


CISC

- > Geringere Größe des Programms
- > ... dafür längere Ausführungsduer der Befehle ("Multi Clock Cycle")
- > Im Prinzip: Große Vielzahl an Befehlen, die teilweise "mehr machen" als RISC-Befehle, aber dafür länger dauern.
- > Schlecht für Pipelining

RISC

- > Größere Programme (mehr Instruktionen nötig)
- > ... aber Instruktionen dauern nicht so lange ("Single Clock Cycle")
- > Durch Pipelining ein "Ergebnis" pro Taktzyklus (siehe Abschnitt "Befehlsausführung")
- > Kleinerer Befehlssatz als CISC, mit spezialisierten Befehlen für Arithmetik und Speicherzugriff

von Neumann-Architektur

- Es gibt einen einzigen "Systembus", bestehend aus Adressbus, Datenbus und Steuerbus, der CPU, Speicher, I/O und Peripherie verbindet
- Programm- und Datenspeicher teilen sich einen gemeinsamen Adressbereich

Harvard-Architektur

- Bei der Harvard-Architektur gibt es getrennte Systembusse für Programm- und Datenspeicher
- Dies ermöglicht parallelen Zugriff auf Befehle und Daten, was eine höhere Ablaufgeschwindigkeit mit sich bringt ...

xPSR: Status-Flags

- N: 'Negative' → Ergebnis der letzten Operation war < 0
- Z: 'Zero' → Ergebnis der letzten Operation war = 0
- C: 'Carry' → Letzte Operation liefert ein Carry-Bit (z.B. wenn bei ADD das Ergebnis > 32 Bit ist)
- V: 'Overflow' → Ergebnis der Addition zweier positiver Zahlen wird negativ (oder Umgekehrt) → V-Bit wird gesetzt
- Q: 'DSP Overflow' → Nur bei speziellen Instruktionen, für uns irrelevant



Status Flags

Flags, 4-Bit ALU Beispiel

$5+2 = 7$	$5+4 = 9$	$8+8 = 16$	$5-7 = (-2)$	$5-4 = 1$
-----	-----	-----	2K von 7: 1001	2K von 4: 1100
0101	0101	1000	+ 0100	+ 0101
+	+	-----	-----	-----
C 0000	C 1000	C 10000	+ 1001	+ 1100
0111	1001	0000	C 0010	C 11000
C=0, Z=0 V=0, N=0 Y=1, N=1	C=0, Z=0 V=0, N=0 Y=1, N=1	C=1, Z=1 V=1, N=0	C=1, Z=0 V=0, N=1	C=1, Z=0 V=0, N=0
Signed: 7 Unsigned: 7	Signed: -7 Unsigned: 9	Signed: 0 Unsigned: 0	Signed: -2 Unsigned: 14	Signed: 1 Unsigned: 1
(Signed wäre die Rechnung: (-8) + (-8) = -16)				

Variablen, Daten und ihre Platzierung

- Was sind eigentlich 'Daten' im Kontext von Programmen?
 - > Veränderliche Daten ('Variablen')
 - Z.B. Zustandswerte, Funktionsvariablen, strukturierte Daten, ...
 - Read/Write
 - > Unveränderliche Daten ('Konstanten')
 - Z.B. Parameter, Zeichenketten, Zustandstabellen, sonstige Tabellen (LUT) ...
 - Read-Only
 - > Globale Variablen
 - > Lokale Variablen
 - > Statische Variablen
- Je nach Typ liegen Daten in unterschiedlichen Speicherbereichen (und teilw. auch Speichern)
 - > normaler RAM, Stack, Heap oder ROM

Shift-Operationen

- 1sl r0,r1,#n $R0 = R1 \ll n$ Logischer Shift nach links
- 1sr r0,r1,#n $R0 = R1 \gg n$ Logischer Shift nach rechts
- asr r0,r1,#n $R0 = R1 \gg n$ Arithmetischer Shift nach rechts

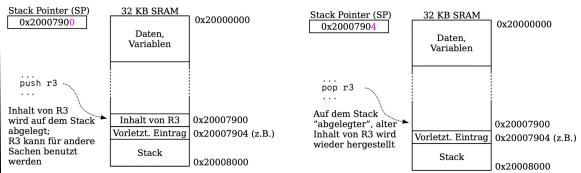
Wofür braucht man RAM?

- RAM ist schnell aber volatil. → Im RAM liegen Daten mit denen das Programm arbeitet (Variablen) → 'Arbeitspeicher'

Wofür braucht man ROM?

