aufgehoben wird, bspw. gibt es neben BTrees andere Strukturen, z. B. Tries oder Hashmaps, die ebenfalls eine effiziente Durchsuchbarkeit gewährleisten könnten.

Das gemeinsame Problem der weniger ressourcenintensiven Designs ist die im Vergleich zum einfachen Speicher gesunkene Performance. Je nach Implementierung sollte bei der simplen Umsetzung der Zugriff auf den Tagspeicher maximal genauso lange dauern wie derjenige auf den Restspeicher. Es entstehen also durch den Zugriff selbst keine Performanceeinbußen, diese können erst durch den Vergleich im Nachhinein und auf höherer Ebene durch zusätzlich benötigte Instruktionen auftreten. Bei den anderen Ansätzen ist eine vorherige Adressübersetzung oder -findung notwendig, die den Zugriff an sich hinauszögert und damit auch länger dauern kann als der Zugriff auf den Restspeicher.

Dieses Problem ließe sich mit der Implementierung eines Tagcaches oder der Erweiterung eines eventuell ohnehin vorhandenen Caches des Prozessors abschwächen. Die Annahme liegt nahe, dass bspw. kürzlich verwendete Speicherbereiche und deren Nachbarn relativ bald wieder angefragt werden. Denkbar wäre also die Einrichtung eines Tagcaches, in dem kürzlich aufgefundene Tags bereitstehen. Dadurch wäre der nächste Zugriff auf diese deutlich verkürzt im Gegensatz zu einem Suchverfahren. Ein Nachteil ist, dass durch das Nachschlagen im Cache der Zugriff im Falle eines Cache-Misses noch einmal verlängert wird. Darüber hinaus benötigt so eine Cache-Infrastruktur selbst wieder nicht zu vernachlässigende Ressourcen.

Analog wäre auch die Erweiterung eines bestehenden Caches möglich. Hier könnten die bereits vorhandenen Cachelines zum Beispiel einfach um die Länge der Tags vergrößert werden. Dadurch lägen beim Zugriff auf Speicherbereiche im Cache die Tags direkt mit vor, es wäre kein Zugriff auf den eigentlichen Tagspeicher mehr nötig und der Vergleich könnte direkt stattfinden. Die zusätzlich benötigten Ressourcen würden sich hier in Grenzen halten, da die bestehende Infrastruktur mitgenutzt werden kann. Auch die Kosten im Falle eines Cache-Misses sind weniger gravierend, da diese beim Restspeicher genauso auftreten, also nicht neu dazukommen.

4.5 Memory Tagging im Daten- und im Instruktionsspeicher

Die bereits oben beschriebene besondere Speicherarchitektur des SEC-V mit nur lesbarem Instruktions- und davon getrenntem Datenspeicher beeinflusst auch die Implementierung des Memory Taggings. Durch diese Aufteilung ist das Tagging des Instruktionsspeichers weniger wichtig, da aus dem Datenspeicher heraus keine Instruktionen überschrieben werden können. Denkbar ist eine Manipulation einer Sprungadresse, um in den Instruktionsbereich eines anderen Threads zu gelangen. Zugriff auf die zugehörigen Daten hätte man dadurch allerdings trotzdem bei einem mit korrekten Tags versehenen Datenspeicher nicht, da die Hart-ID bei einem entsprechenden Zugriff nicht übereinstimmen würde. Die hier entwickelte Memory-Tagging-Erweiterung sichert daher nur den Datenspeicher ab, nicht aber den Instruktionsspeicher.

4.6 Taggenerierung und Instruktionen

Die letzte hier zu besprechende Frage ist die nach der Herkunft der Tags. Dazu gibt es verschiedene Ansätze: So kann man die Auswahl bspw. ausschließlich der Software überlassen oder aber die Hardware generiert die Tags selbst. In beiden Fällen kann man verschiedene Strategien nutzen, um passende Tags auszuwählen, etwa fortlaufend oder zufällig. Auch komplexere Verfahren sind möglich, zum Beispiel in dem man benachbarte Tags mit einbezieht und so garantiert, dass nicht versehentlich zwei nebeneinander liegende Speichersegmente ungewollt denselben Tag erhalten.

Die hier zu entwickelnde Erweiterung soll möglichst viele dieser Strategien ermöglichen, ohne dass die Benutzung unnötig aufwendig wird. Dafür wird es einen Pseudo-Zufallsgenerator in Hardware geben, mit dem Tags generiert werden können, aber auch die Möglichkeit, die Software selbst die Tags bestimmen zu lassen und dadurch andere Verfahren zu unterstützen. Dabei können beide Möglichkeiten im selben Programm gemischt eingesetzt werden. Da in dieser Arbeit Tags nicht nur aus der Farbe bestehen, sondern auch aus einem Berechtigungsanteil, muss es über die Erzeugung der Farbe hinaus auch möglich sein, dem Prozessor den weiter oben beschriebenen Bitvektor zu diesem Zweck mitzuteilen.

Die Umsetzung erfolgt durch drei neue RISC-V-Instruktionen, zwei für die Festlegung von Tags durch die Nutzerin oder den Nutzer und eine für die zufällige Erzeugung durch die Hardware. Alle neuen Instruktionen nutzen den *custom-0*

Basis-Opcode (0001011), der vom RISC-V-Befehlssatz explizit frei gehalten wird, um eigene Anpassungen zu ermöglichen und wird auch in zukünftigen Versionen der Spezifikation nicht anderweitig verwendet werden [31][S. 129]. Sie verwenden dabei den Instruktionstyp R (siehe auch Tabelle 4.1).

4.6.1 `tadr`

Bit	31-25	24-20	19-15	14-12	11-7	6-0
Feld	funct7	rs2	rs1	funct3	rd	opcode
Nutzung	ungenutzt	Farbe und Bitvektor	Adresse	000	Adresse mit Farbe	0001011

Tabelle 4.3: Instruktionsaufbau von `tadr`

Mit `tadr` (***t**ag **a**ddr**e**ss*) hat die Programmiererin oder der Programmierer die Möglichkeit, eine bestimmte Adresse, die im Register `rs1` steht, mit einem vollständigen Tag zu versehen. Dieser besteht aus einer Farbe und einem Bitvektor und muss im Register `rs2` gespeichert sein. Das Feld für das Zielregister (`rd`) wird für die Rückgabe der Adresse mit eingebetteter Farbe genutzt. Diese kann dann von der Anwendung direkt für den zukünftigen Zugriff verwendet werden. Das Feld `funct3` identifiziert die Instruktion als `tadr`.

4.6.2 `tadre`

Bit	31-25	24-20	19-15	14-12	11-7	6-0
Feld	funct7	rs2	rs1	funct3	rd	opcode
Nutzung	ungenutzt	Bitvektor	Adresse mit Farbe	001	Adresse mit Farbe	0001011

Tabelle 4.4: Instruktionsaufbau von `tadre`

Im Gegensatz zur vorhergehenden Instruktion wird bei `tadre` (***t**ag **a**ddress with **e**ncoded tag*) in `rs2` nur der Berechtigungsbitvektor erwartet. Dafür soll in `rs1` nicht nur die Adresse stehen, sondern in den oberen, ungenutzten Bits auch die Farbe kodiert sein. Der Vorteil hier ist, dass die so aufgebaute Adresse dieselbe ist, die später auch für den Zugriff benötigt wird. Wie zuvor wird der Einheitlichkeit wegen die Adresse mit eingebetteter Farbe in `rd` abgelegt, während `funct3` den genauen Operationstypen festlegt.

Bit	31-25	24-20	19-15	14-12	11-7	6-0
Feld	funct7	rs2	rs1	funct3	rd	opcode
Nutzung	ungenutzt	Bitvektor	Adresse	010	Adresse mit Farbe	0001011

Tabelle 4.5: Instruktionsaufbau von `tadrr`

4.6.3 `tadrr`

Diese Instruktion (***t****ag* ***ad****dress with* ***r****andomly generated tag*) wird verwendet, um den Prozessor anzuweisen, selbst zufällig eine Farbe zu generieren. Die erzeugte Farbe wird analog zu den vorherigen Instruktionen in der Adresse kodiert und so ins Register `rd` geschrieben. Auch sonst ähnelt die Instruktion im Aufbau den beiden anderen, jedoch wird in `rs2` nur der Bitvektor erwartet, in `rs1` nur die Adresse. Hier wie bei den anderen bleibt `funct7` ungenutzt, da zur Identifikation der Instruktion `funct3` ausreicht.

