TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

Implementierung und Evaluation eines Thumb- und Thumb2-Architekturmodells für den Befehlssatzsimulator Jahris

Großer Beleg

Ronald Rist

s0200738@mail.zih.tu-dresden.de

Dresden, 24.04.2013

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Inhalt

1 Einleitung

2 Grundlagen

- *Thumb*
- *Jahris*

3 Implementierung

- *Besonderheiten des Thumb-Befehlssatzes*
- *Befehlssatzumschaltung und Sprünge*
- *IT-Block*
- *Lookahead-Logik*

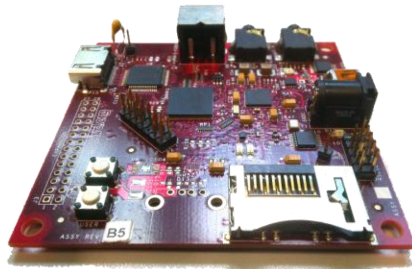
4 Auswertung

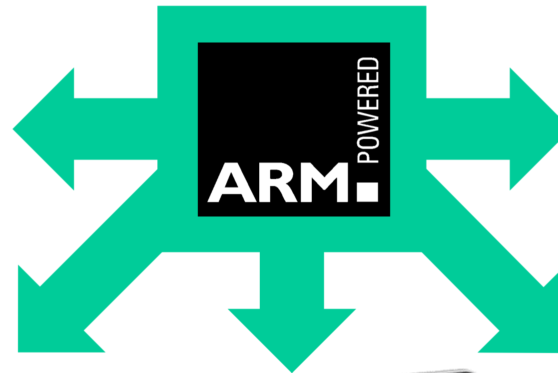
5 Zusammenfassung und Ausblick

6 Ausgewählte Quellen

1 Einleitung

1.1 Hintergrund

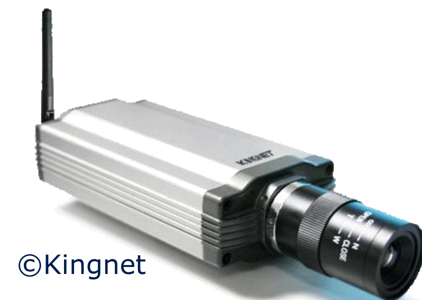

BeagleBoard Project



©Sony

©D-Link

©Samsung


©Kingnet

1.1 Hintergrund (Fortsetzung)

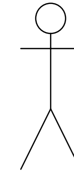
- ARM Ltd. (Acorn/Advanced RISC Machines) vertreibt (heute) ausschließlich IP
- Lizenzierung durch andere Hersteller, Integration in deren Chips/SoCs
- hohe Verbreitung in eingebetteten Systemen, mobilen Geräten, sonstigen low-cost und low-power Anwendungen

- (heute) Unterstützung verschiedener Befehlssätze
  - ARM
  - **Thumb/Thumb2**
  - Jazelle/ThumbEE
  - div. Erweiterungen: *VFP (Floating-Point), Advanced-SIMD, …*

1.2 Motivation

Test/Debug auf Zielsystem

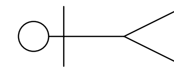
Einsehbarkeit?



Entwickler

Test/Debug im Simulator

Geschwindigkeit?



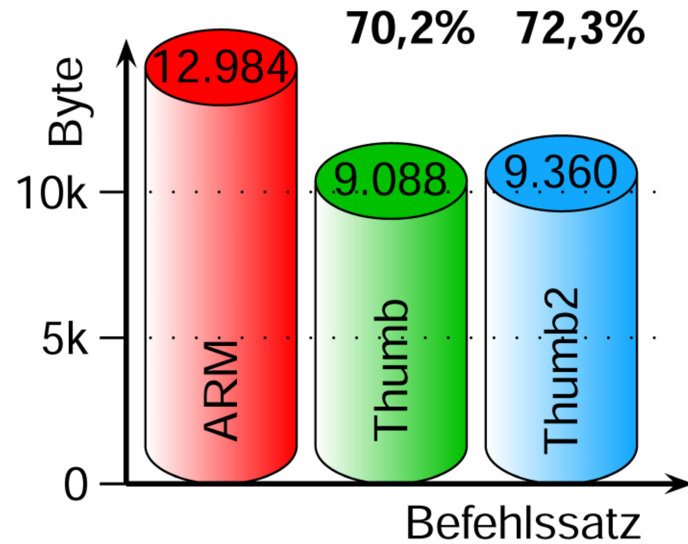
Entwickler

- Vereinbarkeit von Inspizierbarkeit und hoher Simulationsperformanz?
- Thumb in bisherigen Arbeiten kaum berücksichtigt
- Abbildbarkeit der Konzepte von Thumb im Befehlssatzsimulator Jahris?
- Konkurrenzfähigkeit zu anderen Systemen?