- ROM ist nichtvolatil, d.h. es behält seinen Inhalt nach dem Ausschalten.
- ROM ist (wie der Name schon sagt) read-only, d.h. unveränderlich. (Man kann ROM schon verändern, aber das geschieht nicht zur Laufzeit des Programms)

Stack



Stack-Nutzung

- Der Stack wird u.a. genutzt, um (vor allem in Unterprogrammen) Registerinhalte "zwischenzuspeichern" und später wiederherzustellen
- > Ein Unterprogramm "weiß" nicht, welche Register im Hauptprogramm benutzt werden

Speicherbefehle

Unterschiedliche Größen (STR analog wie LDR)

```

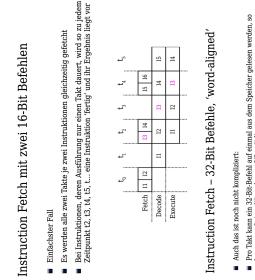
ldrdb r0, [r1]      /* 8 Bit (Byte) von *R1 → R0 */
ldrh r0, [r1]        /* 16 Bit (Half-Word) von *R1 → R0 */
ldr r0, [r1]         /* 32 Bit (Word) von *R1 → R0, das Übliche */
ldrd r0, r1, [r2]    /* 64 Bit (DWord) von *R1 → (R0:R1) */
ldr r0, [r1, #4]    /* Lädt Word von *(R1+4), R1 wird nicht verändert (Pre-Indexing) */
ldr r0, [r1], #4    /* Lädt Word von *R1, danach wird R1 um 4 erhöht (Post-Indexing) */

```

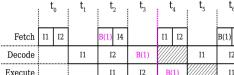
- > Mit einem '!' kann außerdem beim Pre-Indexing das Adressregister auch auf den neuen Wert geschrieben werden

```
ldr r0, [r1, #4!] /* Lädt Word von *(R1+4!), R1 wird nach der Operation =R1+4 (Pre-Indexing)! */
```

Die Pipeline des ARM Cortex-M4



Die Pipeline und Sprungbefehle



Bereits nach t3 wird Sprungziel an die Fetch-Unit gemeldet, die sofort den PFB neu befüllt

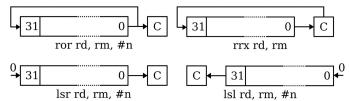
Zusammenfassung:

- Bei logischen Shifts (z.B. zur Bitfeldmanipulation): 1sl und 1sr
- Bei arithmetischen Shifts (Multiplikation mit oder Division durch 2^n) mit Vorzeichen: 1sl und asr
- Bei vorzeichenloser Arithmetik: 1sl und 1sr

Rotate-Operationen

`ror r0,r1,#n R0 = [R1]rot(n)` Rotation um n
`rrx r0,r1,#n R0 = [R1 : C]rot(1)` Rotation um 1, R1 mit C verlängert

Shift und Rotate



Endianness und Reverse-Befehle

Das Folgende Beispiel zeigt die Zahl `0xa1b2c3d4` einmal als Little- und einmal als Big-Endian:

	Little Endian	Big Endian
	0x203 0x202 0x201	0x203 0x202 0x201
Little Endian	0xa1 0xb2 0xc3 0xd4	0xd4 0xc3 0xb2 0xa1
	0x200	0x200
<code>rev r0,r1 R0 = rev(R1)</code>	Umkehrung der Bytereihenfolge	
<code>rev16 r0,r1,#n R0 = rev([R1 31..16])</code>		Umkehrung der Bytes beider Hälften
<code>: rev[R1 15..0]</code>		

Bedingungscodes

Bedingung	Beschreibung	Flags
<code>eq, ne</code>	Equal / Not Equal	<code>Zero = {1,0}</code>
<code>cs, cc</code>	Carry Set / Carry Clear	<code>Carry = {1,0}</code>
<code>mi, pl</code>	Minus / Plus	<code>Negative = {1,0}</code>
<code>vs, vc</code>	Overflow Set / Clear	<code>Overflow = {1,0}</code>
<code>gt</code>	Größer als (signed)	<code>Z = 0, N = 0, V = 0 oder Z = 0, N = 1, V = 1</code>
<code>lt</code>	Kleiner als (signed)	<code>Z = 0, N = 1, V = 0 oder Z = 0, N = 0, V = 1</code>
<code>ge, le</code>	Gr. gleich / Kl. gleich	Wie <code>gt, lt</code> aber <code>Z</code> egal

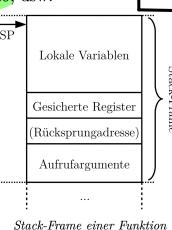
ARM Calling Conventions

Bei ARM definiert der AAPCS die Regeln:

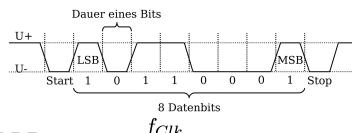
- Für die Übergabe weniger Argumente werden `R0` bis `R3` genutzt
- Damit können vier 32-Bit Argumente übergeben werden
- Bei 64-Bit-Zahlen werden zwei Register zusammengefasst
- Rückgabewerte: 32 Bit in `R0`, 64 Bit in `R1 : R0`, usw.