5 Implementierung

Dank des modularen Designs des SEC-V war es möglich, die Implementierung der Erweiterung weitgehend in eigenen, gekapselten Einheiten vorzunehmen. Zu beachten waren dabei vor allem die korrekten Interfaces und die Kompatibilität mit dem grundsätzlichen Aufbau des Prozessors, um sie an der richtigen Stelle anbinden zu können. Erst bei der tatsächlichen Integration der neuen Komponenten mussten bestehende Elemente des SEC-V leicht angepasst werden.

5.1 Konfiguration

Alle neuen Module bieten die Möglichkeit verschiedene Aspekte des Memory Taggings zu konfigurieren. Das ist gerade im zukünftigen Zusammenspiel mit dem konfigurierbaren SIMT eine erste Neuerung gegenüber bestehenden Ansätzen. Einstellbar sind die Anzahl der zu unterstützenden Hardware-Threads als Parameter `HARTS`, außerdem die Länge der Tags (`TLEN`) in Bits und die Granularität des Taggings (`GRANULARITY`). Diese ist als Anzahl der Stellen angegeben, um welche die eigentliche Speicheradresse nach rechts verschoben werden soll, um die zugehörige Tagspeicheradresse zu erhalten. Bei dem hier vorliegenden Byte-addressierbaren Speicher folgt daraus, dass eine Granularität von n Stellen 2^n Byte großen Granules entspricht. Mithilfe der tatsächlichen Adressbreite und der Granularität lässt sich durch einfache Subtraktion letzterer von ersterer auch die maximal notwendige Adressbreite des Tagspeichers berechnen. Darüber hinaus ergibt sich aus der Taglänge abzüglich der Threadzahl die Länge des Memory-Color-Anteils.

5.2 Tagging Unit

Dieses Modul (zu finden im digitalen Anhang in der Datei `hdl/mtag.sv`) dient dazu, Tags Adressen zuzuordnen. Dazu ist es als zusätzliche Funktionseinheit des Prozessors aufgebaut und ergänzt damit die im Kapitel *Entwurf* beschriebe-

nen Einheiten ALU, Speicher und CSR. Wie diese ist auch die Tagging Unit über denselben *function unit bus* angebunden; sowohl für die Eingangs-, als auch die Ausgangssignale. Darüber hinaus gibt es eine Anbindung an den Tagspeicher und einen Eingang für den Zufallszahlengenerator.

Die Logik der Einheit erlaubt es – analog zu den ebenfalls im *Entwurf* beschriebenen Instruktionen – den zu speichernden Tag auf drei verschiedene Weisen zu konstruieren: Einmal wird der gesamte Tag, bestehend aus Farbe und Bitmaske, aus `src2` des *function unit bus* gelesen. Dort liegt in diesem Fall der Inhalt aus Register `rs2` an. Außerdem wird ein Tag aus den oberen Bits der Adresse, die in `src1` des *function unit bus* erwartet wird, und dem Bitvektor, der in `src2` anliegen soll, zusammengesetzt. Die letzte Version des Tags wird aus demselben Bitvektor erzeugt, hier aber mit einer Farbe kombiniert, die aus dem vom Zufallszahlengenerator anliegenden Wert generiert wird.

Wird die Einheit nun über den *function unit bus* aktiviert, erhält sie von diesem auch die auszuführende Operation (analog zu den Instruktionen `tadr`, `tadre` und `tadrr`). Dieser entsprechend wird der korrekte Tag aus den drei genannten ausgewählt und im Tagspeicher abgelegt. Die Adresse, die dafür notwendig ist, wird berechnet, indem die angegebene Speicheradresse um die vorher als `GRANULARITY` konfigurierte Anzahl an Stellen nach rechts verschoben wird. Darüber hinaus wird die Adresse mit eingebetteten Farbanteil des Tags dem *function unit bus* als Ergebnis (`res`) übergeben.

5.3 Tag Checking Unit

Die Tag Checking Unit (in `hdl/mtag_chk.sv`) übernimmt die Aufgabe, die gespeicherten Tags zu nutzen, um zu überprüfen, ob auf eine Speicheradresse zugegriffen werden darf oder nicht. Dafür müssen eine Adresse, bei der die Farbe in den oberen Bits kodiert ist, sowie die ID des zugreifenden Harts angelegt werden. Anhand der Adresse berechnet sie, wie auch die Tagging Unit, die zugehörige Tagspeicheradresse und liest deren Inhalt. Im Falle eines korrekten Zugriffs, stimmt der erste Teil des Tags mit der Farbe aus der Adresse überein. Im Bitvektoranteil des Tags ist das der Hart-ID entsprechende Bit 1.

Ist eine dieser beiden Bedingungen nicht erfüllt, wird entsprechend entweder ein `color_mismatch` oder ein `hart_mismatch` gemeldet, wobei letzteres Vorrang vor ersterem hat. Sind beide Bedingungen erfüllt, gibt es keinen weiteren Handlungsbedarf.

5.4 Tagspeicher

Der Tagspeicher (in `hdl/mtag_mem.sv`) hält alle Tags. Die Wortlänge entspricht dabei der Länge eines Tags, die Adressbreite ist wie oben beschrieben abhängig von der realen Adressbreite des eigentlichen Datenspeichers sowie der gewünschten Granularität. Es existiert ein Interface, mit dem nur gelesen, und eines, mit dem nur geschrieben werden kann. Je Taktzyklus ist zwar in der vorliegenden Implementierung beides möglich, tatsächlich aber nur eines nötig. Dadurch, dass selbst bei einer überlappenden Pipelineausführung im Sinne des IMT (siehe auch das Kapitel *Entwurf*) immer nur eine Funktionseinheit aktiv ist, kommt es nie gleichzeitig zu lesenden und schreibenden Zugriffen auf den Speicher. Daher ist auch die Reihenfolge irrelevant, in der gelesen und geschrieben wird, da nie beides im selben Takt geschieht.

Die Implementierung selbst ist sehr einfach gehalten: Bei jeder steigenden Taktflanke wird geprüft, ob Lese- oder Schreibzugriff aktiviert sind, und dann dementsprechend an der gegebenen Lese- bzw. Schreibadresse auf den Speicher zugegriffen. Ist die Operation beendet, wird ein dieser entsprechendes Bestätigungssignal gegeben, das die Fertigstellung anzeigt.

5.5 Pseudo-Zufallszahlengenerator

Nicht spezifisch für Memory Tagging ist der Pseudo-Zufallszahlengenerator (in `hdl/lfsr_rng.sv`). Da aber im SEC-V so etwas nicht existiert und der *Entwurf* zufällig generierte Tags vorsieht, war es notwendig, einen simplen Generator zu diesem Zweck zu implementieren. Dabei handelt es sich um ein sogenanntes *linear-feedback shift register* in Galois-Konfiguration [23][S. 372ff], das auf Basis eines *Seeds* taktweise eine große Anzahl gleichmäßig verteilter Zahlen generieren und damit als Pseudozufallsgenerator verwendet werden kann.

5.6 Einbettung im SEC-V

Ein Überblick, an welchen Stellen Änderungen und neue Kommunikationsverbindungen notwendig geworden sind, ist in Abbildung 5.1 zu finden. Besonders die Speichereinheit (in der Abbildung als *MEM* gekennzeichnet) wurde angepasst, aber auch der Instruktionsdecoder und das übergeordnete Kernmodul `secv`, in dem alle anderen Module verknüpft sind. Darüber hinaus war die Vorbereitung der passenden Exceptions für den Fehlerfall notwendig.