2 Grundlagen

2.1 Thumb – Zielstellung

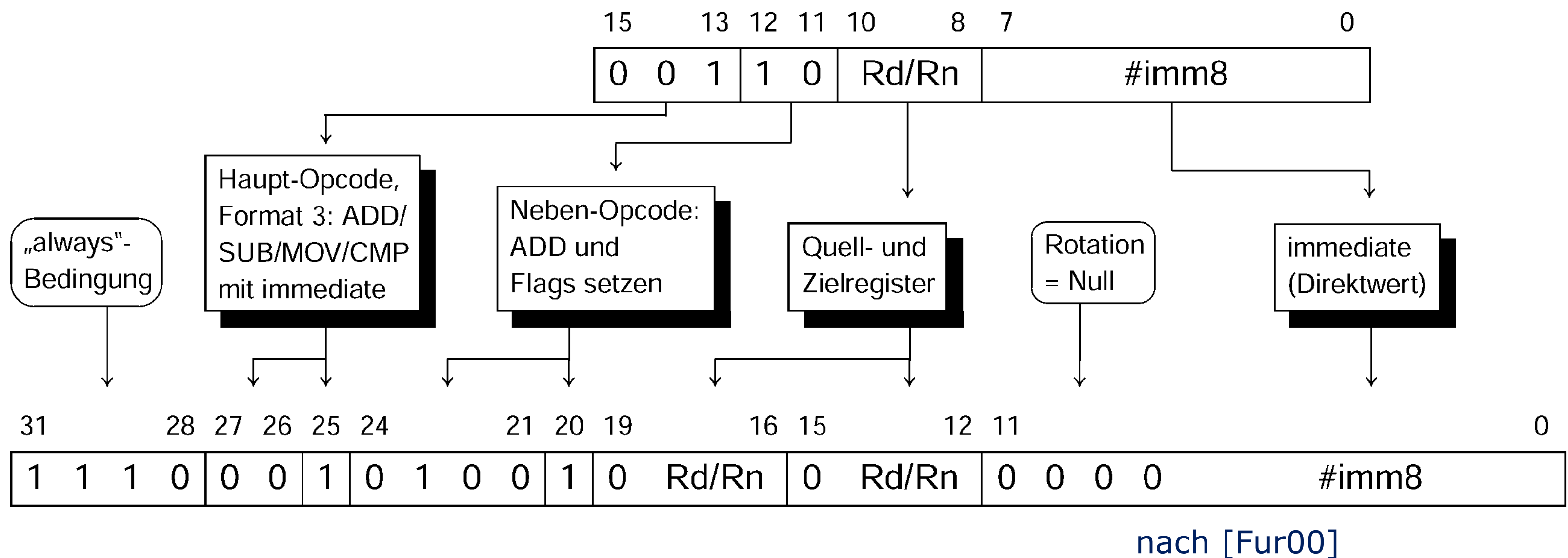
- komprimierte Untermenge des klassischen ARM-Befehlssatzes ab ARMv4T
- Erhöhung der Codedichte: vornehmlich 16 Bit statt 32 Bit
- 70% der Codegröße von ARM (140% der Instruktionsanzahl)
- Möglichkeit der Umschaltung zw. den Befehlssätzen
- ab Thumb2 (ARMv7): große Zahl neuer 32-Bit Instruktionen



Memory Footprint
Tetris Demo `.text`-Sektion
(Sourcery Toolchain, Optimierungsstufe O2)

2.1 Thumb – Physische Interpretation

- Abbildung (Dekompression) der Thumb-Befehle auf ARM-Instruktionen zur Verarbeitung über normale Prozessor-Pipeline



nach [Fur00]

- weniger Information resultiert in Implizität: keine Kondition, kein Shift des Direktwertes, kein Einfluss auf Flag-Setzung(!)

TECHNISCHE UNIVERSITÄT DRESDEN

2.1 Thumb – Aufbau

15					10	9	8 … 6	5 … 3	2 … 0
0	0	0	1	1	0	A	Rm	Rn	Rd

(1) ADD|SUB Rd,Rn,Rm

15					10	9	8 … 6	5 … 3	2 … 0
0	0	0	1	1	1	A	#imm3	Rn	Rd

(2) ADD|SUB Rd,Rn,#imm3

15		13	12 … 11	10 … 8	7 … 0
0	0	1	Op	Rd/Rn	#imm8

(3) <Op> Rd,Rn,#imm8

15		13	12 … 11	10 … 6	5 … 3	2 … 0
0	0	0	Op	#sh	Rn	Rd

(4) LSL|LSR|ASR Rd,Rn,#sh

15					10	9 … 6	5 … 3	2 … 0
0	1	0	0	0	0	Op	Rm/Rs	Rd/Rn

(5) <Op> Rd/Rn,Rm/Rs

15					10	9 … 8	7	6	5 … 3	2 … 0
0	1	0	0	0	1	Op	D	M	Rm	Rd/Rn

(6) ADD|SUB|MOV Rd/Rn,Rm

15			12	11	10 … 8	7 … 0
1	0	1	0	R	Rd	#imm8

(7) ADD Rd,SP|PC,#imm8

15							8	7	6 … 0
1	0	1	1	0	0	0	0	A	#imm7

(8) ADD|SUB SP,SP,#imm7

nach [Fur00]

2.1 Thumb – Aufbau (Fortsetzung)

- hohe Codedichte erzeugt vergleichsweise viele unterschiedliche Formate, resultierend in aufwändigerer Dekodierung
- nur 2-Adress-Format und begrenzter Registerzugriff

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15
niedere Register										SL	FP	IP	SP	LR	PC

limitierter Zugriff

nach [Fur00]

2.1 Thumb – IT-Block (Thumb2)

- neue Variante der bedingten Ausführung:
  `IfThen`-Instruktion macht die nächsten ein bis vier Befehle konditional
- Bedingung kann alterniert werden

15 – 12	11 – 8	7	6	5	4	3	2	1	0
1 0 1 1	1 1 1 1	c_a	c_b	c_c	P_1	P_2	P_3	P_4	1
		first cond.				mask			
		IT[7…0]							

