TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Alexander Kemnitz
Fakultät Informatik Institut für Technische Informatik, Professur für VLSI-EDA

Modellierung eines MultiCore-Prozessors auf Basis der Simulationsplattform RUBICS

Zwischenvortrag zur Diplomarbeit // 6. September 2018

Inhalt

Übersicht der Simulationsplattform RUBICS
- Aufbau
- Architekturbeschreibungssprache
- Interne Zwischendarstellung
- Bus- und Speichersystem
- Simulationsarten

Wesentliche Architekturunterschiede bei der Erweiterung zu MultiCore

Mögliche Ansätze zur Erweiterung von RUBICS
- Architekturbeschreibungssprache
- Synchronisation
- Speichersystem

Zusammenfassung der Zwischenergebnisse

Ausblick

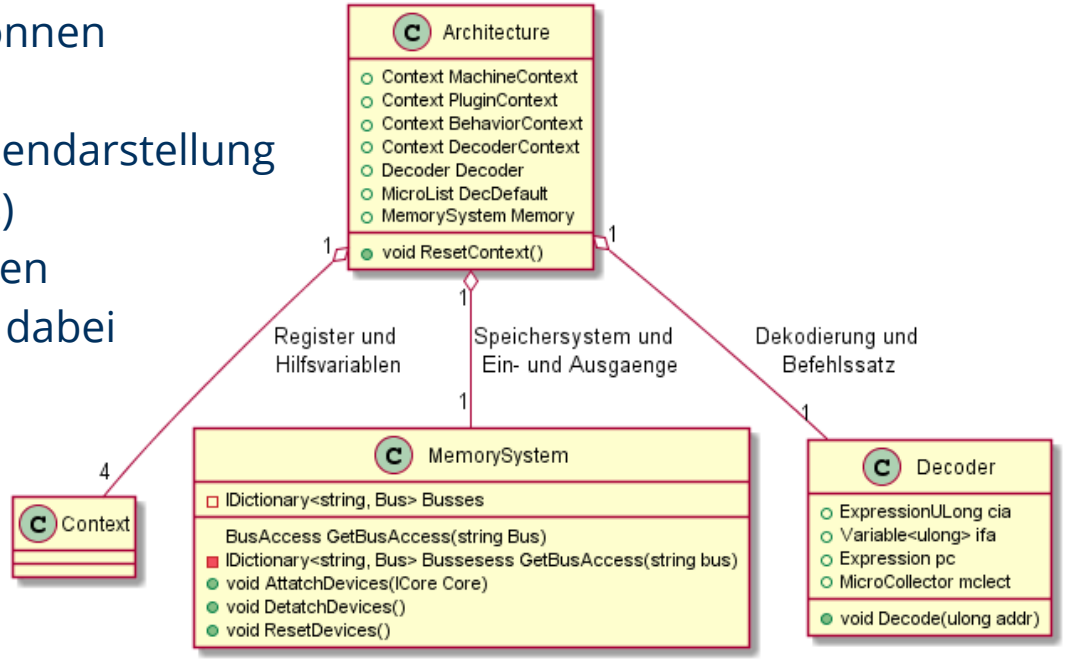
Übersicht
der Simulationsplattform RUBICS

Aufbau
Aktueller Stand von RUBICS

Das Projekt besteht aus 3 Komponenten
- Arcore: Simulationskern
- Rush: Steuerung der Simulation
- Plugins: Erweiterungen, die eingebunden werden können

Der Simulationskern umfasst
- Repräsentation des Befehlssatzes in interner Zwischendarstellung
- Bussystem für Speicher und Externe Geräte (Plugins)
- Zwei Arten Simulation: Interpretieren und Kompilieren

Die Kernklasse des Simlationskerns ‚*Architecture*' enthält dabei
- Registervariablen
- Hilfsvariablen
- Plugins
- Decoder/Befehlssatz
- Bus- und Speichersystem



Architekturbeschreibungssprache
Übersicht

5 Grundbereiche
- Plugin-/Importdefinitionen
- Busse (Speicher und I/O-Geräte)
- Prozessorzustand (Register und Flags)
- Zustandsübergangs-Decoder
- Zustandsübergangs-Verhalten

Parser und Resolver wandeln die Architekturbeschreibung in die interne Zwischendarstellung (im weiteren IR) und Speichersystem um

```
import example.plugin;
...
Bus <BusId> { ... // Bus für Speicher und I/O-Geräte
}
Context { ... // Prozessorzustand
}
Decoder { ... //
}
Behaviour { ...
}
```

Architekturbeschreibungssprache
Busdefinition

Definition der Grundeigenschaften des Bus
- Minimale Zugriffsgröße
- Bytereihenfolge
- Definition der Zugriffsschlüsselwörter

Definition der angeschlossenen Geräte
- Verwendbar sind RAM und importierte Plugins
- Jedem Gerät wird ein Adressraum zugeordnet

```
import <importId0>;
bus <busId> {
    unit = 8;
    endian = little;
    acccess { //definiert die verschiedenen
        ...         //Zugriffsgrößen (z.B. word)
        <accessId> = 4;}
    devices {
        ram ram0 {...}
        <importId0> example {...}
    }
}
```

Interne Zwischendarstellung
Übersicht

Die Dekodierung wird als Liste von Mikrooperationen *(Micro/MicroList) durchgeführt*
- Kontrollflussstrukturen wie Switch-Anweisungen
- Jede Kontrollflussstruktur verweist wieder auf mindestens eine MicroList
- Eine *Micro* für Variablenzuweisungen (*MicroAssign)*
- *Verhaltens*-Schritte/Operationen/Micros* werden mit *MicroDrop* gesammelt