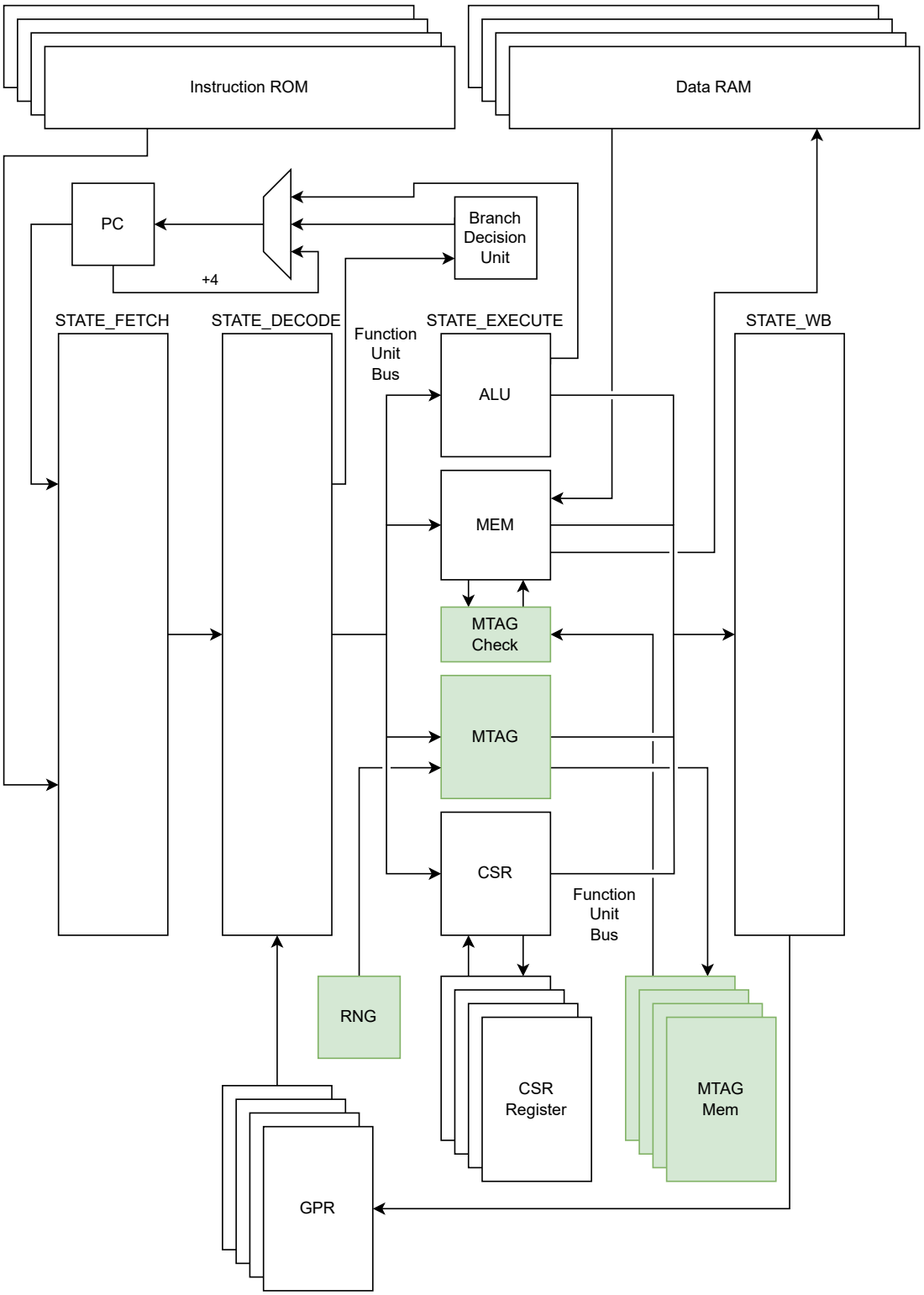

Abbildung 5.1: Erweiterung des SEC-V im schematischem Überblick. Neue Module sind grün eingefärbt.

5.6.1 Decoder

Wie auch ALU und Speichereinheit nutzt die Tagging Unit ein zweistufiges Dekodierverfahren. Die erste Stufe findet im allgemeinen Decoder (in `hdl/decoder.sv`) statt, der von allen Funktionseinheiten für die gemeinsame Funktionalität verwendet wird. Dazu gehört, je nach vom Opcode geforderten Instruktionsformat (siehe auch Tabelle 4.1), die korrekten Operanden, die passende Funktionseinheit und das Ziel, an dem das Ergebnis der Operation gespeichert werden muss, auszuwählen. Diese Werte werden dann vom *function unit bus* an die Einheiten weitergetragen. Im Falle des Memory Taggings werden hier für den Opcode *custom-0* (0001011) als Operanden die in Instruktionsformat *R* kodierten Register `rs1` und `rs2` gewählt. Die zuständige Funktionseinheit ist die Tagging Unit, im Quelltext als `FUNIT_MTAG` bezeichnet. Als Ziel wird das in der Instruktion angegebene Register gewählt. Der zweite Schritt ist dann die Entschlüsselung der genauen Operation (`tadr`, `tadre` oder `tadrr`) aus dem Inhalt des Instruktionsfelds `funct3`. Das geschieht in einem eigenen Modul, das im digitalen Anhang in der Datei `hdl/mtag_decoder.sv` zu finden ist.

5.6.2 Speichereinheit

In der Speichereinheit (in `hdl/mem.sv`) ist die Tag Checking Unit eingebunden, die parallel zu jedem Speicherzugriff arbeitet und die angegebenen Adressen prüft. Dafür erhält sie von der Speichereinheit die Hart-ID und die Adresse samt kodierter Farbe. Darüber hinaus ist sie über diese Einheit mit dem Tagspeicher verbunden. Aus Gründen der Übersichtlichkeit ist das in der vereinfachten Abbildung 5.1 anders dargestellt. Zuletzt werden auch Fehler der Tag Checking Unit hier verarbeitet: Nicht berechtigte Zugriffe wegen nicht übereinstimmender Hart-IDs oder Farben werden mit den entsprechenden Fehlercodes (`ECODE_MTAG_HART_INVLD` und `ECODE_MTAG_COLOR_INVLD`) an die übergeordnete Einheit weitergegeben.

5.6.3 Kernmodul und Exceptions

Im übergeordneten Modul `secv` (in `hdl/secv.sv`) werden alle Einheiten und Module verknüpft. Auch darüberhinausgehende Logik ist hier enthalten, diese ist aber bis auf den *function unit bus*, über den die Tagging Unit angebunden ist, und die Behandlung von Fehlern für diese Erweiterung weniger relevant. Neben der Tagging Unit sind auch der Zufallszahlengenerator und der Tagspeicher direkt in diesem Modul angebunden. Letzterer ist mit Speiche-

reinheit und Tagging Unit verknüpft, ersterer nur mit der Tagging Unit. Dem bereits vorhandenen Teil der Fehlerbehandlung mithilfe von Exceptions wurden zwei neue Auslöser hinzugefügt: `EX_CAUSE_MTAG_COLOR_INVLD` und `EX_CAUSE_MTAG_HART_INVLD`, die zukünftig verwendet werden können, um mit diesen Fehlertypen umzugehen und bspw. den normalen Programmablauf zu unterbrechen.

5.7 Tests

Für alle hier vorgestellten Module existieren Unit Tests (in `hdl/tests-svut/`). Diese sind jeweils als *test bench* umgesetzt und nutzen wie der restliche Prozessor das *SVUT*-Testframework zur einfacheren Ausführung und Umsetzung. Die Tests gehen die üblichen Anwendungsfälle durch, möglicherweise sind aber nicht alle Grenzfälle abgedeckt. Zukünftig müssten hier zusätzliche Testfälle eingeführt werden, wenn Probleme auftreten, die bisher nicht bedacht wurden. Genauere Informationen zur Handhabung der Tests finden sich im *Anhang*.

6 Auswertung

Die Umsetzung des Entwurfs wird unter den Gesichtspunkten der erreichten Funktionalität und Sicherheit, sowie den verursachten Hardware- und Laufzeitkosten. Für ersteres wurde auf Testprogramme zurückgegriffen, für letzteres soweit möglich auch auf Hardwarenutzungsbenchmarks und andere quantitative Methoden. Auch durch die Entwicklung neu aufgeworfene Fragen sind hier dargestellt.

6.1 Funktionalität und Sicherheit

Um die Funktionsweise der Erweiterung zu evaluieren, wurde auf zwei verschiedene Methoden zurückgegriffen: Einerseits wurden die bereits im letzten Kapitel kurz erwähnten *test benches* verwendet, andererseits aber auch verschiedene Assemblerprogramme, mit denen die korrekte Ausführung der neuen Instruktionen sowie die Erkennung von Fehlern getestet werden kann. Bei den *test benches* ist besonders hervorzuheben, dass die Module hier unter der Annahme getestet werden, dass der umgebende Prozessor vier Harts nutzt. Tests mit anderen Hart-Konfigurationen sind ebenfalls erfolgreich gewesen, wurden allerdings nicht per Unit-Test automatisiert. Das ist in den Tests zur Tagging Unit (im digitalen Anhang in der Datei `hdl/tests-svut/mtag_testbench.sv`) und zur Tag Checking Unit (in `hdl/tests-svut/mtag_chk_testbench.sv`) nachvollziehbar. Hier zeigt sich also, dass die Erweiterung tatsächlich in der Lage ist, mehr Threads zu unterstützen, als der Prozessor aktuell bietet.

