Die ARM-Architektur - Ein Einblick

Steven Reim

12.11.2010

Zusammenfassung

ARM-Prozessoren sind aktuell in zahlreichen Mobilfunkgeräten und Eingebetteten Systemen vertreten. Zahlreiche verschiedene ARM-Familien sind mit in Unterschiedlicher Kombination implementierten Technologien auf dem Markt und richten sich mit stets verschiedenen Performance-Eigenschaften an unterschiedliche Marktsegmente. Wir werfen einen Blick auf die ARM-Architektur, das Instruction Set und Technologie-Erweiterungen wie Thumb oder Jazelle und wir vergleichen Eigenschaften zwischen ARM- und X86-Prozessoren.

1 Einleitung

Immer öfter hören oder lesen wir in den Medien Neuigkeiten über programmierbare Waschmaschinen, netzwerkfähige Fernsehgeräte oder verschiedenste entwicklungstechnische Fortschritte bei Kraftfahrzeugen. Beinahe im Monatszyklus erscheinen Handys, Handhelds oder MP3-Player, die immer kleiner, immer schneller und immer günstiger werden. Dies alles ist nur realisierbar, weil eingebaute CPUs fortwährend kompakter, energiesparender und effizienter werden.
Eine aktuell weit verbreitete Architektur ist die ARM(Advanced RISC Machine)-Architektur.

ARMs gehören mittlerweile zu den etabliertesten Prozessoren auf dem Mobilfunk-Markt und sind in den meisten Smartphones wie dem iPhone oder verschiedenen Android-Geräten und auch in zahlreichen Embedded Devices verbaut.
Seit der Entwicklung der ersten ARM in den 80er Jahren sind mittlerweile verschiedenste Arten von ARM-Familien entstanden. Sie alle unterscheiden sich stark in ihren Eigenschaften[1] wie Energieverbrauch, Frequenzleistung, Cache-Größe oder dem Vorhandensein verschiedenster Technologie-Erweiterungen und richten sich auch architekturweise an jeweils unterschiedliche Gruppen von Umgebungen. So sollen ARM9, ARM11 und der Cortex-A ihren Platz hauptsächlich in Mobilfunkgeräten und Smartphones. Die Cortex-R-Gruppe soll sich - mit Fokus auf Echtzeitanwendungen - an latenzkritische Systeme richten. Der Hersteller führt hier vor allem die Automobilbranche mit ihren Airbag-, Brems- und Motormanagementsystemen, Speichercontroller, Cameras oder medizinische Geräte auf. Schließlich richtet sich die Cortex-M-Gruppe an das Microcontroller-Segment. Vor allem für Eingebettete Systeme ist diese Gruppe relevant - seien es Waschmaschinen, Taschenrechner oder DVD-Recorder.

Dass bei bei mobilen Endgeräten und Eingebetteten Systemen die Anforderungen an eine Prozessor-Architektur anders ausfallen als bei Server- oder Desktop-Umgebungen, ist offensichtlich. So ist hier die Anforderung an die Performance und Geschwindigkeit nur zweitrangig. Je nach Anforderung ist hier eine kluge Kombination aus Durchsatz und Latenz zu wählen. Eingebettete Systeme agieren in der Regel Single Threaded, weswegen hier ein guter Durchsatz zu wählen ist. MultiThreading-fähige Geräte sollten dagegen Hauptaugenmerk auf die Latenz, also eine faire Verteilung der Systemressourcen an alle Prozesse, gerichtet werden [4].
Vorrangig bei der Verarbeitung zu erwägen sind allerdings Eigenschaften wie Energieverbrauch, Kosten und Sicherheit. Da mobile Geräte in der Regel batteriebetrieben funktionieren, muss eine Architektur gewählt werden, die so energiesparend wie nur möglich rechnet. Um konkurrenzfähig

zu sein, muss auch der Preis des Produktes eines Herstellers so gering wie möglich sein. Dies ist besonders bei Mobilgeräten ein nicht zu verachtender Punkt, da speziell hier der Markt und die Menge an Konkurrenten groß ist.

Im Folgenden möchte ich mich auf eine spezielle Menge von ARM-Architektur-Familien beziehen. Ich werde Unterschiede zwischen diesen Gruppen erläutern und auf die Anwendungsbereiche der Familien eingehen. Es wird einen umfassenden Einblick in die ARM-Architektur und den ARM-Assembler geben, und wir werden verschiedene Technologien wie VFP, Jazelle oder Thumb(-2) begutachten.