Da `R0` bis `R3` als Übertragungsregister verwendet werden, gelten einige Besonderheiten was das Sichern von Registern betrifft:

- `R0 – R3` werden vom Unterprogramm grundsätzlich *nicht* gesichert (also nicht auf den Stack gepusht)
- Die aufrufende Funktion (*Caller*) muss sich ggf. um das Sichern und Wiederherstellen vor bzw. nach Aufruf kümmern. (*Caller-Save-Register*)
- Die anderen Register `R4 – R11` müssen vom Unterprogramm (*Callee*) gesichert und wiederhergestellt werden (*Callee-Save-Register*)



UART und serielle Kommunikation



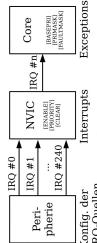
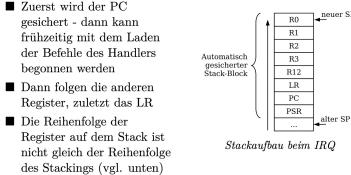
$$IBRD = \text{integer}(BRD)$$

$$FBRD = \text{integer}(\text{Kommastellen} * 64 + 0.5)$$

Interrupt-Quellen

- Exceptions* werden vom Prozessor selbst erzeugt: z.B. Fehler, Traps, Reset, SysTick, Software-Interrupts
- Interrupts* werden asynchron durch (interne) Peripherie erzeugt: z.B. Timer, GPIOs, Schnittstellen, ADC, ...
- Peripherie wird so konfiguriert, dass sie IRQ bei NVIC auslösen kann
 - Beim ARM Cortex-M beträgt sie 12 Zyklen.
- Der NVIC löst - wenn der entsprechende IRQ freigegeben ist - bei der CPU einen Sprung an die Handler-Adresse aus
- Exceptions* werden durch den Prozessor selbst ausgelöst

Ablauf beim Stacking



Tail-Chaining

Wird ein niederprioriorer IRQ während eines höherpriorioren Handlers ausgelöst, bleibt der niederprioriorer IRQ währnddessen *pended* (also hinten angestellt).

Die Register werden dabei nicht unstacked und erneut gestacked, sondern es wird direkt der nächste Vektor gelesen. Man spricht von *Tail-Chaining*.

Interrupt-Nesting

Tritt ein höherprioriorer IRQ auf, während bereits ein Handler ausgeführt wird, so unterbricht der Handler des höherpriorioren IRQ diesen. Man spricht von *Nesting*.

Beim Wechsel in den höherpriorioren Handler werden erneut die Register gesichert.

Die Hardware des SysTick-Timers

Der *SysTick* ist ein 24-Bit Abwärtszähler mit automatischem Reload. Erreicht er Null, kann Interrupt #15 ausgelöst werden.

$$T_{\text{ick}} = \text{RELOAD} * T_{\text{CLK}} = \frac{\text{RELOAD}}{f_{\text{CLK}}}$$

Im ersten Fall ist $f_{\text{CLK}} = 4 \text{ MHz}$ und damit $T_{\text{CLK}} = 250 \text{ ns}$. Soll z.B. der SysTick-Interrupt alle 10 ms kommen, dann gilt:

$$\text{RELOAD} = 10 \text{ ms} * 4 \text{ MHz} = 40000 \quad (2)$$

A/D-Wandlung

- Der ADC des TM4C hat eine Auflösung von 12 Bit
- D.h. es stehen 12 Bit (0x000 bis 0xFFFF) zur Verfügung, um den Wertebereich 0 V bis 3,3 V als digitale Zahlen abzubilden
- Es stehen somit $2^{12} = 4096$ diskrete Werte zur Verfügung, womit sich eine Auflösung von $\frac{3.3 \text{ V}}{4096} = 805, 6 \mu\text{V}$ ergibt

Sample Sequencer

- Der ADC wandelt eine Reihe von Messungen von unterschiedlichen Kanälen direkt hintereinander, die Reihenfolge lässt sich im Sample Sequencer programmieren

Eine Sample-Sequenz läuft auf unterschiedliche Weise starten, u.a.

- 'manuell' durch Software, durch setzen eines Registerbits
- durch einen Timer -> periodisches Sampling
- durch ein externes Signal an einem GPIO-Pin
- durch einen analogen Komparator (Trigger, sobald Spannung größer oder kleiner als vorgegeben)
- kontinuierlich, also ununterbrochenes Sampling

Ablauf von A/D-Wandlungen

Aus den Beispielen sieht man, dass die eigentliche Wandlung bei n Bit Auflösung n Takte dauert.

Für eine einzelne A/D-Wandlung bei 16 MHz ADC-Takt ergibt sich folgendes Timing:

