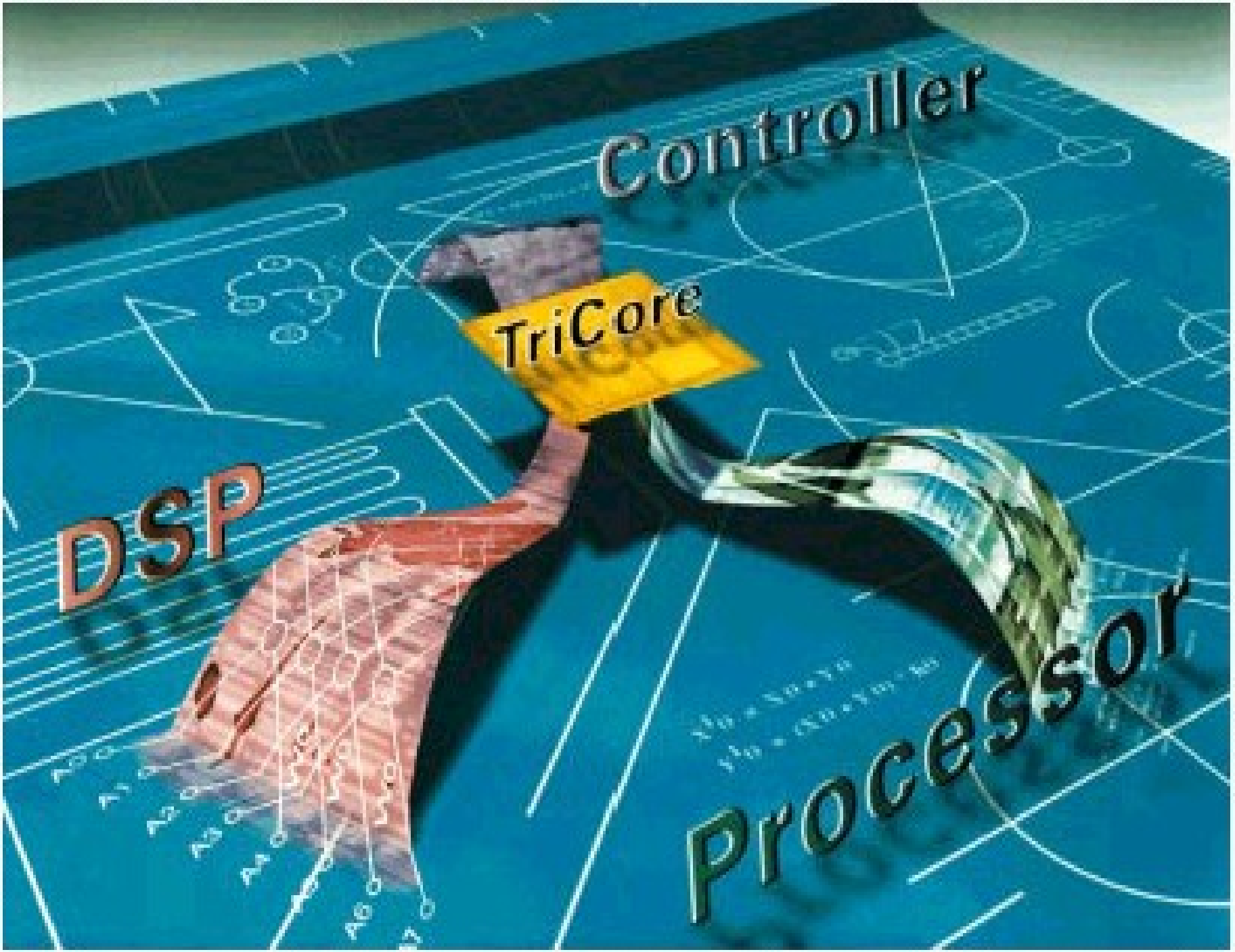

Controller
TriCore
DSP
Processor


# TRICORE 4th!?

Von: Antoni Gazali

Clarkson Laryea

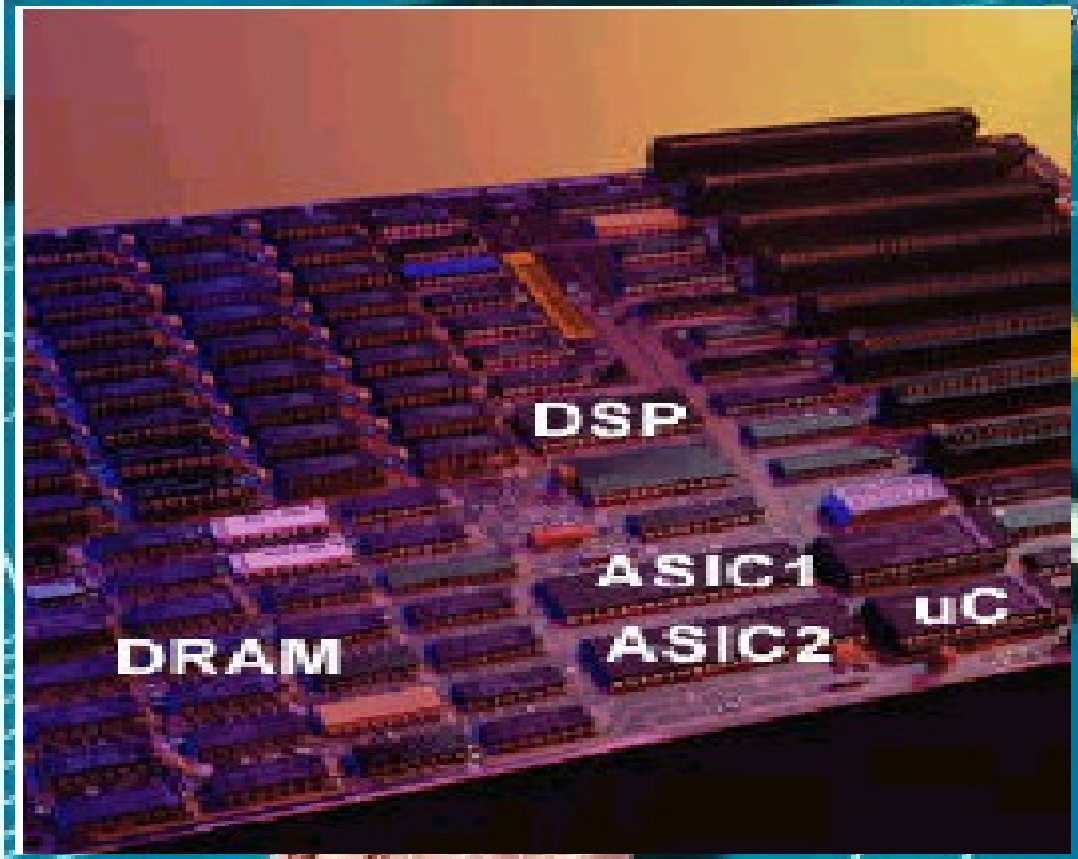
Karsten Roederer

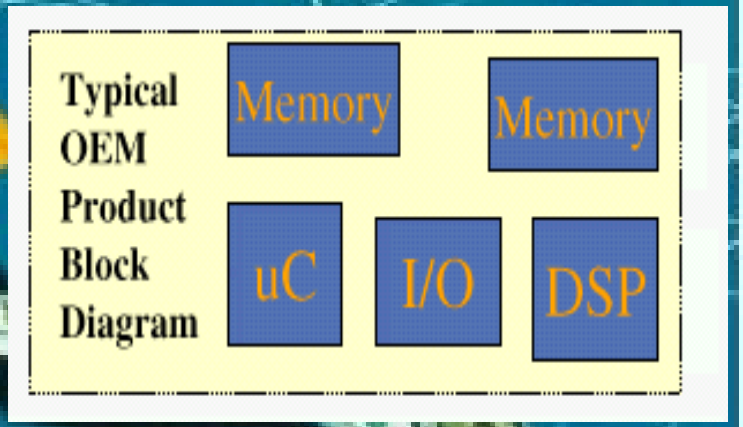
# Einsatzbereich µC&DSP





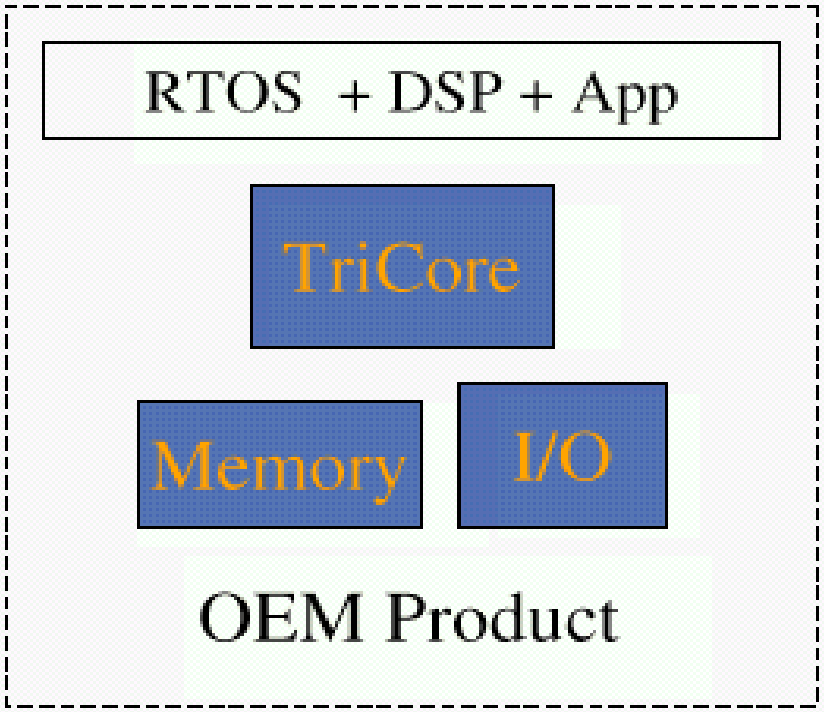
- Vernetzte Aktoren und Sensoren
- Karosserie und Maschinen Kontrolle
- Bildverarbeitung
- Sprache Kodierung und Komprimierung
- Spracherkennung




- Typische Implementierungen benutzen 2 CPU Systeme; einen µC und einen DSP
- Jeder hat seinen eigenen Speicher, Betriebssystem und Werkzeug
- höhere Kosten und Komplexität

# TRICORE

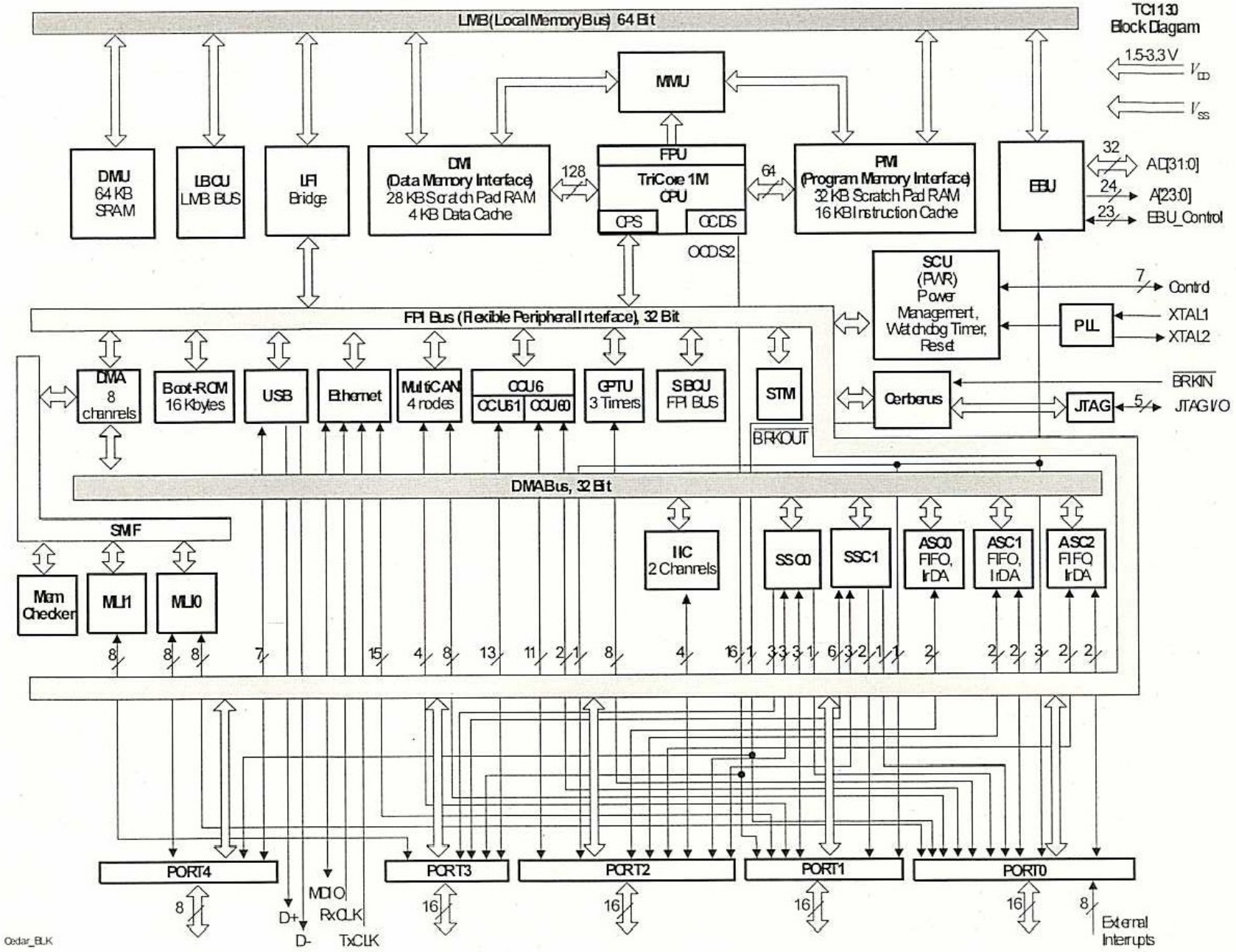



- µC, DSP Entwicklung und debug Unterstützung auf dem Chip.
- Kleiner

# Wieso TRICORE?

- Ein Prozessor, der die DSP , µC und RISC Prozessor-Eigenschaften unter einem Core bringt
- RISC Prozessor Eigenschaften:
  - 32 Bit load/store Harvard Architektur
  - 16 Adress & 16 Daten Register
  - Super-Skalar Ausführung
  - komprimierte/SIMD Instruktion
  - HLL und RTOS Unterstützung
- µC Eigenschaften:
  - 16-Bit und 32-Bit Instruktion Format
  - BMU (Bit Manipulation Unit)
  - Schneller Kontext-Wechsel
  - Schnelle Interrupt Behandlung
  - Integrierte Peripherie Unterstützung