2 Die Advanced RISC Machine

2.1 Grundlagen

Das Kürzel ARM steht für *Advanced RISC Machine*. Daraus lässt sich bereits die Architekturgrundlage ableiten: Der RISC - also *Reduced Instruction Set Computer*. Die ARM-Architektur ist also eine 32-Bit RISC-Architektur, welche mehrere Eigenschaften der RISC-Maschine übernommen hat.[4]
So sind beide Architekturen Load-/Store-Architekturen. Was dies genau bedeutet, werde ich im Abschnitt 2.3 genauer beschreiben. Weiters verzichten beide Architekturen auf die Unterstützung *falsch ausgerichteter Speicherzugriffe*. Dieser Support wird erst beim ARMv6 mit Ausnahmen eingeführt. Außerdem wird auf einheitliche 16 x 32 Bit Register Files gesetzt. Die meisten Ausführungen geschehen in beiden Architekturen in single cycles bei grundsätzlich festgelegten 32-Bit-Instruktionen, wodurch vor allem Pipelining und Decoding vereinfacht werden. Allerdings wirkt sich dies auch negativ auf die Code-Dichte aus. Um dem entgegenzuwirken, entstand mitunter die Technologie Thumb (2.4.1). Auf das Instruction Set wird auch im Abschnitt 2.3 weiter eingegangen.

Zusätzlich werden auch einige weitere, nicht RISC-typische, Features genutzt[10]:

- bedingte Ausführung der meisten Instruktionen: Instruktionen besitzen meist ein Conditional Flag. Wenn der Conditional Flag nicht mit dem CPSR (2.3.2) übereinstimmt, wird die Instruktion nicht ausgeführt und entspricht einem NOP.
- Arithmetische Instruktionen modifizieren Statusregister nur, wenn dies ausdrücklich erwünscht ist
- ein 32-Bit Barrel Shifter (siehe auch 2.3.4)
- ein 2-Prioritäten-Level Interrupt System mit verschalteten Register-Bänken (Näheres in 2.3.2)

ARM-Prozessoren sind dank ihrer Einfachheit besonders für Anwendungen mit weniger Rechenbedarf geeignet. Diese Einfachheit ergibt vor allem einen geringen Energieverbrauch bei gleichzeitig relativ geringen Kosten[4]. Aus diesem Grund haben sich ARM-Kerne besonders im mobilen Bereich, beispielsweise bei Smartphones, Handhelds oder MP3-Playern, aber auch im Bereich der Eingebetteten Systeme etabliert.

Hierbei ist zu erwähnen, dass ARM-Produkte keineswegs final sind, also die Großkundenabnehmer durchaus noch die Möglichkeit besitzen, selbst Features und Techniken zu implementieren doer Designänderungen zu erwirken [10].

2.1.1 Unterschiede zur X86-Architektur

Zuallererst muss klargestellt werden, dass beide Architekturen sich an unterschiedliche Zielgruppen richten. Während sich der x86 für ëchte Computer", also PCs, Notebooks, Netbooks oder Server eignen soll, richtet sich der ARM an Eingebettete Systeme und Mobilgeräte. Folglich variieren also auch die Anforderungen an diese Architekturen.

Der architektonische Hauptunterschied der ARM- zur x86-Architektur ist sicherlich der Befehlssatz. Wie bereits erwähnt und der Name es verdeutlicht (2.1), entspricht die ARM einer vollwertigen RISC-Architektur, wogegen der x86 auf einen CISC-Befehlssatz aufbaut. Dies heißt also auch, dass der x86 deutlich mehr Instruktionen offeriert, welche - im Gegensatz zum ARM - nicht einheitlicher Länge sein müssen.

Die Ur-x86-Architektur enthielt 14 16-Bit-Register, wogegen die ARM-Architektur insgesamt 37 32-Bit-Register besitzt, von denen in jedem Modus stets 16 davon angesprochen werden können (siehe auch 2.3.2).
Der x86 bietet Datenspeicherung nur im *Little Endian*-Format an, wogegen - abhängig von einer vorherigen Vereinbarung - bei der ARM sowohl Big, als auch Little Endian angewandt werden können.

Auch die Implementierung von Prozeduren sieht bei der ARM etwas anders aus: Wird die Rücksprungadresse bei Prozeduraufrufen in der x86-Architektur in einen Stack geschrieben, dient bei der ARM-Architektur einfach der Link Register (R14) dafür. Dadurch verkürzt sich zwar die Zeit, die benötigt wird, um von einer zur nächsten Prozedur zu springen, schränkt aber auch die Flexibilität stark ein.
Weiterhin bietet die ARM-Architektur auch das Feature der bedingten Instruktionen, welche in 2.1 bereits erläutert worden sind.

Im Gegensatz zum x86 kann die ARM als Load-/Store-Architektur (2.3) nur vom Speicher lesen und auf den Speicher schreiben. Für direkte Operationen auf Daten des Speichers müssen diese erst in Register geschrieben, dort verarbeitet und wieder zurückgeschrieben werden. Der x86 bietet hingegen deutlich mehr Instruktionen, welche direkt auf dem verwendeten Speicher operieren können.