`IfThen`-Instruktion

$x = 2^{nd}$ *cond.* $= c_a\, c_b\, c_c\, P_2$

$y = 3^{rd}$ *cond.* $= c_a\, c_b\, c_c\, P_3$

$z = 4^{th}$ *cond.* $= c_a\, c_b\, c_c\, P_4$

$P_n = P_1 \longrightarrow n^{th}$ *cond.* $= 1^{st}$ *cond.*

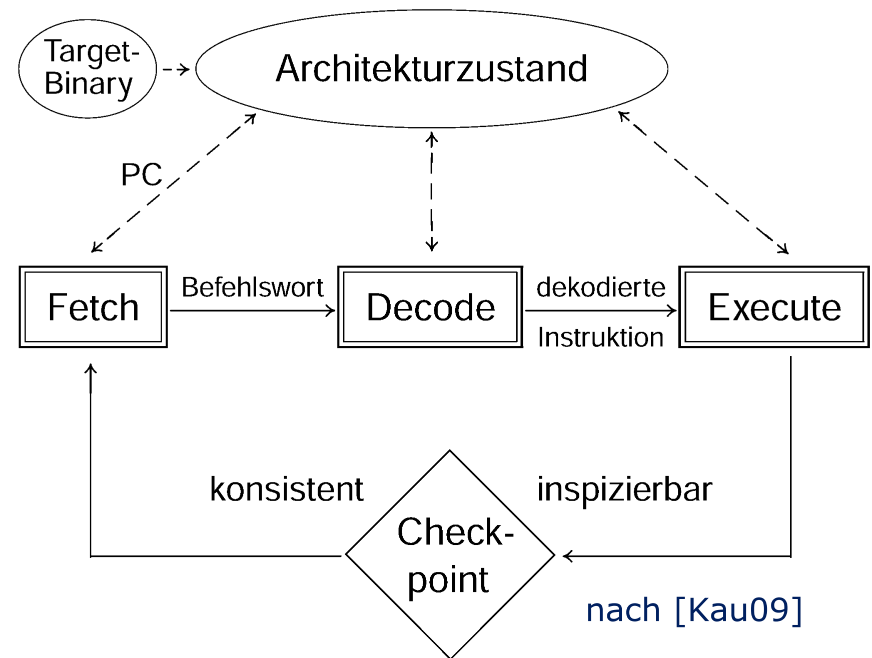
$P_n = \neg P_1 \longrightarrow n^{th}$ *cond.* $= \neg 1^{st}$ *cond.*

IT Bits						
[7 ... 5]	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**	**Beginn eines IT-Blocks**
$c_a\, c_b\, c_c$	P_1	P_2	P_3	P_4	1	IT-Block der Länge 4
$c_a\, c_b\, c_c$	P_1	P_2	P_3	1	0	IT-Block der Länge 3
$c_a\, c_b\, c_c$	P_1	P_2	1	0	0	IT-Block der Länge 2
$c_a\, c_b\, c_c$	P_1	1	0	0	0	IT-Block der Länge 1

nach [Arm11]

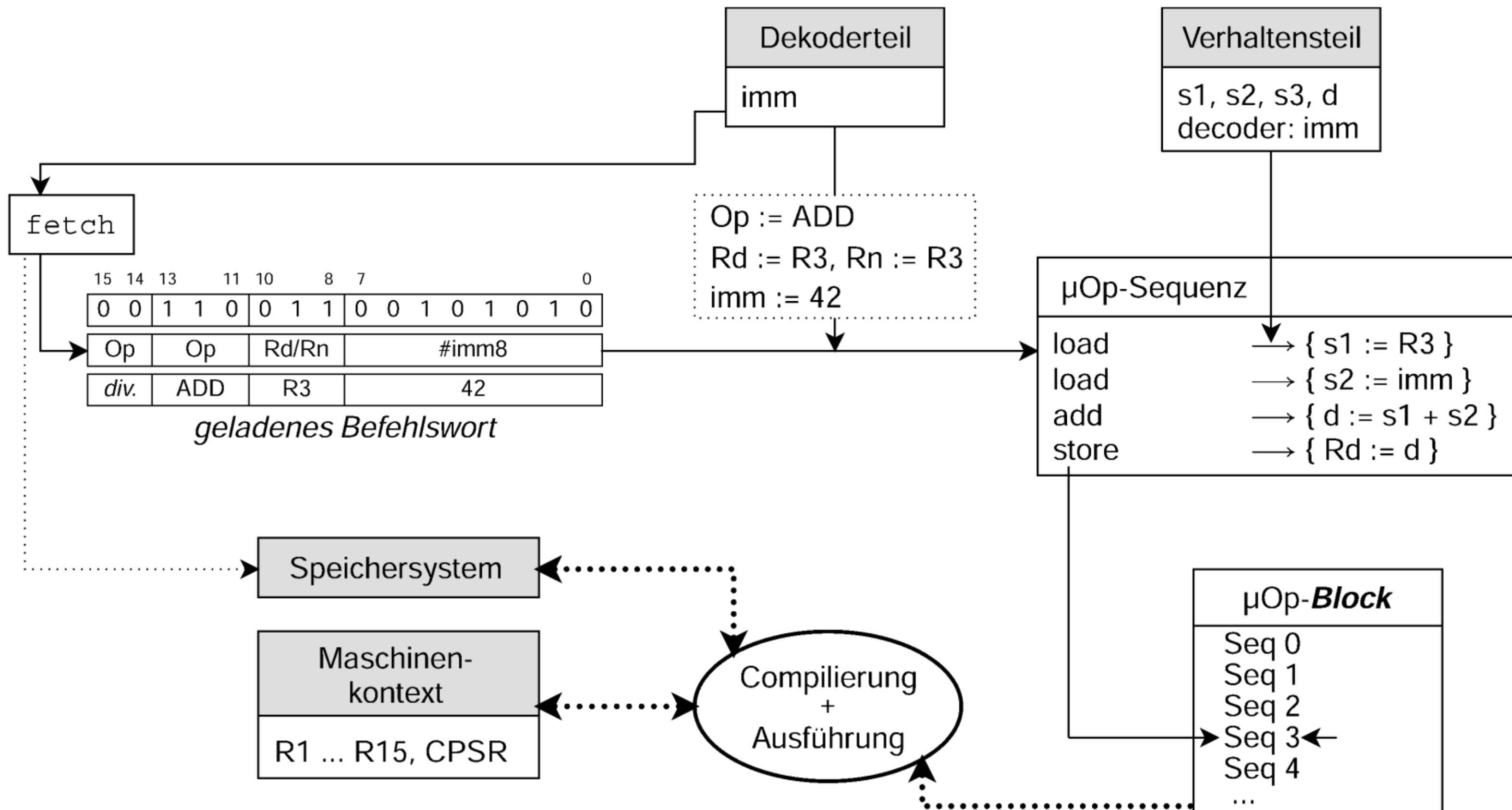
2.2 Jahris – retargierbarer Befehlssatzsimulator