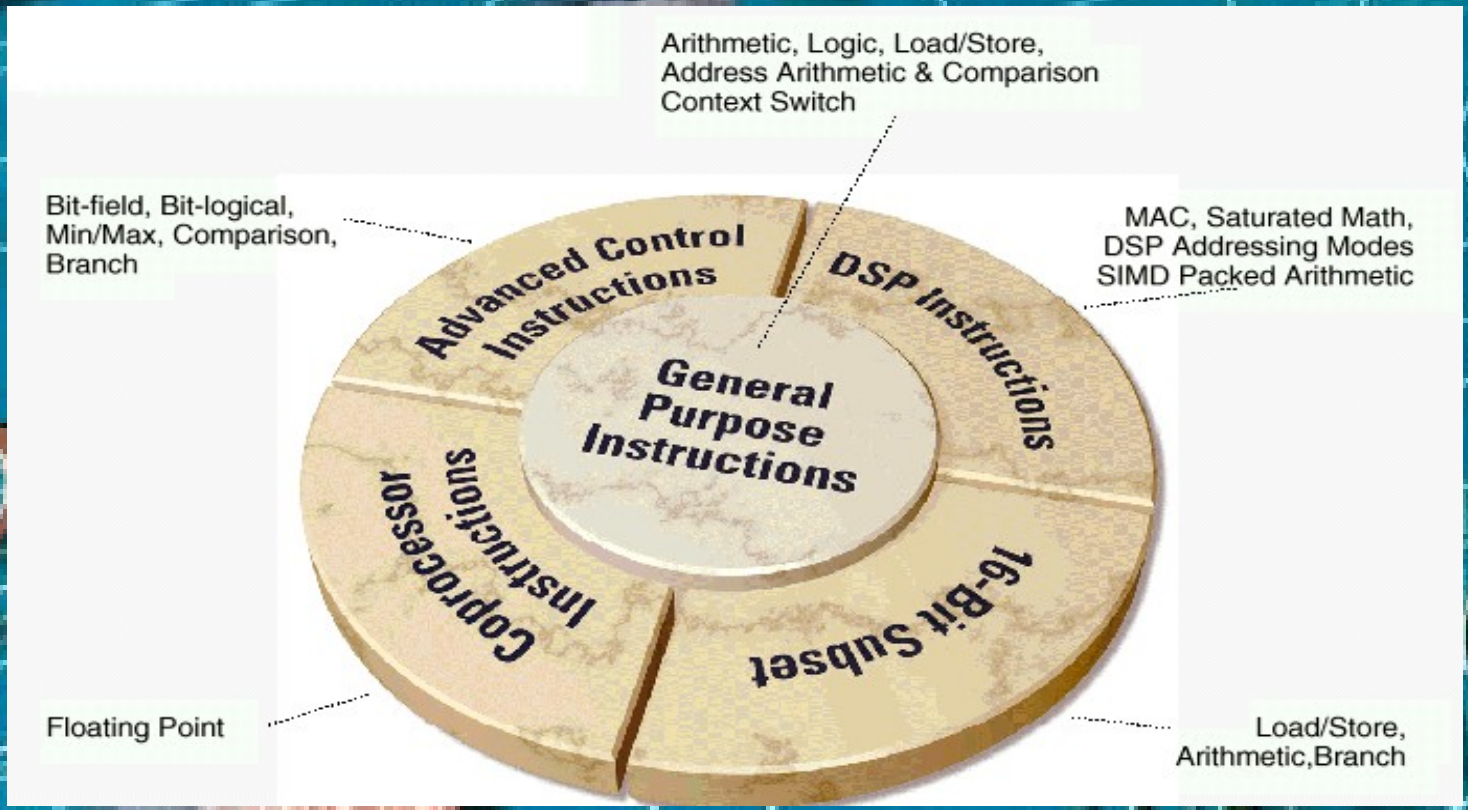
- DSP Eigenschaften:
  - spezifische DSP Adressierung Mode (circular, bit reverse)
  - Zero overhead loops
  - single-cycle dual MAC (multiply accumulator)
  - Abrundung


TC1130
Block Diagram
LMB (Local Memory Bus) 64 Bit
MMU
1.5-3.3 V
$V_{DD}$
$V_{SS}$
DMU
64 KB
SRAM
LBCU
LMB BUS
LFI
Bridge
DMI
(Data Memory Interface)
28 KB Scratch Pad RAM
4 KB Data Cache
128
FPU
TriCore 1M
CPU
CPS
OCDS
64
PMI
(Program Memory Interface)
32 KB Scratch Pad RAM
16 KB Instruction Cache
EBU
32
A[31:0]
24
A[23:0]
23
EBU_Control
OCDS2
SCU
(PWR)
Power
Management,
Watchdog Timer,
Reset
7
Control
PLL
XTAL1
XTAL2
FPI Bus (Flexible Peripheral Interface), 32 Bit
DMA
8
channels
Boot-ROM
16 Kbytes
USB
Ethernet
MultiCAN
4 nodes
CCU6
CCU61
CCU60
GPTU
3 Timers
SBCU
FPI BUS
STM
Cerberus
BRKIN
JTAG
5
JTAG I/O
BRKOUT
DMA Bus, 32 Bit
SMF
Mem
Checker
MLI1
MLI0
IIC
2 Channels
SSC0
SSC1
ASC0
FIFO,
IrDA
ASC1
FIFO,
IrDA
ASC2
FIFO
IrDA
8
8
8
7
15
4
8
13
11
2
1
8
4
16
1
3
3
3
1
6
3
2
1
1
2
2
2
2
3
2
2
PORT4
PORT3
PORT2
PORT1
PORT0
8
16
16
16
16
8
D+
D-
MDIO
RxCLK
TxCLK
External
Interrupts
Oxtar_BLK

# Tricore Eigenschaften

- Unified Instruction Set
  - unterstützt Kontroller und DSP Aufgaben in einem Core durch eine größere Verfügbarkeit von Datentypen und Adresse-Mode
- Schnelle und hardware-gestützte Aufgaben Wechsel
  - von Kontroller- zu DSP Aufgaben innerhalb von 2 cyclen (weniger als 100ns)- ermöglicht wird das durch breiten Bus zum Speicher
- Größe des On-chip Speicher:
  - bis 2 Mbyte DRAM and 512 Kbyte Flash
- Tricore:
  - Virtueller Multi-Prozessor
  - Real-time Behandlung von komplexen Operationen

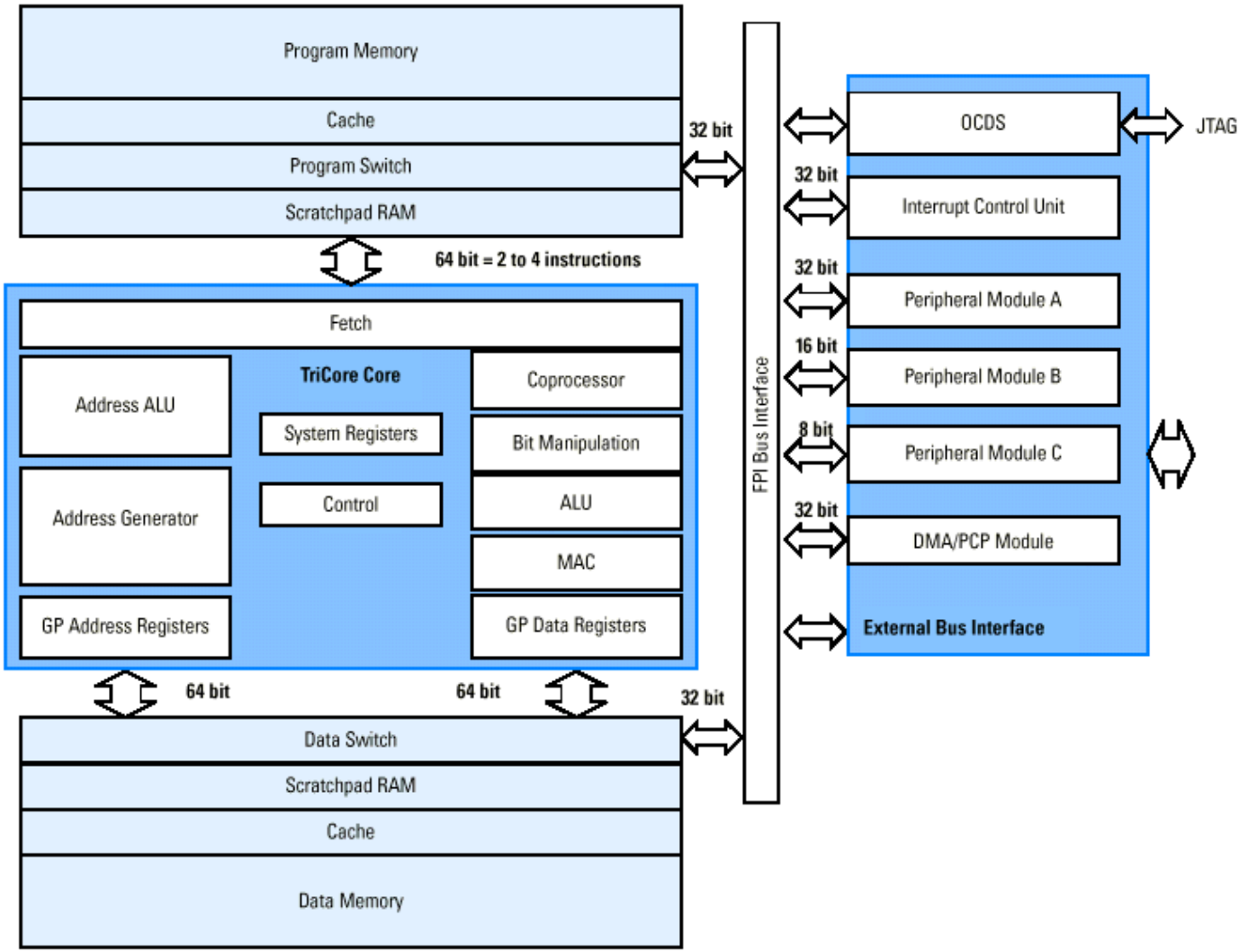
# Unified Instruction Set



# Tricore-1 Core & Module

- Benutzt: 0.35µ Technologie, 3.3V
- Prozessor Core
- On-Chip Memory
  - Program Memory
    - SRAM(1Kbyte)
    - Cache (1 Kbyte)
    - DRAM (128 Kbyte)
  - Data Memory
    - SRAM (2 Kbyte)
    - Cache (2 Kbyte)
    - DRAM (64 Kbyte)
- Serial Port, Timers, Interrupt Controller, Debug
- FPI Bus, External Memory
- PCP: Programmable I/O Controller

# Tricore Architektur



# Flexible Peripheral Interconnect Bus (FPI)

- Für on-chip Verbindungen (Transfer daten Memory und I/O)
- Verbindet Tricore mit CPUs, externe und interne Peripherie, Coprozessoren, Ports, Memory, Externe Bus Controller (EBC), u.s.w.
- Demultiplexed Bus mit bis zu 32 Bit-Adresse und 64 Bit-Daten
- Datendurchsatz bis 800 MBytes/s bei 100 MHz
- Es können beliebig viele Peripherien angeschlossen werden
- Multi-master Fähigkeiten
- Clock synchronisiert
- 8-16-32- und 64-Bit Daten Übertragung
- Geschwindigkeit Anpassungsfähigkeit (wichtig für langsame Peripherien)

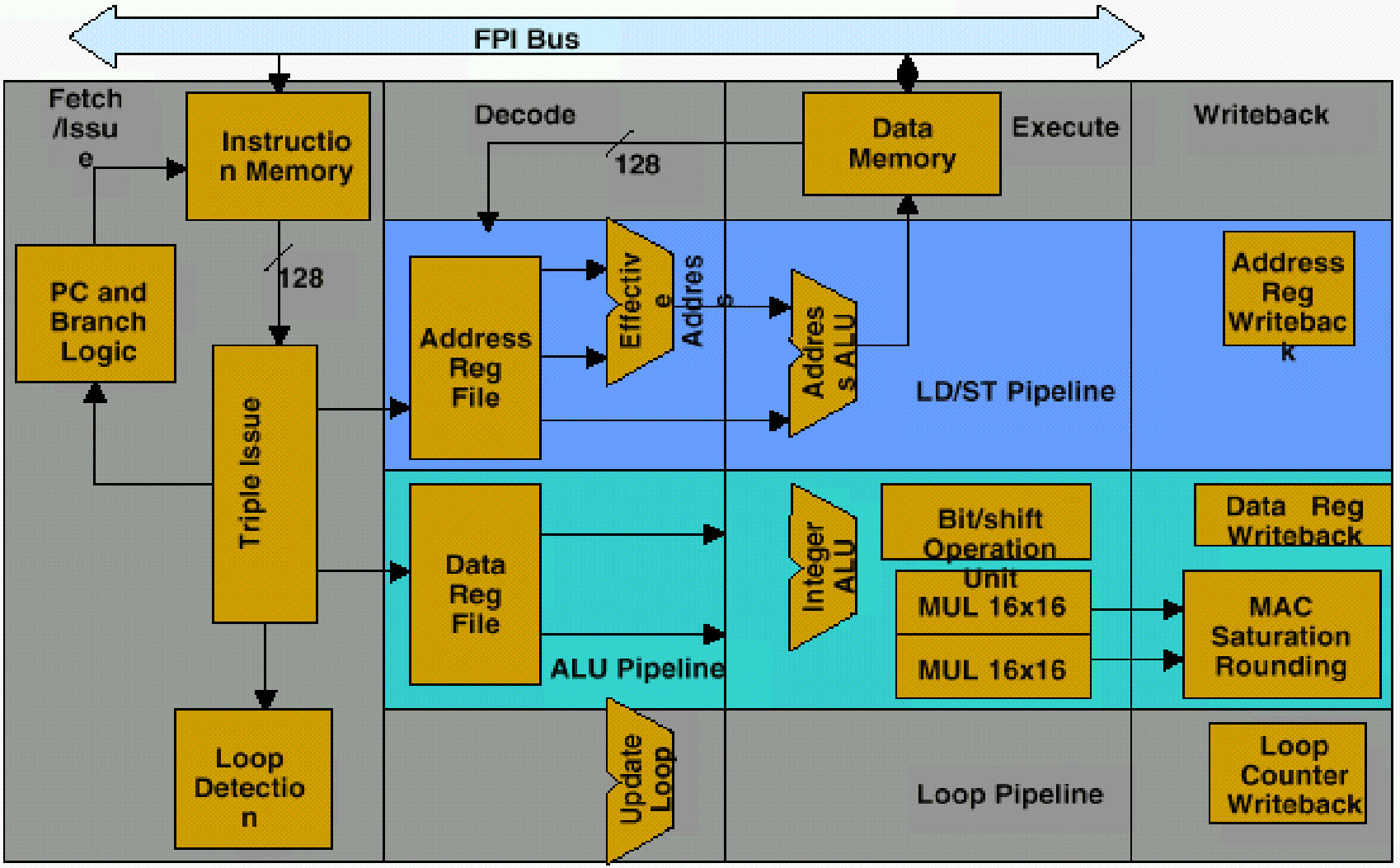
# DMA/PCP

- I/O Kontroll-Prozessor, der die typischen Aufgaben vom DMA und Interrupt Routine von CPU übernimmt
- überträgt und überwacht einfache Daten und verarbeitet diese
- Kann bis zu 64 Peripherien parallel zu CPU bedienen
- kann die Ergebnisse von CPU Operationen  überprüfen

# Tricore-1 core CPU

- RISC load/store Maschine
- Harvard Architektur
  - getrennte Adress- und Datenbus für Programm- und Datenspeicher
- Little-endian byte ordering
- 2 Major Pipelines:
    - integer Operation
    - load/store Operation
- 1 Minor Pipeline:
    - Optimierung der DSP Schleifen-Operation

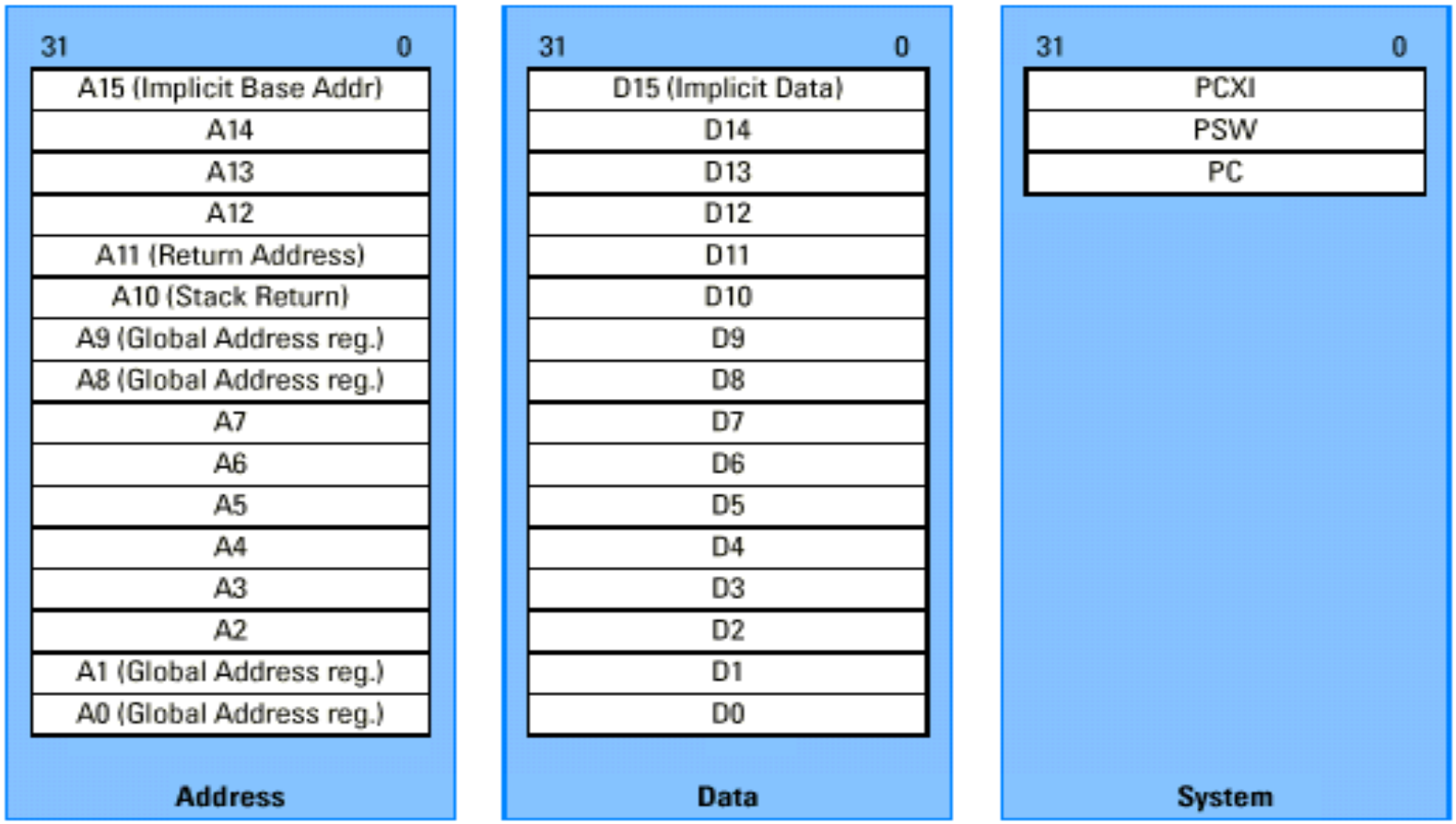
# Superscalar/Pipeline Architektur



# Register Architektur

- hat 32 General Purpose Register mit je 32-bit breite
  - sind in 16 Daten Register (DGPRs),(D0-D15)
  - und 16 Adressen Register (AGPRs),(A0-1515) unterteilt
- hat zwei 32-bit Register für Programm Status Informationen
  - PCXI –Programm Context Information Register
  - PSW –Programm Status Word
- ein 32-bit Register für den Programmcounter
- 4 GPRs haben spezielle Funktionen
  - D15 wird als implizites Daten Register benutzt
  - A15 wird als implizites Adressen Register benutzt
  - A10 wird als Stack Pointer benutzt
  - A11 wird als Return Stack Pointer benutzt
- Registers A0,A1,A8 und A10 sind als System Global Register definiert (dessen Inhalte dürfen gespeichert oder gelöscht werden während eines call, Interrupt usw..
- Es gibt keine Floating-Point Register. Daten Register werden für Floating-Point Operationen verwendet.

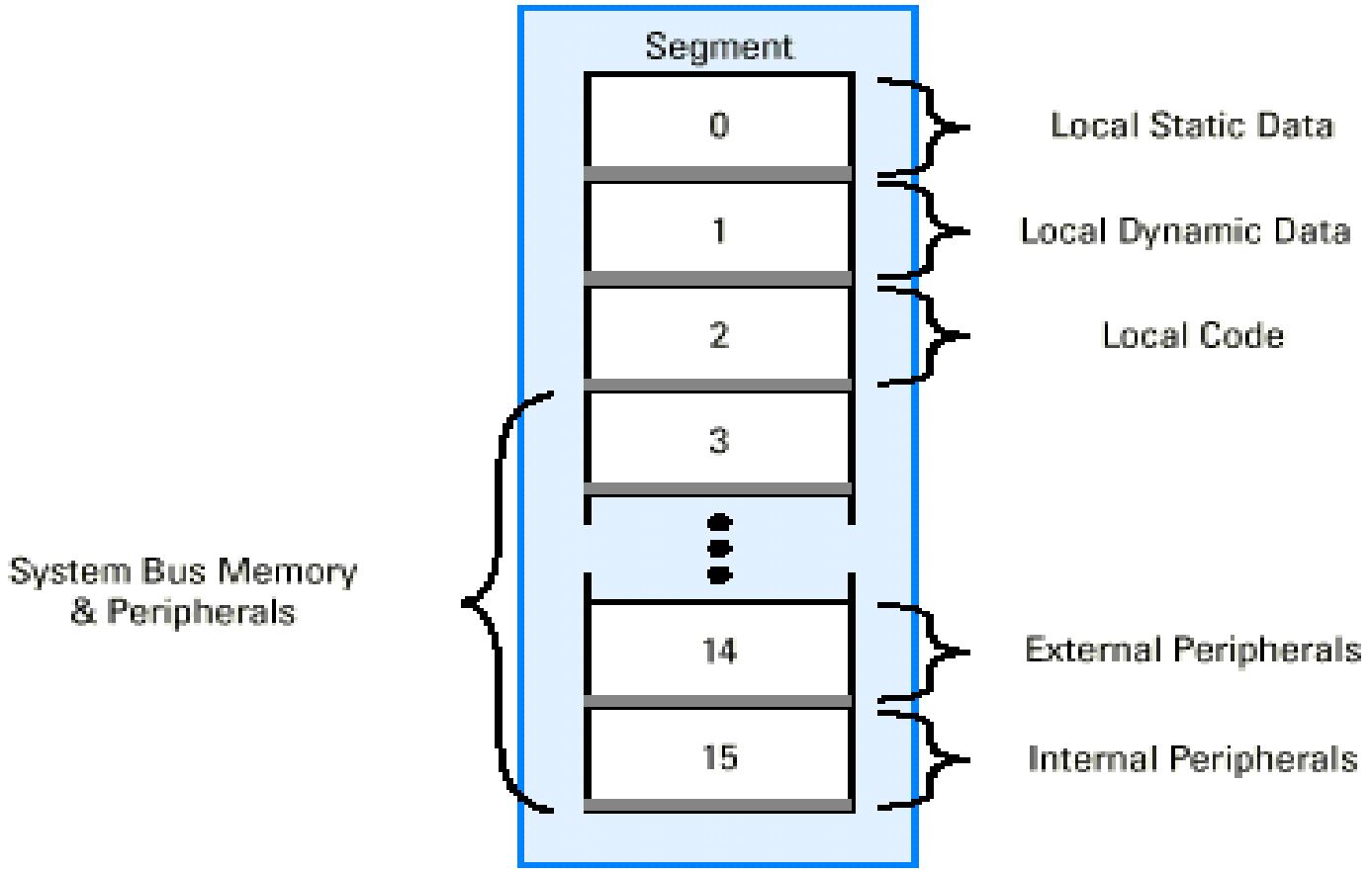
# Register


31
0
A15 (Implicit Base Addr)
A14
A13
A12
A11 (Return Address)
A10 (Stack Return)
A9 (Global Address reg.)
A8 (Global Address reg.)
A7
A6
A5
A4
A3
A2
A1 (Global Address reg.)
A0 (Global Address reg.)
Address
31
0
D15 (Implicit Data)
D14
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
Data
31
0
PCXI
PSW
PC
System