2.1.2 ARM und Unix

Nachdem sich ARM im mobilen Bereich stark etablierte, zeigten mit der Zeit auch viele Unix-Betriebssystemanbieter Interesse daran, ihr Produkt ARM-tauglich zu machen. So bieten aktuell beispielsweise Ubuntu, Gentoo, Debian oder Slackware für ihre Linux-Distributionen eine ARM-Version an [11, 14, 12, 13]. Aber auch BSD- und Solaris-Distributoren wie FreeBSD und OpenSolaris bieten seit Längerem ARM-Support an [15, 16].

2.2 ARM-Familien und ihre Interessenten

Seit der Vorstellung der ersten ARM-Familie ARM1 1983 sind bereits viele Jahre vergangen. Seitdem wurde auch eine große Zahl von weiteren ARM-Familien auf den Markt gebracht, auf deren verbreitetesten, den ARM7, ARM9, ARM11 und den Cortex nun im Folgenden eingegangen wird. Im Anhang befindet sich die Tabelle 1, welche eine zusätzliche Übersicht über die ARM-Gruppen bieten soll.

Zu einer allgemeinen Bekanntheit verhalf erstmalig der ARM7. Dieser ist auch der erste, welcher die Thumb-Technologie (2.4.1) implementierte. Diese verhalf dem ARM zu einer großen Geschwindigkeitssteigerung bei gleichbleibendem Energieverbrauch, weswegen sich fortan auch große Firmen wie Apple und Nintendo für die ARM interessierten. Der ARM7 ist der erste ARM

mit ARMv4-Architektur (2.3).
Seit dem ARM9 schaffte es ARM in den Großkunden-Mobilfunkmarkt. Mit einer nun fünfstufigen Pipeline (2.3.3) erhielt ARM mitunter Bestellungen von Nokia, Sony Ericsson und LG.
Der ARM11 ist auch heutzutage noch breit vertreten, zum Beispiel fand er Platz im iPhone 3G, im HTC Hero oder in Microsofts MP3-Player Zune. Erstmalig schaffte es ein ARM auf über 1000 DMIPS (*Dhrystone Million Instructions per Second* ist die Einheit des allgemeinen Dhrystone-Benchmarks). Erstmalig sind hier Thumb-2 (2.4.2) und Jazelle DBX (2.4.3) implementiert worden.
Die momentan (Stand 2010) aktuellste ARM-Familie ist der Cortex. Hier ist in drei Gruppen zu unterteilen, Den Cortex-A, Cortex-R und Cortex-M. Der Cortex-A steht für *Application* und soll auf den ARM11 aufbauen, also seinen Platz in Mobilgeräten finden. Dies ist auch geglückt. So ist der Cortex-A8 aktuell mitunter im iPhone 4 und 3GS eingebaut. Der neuere Cortex-A9 kann bis zu vier Kerne enthalten und erreicht bereits bis zu 2 GHz bei einer Performance von 2,5 DMIPS/(Mhz pro Kern). Cortex-R steht für *Real Time Application*, angestrebte Marktsegmente sind hier beispielsweise die Automobilindustrie (bspw. Airbag-Kontrolle) oder die Netzwerktechnik (Router). Cortex-M steht für *Microcontroller*. Hier stehen vor allem die Kosten im Vordergrund und weniger die Leistung, weswegen auf eine hohe Frequenz und Technologien wie Jazelle oder DSP (2.4.5) verzichtet wurde.[1, 10]

2.3 Die ARM-Architektur

Der ARM besitzt eine Load-/Store-Architektur. Dies bedeutet, dass Daten zuerst in die Register geladen werden müssen, bevor sie von der ALU verarbeitet werden können. Anschließend müssen sie dann über den Bus wieder zurück in den Speicher geschrieben werden. Der ARM kann also nicht direkt mit den Daten im Speicher umgehen.
Um dies als Entwickler realisieren zu können und um Programme auf ARM-Grundlage zu entwickeln, gibt es die ARM-Assemblersprache. Diese entspricht nahezu vollständig dem ARM-Instruction Set.

Im Folgenden soll anhand eines kurzen Assembler-Beispiels die Funktionsweise der Load-/Store-Architektur erklärt werden.

```
1 LDR R1, =0x400
2 LDR R2, =0x404
3 ADD R3, R2, R1
4 STR R3, =0x408
```

Listing 1: ARM Assembler Beispielcode

Zu Beginn müssen also erst die Daten aus den exemplarischen Speicher-Adressen 0x400 und 0x404 geladen werden und in die Register R1 und R2 gespeichert werden. Erst dann kann die ALU die Daten benutzen. Im hiesigen Fall werden also die Daten aus R1 und R2 wieder geholt, um im Anschluss die Summe beider in R3 zu packen. Zum Schluss wird das Ergebnis aus R3 genommen und im Speicher unter der Adresse 0x408 hinterlegt.

Wie das Laden und Speichern von Daten und Instruktionen nun tatsächlich in der Praxis abläuft, hängt zusätzlich auch vom Grundaufbau der Architektur ab. Bis einschließlich dem ARM7 wurde auf die von Neumann-Architektur gesetzt. Seit dem ARM9 wechselte man zur Harvard-Architektur. Der Vorteil hier war, dass - im Gegensatz zur von Neumann-Architektur, welche einen gemeinsamen Bus für Daten und Instruktionen besitzt - die Harvard-Architektur dank der zwei getrennten Busse das Holen von Instruktionen und Daten simultan erlaubt. Dies bewirkt natürlich einen erheblichen Geschwindigkeitsschub, allerdings mit dem Nachteil, dass die Harvard-Architektur keinen selbstmodifizierenden Code ermöglicht. Zusätzlich steigen aufgrund des zweiten Busses die Hardware-Kosten.

2.3.1 Prozessor-Modi

Bis zum ARMv4 besaß die ARM sechs verschiedene Operationsmodi [5, 2]:

- *User* (Eingeschränkter Modus, unter welchem die meisten Prozesse arbeiten)
- *FIQ* (Fast Interrupt Request; wird bei einem hochprioritärem Interrupt aufgerufen)
- *IRQ* (Interrupt Request; normalprioritärer Interrupt-Modus)
- *Supervisor* (wird bei einem Reset oder einem ausgeführtem Interrupt betreten)
- *Abort* (um Memory Access Violations zu verarbeiten)
- *Undef* (für undefinierte Instruktionen)

Seit ARMv4 existiert zusätzlich noch ein siebter Modus:

- *System* (Vollpriviligierter Modus, welcher auf die gleichen Register wie der User-Modus zugreift)

2.3.2 Der ARM-Registersatz

Der ARM besitzt insgesamt 37 Register, jeder ist 32 Bit groß. Jeder Modus kann auf genau 16 Register plus dem CPSR (Current Program Status Register) zugreifen. Priviligierte Modi erhalten zudem Zugriff auf den SPSR (Saved Program Status Register). Diese 16 Register sind im Einzelnen:[5, 2]

- R0 bis R12 sind Universalregister
- R13 ist der Stack Pointer, welcher für Push und Pop die Stack-Adresse liefert
- R14 ist der Link Register, welcher bei Unterprozeduraufrufen die Rücksprungadresse bereithält
- R15 ist der Program Counter; enthält je nach Modus die Adresse der folgenden oder der übernächsten Instruktion. Deswegen wird er auch Instruction Pointer genannt.

Für eine genaue Übersicht, welcher Modus auf welche Register zugreifen kann, befindet sich anhängend eine Grafik des *ARM University Program* (Abbildung 2).

2.3.3 Pipelining

Bis zum ARM7 war die eingesetzte Befehlspipeline dreistufig: Instruction Fetch, Instruction Decode und Instruction Execution[1, 8]. Es ist ersichtlich, dass der Instruction Execution-Schritt der umfangreichste ist: zusätzlich zur eigentlichen ALU-Operation muss vorher der Datensatz aus den Registern gelesen werden; im Anschluss muss das Ergebnis wieder zurückgeschrieben werden. Dies alles geschieht in einem Taktzyklus.

Da bei diesem Aufbau nur maximal drei Befehle zeitgleich bearbeitet werden können (zu einem Taktzeitpunkt ist Instruktion 1 im Execution Stage, Instruktion 2 im Decode Stage und Instruktion 3 im Fetch Stage), ist der Geschwindigkeitsetrag nur minimal.
Aus diesem Grund wurde die Pipeline zum ARM8 um zwei Stages und ein neues Cache-Interface erweitert. Der Execution Stage wird in drei kleinere Stages geteilt; nebst der eigentlichen Execution kamem Stages für Memory Access und Register Writes hinzu, Register Read wurde in den Instruction Decode-Stage verschoben. Dies vereinfacht ein optimales Auslasten der Pipeline, da mehr Instruktionen semi-parallel abgearbeitet werden können.

Der Cortex-A8 verfügt bereits über 13 verschiedene Pipeline-Stages [10]. Dies hat vor allem positive Auswirkungen auf das Branch Prediction: Bei vielen Pipeline-Stufen kann die Hardware leichter ermitteln, welche Instruktionen in den nächsten Takten welche Aktionen ausführen, wodurch sich Vorhersagen deutlich öfter als korrekt erweisen.
Weiterhin ermöglicht eine längere Pipeline mehr Spielraum beim Forwarding. Folglich können deutlich öfter erst später folgende Instruktionen bevorzugt werden, während eine aktuelle Instruktion auf eine Ressource oder ein Datum wartet. Die Folge sind weniger Hazards und weniger einzusetzende NOPs, wodurch der Durchsatz deutlich steigt.

2.3.4 Der Barrel Shifter

Die ARM hat keine eigentliche Funktion, um Bit-Shifting durchzuführen. Stattdessen bietet sie einen Barrel Shifter an, welcher eine Möglichkeit bietet, Shifts direkt als Teil weiterer Instruktionen durchzuführen.