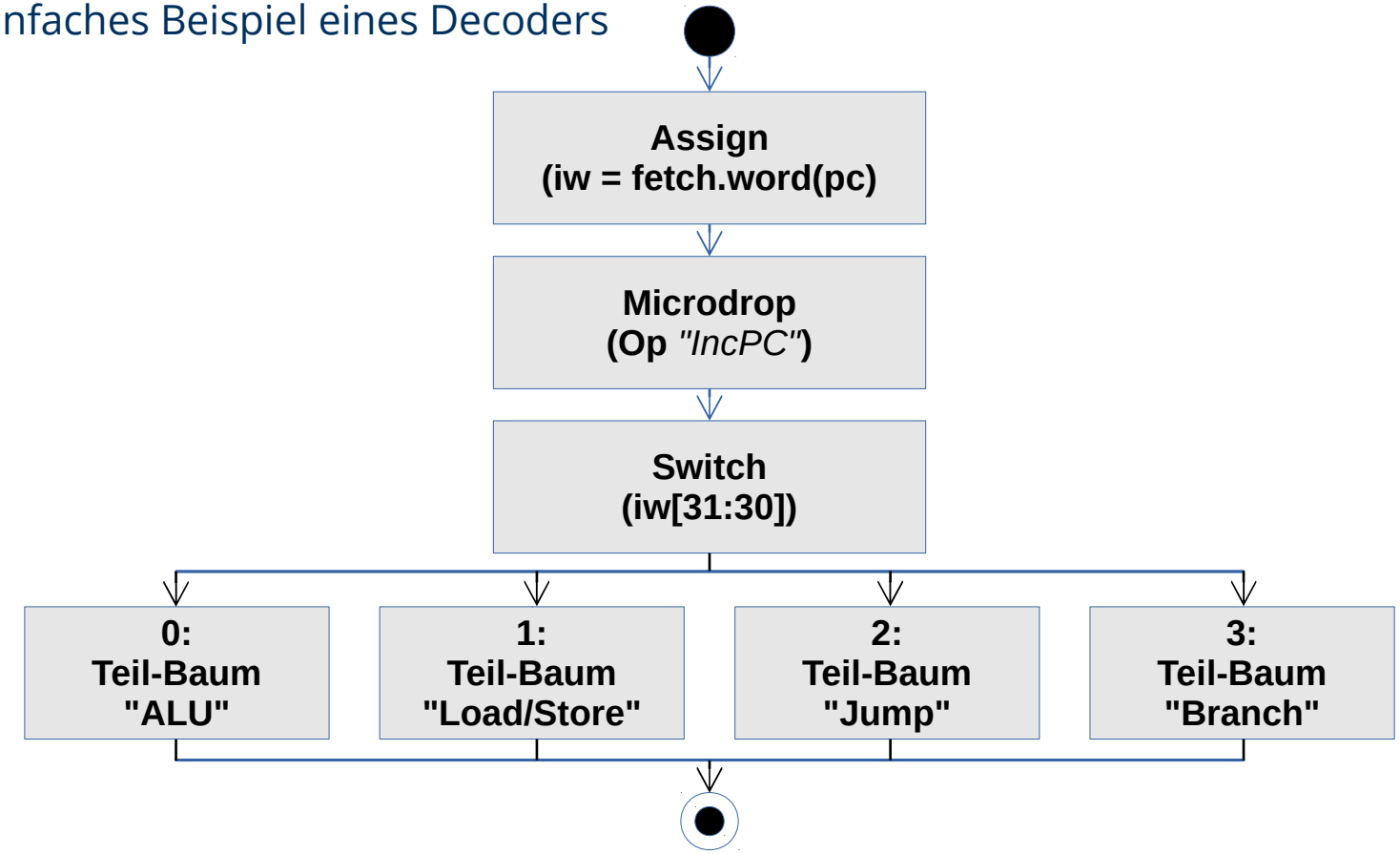
Eine Zuweisung besteht aus einem Ausdrucksbaum *(Expression)/MicroAssign* verweist auf *Expression*-Baum
- Die Meisten *Expressions* repräsentieren Operationen oder Zuweisungen
- Die Knoten sind Variablen