- von Marco Kaufmann in Java entwickelt, plattformunabhängig
- Simulation verschiedenster Architekturen/Befehlssätze, diese werden in HPADL beschrieben
- Beschränkung auf befehlsgenaue Simulation
- ausgelegt auf hohe Performanz durch Just-in-Time-Compilierung größerer Codeabschnitte
- interpretierende schrittweise Simulation ebenfalls möglich



nach [Kau09]

2.2 Jahris – HPADL



2.2 Jahris – HPADL (Fortsetzung)

- Maschinenkontext, Dekoderteil (Zerlegung in Mikrooperationen), Verhaltensteil (Beschreibung der Mikrooperationen)

- strikte Trennung von Dekodierung und Ausführung, sowohl zeitlich als auch bei Implementierung
- nur zur Ausführungszeit (somit nur in µ-Ops) Zugriff auf Maschinenzustand
- zur Dekodierzeit (in Dekoderteil) nur aktuelles Befehlswort bekannt

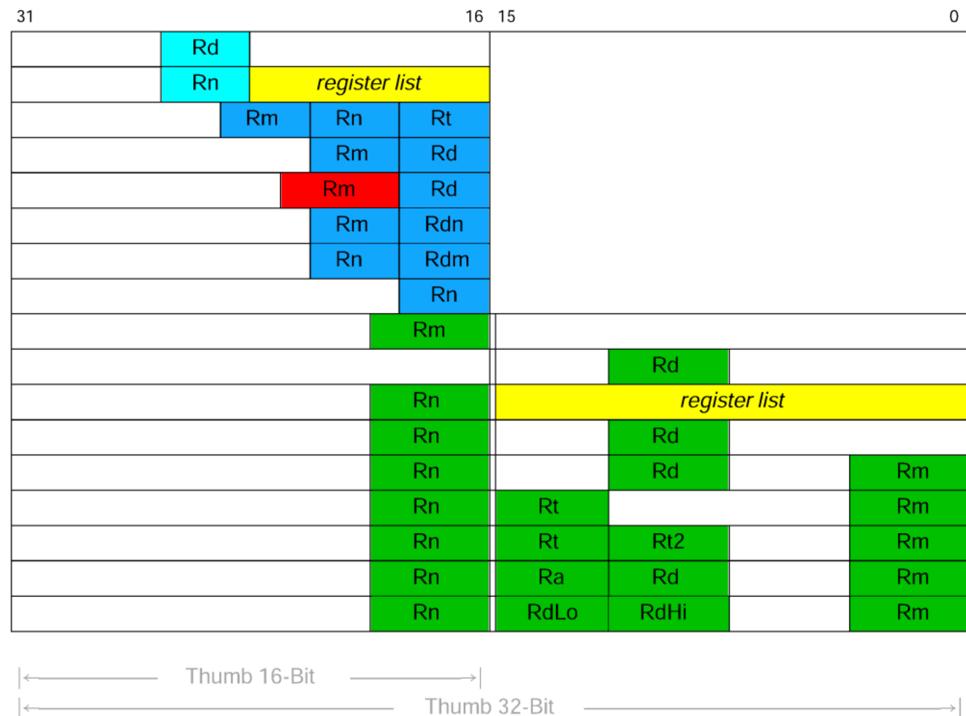
- bei Ausführung liegt nur mehr dekodierte und übersetze µOp-Folge vor
- Zusammenfassung aufeinanderfolgender Befehle zu größeren µOp-Blöcken