Jede Instruktion der ARM-ISA operiert auf entweder einem oder zwei Operanden, zu sehen in Abbildung 1. Operand 1 gelangt aus dem Register unverändert direkt in die *Arithmetic Logic Unit*, Operand 2 wird wegen eines eventuellen Shiftings in den Barrel Shifter geschickt, bevor er in die ALU für die Berechnung gelangen kann. Der Barrel Shifter ist in der Lage Shift-Instruktionen grundsätzlich in einem Takt durchzuführen. Er benötigt also keinen iterativen Ansatz für Shifts um mehrere Bits [6, 5].

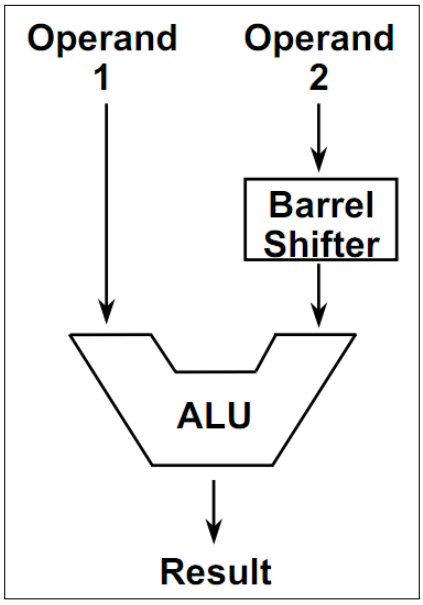

Abbildung 1: ARM-Instruktionsdesign [5]

2.4 Technologien der ARM-Architektur

Aktuell existieren bereits verschiedene ARM-Technologien, die allesamt auf verschiedenen Ebenen ansetzen und unterschiedliche Technologien implementieren. Für eine Übersicht, welcher ARM-Kern welche Technologie implementiert hat, soll Tabelle 1 dienen.
Im Folgenden möchte ich verschiedene Erweiterungen im Einzelnen erläutern.

2.4.1 Thumb

Grundsätzlich alle ARM-Instruktionen sind 32Bit-Instruktionen. Diese Vereinheitlichung macht vieles einfacher und verringert die Komplexität. Jedoch hat dies auch zur Folge, dass Leistung verschenkt wird, da nicht jede Instruktion unbedingt 32Bit benötigt. Thumb schafft hier Abhilfe.

Thumb liefert ein vollständig 16-Bit-weites Instruction Set, was bewirkt, dass mit der gleichen Menge über den Bus geladener Bits nahezu doppelt so viele Instruktionen geladen werden können. Dies hat also Auswirkungen auf die Code-Dichte [4]. Wohl zu beachten ist hierbei, dass Thumb nicht das darunterliegende Instruction Set verändert oder erweitert, sondern dass es diesen mit seinem eigenen Instruction Set mit beschränktem Zugriff repräsentiert [2].

Beim Ausführen von Thumb-Instruktionen stehen 8 Register zur Verfügung - R0 bis R7. Diese sind äquivalent zu den ARM-Registern R0-R7. Außerdem haben einige Thumb-Instruktionen Zugriff auf die ARM-Register R13, R14 und R15, einige auch auf den gesamten *high register*-Satz R8-R15.
Ob eine Thumb-Instruktion angewandt, ist am T-Flag (bit[5]) im CPSR (Current Program Status Register, 2.3.2) zu erkennen. Eine 1 heißt, es handele sich um eine Thumb-Instruktion [7].

2.4.2 Thumb-2

Thumb-2 ist eine Erweiterung von Thumb und kommt mit einer zusätzlichen Palette von 16Bit- und 32Bit-Instruktionen daher. Das Ziel von Thumb-2 ist, bei einer etwa gleichhohen Code-Dichte wie Thumb dennoch ebenso komplexe Instruktionen wie die ARM-Instruktionen anbieten zu können.
Thumb-2 erweitert sowohl das Thumb-, als auch das ARM-Instruction Set um einige weitere Instruktionen, wie beispielsweise Bitfeldmanipulationen, table branches und Bedingungsausführungen.
Die neue zu Thumb-2 zugehörige *Unified Assembly Language* kann ebenso Thumb-2-, wie auch ARM-Instruktionen aus demselben Assembler-Code erzeugen.

2.4.3 Jazelle

ARM-Jazelle ist eine ARM-Hardware- und -Software-Lösung, um Java-Unterstützung zu realisieren. Die Jazelle-Software entspricht einer vollfunktionsfähigen Java VM mit Multi-Tasking-Unterstützung, welche es möglich macht, Java-Programme auf der Jazelle-Hardware laufen zu lassen. Jazelle benutzt eine low-level-basierte, binäre Übersetzung, welche als extra-Stage in der Prozessor-Pipeline (2.3.3) zwischen dem Instruction Fetch und Instruction Decode agiert [9].

Jazelle DBX (Direct Bytecode eXtension) ist eine Erweiterung von Jazelle, welche es dem Prozessor ermöglicht, Java Bytecode direkt auszuführen und auf die Jazelle-VM-Software verzichten zu können [1].

Weiterhin fügt die Erweiterung Jazelle RTC zwei weitere Features hinzu: Die Ahead-of-Time- (AOT) und die Just-in-Time-Compilation (JIT) [1].
Die JIT-Compilation ermittelt den am häufigsten benutzten Code und übersetzt ihn, direkt zum Bedarfszeitpunkt, in nativen ARM-Code. Profiling und Übersetzen sind sehr zeitintensiv, im Endeffekt ergibt sich aber dennoch eine Geschwindigkeitssteigerung, da nur der meistverwendete Code übersetzt wird und nativer Code deutlich schneller ausgeführt werden kann als Code, der von Interpretern erst analysiert und verarbeitet werden muss.
Die Alternative ist der AOT-Compiler, welcher jeglichen Code bereits vor der Ausführung in nativen Maschinencode übersetzt. Vorteil ist, dass tatsächlich das gesamte Programm nativ ausgeführt werden kann. Jedoch kann dies auch den Speicher enorm aufblähen, weswegen AOT-Compiling prinzipiell nur selektiv angewandt wird [1].

2.4.4 Vector Floating Point

Der Vector Floating Point (VFP) Coprocessor ist eine zusätzliche optionale Einheit, welche Unterstützung für einfach und doppelt präzise IEEE 754 Gleitkommaoperationen bietet. Zusätzlich werden Vektor-Operationen unterstützt. Zusätzlich zu den Basisoperationen gibt es mitunter

Anbindungen für Multiplikations-Additionen und -Subtraktionen, Quadratwurzeloperationen, Vergleiche und Absolutwerte oder auch Umwandlungen zwischen Gleitkomma-Formaten und zwischen Integer- und Gleitkomma-Formaten [9].

2.4.5 DSP enhancement instructions

Dem Digitalen Signalprozessor bei dieser Erweiterung einige zusätzliche Features gegeben, um weitere Instruktionen behandeln zu können. Einige Neuerungen sind beispielsweise Instruktionen für vorzeichenbehaftete 16x16Bit- und 16x32Bit-Multiplikationen und -Multiplikations-Additionen, gesättigte Additionen und Subtraktionen und Instruktionen zum Laden und Speichern von Registerpaaren [9, 4].

2.4.6 TrustZone Security Extensions

Die ARM-TrustZone Security Extension hat das Ziel, zwei separate Adressbereiche zu erzeugen: Einen sicheren und einen unsicheren Bereich. Eine Instruktion aus dem unsicheren Bereich hat dann nur Zugriff auf ihren eigenen Bereich und ist nicht berechtigt Daten in der secured Zone zu modifizieren, löschen oder hinzuzufügen. Dies soll die Gefahr eines Buffer Overflows oder ähnlicher Effekte in der TrustZone deutlich verringern[4].

3 Fazit

Die ARM ist nicht für gewöhnliche Desktop-Umgebungen entwickelt worden. Die Anforderungen sind ganz andere - dies macht die Architektur deutlich. Ein mobiles Endgerät benötigt nicht unzählige Instruktionen, wenn stattdessen mit RISC-typisch deutlich weniger Instruktionen und einer einheitlichen Instruktions- und Registergröße von 32Bit Komplexität verringert und dadurch Speicherbedarf und Durchsatz erhöht werden kann.

Eine ARM kann nicht einfach eine ebenso hohe Frequenz wie ein x86-Prozessor einsetzen. Der Verbrauch wäre deutlich zu hoch für Geräte, die mit Batterie oder Akku betrieben werden. Auch Embedded Devices, Router oder Mikrocontroller können es sich nicht erlauben, Unmengen an Strom zu verbrauchen.
Dewegen muss mit niedrigstmöglichem Energieverbrauch eine höchstmögliche Leistung erzielt werden. Dies kann nur gelingen, wenn anstelle der Frequenz Optimierungen bevorzugt an der architektonischen Implementierung vorgenommen werden.
Dies entspricht der Entwicklung der Advanced RISC Machine.