Die Inhalte von Registern und Zwischenergebnisse sind Variablen *(Variable)*

** OF, EX, LS und WB*

Interne Zwischendarstellung
Decoder Micro

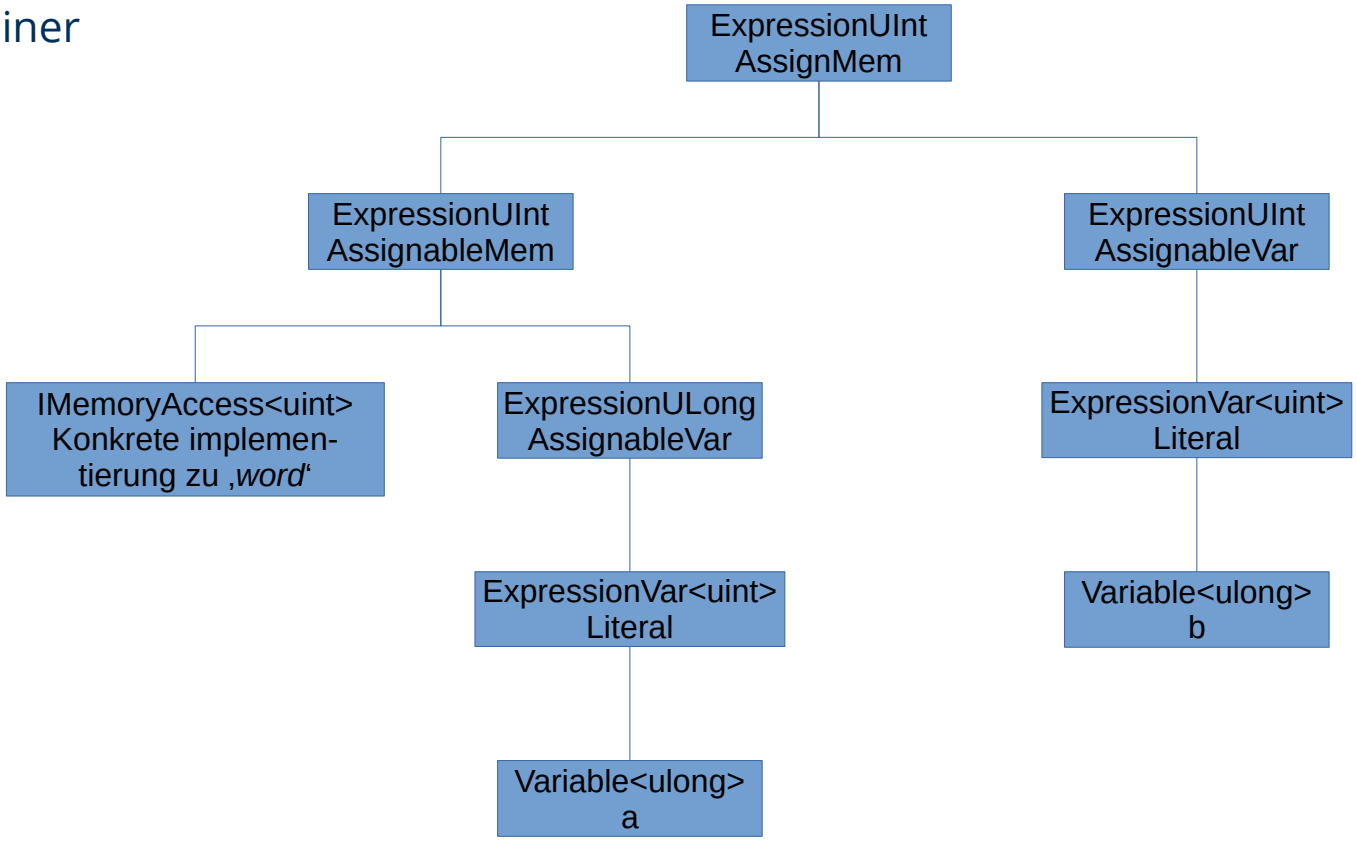
Einfaches Beispiel eines Decoders



Interne Zwischendarstellung
Assign-Expression

Einfaches Beispiel eines *Expression*-Baums einer Speicherzuweisung

- mem.word[a] = b
- ‚*word*' wurde als 32 Bit definiert

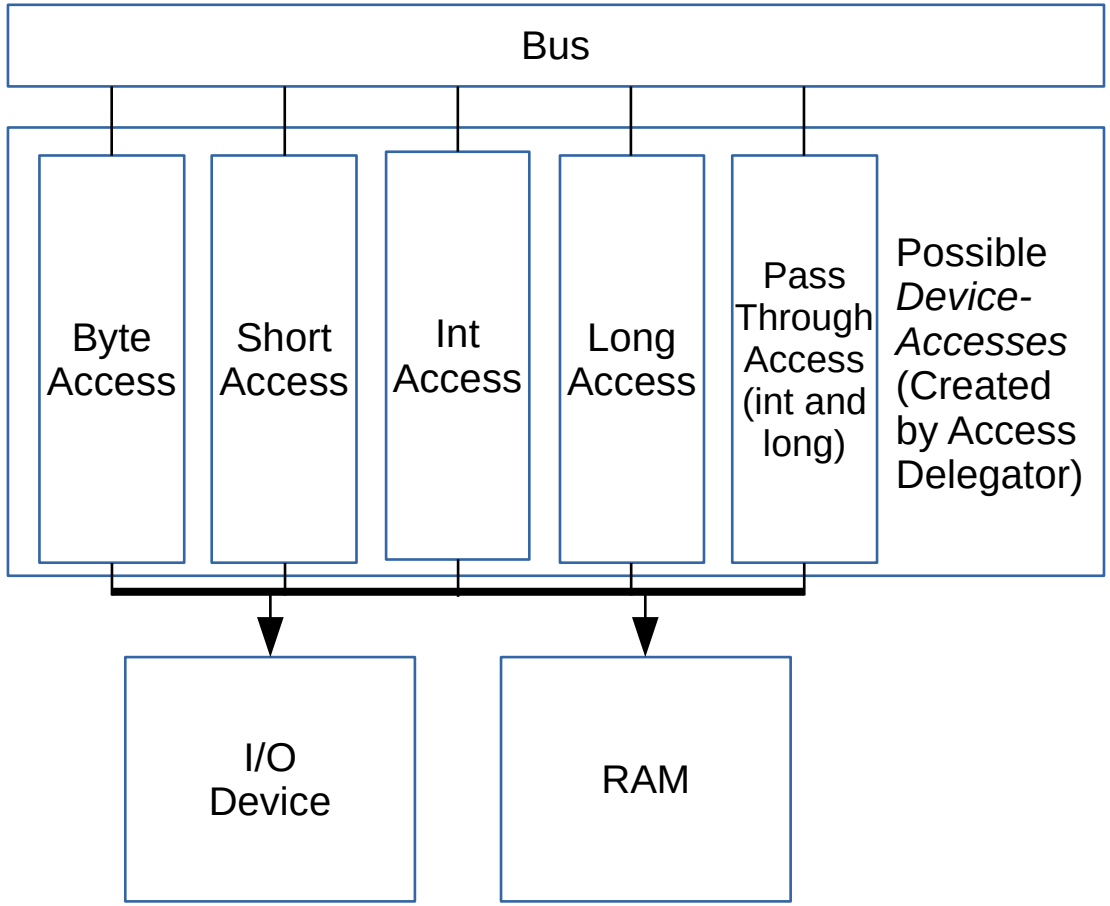


Speichersystem
Busstruktur aus Sicht der IR

Liste von Zugriffen (DeviceAccess)
- Vielfache der *unit (8 oder 32 Byte)*
- Nur 8, 16, 32 und 64 Byte vorgesehen

Liste von angeschlossenen Geräten
- Speicher (RAM)
- I/O-Geräte und andere Plugins
- Ansprechbar durch definierten Adressraum

Zugriffsweg
- *Expression enthält den DeviceAccess*
- *Das Gerät wird durch die Adresse bestimmt*
- *In ADL wird der Bus per <BusId>* ausgewählt und der *DeviceAccess* wird per *<AccessId> ausgewählt*



Simulationsarten
Interpretierende Simulation

Die Interpretierende Simulation ist die Basisvariante der Simulation

Hier wird die Struktur der Internen Zwischendarstellung der *Decoder-Micro vollständig durchlaufen*
- (Wählt an Kontrollflussstrukturen nur den notwendigen Weg)
- Sammelt die Verhaltens-Micros im *MiroCollector*

Gesammelte Micros werden anschließend Nacheinander Ausgeführt

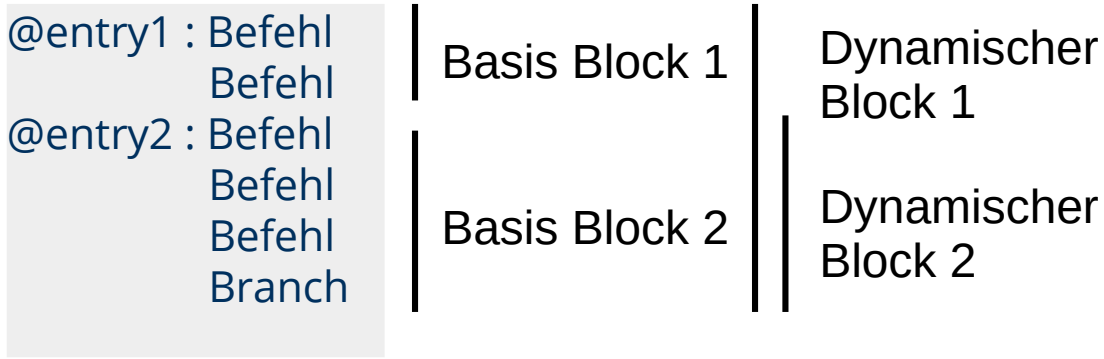
Nachteil
- Wiederverwendete Code-Abschnitte werden jedes mal Dekodiert
- ➔ Hoher Rechenaufwand

Simulationsarten
Compilierende Simulation

Compiliert gesammelte Befehle des *MicroCollectors*
- Erstellt ‚Byte-Code'
- Fasst den Byte-Code eines Blocks zusammen
- Sammelt den Bytecode in einem ‚Cache'

Teilt den Code in Blöcke ein
- Trennung nach Verzweigungen
- Ein Dynamischer Block ist der längste mögliche Weg nach einem Einstiegspunkt

Nachteil
- Hoher Speicherbedarf



=> Zusammenspiel der beiden Simulationsarten zur Optimierung von Speicherbedarf und Rechenzeit
- Zählen der Häufigkeit
- Umschalten auf Compilierende Simulation, wenn ein Schwellwert erreicht wird

Wesentliche Architekturunterschiede
Bei der Erweiterung zum MultiCore

Erweiterung zum Multicore
Allgemein zu beachten

Cache-Kohärenz
- RUBICS simuliert keinen Cache

Topologie
- Bestimmt Zugriffszeiten
- RUBICS simuliert nicht Zeit(punkt)-genau

Synchronisationsmechanismen
- Synchronisation via Speicher => atomare Befehle
- Explizite Kommunikation?
- Interrupts und Events?

Multicore
Speichersynchronisation durch atomare Befehle

Zugriffssperren
- Allgemein: Sicherstellung dass nur ein Thread eine Ressource belegt
- MultiCore-Synchronisation: Sicherstellung dass nur ein Prozessor einen Speicherbereich bearbeitet
- Sperrvariable die in atomaren Befehlen ausgelesen und überschrieben werden

Swap
- Liest Alten Wert und schreibt neuen Wert in einem Befehl
- Ausschließlich für Sperrvariablen vorgesehen

Exklusive Zugriffe
- Besteht aus Lade- und Schreibbefehl-Paar
- Durch Laden wird Anspruch auf den exklusiven Zugriff auf eine Adresse erhoben
- Schreiben erfolgt nur bei exklusiven Recht
- Man erhält die Information, ob der exklusive Zugriff erfolgreich war
- Normalerweise für Sperrvariablen gedacht

Multicore
Umsetzung des exklusiven Zugriff

Local Monitor
- Gibt Adresse des aktuellen exklusiven Zugriffs an

Global Monitor
- Teilt Speicher in Blöcke
- Jeder Block hat einen exklusiver Besitzer
- Wenn ein exklusiver Besitzer auf den Block exklusiv schreibt, hat er Erfolg (Besitzer wird zurückgesetzt)
- Exklusives Lesen macht zum Besitzer

Mögliche Ansätze
zur Erweiterung von RUBICS

Erweiterung zum Multicore
Teilaufgaben

Erweiterung der Architekturbeschreibung
- Strukturelle Veränderungen der ADL
- Busstruktur
- Atomare Befehle

Umsetzung der Synchronisation

Anpassung des Speichersystems

Architekturbeschreibungssprache
Gesamtansatz zur Erweiterung der ADL

Trennung von Strukturbeschreibung und Befehlssatzbeschreibung
- Wiederverwendbarkeit der Befehlssatzbeschreibung
- Übersichtlichkeit

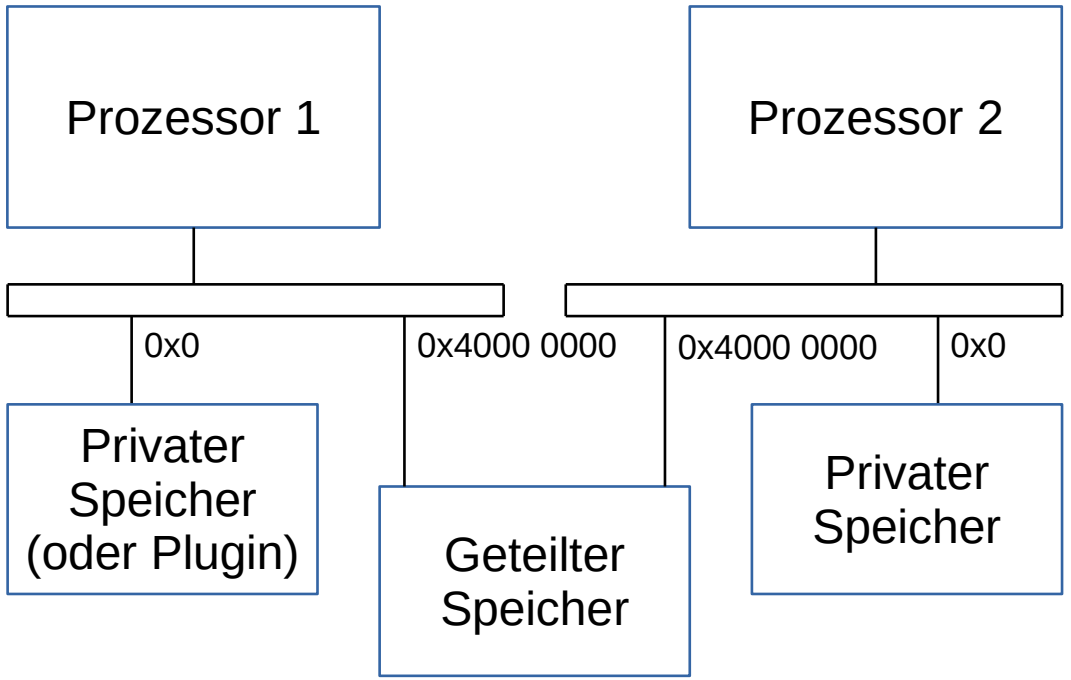
Strukturbeschreibung
- Vernetzung von Prozessoren mit gemeinsamen und privaten Speichern (und Plugins)

Befehlssatzbeschreibung
- Core-spezifisch
- Synchronisationsmöglichkeiten/Synchronisationsbefehle

Architekturbeschreibungssprache
Strukturbeschreibung

Mögliche Ansätze zur Beschreibung von MultiCore

- Definition einer Umgebung für geteilte Busgeräte
- Verwenden des vorhandenen ‚*connect*' für geteilte Geräte
- Bus-zu-Bus-Verbindungen um geteilte Geräte auf geteilten Bussen zu definieren
- Bereitstellen lokaler Bereiche:
  - Zusammenfassung von Prozessorgruppen
  - Darstellungshilfe bei Debugging
  - kein funktioneller Wert



Beispiel einer zu beschreibenden Struktur

Busgeräte Geräte sind meist Speicher, können aber auch Plugins sein

Architekturbeschreibungssprache
Strukturbeschreibung – Geteilte Geräte

```
shared {
   ram sharedMem {
      size = 0x40000000;  }
   … }
…
bus bus0 {
   devices {
      shared ram sharedMemory{
         base = 0x40000000 ;
         size = 0x40000000 ;}
      …
   }
}
```

Beispiel für Definition einer *Shared*-Umgebung

```
bus bus0 {
   …
   devices {
      ram sharedMem {
         base = 0x40000000 ;
         size = 0x40000000 ;}
…
bus bus1 {
   …
   devices {
     connect ram bus0.sharedMem{
         base = 0x40000000 ;
         size = 0x40000000 ;} …
```

Nutzung der bereits vorhandenen *connects*

```
bus sharedBus {
   ….
   devices {
      ram sharedMem {…}
….
bus bus0 {
   ….
   devices {
      map sharedBus {
         …}//adressraum Infos
   …
   }
}
```

Busverkettung

Architekturbeschreibungssprache
Strukturbeschreibung – Gruppierung von Prozessoren

```
cluster cluster0 {
   core cpu0 implements armv4 {...}
   ...
   core cpu3 implements armv4 {...}
}

cluster cluster1 {
   core cpu0 implements armv7 {...}
   ...
}
```

Beispiel für Gruppierungen von Prozessoren

```
shared {
   ram globalSharedMemory  {...}
   ...}

cluster cluster0 {
   shared {
      ram localSharedMemory {...}
      ... }
   core cpu0 implements armv7 {...}
   ...
}
```

Kombination eines *Clusters* mit der S*hared*-Umgebung zur Erstellung von einer lokalen und einer globalen Umgebung für gemeinsame Geräte

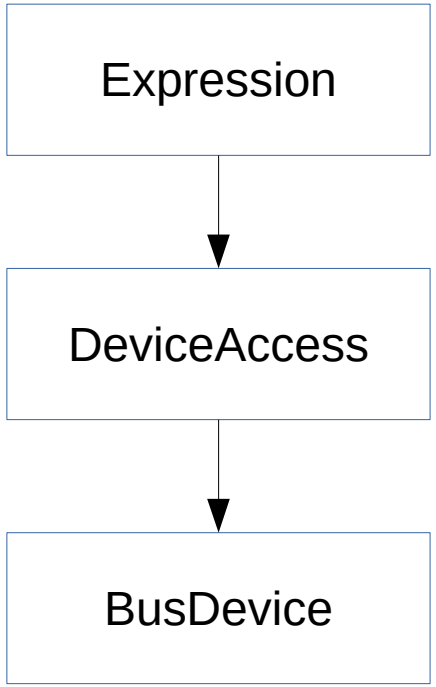
Architekturbeschreibungssprache
Befehlssatzerweiterung um Synchronisationmechanismen

Synchronisation über Speicherbefehle
- Swap : Erweiterung der Sprache um einen Befehl
- Exklusive Zugriffe: viele Realisierungsmöglichkeiten
- Die beiden Befehle sind aufeinander abbildbar
- ➔ Zunächst eine allgemeine Betrachtung für Synchronisation
- ➔ ADL Wird erst danach behandelt

Zu betrachtende weitere Mechanismen
- Interrupts
- Events
- Explizite Kommunikation

Synchronisation
Atomare Zugriffe – Mögliche Ansatzpunkte

Beteiligte Klassen bei einem Speicherzugriff durch einen Befehl



Umsetzen mit Lock-Variable in C#

BusDevice
- Vorteil: Einfachste Umsetzung
- Nachteil: Externe Busgeräte / Plugins sind selbst für Zugriffssperren zuständig

DeviceAccess
- Vorteil: Einmalige Definition im *IAccessDelegator*
- Nachteil: Bei *PagingAccessDelegato*r erhöhter Rechenaufwand oder mögliche Zusammenfassung mehrerer Busgeräte

Expression (Micro)
- Vorteil: Zusätzliche Möglichkeiten, wie ein Schlüsselwort Critical
- Nachteile: - Umfangreicher Lock
  - potenziell viel längere Lockzeiten

Synchronisation
Zeitabschätzungen für Zugriffssperren

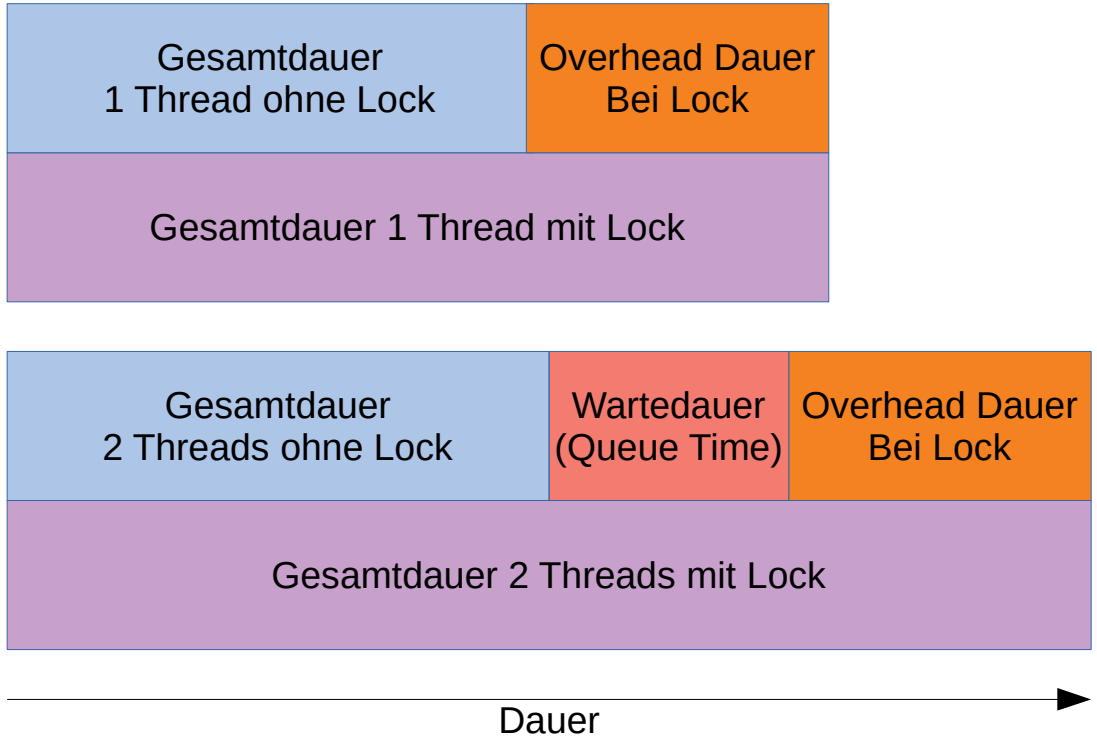
Szenario
- Einfache Lese- und Schreibzugriffe verwenden Locks
- Einfache Vektoraddition
- 2 Lade- und 1 Schreibbefehl, 6 Arithmetische Operationen und eine Verzweigung pro Iterationsschritt
- Ausführung in interpretierender und compilierender Simulation
- 1 Thread (Verktorgröße 4096) und 2 Threads (Vektorgröße 8192)
- Arbeiten auf einem gemeinsamen Speicher
- Vorhandene Architektur: ArmV4

Verglichene Zugriffsperren:
- Keine Sperre
- Sperre im Speicher (Busgerät)
- Sperre beim *DeviceAccess*
- Sperre in der Zugrifffs-*Expression* (nur Interpretierende Simulation)

Synchronisation
Definitionen von Messgrößen und Annahmen

Overhad
- Zusätzlicher Rechenaufwand für Sperren und Freigabe

Wartedauer (Queue Time)
- Dauer die Prozesse durch Zugriffssperren nicht arbeiten



Mehrzeit durch Zugriffssperren
Interpretierende Simulation

Aufnahme der Messwerte
- Mittelwert über 10 Messungen je 50 Durchläufe der Vektormultiplikation
- Messung je mit nur Child- nur Parent-Thread und beiden

Zusammenfassung
- Der Overhead durch *Expression-Lock* ist unerwartet viel höher
- Der Anteil Queue Time ist bei *DeviceAccess-Lock* unerwartet höher

Lock	Overhead	Queue Time	Mehrzeit
Device	2,01 %	1,61 %	3,62 %
Access	2,29 %	1,98 %	4,26 %
Expression	4,84 %	1,68 %	6,xx %

Tabelle: Erhöhung Dauer, wenn 3 von 10 Operationen sperrende Speicherzugriffe sind

Mehrzeit durch Zugriffssperren
Interpretierende Simulation - Hochrechnung

Lock \ N	10	30	100
Device min	0,761%	0,204%	0,056%
Device max	1,43%	0,278%	0,063%
Access min	0,913%	0,248%	0,068%
Access max	1,68%	0,333%	0,076%
Expression min	0,911%	0,225%	0,059%
Expression max	1,97%	0,343%	0,070%

Tabelle: Geschätzte Erhöhung der Rechendauer bei 1 von N Atomare Zugriffen mit minimalen und maximalen Schätzwert

Annahmen für die Hochrechnungen

- Der Overhead für Locking ist unabhängig von Lese-, Schreib- und Lese und Schreibbefehle
- Der Overhead ist Liniear zur Anzahl der Sperrenden Befehle
- Der Dauer, die ein Atomare Befehl sich in einem Kritischen Abschnitt befindet beträgt mindestens die Dauer eines L/S-Befehls und maximal die Dauer von 2 L/S-Befehlen
- Die Queue Time wächst Quadratisch mit dem Zeitanteil von Kritischen Abschnitten

Mehrzeit:

- Die Zeit, die DualCore-Simulation durch das Verwenden von Zugriffssperren länger benötigt als ohne die Verwendung von Zugriffssperren

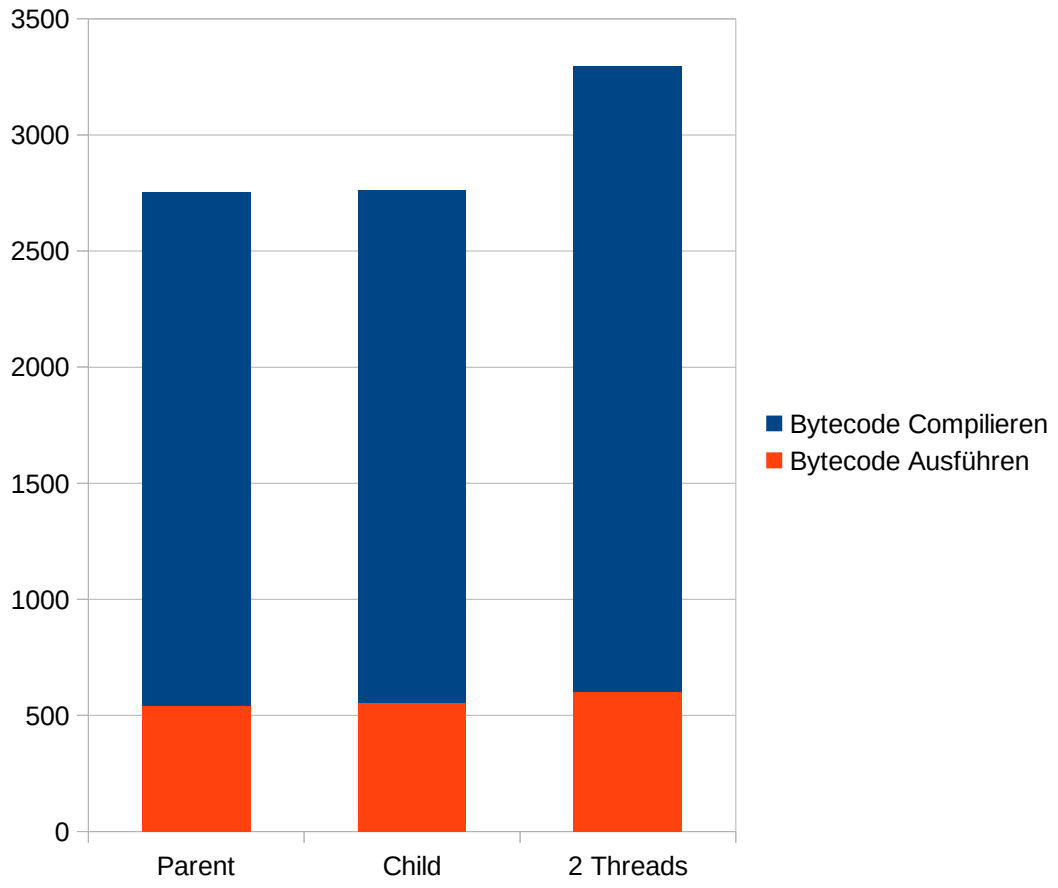
Mehrzeit durch Zugriffssperren
Basisdauer Compilierende Simulation ohne Zugriffssperren

Aufnahme der Messwerte

- Mittelwert über 10 Messungen je 2500 Durchläufe der Vektormultiplikation
- Pro Durchlauf und Thread 1 Bytecode-Generierung (Compile)
- Messung je mit nur Child- nur Parent-Thread und beiden

Messung Ohne Lock

- Selbst bei 4096 Schleifendurchläufen verursacht die Bytecodegenerierung 80% der Dauer
- Der Erhöhte Zeitaufwand für 2 Threads beträgt ca. 20% (ca. 10% bei der tatsächlichen Laufzeit)



Mehrzeit durch Zugriffssperren
Overhead und Busy Waiting in der Kompilierenden Simulation

Overhead durch den Vergleich von Singlethread
- Zeitanteil relativ zu ohne Lock (1 Thread)
- Der relative Overhead ist deutlich größer als in der interpretierenden Simulation
- Der absolute Overhead beträgt ca. ¼ der Interpretierenden Simulation
- Kaum unterschied zwischen den Lockingvarianten

Queue Time
- Zeitanteil relativ zu Ohne Lock / 2 Threads
- Die Absolute Dauer ist vergleichbar mit der Interpretierenden Simulation
- Die Relative Zeit ist um die Größenordnung Faktor 50 Höher
- Device-Lock ist auffällig besser

Lock	Compile	Bytecode Ausführen	Overhead Gesamt
Device	0%	121%	23,8%
Access	0%	122%	24,2%

Overhead durch Zugriffssperren

Lock	Anteil Bytecode-Ausführzeit	Anteil Gesamtzeit
Device	345%	63,0%
Access	404%	75,8%

Queue Time des Simulators durch Zugriffssperren

Mehrzeit durch Zugriffssperren
Abschätzung für Atomare Zugriffe in der Compilierenden Simulation

Dieselben Annahmen für die Hochrechnung wie bei der interpretierenden Simulation

→ Zwischen N=32 und N=100 wird die Erhöhung der Rechendauer durch Locking bei atomaren Zugriffen vernachlässigbar

Lock / N	10	32	100	320
Device (min)	5,73%	0,573%	0,0627 %	0,00742 %
Device (max)	22,7%	2,23%	0,233%	0,0240 %
Access (min)	6,71%	0,668%	0,0725 %	0,00836 %
Access (max)	26,6%	2,61%	0,272%	0,0278 %

Tabelle: Geschätzte Erhöhung der Rechendauer bei 1 von N Atomare Zugriffen mit minimalen und maximalen Schätzwert