Im Zusammenspiel mit dem gesamten Prozessor lässt sich aktuell zeigen, dass ein Fehler ausgelöst wird, wenn dem einzigen Thread des Prozessors der Zugriff auf eine Speicherstelle untersagt wird. Dieser Fall ist im Assemblerprogramm `programs/illegal-hart/illegal-hart.asm` modelliert und kann damit getestet werden. Genaueres zum Umgang mit den Assemblertestprogrammen findet sich im *Anhang*. Darüber hinaus wurde die Funktionalität mithilfe weiterer Testprogramme evaluiert. Zwei weitere davon sollen wie das eben beschriebene die Sicherheitsmechanismen testen. Eines simuliert einen Pufferüberlauf (in `programs/buffer-overflow/buffer-overflow.asm`),

bei dem vier aufeinanderfolgende Granules getaggt werden, um dann in einer Schleife auf die zugehörigen Speicheradressen zuzugreifen, bis dadurch ein Fehler ausgelöst wird, da die Granulegrenzen überschritten wurden. Damit wird gezeigt, dass räumliche Speicherfehler erkannt werden können.

Das andere simuliert ein *use-after-free* (in `programs/use-after-free/use-after-free.asm`), also einen temporalen Speicherfehler, wo ein eigentlich ungültiger Zeiger verwendet wird, nach dem der zugehörige Speicher deallokiert sein sollte. Die C-Bibliotheksfunktion `free` wird im Assemblerprogramm dadurch dargestellt, dass eine zuvor getaggte Speicherstelle wieder auf den Tag `0` gesetzt wird. Der Zugriff mit dem alten Zeiger scheitert dadurch, dass sowohl die Farbe nicht mehr übereinstimmt, als auch dadurch, dass bei einem `0`-Tag kein Thread Zugriffsrechte hat.

Das letzte Testprogramm führt einmal alle Instruktionen aus und testet sie, außerdem werden beide Fehlerfälle (*hart mismatch* und *color mismatch*) einmal ausgelöst. Da der Prozessor noch keine echte Handhabung von Exceptions unterstützt, kann die korrekte Ausführung der Programme nur durch die Untersuchung der Simulationsergebnisse als sogenannte *wave forms* verifiziert werden.

In den eben genannten Testprogrammen wird aber auch ein Problem deutlich: Zwar werden unerlaubte Speicherzugriffe erkannt, allerdings im Falle von schreibenden Zugriffen nicht unterbunden, da Schreib- und Prüfvorgänge parallel ablaufen, und der Wert bereits geschrieben ist, wenn festgestellt wird, dass er nicht hätte geschrieben werden dürfen. Bei lesenden Zugriffen kann dagegen einfach das Gelesene verworfen und nicht ins Register zurückgeschrieben werden. Um das Problem beim Schreiben zu umgehen, gäbe es verschiedene Möglichkeiten. Eine wäre, im Falle einer Exception die Adresse, an der diese ausgelöst wurde, einfach mit 0 zu überschreiben, also den unerlaubten, potenziell böswilligen Zugriff unschädlich zu machen. Eine andere würde einen grundlegenderen Umbau des Prozessors erfordern, der auch aus anderen Gründen bereits im Gespräch ist: Streckt man die Pipeline auf fünf Stufen, führt also eine zusätzliche Stufe für Speicherzugriffe ein, könnte die Prüfung vor dem eigentlichen Zugriff, während der Ausführungsphase, stattfinden und so der Speicherzugriff im Fehlerfall direkt unterbunden werden. Diese Erkenntnis ist ein wichtiges Argument für die Änderung der Pipelinekonfiguration, um eine höhere Sicherheit zu gewährleisten. Durch das frühe Entwicklungsstadium, in dem sich der Prozessor befindet, ist es möglich diese anzupassen, bevor höhere Umbaukosten entstehen.

Weiterhin zeigt sich, dass der Ansatz, den Speicherzugriff standardmäßig allen Threads zu untersagen, positive Auswirkungen auf die Sicherheit des Ansatzes hat, da die Nutzung damit erzwungen wird: Es ist Programmiererinnen und Programmierern nicht möglich, das Tagging einfach zu ignorieren. Reali-

stisch für alltägliches Arbeiten ist dieser Ansatz jedoch nicht in jedem Fall; u. a. aus Gründen der Leistungsfähigkeit (vgl. dazu auch das Unterkapitel zu den *Laufzeitkosten*). Eine alternative Umsetzung wäre mit geringem Entwicklungsaufwand dadurch möglich, zulässige Threads im Bitvektor nicht mit einer 1, sondern mit einer 0 zu kennzeichnen. Dadurch wäre standardmäßig jeder Thread zugriffsberechtigt, wenn eine Adresse mit 0-Tag versehen ist. Außerdem werden so auch durch die Farbe nicht direkt Fehler ausgelöst, da ein unveränderter Adresszeiger, mit Nullen in den oberen Bits korrekt mit dem dazugehörigen 0-Tag verglichen wird, es also nicht ohne vorherige Interaktion zu einem *mismatch* kommt. Aus diesen Gründen wäre es dann möglich, Tagging nur als zusätzlichen Mechanismus in ausgewählten Fällen einzusetzen, z. B. ausschließlich im Heapspeicher, und nicht grundsätzlich. Auch die Verwendung nur einer der Mechanismen oder eine hybride Form der beiden je nach Bedarf ist denkbar.

Dank der 64 Bit langen Adressen ermöglicht zudem sehr lange Tags. Gerade unter dem Gesichtspunkt der maximal notwendigen Adressbreite bei Mikrocontrollern wie dem SEC-V. Diese ist oft sehr niedrig, bspw. acht oder 16 Bit, da der zur Verfügung stehende Speicher i. d. R. nicht so groß ist, wie bei leistungsfähigeren Plattformen. Dadurch bleibt viel Platz in den Adresszeigern für Tags. Im Gegensatz zu den im Kapitel *Stand der Technik* vorgestellten Umsetzungen ADI und MTE sind hier statt nur vier Bit ohne Probleme deutlich längere Tags möglich. Die standardmäßige Konfiguration der vorliegenden Umsetzung nutzt bspw. 16 Bit, davon 15 Bit für die Farbe und ein Bit für den Threadzugriff. Die Wahrscheinlichkeit von zwei zufällig gleichen Tags sinkt dadurch beträchtlich: Bei den kurzen Tags liegt sie bei 6,25 %, bei den hier verwendeten längeren nur noch bei etwas mehr als 0,003 %. Das sichert deutlich besser gegen Speicherfehler ab, besonders gegen temporale, bei denen die Abwehr von unerlaubten Zugriffen nicht durch sorgfältige Tagauswahl unterstützt werden kann. Bei räumlichen Speicherfehlern ist die Art der Auswahl dadurch auch deutlich unwichtiger, da es mit steigender Taglänge unwahrscheinlicher wird, dass zufällig gleiche Tags für aneinander angrenzende Speicherbereiche gewählt werden. Außerdem sind Angriffe, wie das systematische Durchprobieren aller möglichen Tags, also Brute-Force-Angriffe, deutlich erschwert: Müssen mit vier Bit langen Tags nur 16 Varianten durchprobiert werden, sind es bei 15 Bit 32.768 Tags. Diese Faktoren ermöglichen es weniger Aufwand bei der Tagauswahl zu betreiben. So kann, wenn nicht höchstmögliche Sicherheitsgarantien nötig sind, darauf verzichtet werden, bei der Vergabe neuer Tags auch die benachbarten Tags zu betrachten und in die Erzeugung mit einzubeziehen. Je nachdem, wo die Erzeugung genau geschieht, spart man dadurch Hardware- und/oder Laufzeitkosten.

6.2 Hardwarekosten

Zusätzlich benötigte Hardware fällt bei der Erweiterung in zwei Kategorien an: zum einen in der Logik, also vor allem der Tagging und der Tag Checking Unit, zum anderen im Tagspeicher. Um die Kosten besser einschätzen zu können, wurden sowohl der unveränderte SEC-V, als auch der Prozessor mit der Erweiterung mithilfe der Software *Vivado* [30] synthetisiert und die dabei entstehenden *utilization reports* als Grundlage der Betrachtung verwendet. Die genauen Ergebnisse finden sich im Anhang, außerdem sind die vollständigen *utilization reports* als Textdateien im digitalen Anhang zu finden. Neben den beiden eben genannten Konfigurationen wurde außerdem eine weitere synthetisiert, bei welcher der eigentliche Tagspeicher auskommentiert wurde, dementsprechend also nicht mit in die Ressourcenschätzung eingeflossen ist. Einen Überblick über die Ergebnisse bietet die Abbildung 6.1.

Prozentual ist bei einem Vergleich der Gesamtsumme aller verwendeten Elemente ein Zuwachs an Hardwarebelegung durch die Tagging-Erweiterung von knapp 13 % zu erkennen; werden die Kategorien untereinander verglichen und daraus den Durchschnitt gebildet, zeigt sich ein Zuwachs von knapp 11 %. Die größte Zunahme ist bei den Latch-Registern zu beobachten, von denen mit der Tagging-Erweiterung fast 30 % zusätzlich verwendet werden. Auf der anderen Seite nutzt die Erweiterung drei F8 Muxes weniger als der unveränderte SEC-V. Der Grund dafür ist wahrscheinlich die Nutzung zusätzlicher F7 Muxes, wodurch weniger F8 Muxes benötigt werden.