3 Implementierung

3.1 Besonderheiten des Thumb-Befehlssatzes

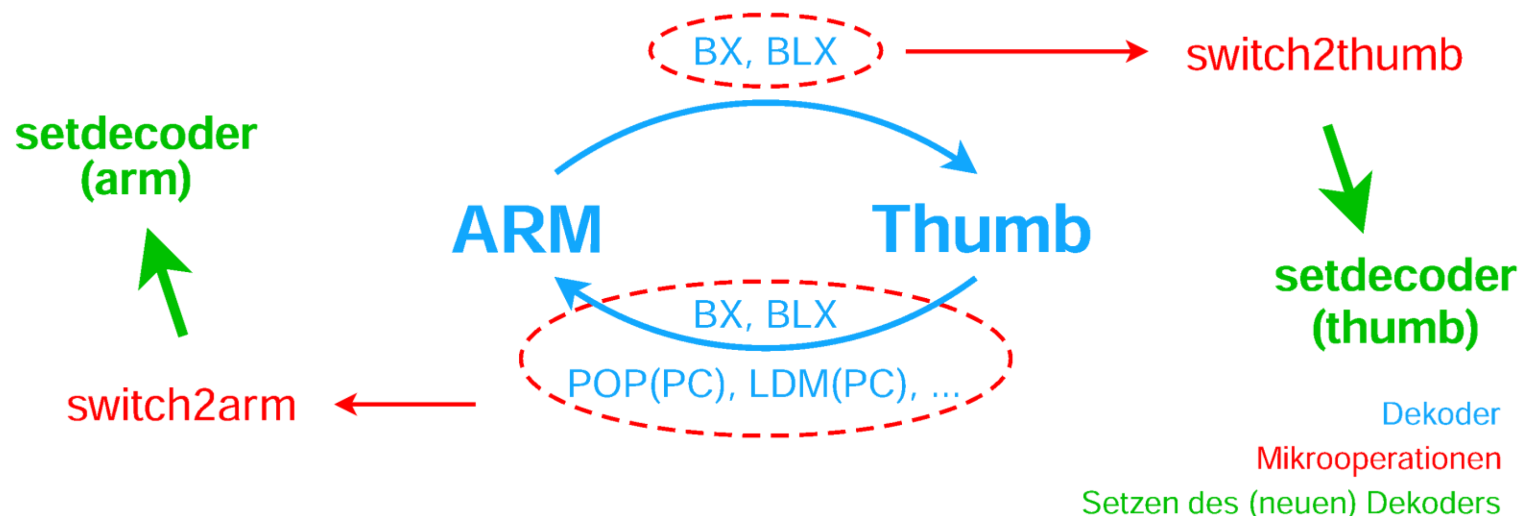
- unregelmäßiger Aufbau erfordert komplexen Dekoderbaum
- stufenweise Dekodierung von OpCode und Operanden

- Entscheidung: hohe Treue zur Referenz-beschreibung zu-gunsten *Wartbarkeit*



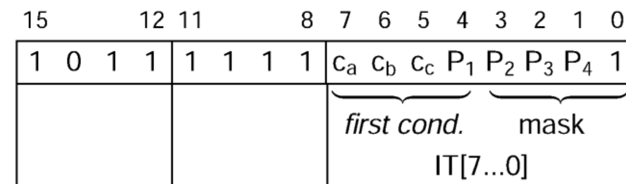
3.2 Befehlssatzumschaltung und Sprünge

- Umschaltung zwischen ARM und Thumb auf physischen Prozessoren vorgesehen
- nicht trivial: unsaubere Varianten vorgesehen
- Anpassung von Jahris/HPADL erforderlich: `setdecoder(<ident>)`
- nicht trivial: Zielbefehlssatz u.U. erst zur Ausführung bekannt



3.3 IT-Block (Thumb2)

- sprengt RISC-Konzept: Umsetzung eines Befehls nicht mehr nur von diesem selbst abhängig
- Konflikt mit Befehlsstromdekodierung in Jahris; Realisierung mit vorhandenen Möglichkeiten zu performanzlastig
- Anpassung von Jahris/HPADL erforderlich: *statische Dekodervariablen* **`static <type> <ident>`**
- diese ermöglichen performante Realisierung: dekoderinterner Indikator/Zähler, Zwischenspeichern der Kondition(en)
- sind problematisch bei Debugging, *Kompromisslösung*



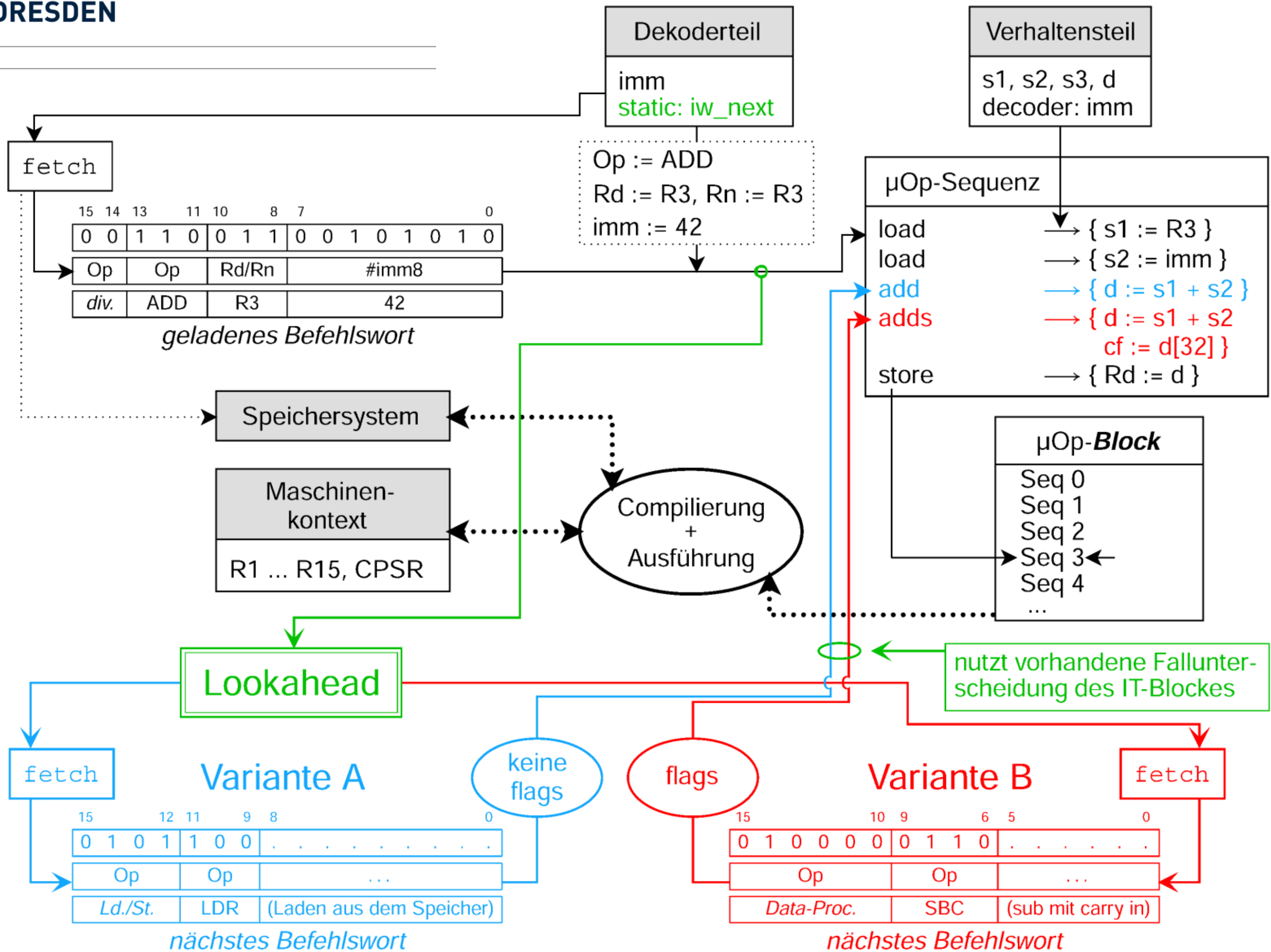
IfThen-Instruktion

IT Bits						
[7 ... 5]	[4]	[3]	[2]	[1]	[0]	Beginn eines IT-Blocks
c_a c_b c_c	P_1	P_2	P_3	P_4	1	IT-Block der Länge 4
c_a c_b c_c	P_1	P_2	P_3	1	0	IT-Block der Länge 3
c_a c_b c_c	P_1	P_2	1	0	0	IT-Block der Länge 2
c_a c_b c_c	P_1	1	0	0	0	IT-Block der Länge 1

nach [Arm11]

3.4 Lookahead-Logik

- erkennt und überspringt verzichtbare Flagsetzungen
- vier Beobachtungen:
  - 16-Bit Thumb Instruktionen setzen stets Flags mangels S-Bit (Indikator)
  - **Ausnahme:** o.g. IT-Block → Fallunterscheidung dennoch benötigt
  - Flags arithmetischer 16-Bit Instruktionen werden *seltenst* ausgewertet
  - arithmetische 16-Bit Instruktionen kommen jedoch insg. *sehr häufig* vor
- Vermutung: Performanzanstieg durch Auslassung unnötiger Mikrooperationen
- Funktionsweise: „pre-"`fetch` der Folgeinstruktrion, teilweise Dekodierung (Flags benötigt?), Zwischenspeicherung
- Ausnutzung der durch die IT-Block-Umsetzung sowieso vorhandenen Fallunterscheidung sowie der statischen Dekodervariablen
- zu-/abschaltbar über Environment-Variable
- nutzt ausschließlich vorhandene Möglichkeiten von HPADL


Dekoderteil
imm
static: iw_next
Verhaltensteil
s1, s2, s3, d
decoder: imm
fetch
Op := ADD
Rd := R3, Rn := R3
imm := 42
15 14 13 11 10 8 7 0
0 0 1 1 0 0 1 1 0 0 1 0 1 0 1 0
Op Op Rd/Rn #imm8
div. ADD R3 42
geladenes Befehlswort
µOp-Sequenz
load → { s1 := R3 }
load → { s2 := imm }
add → { d := s1 + s2 }
adds → { d := s1 + s2
cf := d[32] }
store → { Rd := d }
Speichersystem
Maschinen-
kontext
R1 ... R15, CPSR
Compilierung
+
Ausführung
µOp-Block
Seq 0
Seq 1
Seq 2
Seq 3
Seq 4
...
Lookahead
nutzt vorhandene Fallunter-
scheidung des IT-Blockes
fetch
Variante A
keine
flags
flags
Variante B
fetch
15 12 11 9 8 0
0 1 0 1 1 0 0 . . . .
Op Op ...
Ld./St. LDR (Laden aus dem Speicher)
nächstes Befehlswort
15 10 9 6 5 0
0 1 0 0 0 0 0 1 1 0 . . . . . .
Op Op ...
Data-Proc. SBC (sub mit carry in)
nächstes Befehlswort

3.4 Lookahead-Logik (Forts.)

```
1  // Thumb: Add/Sub reg and imm3
2  operation ADDSUBr3_th {
3    rn_mod = iw[19 to 21];
4    ls1rn_th;
5    $if defined(LOOKAHEAD) $then
6      lookahead;
7      if ((inITcount == 0) && (needflags == 1)) {
8    $else
9      if (inITcount == 0) {
10   $end
11     decode iw[25 to 26] {
12       0: { rm_mod = iw[22 to 24]; ls2rm_th; ADDS; } // ADDreg
13       1: { rm_mod = iw[22 to 24]; ls2rm_th; SUBS; } // SUBreg
14       2: { imm = iw[22 to 24];    LS2_IMM;  ADDS; } // ADDimm3
15       3: { imm = iw[22 to 24];    LS2_IMM;  SUBS; } // ADDimm3
16     }
17   } else { //in case in IT-Block: no flags
18     decode iw[25 to 26] {
19       0: { rm_mod = iw[22 to 24]; ls2rm_th; ADD; } // ADDreg
20       1: { rm_mod = iw[22 to 24]; ls2rm_th; SUB; } // SUBreg
21       2: { imm = iw[22 to 24];    LS2_IMM;  ADD; } // ADDimm3
22       3: { imm = iw[22 to 24];    LS2_IMM;  SUB; } // ADDimm3
23     }
24   }
25   rd_mod = iw[16 to 18];
26   sdrd_th;
27 }
```

3.4 Lookahead-Logik (Fortsetzung)

- Dekodierung der Folgeoperation: Zurücklesen aus statischer Dekodervariable
- ebenfalls problematisch bei Debugging, *Kompromisslösung*

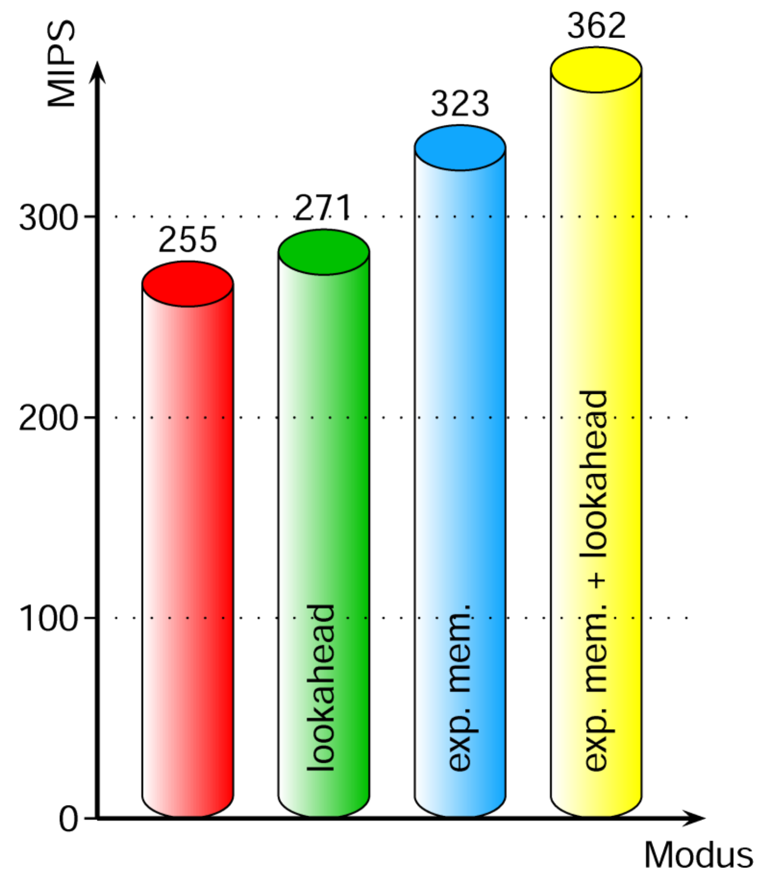
- Möglichkeit des Versagens: Vorausschau der Tiefe 1 war in Praxis zweimal nicht ausreichend

```
311a:    191b         adds              ( 16 Bit; s              → set flags )
311c:    f108 32ff    add.w    f..f     ( 32 Bit; s = no S-Bit → no flags  )
3120:    d236         bcs.n             ( 16 Bit; on carry set → branch    )
```

4 Auswertung

4.1 Performanz von/in Jahris

- Testprogramm Sieve:
  100.000 Iterationen Primzahlsuche
  „Sieb des Eratosthenes“ bis 2^{14}-1,
- 18.130.200.052 Thumb-Instruktionen,
  erzeugt von *gcc* mit Opt.-st. O3
- Testsystem:
  Core 2 Duo E6750 mit 2,66 GHz
- neben Lookahead ebenfalls neues experimentelles ARM-Speichersystem (v. Marco Kaufmann) gestestet



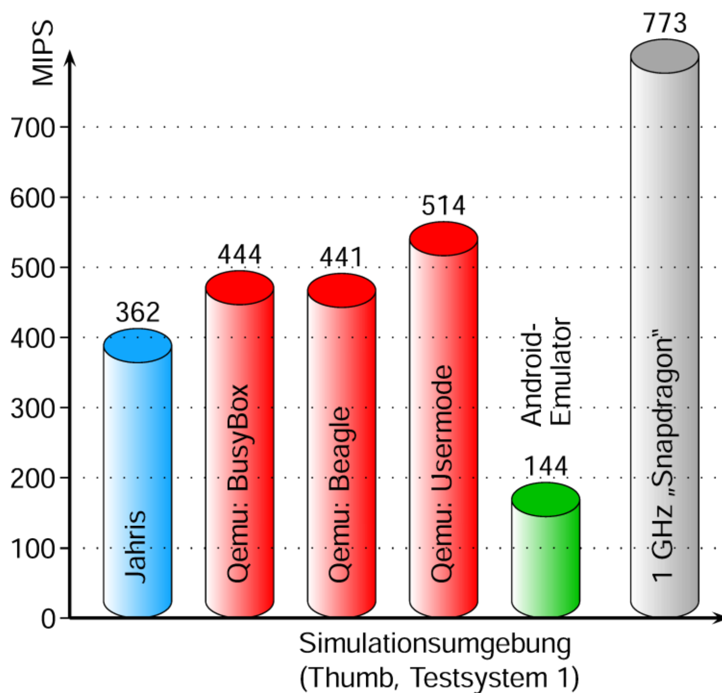
4.2 Vergleich mit anderen Simulatoren

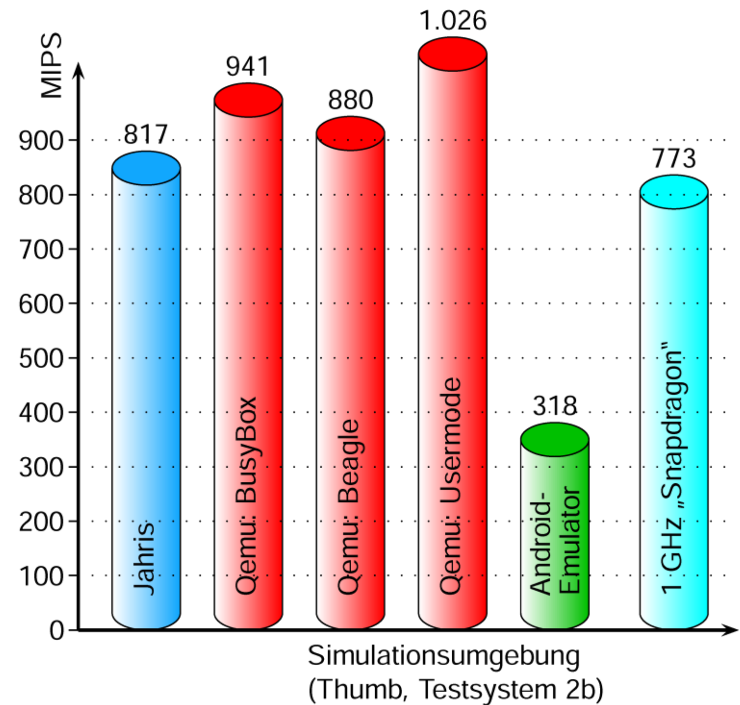
- eingeschränkte Verfügbarkeit von Simulationsumgebungen anderer Arbeiten
  keine Unterstützung von Thumb in anderen Arbeiten
- deshalb primärer Vergleich mit Qemu und Android-Emulator

Qemu	qemu(-system)-arm Version 1.0.50,	
• BusyB		Emulierte CPU: ARM926EJ-S rev. 5 (ARMv5TEJ), Gast-OS: Linux version 2.6.17-rc3 with BusyBox Utilities
• Beagle		Emulierte CPU: Beagle (OMAP3530, Cortex-A8, ARMv7 rev. 3), Gast-OS: Linux version 3.1.0-2-linaro-omap
• Userm		Usermode-Emulation (Emulation ohne Gastsystem)
Android	Version 20.0.3.0, Emulierte CPU: Cortex-A8, Gast-OS: Android 4.1.2 (API 16) / Linux version 2.6.29-gc497e41	
Snapdragon	MK16i mit 1 GHz ARMv7-Prozessor „Snapdragon“, RAM: 512 MB OS: Android 4.0.4 / Linux version 2.6.32.9-perf (gcc version 4.4.3)	

4.2 Vergleich mit anderen Simulatoren (Fortsetzung)

- Rückstand von Jahris ggü. Qemu auf Core i7 deutlich geringer (7-20 vs. 18-30%)
- Android-Emulator wird um Faktor 2,5 überboten





5 Zusammenfassung und Ausblick

- funktionsfähiges Thumb/Thumb2-Architekturmodell wurde erstellt
- Erweiterung: Lookahead-Logik
- Konkurrenzfähigkeit zu anderen Simulationssystemen, Performanz erreicht auf modernem Host Level physischer Prozessoren
- Abbildungsfähigkeiten von HPADL überprüft; ggf. Erweiterungen

- Ausbaumöglichkeiten:
  - Umsetzung weiterer Architektur-Extensionen: Jazelle/Thumb-EE, VFP, …
  - mehr Benchmark-Programme und Untersuchung von Befehlshäufigkeiten
  - ist der Dekoderbaum mit technischen Mitteln optimierbar?
  - Erweiterung der Lookahead-Logik auf Tiefe x (Umgebungsvariable)

6 Ausgewählte Quellen

[Arm11] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition, Errata markup, ARM Limited, 2004-2011, ARM DDI 0406B

[Bel05] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, 2005, USENIX Annual Technical Conference, S. 41-46

[Fur00] Furber, Steve B.: ARM System-On-Chip Architecture, Addison-Wesley, 2000, ISBN 978-0-201-67519-1, insb. S. 189-206

[Kau09] Kaufmann, M.: Erschließung von Just-in-Time-Compilierungstechniken in der Realisierung eines retargierbaren Architektursimulators, TU Dresden, 2009