# Speicher Mode

- Tricore Architektur kann bis zu 4Gbytes große Programm- und I/O Adressräume verwalten
- Adressbreite ist  32-bit
- Adressraum ist in 16 Segmente geteilt (0 – 15) je Segment von 256 MBytes
- Die höchsten 4-bits selektieren die Segmente.
- Die ersten 16Kbytes vom jedem Segmente werden für absolute Bit Adressierung mit **set-bit** oder **clair-bit** Instruktionen benutzt.

# Memory


Segment
0
1
2
3
14
15
Local Static Data
Local Dynamic Data
Local Code
External Peripherals
Internal Peripherals
System Bus Memory
& Peripherals

# Adresse Mode

| Addressing Mode | Address Register Use | Offset Size (bits) |
|---|---|---|
| Absolute | None | 18 |
| Base + Short Offset | Address Register | 10 |
| Base + Long Offset | Address Register | 16 |
| Pre-increment | Address Register | 10 |
| Post-increment | Address Register | 10 |
| Circular | Address Register Pair | 10 |
| Bit-reverse | Address Register Pair | — |

- Tricore unterstützt 7 Adressierungs Modes

# Datentypen und Format

- Tricore unterstützen Operationen von
  - Boolean
  - Bit String
  - Signed/Unsigned Integer
  - Charakter
  - Signed Fraction
  - single-precision floating-point numbers
- meisten Instruktionen arbeitet mit einem spezifische Datentypen
- den Rest manipuliert verschiedene Datentypen

# Instruktion Set

- Tricore unterstützt 16-Bit und 32-Bit Instruktionsformate
- 16-Bit Instruktionen werden mit Nullen auf 32-Bit ergänzt (Saturation)

| Mnemonic | Definition | Mnemonic | Definition |
|---|---|---|---|
| LDMDST | Load modify store | OR | Logical OR |
| LDUCX | Load upper context | OR.comp | Compare, OR and accumulate |
| LEA | Load Effective address | OR.logic | Bit OR logical accumulate |
| LOOP | Loop | ORN | Logical OR Not |
| LT | Less than | RET | Return from call |
| MADD | Multiply/add; 32-bit result | RFE | Return from Exception |
| MADDM | Multiply/add; 64-bit result | RSLCX | Restore lower context |
| MADDMS | Multiply/add with saturation; 64-bit result | RSTV | Reset overflow flags |
| MADDR.0 | Multiply/add/round; 16-bit result | RSUB | Reverse subtract |
| MADDRS.0 | Multiply/add/round with saturation; 16-bit result | RSUBS | Reverse subtract with saturation |
| MADDS | Multiply/add with saturation; 32-bit result | SAT | Saturate result |
| MAX | Maximum value | SEL | Select |
| MFCR | Move from Core Register | SELN | Select Not |
| MIN | Minimum value | SH | Shift |
| MOV | Move | SH.comp | Compare accumulate and shift |
| MOVH(.A) | Move halfword to address | SH.logic | Bit shift logical accumulate |
| MOVZ.A | Move zero to address | SHA | Arithmetic shift |
| MSUB | Multiply/subtract; 32-bit result | SHAS | Arithmetic shift with saturation |
| MSUBS | Multiply/subtract with saturation; 32-bit result | ST | Store |
| MSUBM | Multiply/subtract; 64-bit result | STLCX | Store lower context |
| MSUBMS | Multiply/subtract with saturation; 64-bit result | STUCX | Store upper context |
| MSUBR | Multiply/subtract/round; 16-bit result | SUB | Subtract |
| MSUBRS | Multiply/subtract/round with saturation; 16-bit result | SUBC | Subtract with carry |
| MTCR | Move to Core Register | SUBS | Subtract signed with saturation |
| MUL | Multiply: 32-bit result | SUBSC | Subtract scaled address |
| MULS | Multiply with saturation: 32-bit result | SUBX | Subtract extended |
| MULM | Multiply; 64-bit result | SVLCX | Save lower context |
| MULR | Multiply/round: 16-bit result | SWAP | Swap |
| NAND | Logical NAND | SYSCALL | System call |
| NE | Not equal | TRAPV | Trap on overflow |
| NEZ.A | Not equal zero address | TRAPSV | Trap on sticky overflow |
| NOP | No operation | XNOR | Logical exclusive NOR |
| NOR | Logical NOR | XOR | Logical exclusive OR |
| NOT | Bitwise complement | XOR.comp | Compare, XOR and accumulate |

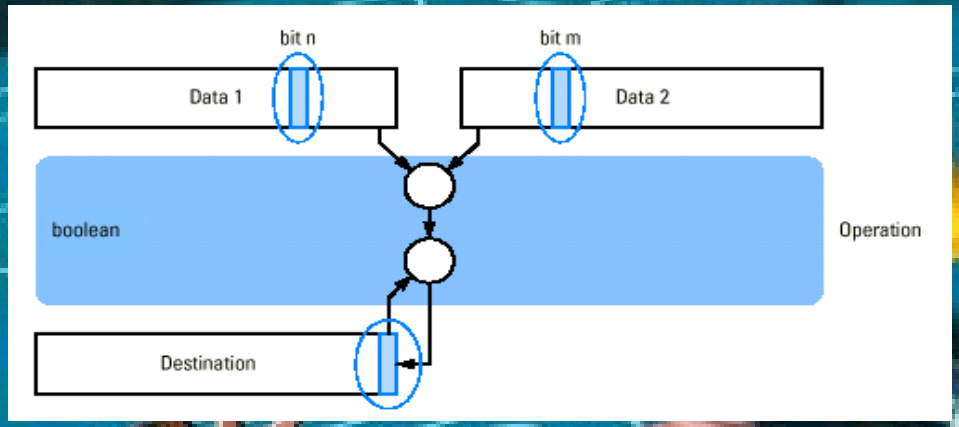
# Instruktion Set

- Load /Store Instruktion
  - Benutzt die 7 Adressierungs Modes um Daten zwischen Register und Speicher zu bewegen
- Arithmetische Instruktionen
  - Arbeiten mit Daten und Adressen im Register.
  - Status Informationen über die Ergebnisse der Arithmetische Operationen werden in die 5 Status Flags eingetragen.
- Arithmetische Instruktion wird in drei Kategorien unterteilt
  - Integer Instruktion
  - DSP Arithmetische Instruktionen
  - Packed Arithmetische Instruktionen