Es muss auf alles verzichtet werden, was nicht unbedingt nötig ist, aber dafür Energie verbraucht. Wie zum Beispiel ein Repertoire vieler komplexer Instruktionen. Mit der Entscheidung auf das Design der *Reduced Instruction Set Computer* zurückzugreifen und auf viele mächtige und komplexe Instruktionen zu verzichten, wird beim RISC dank der relativ wenigen und einheitlich langen Instruktionen ein weit niedrigerer Decodieraufwand und dadurch eine höhere Ausführungsgeschwindigkeit erreicht.
Auch durch die Entscheidung, die ARM als Load-/Store-Architektur zu entwickeln, welche also keine Operationen direkt auf dem Speicher, sondern nur Operationen auf Registern erlaubt, wird eine verringerte Komplexität und dadurch ein geringerer Rechenaufwand erreicht.
Um weitere Ressourcen einzusparen, verzichtet man auf ein umfangreiches Interrupt-System und bietet nur zwei Interrupt-Modi an: *IRQ* und *FIQ* (2.3.1). Auch auf eine umfangreiche Subroutinen-Verwaltung wird verzichtet: Anstatt Rücksprüngadressen zusammen mit Parametern, Rückgabewerten und Unterprogrammdaten im Stack unterzubringen, dient für die Rücksprungadresse lediglich der Link Register R14 (2.3.2).

Aber Leistungssteigerung darf natürlich nicht nur durch Einsparungen erwirkt werden. Auch muss eie für den Bereich der Mobile Devices und Embedded Systems sinnvolle Architektur entworfen werden - und zusätzliche Technologien entwickelt werden, die die Ausführungszeit deutlich vermindern und den Energiebedarf nicht zu weit in die Höhe treiben.
Hier wird zum Beispiel bereits beim Pipeline-Modell (2.3.3) angesetzt. Umso mehr Pipeline-Stage-Unterteilungen gemacht werden, desto effektiver kann das System Instruction Forwarding anwenden und präzisere Sprungvorhersagen treffen.
Aber auch Technologien wie Thumb, welches bei optimalen Voraussetzungen die Code-Dichte um 40-50% erhöhen kann, oder Jazelle, was - mit den Erweiterungen DBX und RCT ausgestattet - selbst Java-Code teilweise oder komplett nativ auf dem Prozessor laufen zu lassen, helfen der ARM dabei, trotz geringer absoluter Leistung eine erhöhte Performance darzubieten.

Dies sind allesamt Gründe, welche der *Advanced RISC Machine* dabei halfen, im Markt der Mobilen Geräte Fuß zu fassen. Denn selbst Apple, HTC und Nokia zählen zu den Käufern der ARM-Produkte.

Literatur

[1] http://www.arm.com

[2] David Seal - Architecture Reference Manual

[3] Stephen Furber - ARM System-on-chip-architecture

[4] Peter Kirchmair - Processors for Embedded Systems - `http://informatik.uibk.ac.at/teaching/ws2009/esa/processors.pdf`

[5] The ARM Instruction Set - `http://simplemachines.it/doc/arm_inst.pdf`

[6] Ingo Sander - The ARM Architecture and Instruction Set - `http://www.imit.kth.se/courses/2B1445/Lectures/Lecture2/2B1445_L2_InstructionSet.pdf`

[7] Arvind Krishnaswamy, Rajiv Gupta - Profile Guided Selection of ARM and Thumb Instructions - `http://portal.acm.org/citation.cfm?id=566225.513840`

[8] Dave Jaggar - ARM Architecture and Systems `http://www.ee.ryerson.ca/~courses/coe718/Data-Sheets/ARM/ARM-Architecture.pdf`

[9] Carl von Platen - ARM Analysis `http://www.control.lth.se/user/karlerik/Actors/d2a.pdf`

[10] `http://en.wikipedia.org/wiki/ARM_architecture`

[11] `http://www.ubuntu.com/news/arm-linux`

[12] `http://www.debian.org/ports/arm/`

[13] `http://www.armedslack.org/`

[14] `http://www.gentoo.org/proj/en/base/arm/index.xml`

[15] `http://www.freebsd.org/platforms/arm.html`

[16] `http://hub.opensolaris.org/bin/view/Project+osarm/WebHome`

4 Anhang

Familie	Architektur	Kern	Technologien	Performance	Verwendung
	ARMv4T	ARM7TDMI	Thumb	85-204 MHz 94-189 DMIPS	Apple iPod GameBoy Advance
ARM7	ARMv5TEJ	ARM7EJ-S	Thumb	100-210 MHz	
ARM9	ARMv5TE	ARM926EJ-S	Thumb DSP Jazelle	200-470 MHz 220-517 DMIPS	SonyEricsson K/W LG Arena Benq x65 Series
		ARM946E-S	Thumb DSP	230-441 MHz 275-529 DMIPS	Nintendo DS Nokia N-Gage
		ARM968E-S	Thumb DSP	180-470 MHz 198-517 DMIPS	
ARM11	ARMv6	ARM1136	Thumb Jazelle DBX DSP	610 MHz 762 DMIPS	Zune HTC Hero Nokia N97
		ARM1156	Thumb (-2) DSP	600MHz 846 DMIPS	
		ARM1176	Thumb DSP Jazelle DBX	482-990 MHz 603-1238 DMIPS	Apple iPhone 3G iPod Touch 1G/2G Samsung Omnia II
		ARM11MPcore	Thumb DSP Jazelle DBX	427-865 MHz 1060-1075 DMIPS	NVidia APX 2500
Cortex	ARMv7-A	Cortex-A5	Thumb (-2) DSP Jazelle DBX&RCT VFPv3 (optional)	530MHz/ >1GHz 832 - 1530 DMIPS	
		Cortex-A8	Thumb (-2) VFPv3	500 MHz - 1 GHz 1000-2000 DMIPS	iPhone 3GS/4 iPod Touch 3G
		Cortex-A9	Thumb (-2) Jazelle DBX&RCT DSP	0,8-2 GHz 2k-10k DMIPS	Samsung Orion NVidia Tegra 2
	ARMv7-R	Cortex-R4	Thumb (-2) DSP	270-620 MHz 450-1030 DMIPS	
	ARMv6-M	Cortex-M1	Thumb (-2)	70-200 MHz	
	ARMv7-M	Cortex-M3	Thumb (-2)	50-275 MHz 75-340 DMIPS	
	ARMv7E-M	Cortex-M4	Thumb (-2)	150-300 MHz 185-375 DMIPS	