Mehrzeit durch Zugriffssperren
Zusammenfassung

Vergleich der beiden Simulationsarten

- In der compilierenden Simulation ist die Erhöhung der Dauer deutlich auffälliger
- Wenn weniger als eine von 100 Operationen ein atomarer Befehl ist, ist die Mehrzeit durch Zugriffssperren vernachlässigbar
- Zugriffssperren sind für die Sicherstellung eines korrekten Programmablaufs nicht zu vermeiden
- Vor- und Nachteile von den Ansatzpunkten *DeviceAccess*- und *BusDevice*-Lock werden weiterverfolgt
- *Expression-Lock* wird ausgeschlossen

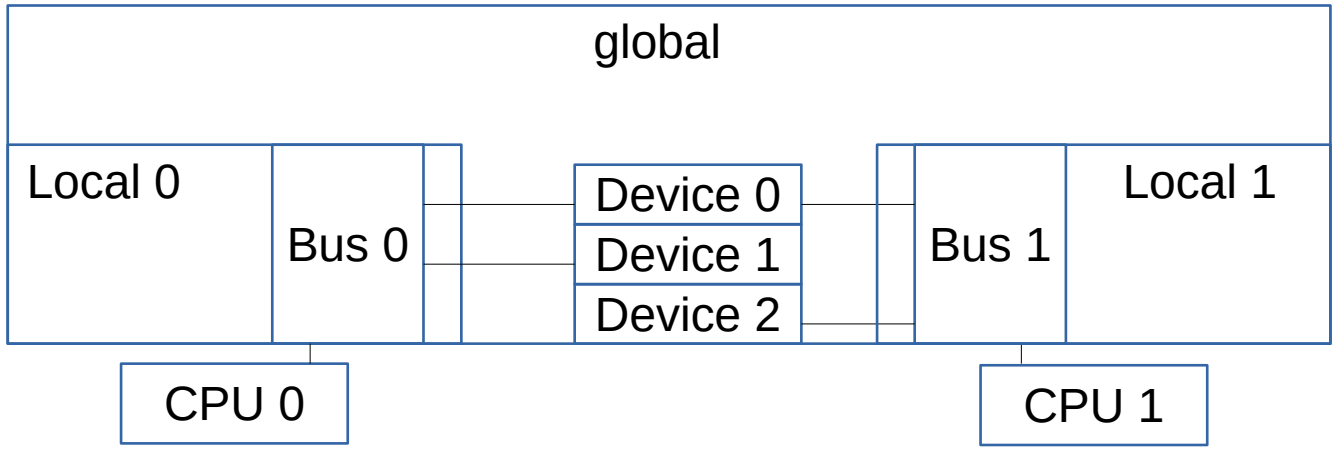
Schätzung des Speedup einer DualCore-Simulation

- Ergibt sich aus Speedup auf realer Hardware, Verlust durch Threading Overhead auf dem Host-Rechner und Verlust durch Zugriffssperren
- Verlust durch Zugriffssperren bei 1 Sperre bei 100 Befehlen ist deutlich kleiner 1%
- Verlust durch Threading Overhead beträgt ca. 3%

Speichersystem
Konzepte

Trennung nach Aufgabenbereichen
- Organisation/Einbinden der Geräte (Attatch, Detatch, Reset Devices)
- Zugriffe (Bus->DeviceAccess, BusAccess)

Trennung nach Prozessorzugehörigkeit
- Ein Speichersystem für jeden einzelnen Prozessor
- Ein Gesamtspeichersystem in dem alle Busse vorhanden sind



Zusammenfassung und Ausblick
Zum Stand der Zwischenpräsentation

Zusammenfassung
Der Bisherigen Arbeit

Spracherweiterung
- Trennung nach Strukturbeschreibung und Befehlssatz
- Verschiedene Konzepte zur Strukturbeschreibung
  - Voraussichtlich werden die Shared Umgebung und die Gruppierungs-Umgebung eingeführt
- Befehle für Atomare Zugriffe
  - bisherige Betrachtung entspricht einem *swap*-Befehl

Umsetzung Atomarer Befehle
- C#-Object-Lock
- Ansatz im *Access* oder Im *BusDevice* selbst werden weiterverfolgt

Speichersystem und Busstruktur
- Voraussichtlich Trennung in globales und Core-bezogenes Speichersystem

Ausblick
Was sind die nächsten Schritte

Erstellen von 1 oder 2 Atomaren Grundbefehlen
- Swap und exklusiver Zugriff
- Umsetzung im Code
- Vergleich der zwei Ansätze Sperre bei *Acccess* oder im *BusDevice*

Erstellen der Busstruktur für Mehrere Prozessoren
- Trennung von Prozessor bezogenen und globalen Aufgaben von *Memorysystem*
- Vergleich der Konzepte für geteilten Speicher und geteilte Busgeräte

Überführung in ein Testbeispiel
- Matrixmultiplikation auf 2 CPUs (n CPUs)

Evaluation der besten Ansatzkombination

Festlegung der Architekturbeschreibung

Anpassen von Parser und Resolver

Vollständige Implementierung eines DualCores mit Testprogramm

Anhang

Testprogramm
Für die Zeitmessung bei atomaren Zugriffen

Assembler

4: @entryPoint: LOAD WORD R5 [r2 + #0]

8: LOAD WORD R6 [r3 + #0]

12: ADD R7 R6 R5 #0

16: STORE WORD R7 [r4 + #0]

20: ADD R0 R0 #1

24: ADD R2 R2 #4

28: ADD R3 R3 #4

32: ADD R4 R4 #4

36: ADD R8 R0 R1 #0 / set flags

40: BRANCH ON NEGATIV @entryPoint

Literaturverzeichnis

Th. Frank. Untersuchung zu JIT-Basierter Binary Translation für die Prozessorsimulation auf Basis von .net CLR Frameworks. 2015

J. Smith, R. Nair. Virtual Machines. Versatile Platforms for Systems and Processes. 2005

A. Tannenbaum, J. Goodman. Computerarchitektur. 2001

ARM. Arm Architecture Reference Manual. Stand 2016

G. van den Braak et al. Simulation and Architecture Improvements of Atomic Operations on GPU Scratchpad Memory. 2013

M. Scott. Shared-Memory Synchronisation. 2013