# Vergleich & Branch Instruktionen

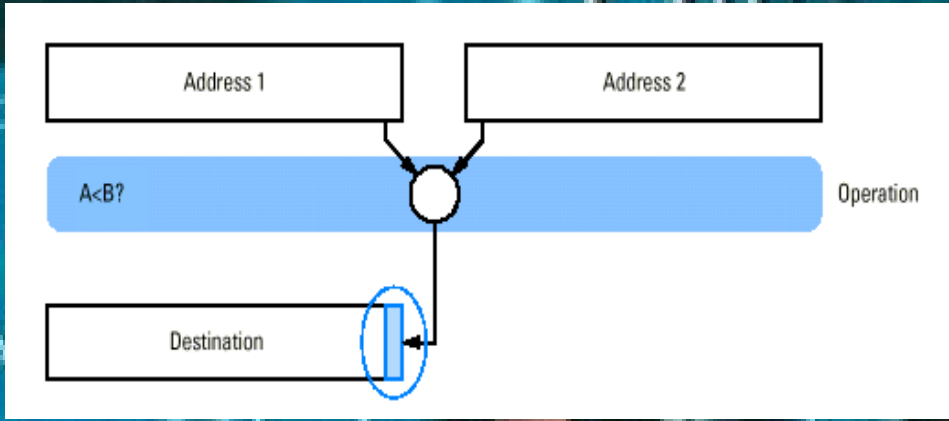
- Vergleich Instruktionen
  - wird benutzt um den Inhalte von zwei Registern zu vergleichen
  - Das boolean Ergebnis (1 = wahr 0 = falsch) wird in dem niedrigsten Bit vom dem Ziel Daten Register abgelegt, die höheren 31 Bit werden mit Nullen gefüllt
- Branch Instruktionen
  - Verändern den Ablauf des Programms durch  modifizieren des Inhalts des Programmcounters
  - Es gibt zwei Arten vom Branch Instruktionen
  - Bedingter Branch
  - Unbedingter Branch
  - Der Branch ist abhängig vom dem Ergebnis der boolean Operation

# Bit Operation




- Bit Operationen
  - 8 Instruktionen für kombinatorische Logik
  - Funktionen mit zwei Inputs
  - 12 Instruktionen mit drei Inputs
  - 1-Bit Ergebnis von zwei Input Funktionen wird in dem niedrigsten Bit vom Ziel-Daten-Register, die höheren 31 werden mit Nullen gefüllt

# Adress-Arithmetik

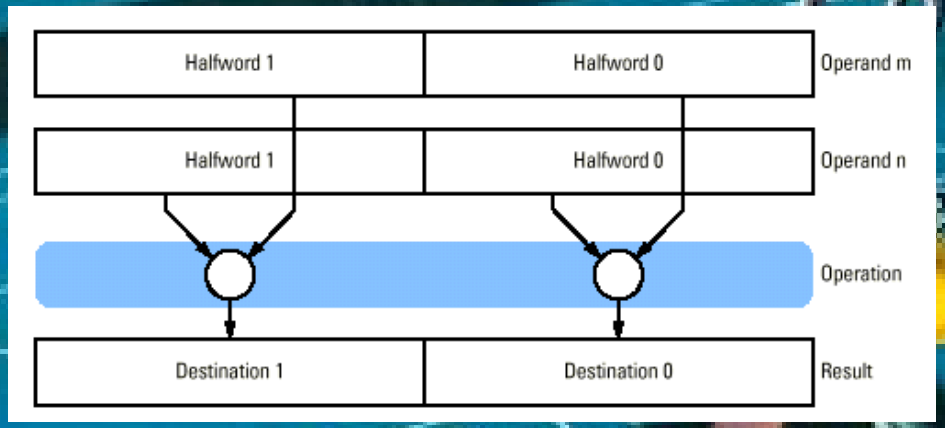


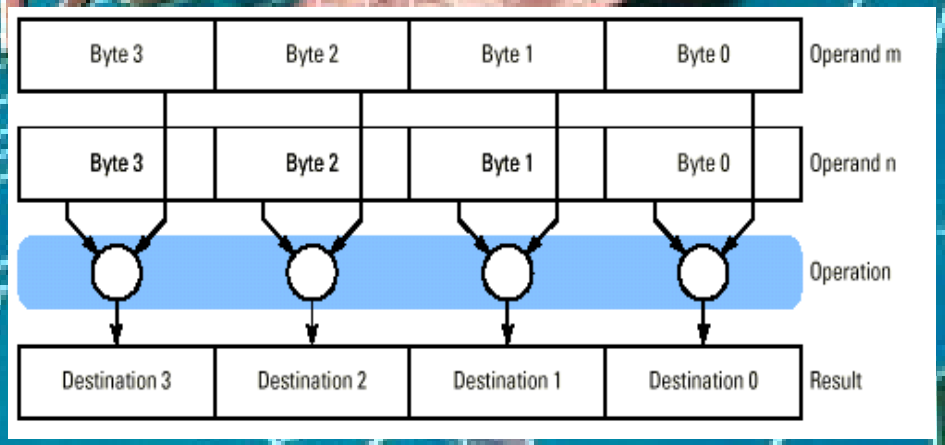

- Wie bei den Branch Instruktionen, die die Daten Register benutzen wird das Ergebnis des Vergleichs von Adresse Registern in dem niedrigsten Bit des Ziel Daten (?) Registers abgelegt; die höheren 31 Bits werden mit Nullen gefüllt.

# Packed Arithmetik

- DSP & Packed Arithmetik
  - DSP Arithmetik Instruktionen verwenden 16-Bit “signed fractional” Daten und 32-Bit “signed fractional” Daten
  - 16-Bit DSP Daten werden in die obere Hälfte des Daten Registers geladen und der Rest mit Nullen gefüllt.
  - Packed Arithmetik Instruktionen teilen ein 32-Bit Wort in mehrere identische Objekte und wird gleich geholt, gespeichert und parallel verarbeitet.
- Tricore unterstützt zwei Packed Format
  - 16- und 32-Bit