Tabelle 1: Vergleich verschiedener ARM-Familien [1][10]

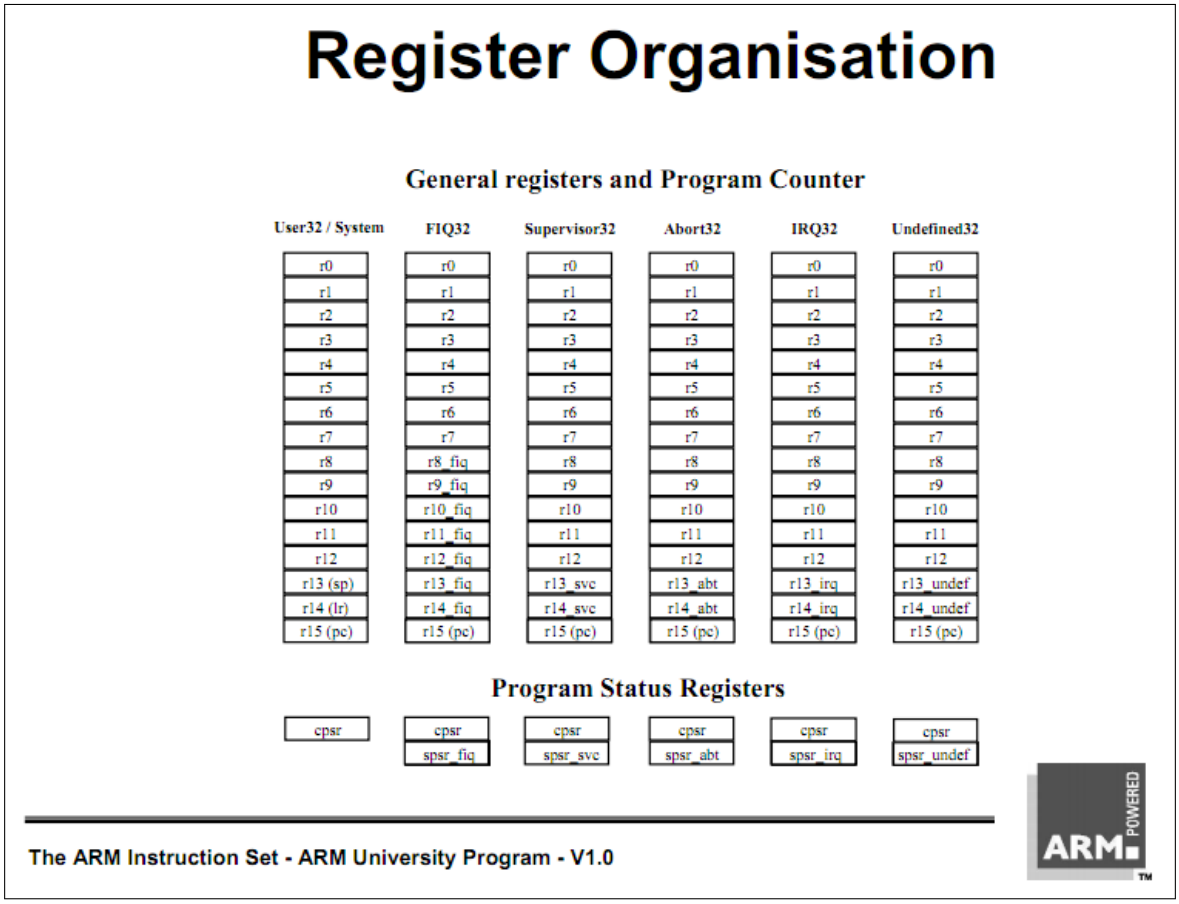

Abbildung 2: Register-Verteilung für die verschiedenen Prozess-Modi [5]