6.2.1 Logik

Durch die Messung, bei welcher der Tagspeicher außen vor gelassen wurde, kann gezeigt werden, wie wenig zusätzliche Hardware für die reine Logik der Erweiterung benötigt wird. Der Zuwachs aller Elemente zusammen beträgt lediglich 3,6 %, während der Durchschnittszuwachs der einzelnen Kategorien nur bei 2,7 % liegt. Besonders auffällig ist der reduzierte Bedarf an *Look-Up Tables* (*LUTs*), die Registerbelegung bleibt dagegen weiterhin recht hoch. Auch hier ist bei den Latch-Registern mit gut 23 % der größte Zuwachs zu beobachten. Insgesamt ist deutlich zu sehen, dass der Großteil des Hardwareaufwandes der Tagging-Erweiterung durch den Tagspeicher verursacht wird.

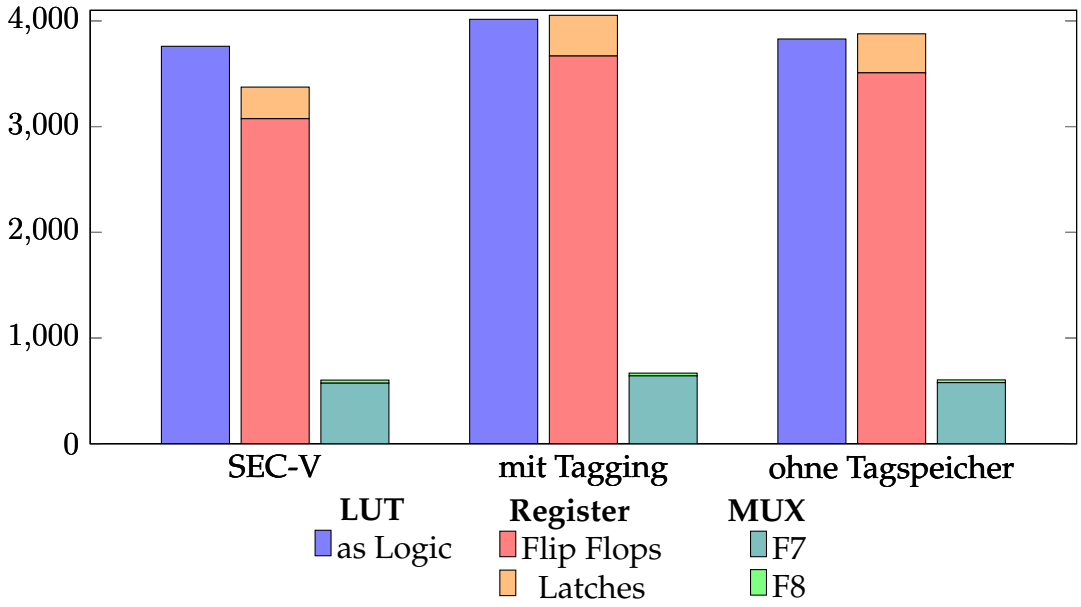

Abbildung 6.1: Verwendete Ressourcen

6.2.2 Speicher

Der zusätzlich benötigte Speicher hängt, wie bereits in Kapitel *Grundlagen* erläutert, von den genauen Implementierungsumständen ab. Im Gegensatz zu den Ausführungen dort, muss in diesem Fall nicht mit dem gesamten theoretisch verfügbaren Speicher gerechnet werden, der mithilfe des größtmöglichen Adressraums verfügbar wäre. Stattdessen ist im SEC-V die tatsächliche Adressbreite des Datenspeichers per Parameter einstellbar. Diese wird im Folgenden als Data Address Width (DAW) bezeichnet. Ein höherer Wert hier führt zu größeren Tagspeichern. Davon abgesehen ist die Granularität (G) zu beachten: Diese wird als Anzahl von Stellen angegeben, um welche die Speicheradresse nach rechts verschoben werden muss, um daraus die zugehörige Tagspeicheradresse zu berechnen. Hier führt ein höherer Wert zu kleineren Tagspeichern. Aus Granularität G und Adressbreite des Datenspeichers DAW lässt sich durch einfache Subtraktion die notwendige Adressbreite des Tagspeichers TAW berechnen:

$$TAW = DAW - G$$

Mithilfe des letzten für die Tagspeichergröße relevanten Parameters, der Länge eines einzelnen Tags in Bit (TS für Tag Size), kann dann die tatsächliche Größe (Tag Memory Size für TMS) in Byte berechnet werden:

$$TMS = \frac{2^{TAW} \cdot TS}{8}$$

Eine höhere Taglänge führt demnach zu größeren Tagspeichern. Darüber hinaus lässt sich erkennen, dass die Größe eines Tags tendenziell weniger Einfluss auf die Tagspeichergröße hat, als die Adressbreite.

In der aktuellen Konfiguration nutzt der Prozessor zehn Bit für die Adressierung des Datenspeichers und eine Granularität von vier. Einem 64 Bit Speicherwort entspricht also ein Tag, hier mit 16 Bit Länge. Den eben besprochenen Formeln zufolge, nutzt der Tagspeicher dementsprechend sechs Bit Adressen und ist 128 Byte groß. Demgegenüber steht der Datenspeicher mit einer Größe von 1024 Byte, also 16 Speicherworten. Wählt man einen größeren Speicher, bspw. mit 16 Bit Adressweite, also 64 KiB Größe, wächst der entsprechende Tagspeicher auf 8 KiB. Je nach den konkreten Anforderungen können die einzelnen Parameter entsprechend angepasst werden.

Vorteilhaft ist, dass bei einem sehr kleinem Hauptspeicher auch der Tagspeicher gering ausfällt. In Fällen, in denen die verfügbare Hardware also bereits knapp ist, hält sich auch die für das Tagging zusätzlich benötigte in engen Grenzen. Eine Schwäche ist, dass selbst, wenn der ganze Speicher mit demselben Tag versehen wurde, der gesamte Tagspeicher dafür verwendet werden muss. Alternativen zu diesem Modell wurden bereits im Kapitel *Entwurf* diskutiert und sind gerade im Falle von Mikrocontrollern interessant ist, da sie den zusätzlich benötigten Speicher so klein wie möglich halten.

6.3 Laufzeitkosten

In der aktuellen Entwicklungsphase des Prozessors lassen sich die tatsächlichen Laufzeitkosten nur schwer abschätzen. Notwendig wäre dafür nicht nur die Portierung von Benchmarks und realen Programmen, sondern auch die Anpassung der C-Bibliothek und des Compilers, um die genauen Auswirkungen zu berechnen. Potenzielle Kosten durch die Regelung der Thread-Berechtigungen zu messen, würde zusätzlich die Fertigstellung des Interleaved Multithreadings erfordern. Es ist allerdings möglich, zu zeigen, in welchen Fällen es zu zusätzlichen Instruktionen im Vergleich zu einem unveränderten SEC-V kommt.

Das betrifft in erster Linie die Allokation und Deallokation von Speicher. Vor dem ersten Zugriff auf einen Speicherbereich, also bei der **Allokation**, ist es notwendig, diesen mit Farbe und Hart-Vektor zu taggen. Ohne vorheriges Taggen ist keinerlei Speicherzugriff erlaubt, da automatisch der Hart-Vektor im-

mer nur aus Nullen besteht, und damit kein Thread Zugriffsrechte hat. Je nach gewünschter Tagging-Operation sind hier verschieden viele Instruktionen notwendig.

Bei **`tadr`** sind es mindestens:

- Adresse in ein Register laden
- Tag in ein Register laden
- `tadr`

Je nach Adresse oder Tag sind unterschiedliche Instruktionen möglich, von denen bei besonders großen Tags und Adressen unter Umständen auch mehrere nötig sein können, um diese zu konstruieren, da nicht beliebig große Konstanten in den jeweiligen Instruktionen kodiert werden können. Die maximale Instruktionsanzahl, um eine 32 Bit große Konstante zu konstruieren, wäre drei (`lui` für die oberen 20 Bits, `addi` für die unteren 12 Bit und `or` um beide zu verknüpfen), für kleinere Konstanten sind auch andere Kombinationen denkbar. Für die Allokation eines Granules sind also bei `tadr` **zwischen drei und sieben** Instruktionen nötig.

Bei **`tadre`** sind es **mindestens sechs**:

- Adresse in ein Register laden
- Farbe in ein Register laden
- Farbe in die oberen Bits verschieben
- Farbe in Adresse kodieren
- Hart-Vektor in ein Register laden
- `tadre`

Wie zuvor kann auch hier die Konstruktion von Adresse und Farbe aufwendiger sein; **maximal** sind **zehn** Instruktionen nötig.

Im Falle von **`tadrr`** sind weniger Instruktionen nötig, **zwischen drei und sechs**:

- Adresse in ein Register laden
- Hart-Vektor in ein Register laden
- `tadrr`

Beim Lesen und Schreiben sind keine zusätzlichen Instruktionen nötig, da das Prüfen der Tags parallel in Hardware passiert.

Die **Deallokation** verhält sich einfacher als die Allokation. Hier kann immer die Instruktion `tadr` verwendet werden. Die Zieladresse kann als bekannt vorausgesetzt werden, da sie bei der Allokation bereits geladen worden sein muss. Dadurch benötigt die Deallokation dann nur **eine Instruktion**, da 0 in Form des Registers `x0` direkt als Tagwert verwendet werden kann.

Auf einem höheren Level, also der Allokation mehrerer Granules am Stück, zeigt sich eine Herausforderung: Hier ist es notwendig, je nach Länge des gewünschten Speicherbereiches, mehrere Bereiche nach dem eben vorgestellten Muster mit demselben Tag zu versehen. Dadurch werden Allokationen verglichen mit der einfachen Erzeugung eines Zeigers auf die erste Adresse des entsprechenden Speicherbereiches deutlich aufwendiger. Dasselbe gilt für die Deallokation, auch wenn die Auswirkungen aufgrund der geringeren Anzahl an benötigten Instruktionen weniger gravierend sind. Sollen bei der Allokation und/oder Deallokation ohnehin alle Speicherinhalte aus Sicherheitsgründen auf 0 gesetzt werden, ist der zusätzliche Aufwand weniger groß, hier wären bspw. noch neue Instruktionen denkbar, bei denen sowohl der Speicherinhalt, als auch der zugehörige Tag zurückgesetzt werden, um diesen Ansatz zu unterstützen. Auch dabei wäre eine fünfstufige Pipelineumsetzung hilfreich, um in der Ausführungsstufe den Tag und in der Speicherphase den entsprechenden Bereich mit 0 zu überschreiben.

Ein anderer Ansatz wäre, für größere Speicherbereiche den Tag nur an einer Stelle den dazugehörigen Granuleadressen zuzuordnen. Dadurch wären die entsprechenden Operationen schneller, da sie jeweils nur einmal durchgeführt werden müssten, anstatt je Granule. Hierdurch könnte, wenn man auf die zusätzliche Sicherheit durch die Überschreibung des Speichers verzichten kann, die Laufzeit dieser Vorgänge verringert werden. Allerdings wird hierbei auch, wie im Kapitel *Entwurf* dargestellt, der einzelne Zugriff auf den Tagspeicher aufwendiger und damit unter Umständen langsamer. Je nach konkreter Umsetzung, kann das zu langsameren Taktgeschwindigkeiten oder zusätzlichen Taktzyklen führen, auch wenn insgesamt weniger Instruktionen benötigt werden.

Die realen Auswirkungen auf Software lassen sich anhand dieser Zahlen und Überlegungen nur schwer zu bewerten. Hier sind mehrere verschiedene Aspekte im Spiel, die sich nicht zu einem eindeutigen Ergebnis formulieren lassen: Wie häufig werden Speicherbereiche allokiert? Wie groß sind Tags und Granules? Auf welche Art werden die Tags konstruiert? Dazu kommt in diesem bestimmten Fall, wie die verschiedenen Threads die Speicherallokation auch untereinander aufteilen können: Speicher, der gemeinsam verwendet wird, muss auch nur einmal allokiert werden. Gerade bei Mikrocontrollern muss hier also

der konkrete Einzelfall betrachtet werden, um die zusätzlichen Kosten einschätzen zu können.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Ansatz vorgestellt, wie sich Memory Tagging und Interleaved Multithreading verbinden lassen, um sowohl vor typischen Speicherfehlern innerhalb eines, als auch zwischen verschiedenen Threads zu schützen. Dafür wurden die theoretischen Grundlagen des Memory Taggings und seine aktuellen Umsetzungen untersucht, um einen Überblick über die damit zusammenhängenden Möglichkeiten und Entscheidungen zu gewinnen. Mithilfe dieser konnte eine Erweiterung des Prozessors SEC-V entworfen werden, die das Tagging-Konzept des Memory Colorings mit einer feingliederigen Zugriffsberechtigungsverwaltung für mehrere Hardware-Threads kombiniert. Dazu wird jeder Speicherbereich nicht nur klassisch mit einer Farbe versehen, mit der garantiert wird, dass nur berechtigte Adresszeiger auf den dazugehörigen Bereich zugreifen dürfen, sondern es wird zusätzlich intern ein von der Programmiererin oder dem Programmierer festgelegter Bitvektor angelegt. An diesem kann bei zukünftigen Zugriffen erkannt, ob der zugreifende Thread die notwendige Berechtigung besitzt. Stimmen entweder der Thread oder die Farbe des Adresszeigers nicht mit den gespeicherten Tags überein, wird im Prozessor eine Exception ausgelöst. Diese kann genutzt werden, um den Zugriff zu ignorieren oder zu unterbinden. Dabei bietet der Bitvektor, der die Threadberechtigungen angibt, sehr flexible Möglichkeiten: Jedem Thread kann jeweils der Zugriff erlaubt oder verboten sein, wobei es dadurch auch ohne Weiteres möglich ist, den Speicherzugriff auf einen bestimmten Bereich vollständig zu untersagen.

Die Auswertung der Implementierung hat gezeigt, wo ihre Stärken und Schwächen liegen: Sowohl Speicherfehler innerhalb von Threads als auch im Zusammenhang mit Zugriffen verschiedener Threads werden erkannt. Das wurde mithilfe einfacher Assembler-Testprogrammen verifiziert. Die Umsetzung der Zugriffsberechtigungen mithilfe von Bitvektoren hat sich als sehr flexibel erwiesen, und bietet auf eine nachvollziehbare Weise sehr genaue Kontrolle über die Zugriffsrechte im Speicher. Das ist besonders angesichts der Tatsache bemerkenswert, dass es sich hier um einen Mikrocontroller handelt, bei dem das Teilen von Daten zwischen und die Isolation von Threads eine oft besonders fehlerbehaftete Aufgabe sein kann. Mithilfe dieser Erweiterung können Annahmen, die entsprechenden Programmen oft implizit zugrunde liegen, explizit gemacht und deren Einhaltung erzwungen werden. Fehler können so schneller bemerkt

und in ihren Auswirkungen begrenzt werden. Die Integration der Erweiterung in den SEC-V erwies sich als weitgehend unproblematisch, an bestehenden Modulen musste wenig bis gar nichts geändert werden, um die Erweiterung zu implementieren.

Nachteile der Erweiterung sind auf der einen Seite die Kosten: Sowohl zusätzliche Laufzeit, als auch Hardware sind notwendig, um Memory Tagging einzusetzen. Gerade der Tagspeicher bietet viele Möglichkeiten der Überarbeitung, um Hardwarekosten zu senken. Analog dazu ist es auf der Laufzeitseite vor allem die Speicherallokation, die noch Optimierungspotenzial bietet. Andere Probleme sind dem frühen Entwicklungsstadium des SEC-V geschuldet: Interleaved Multithreading ist zwar fest eingeplant, als integraler Bestandteil des zukünftigen Entwurfs vorgesehen und teilweise auch schon im aktuellen Design angedacht, der Prozessor besitzt momentan aber nur einen Thread. Darüber hinaus werden ausgelöste Exceptions noch nicht behandelt. Beides macht es schwierig, über das Theoretische oder eng umrissene praktische Tests hinaus die Praxistauglichkeit der Erweiterung zu beurteilen.

Hier können zukünftige Arbeiten ansetzen: Sind diese Funktionen vollständig implementiert, kann die Erweiterung mit voraussichtlich geringem Aufwand angepasst und dann mit realen Programmen getestet werden. Dafür ist außerdem die Implementierung einer zur Erweiterung passenden Software nötig, also bspw. die Anpassung der Speicherallokation in der C-Bibliothek für den Heap- und im Compiler für den Stackspeicher, sodass diese sich das Memory Tagging auch zu nutze machen können. Die geplante Erweiterung des Prozessors auf eine fünfstufige Pipeline bietet ebenfalls interessante Möglichkeiten für die Erweiterung, wie in der *Auswertung* bereits angerissen wurde. Die Effizienz, vor allem aber die Sicherheit des Memory Taggings kann auf diese Weise deutlich verbessert werden. Das gehört zu den zentralen Erkenntnissen dieser Arbeit und kann Argumente für weitere Architekturentscheidungen bzgl. der Pipelinekonfiguration des SEC-V liefern.

Besonders aussichtsreich ist es auch, komplexere Tagspeicherentwürfe umzusetzen. Dadurch könnten sowohl die Hardware- als auch die Laufzeitkosten verringert werden, indem nur Tag, Anfangsadresse und zugehörige Granules für beliebig lange Speicherbereiche gespeichert werden, anstatt je Granule einen separaten Tag zu verwalten. Wie bereits in der *Auswertung* gezeigt wurde, können dadurch sowohl Speicherbedarf, als auch Kosten durch die Verwaltung zahlreicher Tags reduziert werden. Zu diesem Feld gehört auch, die optimalen Werte für die Taglänge und die Granularität auszutarieren, um den sich z. T. widersprechenden Sicherheits-, Performanz- und Kostenanforderungen gerecht zu werden.

Der letzte Bereich für zukünftige Arbeiten, der hier vorgestellt werden soll, ist das Zusammenspiel von Interleaved Multithreading mit dem hier entwickelten Taggingansatz. Durch die Umsetzung der Berechtigungen mit Bitvektoren ist es bspw. möglich, aktuell wartende Threads zu nutzen, um Speicherbereiche zu allokieren, die später von diesen oder anderen Threads benötigt werden. Auch die Abwälzung der Speicherallokation auf einen nur dafür zuständigen Thread ist je nach Anwendung eine Möglichkeit. Bei geteiltem Speicher kann die Allokation auch zwischen den nutzenden Threads aufgeteilt werden, wodurch die zusätzlichen Laufzeitkosten reduziert werden könnten.

Die Untersuchung zeigt das große Potenzial der Weiterentwicklung des in dieser Arbeit vorgestellten Ansatzes. Das gilt gerade auch in Verbindung mit dem SEC-V-Prozessor, der im Laufe seiner Entwicklung noch einige weitere Sicherheitsmechanismen erhalten soll, die sich potenziell mit dem Tagging verbinden lassen. Die vorliegende Arbeit stellt einen Ausgangspunkt für solche und andere neue Entwicklungen im Bereich der Prozessorarchitektur dar. Sie liefert sowohl theoretische Konzepte als auch eine praktische Umsetzung, die als Basis für zukünftige Forschungs- und Entwicklungsarbeiten in diesem sich stets entwickelnden Feld dienen kann.

Abkürzungsverzeichnis

ADI Application Data Integrity

ALU Arithmetic Logic Unit

CSR Control and Status Register

FPGA Field Programmable Gate Array

IC Integrated Curcuit

IMT Interleaved Multithreading

ISA Instruction Set Architecture

LUT Look-up Table

MMU Memory Management Unit

MTE Memory Tagging Extension

RISC Reduced Instruction Set Computer

SIMT Scalabele Interleaved Multithreading

Anhang

Simulation und Tests

Um die Simulation und das Testen des praktischen Teils dieser Arbeit zu erleichtern, sind hier einige Hinweise gesammelt, die bei der Anwendung hilfreich sein können. Alle folgenden Anweisungen gehen von der Ausführung auf einem Linux- oder Unix-Betriebssystem mit einer POSIX-kompatiblen Shell aus.

Voraussetzungen

Es werden mehrere Programme für die Simulation und das Testen der Arbeit benötigt:

- **GNU Make** (`https://www.gnu.org/software/make/`, letzter Zugriff am 31.01.2024)
- **Icarus Verilog** (`https://steveicarus.github.io/iverilog/`, letzter Zugriff am 30.01.2024)
- **SVUT** (`https://github.com/dpretet/svut`, letzter Zugriff am 30.01.2024)
- **RISC-V GNU Compiler Toolchain** (`https://github.com/riscv-collab/riscv-gnu-toolchain`, letzter Zugriff am 30.01.2024)
- **elf2hex** (`https://github.com/sifive/elf2hex`, letzter Zugriff am 31.01.2024)
- **GTKWave** (`https://gtkwave.sourceforge.net`, letzter Zugriff am 30.01.2024) oder ein beliebiges anderes Programm, das zur Anzeige von `.vcd`-Dateien geeignet ist

Mit dem Skript `scripts/pull.sh` können die jeweils aktuellsten Entwicklungsversionen von SVUT und Icarus Verilog heruntergeladen werden. SVUT kann dann verwendet werden, indem das Verzeichnis, in das es heruntergeladen wurde (der Voreinstellung nach `$HOME/lib/svut`), dem Pfad hinzugefügt

wird. Das kann erreicht werden, indem die Datei `scripts/bashrc` mit dem Befehl `source` oder `.` in die aktuelle Umgebung geladen wird. Icarus Verilog kann den Anweisungen im `README.md` unter *Compilation* und *Compiling From GitHub* folgend installiert werden. Diese ist im heruntergeladenen Repository zu finden, das der Voreinstellung nach in `$HOME/lib/iverilog` liegt. In vielen Distributionen ist die aktuelle Version 12 in den offiziellen Paketquellen, diese ist allerdings nicht ausreichend, da einige noch in Enwicklung befindliche Funktionen verwendet werden. Diese Arbeit wurde mit SVUT in der Version des Commits `8f03c39` im Git-Repository und Icarus Verilog entsprechend Commit `5e13989` getestet. GTKWave wird in den meisten Distributionen über die offiziellen Paketquellen angeboten. Alternativ existiert auch eine Version im Flatpak-Format, die distributionsunabhängig angeboten wird.

Zur Übersetzung der Assembler-Testprogramme wird die RISC-V GNU Compiler Toolchain benötigt. In einigen Distributionen steht diese ebenfalls in den Paketquellen bereit, die benötigte Version sollte `riscv64-unknown-elf` als Suffix haben. Der genaue Name ist Entscheidung der jeweiligen Distributoren und kann hier deshalb nicht vorgegeben werden. Alternativ kann sie auch per Hand kompiliert werden. Die dafür nötigen Anweisungen findet man auf der in der Liste verlinkten Seite in den Punkten *Getting the sources*, *Prerequisites* und *Installation (Newlib)*. Darüber hinaus wird `elf2hex` benötigt, um die erzeugte Datei in ein von SystemVerilog lesbares Format umzuwandeln. Auch hiervon wird die aktuellste Version mit `scripts/pull.sh` heruntergeladen, getestet wurde es mit dem Commit `f28a310`. Für die Installation sollte auch hier den Anweisungen in `README.md` direkt im gerade heruntergeladenen Repository unter *Building `elf2hex` from git* gefolgt werden. GNU Make ist auf vielen Systemen bereits installiert und in so gut wie allen Distributionen in den offiziellen Paketquellen.

Die folgenden Programme sind nicht für die Simulation der Erweiterung notwendig, werden aber sonst an irgendeiner Stelle im Projekt verwendet. Sie sind daher optional und hier nur der Vollständigkeit halber erwähnt.

- **Verilator** (`https://www.veripool.org/verilator/`, letzter Zugriff am 30.01.2024) wird im Lint-Skript (`hdl/lint.sh`), sowie als optionale, für diese Erweiterung ungetestete Alternative zu Icarus Verilog in SVUT verwendet.
- **Verible** (`https://chipsalliance.github.io/verible/`, letzter Zugriff am 30.01.2024) wird ebenfalls im Lint-Skript verwendet.
- **SVUnit** (`http://agilesoc.com/open-source-projects/svunit/`, letzter Zugriff am 30.01.2024) dient der Ausführung zusätzlicher Tests, die für die vorliegende Erweiterung nicht angepasst wurden.

Unit Tests

Die Unit-Tests befinden sich im Verzeichnis `hdl/tests-svut/` und können entweder alle mit dem Befehl `make` oder einzeln mit `make <Modulname>` ausgeführt werden. Alle Tests erzeugen dabei jeweils eine `.vcd`-Datei, die z. B. mithilfe von GTKWave betrachtet werden kann, um die genauen Verläufe der Signale im Verlauf der jeweiligen Testsuite zu untersuchen. Der Erfolg oder Misserfolg der einzelnen Testfälle wird aber auch als Text bei der Ausführung ausgegeben.

Assemblierung und Simulation der Testprogramme

Um die Assembler-Testprogramme zu übersetzen, muss zuerst in das entsprechende Verzeichnis gewechselt werden. Die verfügbaren Applikationen befinden sich in Unterverzeichnissen von `programs/`. Befindet man sich in dem gewünschten Verzeichnis, kann mit dem Befehl `make` die `.asm`-Datei in eine `.hex`-Datei übersetzt werden. Diese kann dann in `firmware.hex` umbenannt und in `hdl/tests-svut/` kopiert werden. Dort wird mit `make secv` dann die Ausführung des Programms mit dem Prozessor simuliert. Im Gegensatz zu den anderen *test benches* gibt es hier keine Testfälle, sondern es wird nur die normale Arbeitweise des Prozessors mit dem dazugehörigen Speicher simuliert. Die Simulationsergebnisse selbst müssen händisch durch die Auswertung der Datei `secv_testbench.vcd`, bspw. mit GTKWave, überprüft werden. Dabei sind besonders die Fehlersignale hilfreich, bspw. `hart_mismatch` und `color_mismatch` des Moduls `mtag_check`.

Ergebnisse der *utilization reports*

	SEC-V	Tagging	Ohne Tagspeicher	Kosten	Verhältnis	Kosten ohne Tagspeicher	Verhältnis
Slice LUTs	3760	4015	3829	255	1,068	69	1,018
LUT as Logic	*3760*	*4015*	*3829*	*255*	*1,068*	*69*	*1,018*
LUT as Memory	*0*	*0*	*0*	*0*	*0*	*0*	*0*
Slice Registers	3374	4053	3509	679	1,201	135	1,040
Register as Flip Flop	*3075*	*3668*	*3140*	*593*	*1,193*	*65*	*1,021*
Register as Latch	*299*	*385*	*369*	*86*	*1,288*	*70*	*1,234*
F7 Muxes	573	642	578	69	1,120	5	1,009
F8 Muxes	29	26	26	-3	0,897	-3	0,897
Durchschnitt	**-**	**-**	**-**	**-**	**1,113**	**-**	**1,036**
Gesamt	**7736**	**8736**	**7942**	**1000**	**1,129**	**206**	**1,027**

Tabelle 7.1: Ergebnisse der *utilization reports*

Digitaler Anhang

Der Arbeit liegt ein Datenträger bei, auf dem sich die praktische Umsetzung der Arbeit in Form eines Git-Repositorys befindet. Im dort ausgewählten Branch *memtag* ist der SEC-V inklusive der hier entwickelten Erweiterung zu finden. Die relevanten Dateien werden in den entsprechenden Teilen der Arbeit vorgestellt. Das ursprüngliche Design ist im Branch *main* zu finden. Im Vergleich der beiden Branches kann nachvollzogen werden, welche Anteile Leistung des Autors sind und welche bereits vorlagen.

Literatur

[1] *ADI Cookbook*. Oracle. 2015. URL: https://docs.oracle.com/cd/E37069_01/pdf/E60383.pdf (siehe S. 15).

[2] *adi(2)*. Letzter Zugriff am 03.09.2023. Oracle. 2017. URL: https://docs.oracle.com/cd/E86824_01/html/E54765/adi-2.html (siehe S. 15).

[3] *adi(7)*. Letzter Zugriff am 27.08.2023. Oracle. 2022. URL: https://docs.oracle.com/cd/E88353_01/html/E37853/adi-7.html (siehe S. 15, 16).

[4] Kathirgamar Aingaran u. a. "M7: Oracle's Next-Generation Sparc Processor". In: *IEEE Micro* 35 (2015), S. 36–45 (siehe S. 8, 15).

[5] *Application Data Integrity (ADI)*. Letzter Zugriff am 03.09.2023. Linux Kernel Organization. URL: https://www.kernel.org/doc/Documentation/sparc/adi.txt (siehe S. 15).

[6] *Armv8.5-A Memory Tagging Extension White Paper*. Techn. Ber. ARM, 2021. URL: https://developer.arm.com/documentation/102925/0100/ (siehe S. 3, 5, 8, 11, 12, 16, 17).

[7] Ying Bai. "ARM® Memory Protection Unit (MPU)". In: (2016) (siehe S. 13).

[8] Andrew Ferraiuolo u. a. "HyperFlow: A processor architecture for non-malleable, timing-safe information flow security". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, S. 1583–1600 (siehe S. 18).

[9] Edward A. Feustel. "On The Advantages of Tagged Architecture". In: *IEEE Transactions on Computers* C-22 (1973), S. 644–656 (siehe S. 4).

[10] Samuel Fingeret. "Defeating code reuse attacks with minimal tagged architecture". Diss. Massachusetts Institute of Technology, 2015 (siehe S. 18).

[11] Jörn Hoffmann. *SEC-V - The Secure RISC-V Processor*. Techn. Ber. Letzter Zugriff am 13.12.2023. 2023. URL: https://github.com/joernhoffmann/sec-v/blob/main/README.md (siehe S. 21).

[12] Samuel Jero u. a. "TAG: Tagged Architecture Guide". In: *ACM Computing Surveys* 55 (2022), S. 1–34 (siehe S. 3, 7, 8, 15, 18).

[13] David Kaplan, Jeremy Powell und Tom Woller. "AMD memory encryption". In: *White paper* (2016), S. 13 (siehe S. 13).

[14] Tong Liu u. a. "TMDFI: Tagged memory assisted for fine-grained dataflow integrity towards embedded systems against software exploitation". In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. Ieee. 2018, S. 545–550 (siehe S. 18).

[15] Arjun Menon u. a. "Shakti-T: A RISC-V processor with light weight security extensions". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy*. 2017, S. 1–8 (siehe S. 18).

[16] Matt Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. Letzter Zugriff am 31.01.2024. 2019. URL: https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf (siehe S. 1).

[17] Pascal Nasahl u. a. "CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (2020) (siehe S. 4, 5, 13, 19).

[18] *Oracle SPARC Architecture 2015*. Oracle. 2016. URL: https://www.oracle.com/technetwork/sparc-architecture-2015-2868130.pdf (siehe S. 16).

[19] Christian Palmiero u. a. "Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. Ieee. 2018, S. 1–7 (siehe S. 18).

[20] Aditi Partap und Dan Boneh. "Memory Tagging: A Memory Efficient Design". In: *ArXiv* abs/2209.00307 (2022) (siehe S. 5, 6, 12, 30, 31).

[21] David Patterson und Andrew Waterman. *The RISC-V reader an open architecture atlas*. San Francisco, CA: Strawberry Canyon LLC, 2017. ISBN: 9780999249116 (siehe S. 24).

[22] David A Patterson und John L Hennessy. *Computer organization and design RISC-V edition*. 2. Aufl. The Morgan Kaufmann Series in Computer Architecture and Design. Oxford, England: Morgan Kaufmann, 2021 (siehe S. 25).

[23] Bruce Schneier. *Applied cryptography: Protocols, Algorithms and Source Code in C*. en. 20. Aufl. Nashville, TN: John Wiley & Sons, März 2015 (siehe S. 37).

[24] Kostya Serebryany u. a. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX Annual Technical Conference*. 2012 (siehe S. 13).

[25] Kostya Serebryany u. a. "Memory Tagging and how it improves C/C++ memory safety". In: *ArXiv* abs/1802.09517 (2018) (siehe S. 5, 8, 9, 15).

[26] Chengyu Song u. a. "HDFI: Hardware-assisted data-flow isolation". In: *2016 IEEE Symposium on Security and Privacy (SP)*. Ieee. 2016, S. 1–17 (siehe S. 18).

[27] Jeff Vander Stoep und Chong Zhang. *Queue the Hardening Enhancements*. Letzter Zugriff am 31.01.2024. 2019. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (siehe S. 1).

[28] Gregory T Sullivan u. a. "The dover inherently secure processor". In: *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. Ieee. 2017, S. 1–5 (siehe S. 18).

[29] László Szekeres u. a. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy* (2013), S. 48–62 (siehe S. 5).

[30] *Vivado ML Standard Enterprise Edition*. Letzter Zugriff am 16.02.2024. 2024. URL: https://www.xilinx.com/products/design-tools/vivado/vivado-ml-buy.html (siehe S. 44).

[31] Andrew Waterman und Krste Asanovíc. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation. 2019 (siehe S. 10, 18, 27, 33).

[32] Andrew Waterman, Krste Asanovíc und John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. RISC-V Foundation. 2021 (siehe S. 27).

[33] Samuel Weiser u. a. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V". In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019) (siehe S. 19).

[34] Wei Xu, Daniel C. DuVarney und R. C. Sekar. "An efficient and backwards-compatible transformation to ensure memory safety of C programs". In: *Sigsoft '04/fse-12*. 2004 (siehe S. 4, 5).

[35] Adam Zabrocki u. a. *RISC-V Pointer Masking proposal*. Techn. Ber. Letzter Zugriff am 06.09.2023. 2021. URL: https://github.com/riscv/riscv-j-extension/blob/master/pointer-masking-proposal.adoc (siehe S. 18).

Erklärung

„Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann“.

Ort Datum Unterschrift