# Packed Arithmetik



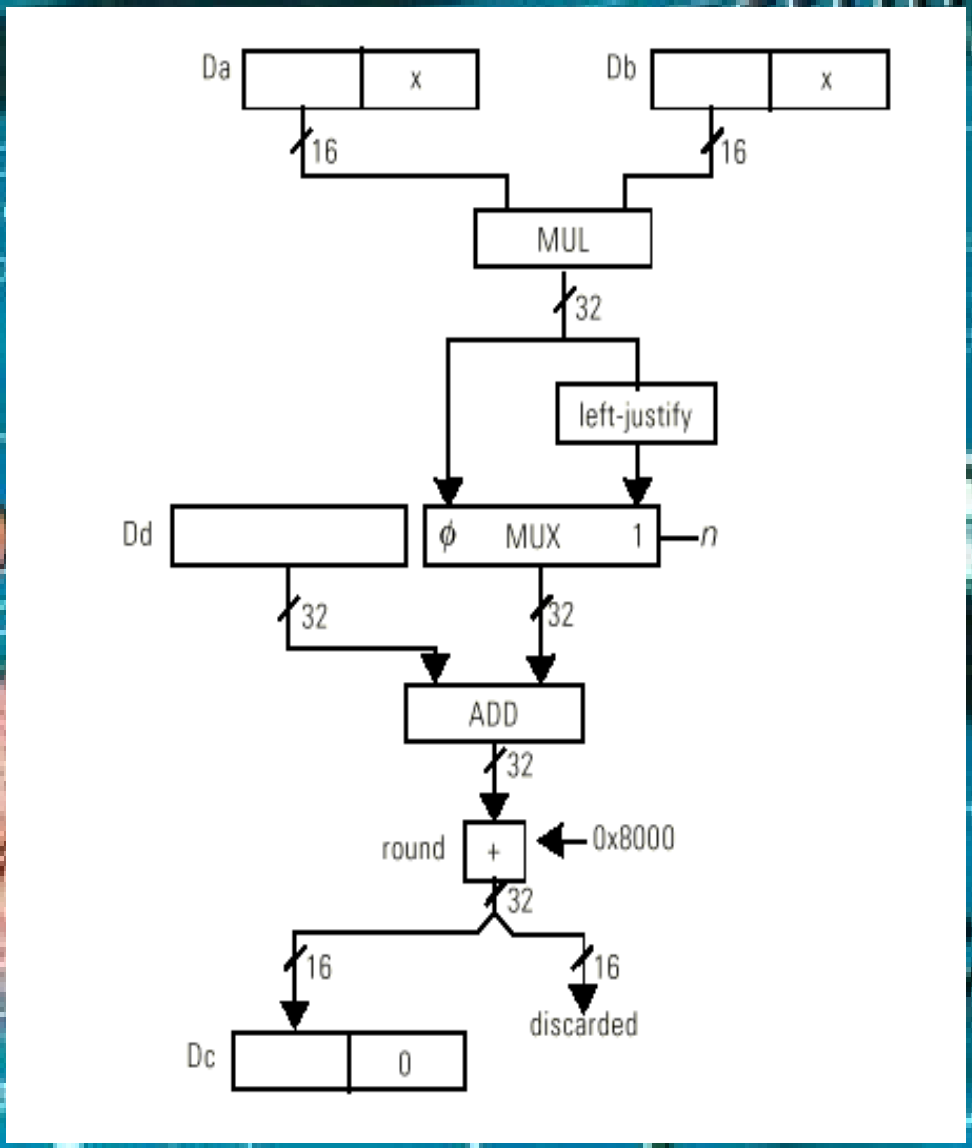




- Der erste Befehlsvorrat teilt die 32-bit in zwei 16-bit werte. Diese Instruktionen haben die Endung “ H” und “ HU”
- Der zweite teilt die 32-Bit in vier 8-Bit werte. Diese Instruktionen haben die Endung “ B” und “ BU”

# MADDR.Q

- Packed Multiply-Add with Rounding  (16x16)+32 =>16 Bit
- syntax:
  - maddr.q Dc, Dd, Da, Db, n(RRR)
- z.B.: maddr.q d3, d4, d1, d2, 0
- Multipliziert das höchste signifikante Halbwort von Daten Register Da mit dem höchsten signifikanten Halbwort vom Daten Register Db
- Das Produkt wird addiert und nach links geschoben wenn n=1 und in den oberen Teil des Register Dd geschrieben
- Inhalt von Register Dd wird gerundet und sein höchstes signifikantes Halbwort wird ins höchste signifikante Halbwort Dc geschrieben und die niedrigeren signifikanten Bits werden auf 0 gesetzt.
- Die Operanden werden als Werte mit Vorzeichen behandelt

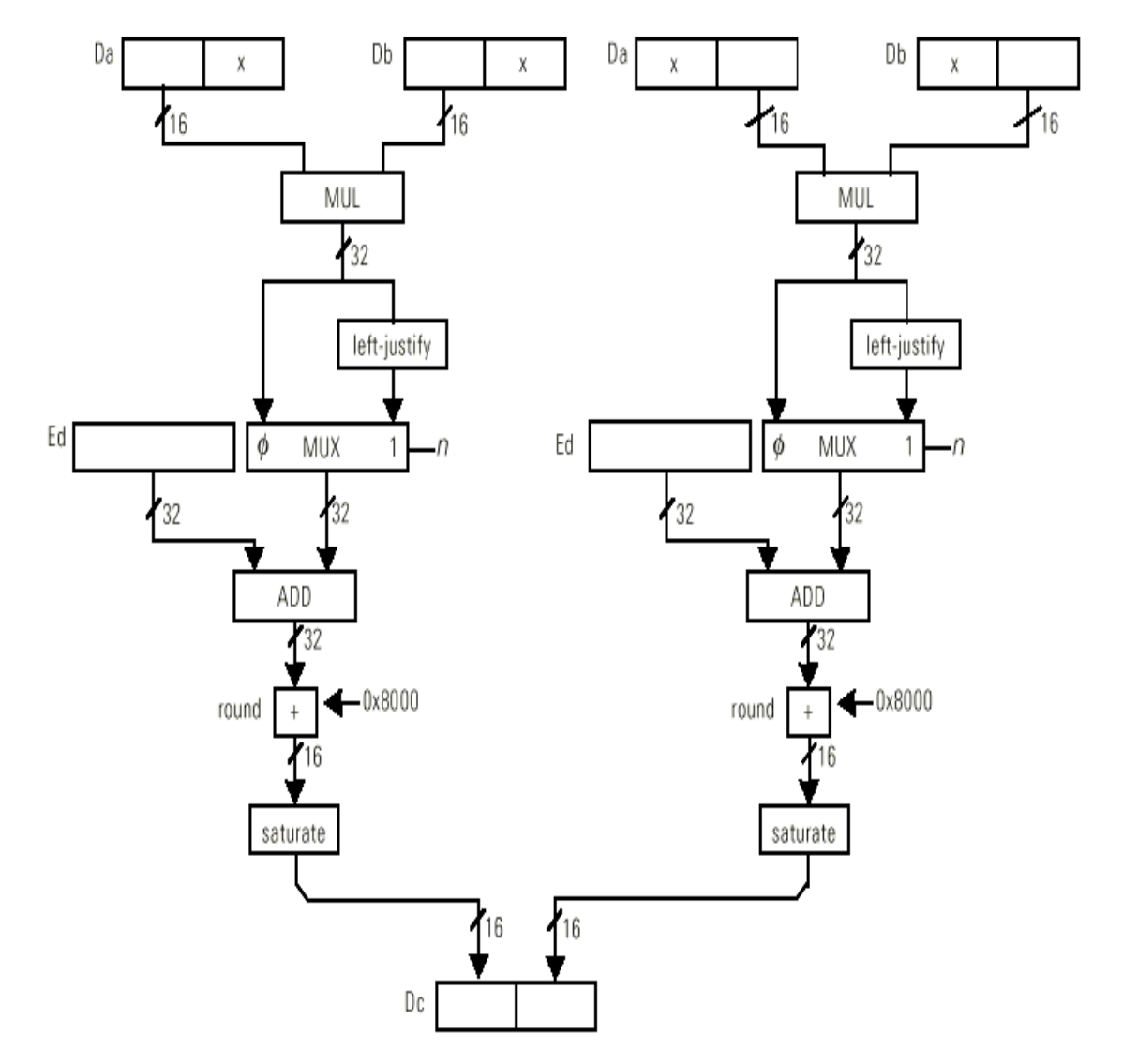
# MADDR.Q


Da
x
Db
x
16
16
MUL
32
left-justify
Dd
ϕ
MUX
1
n
32
32
ADD
32
round
+
0x8000
32
16
16
discarded
Dc
0

# MADDR.H

- Packed Multiply-Add mit Rundung and Saturation
  - (16x16)+32=>16
  - (16x16)+32=>16
- syntax:
  - MADDR.H Dc, Ed, Da, Db, n(RRR)
- z.B.: maddr.h d3, e4, d1, d2, 0
- Multipliziert das höchste signifikante Halbwort von Daten Register Da mit dem höchsten signifikanten Halbwort vom Daten Register Db
- Das Produkt wird addiert, nach links geschoben wenn n=1 ist und in die oberen 32 bit von Register Ed geschrieben
- Der Inhalt von Register Ed wird gerundet und sein höchstes signifikantes Halbwort wird in das höchste signifikante Halbwort von Dc geschrieben
- Multipliziert das niedrigste signifikante Halbwort des Daten Registers Da mit dem niedrigsten signifikanten Halbwort von Db
- Das Produkt wird addiert, nach links geschoben wenn n=1 ist und in die niedrigeren 32 Bit von Register Ed geschrieben
- Der Inhalt von Register Ed wird gerundet und sein höchstes signifikantes Halbwort wird ins niedrigere signifikante Halbwort von Dc geschrieben
- Die Operanden werden als Werte mit oder ohne Vorzeichen behandelt

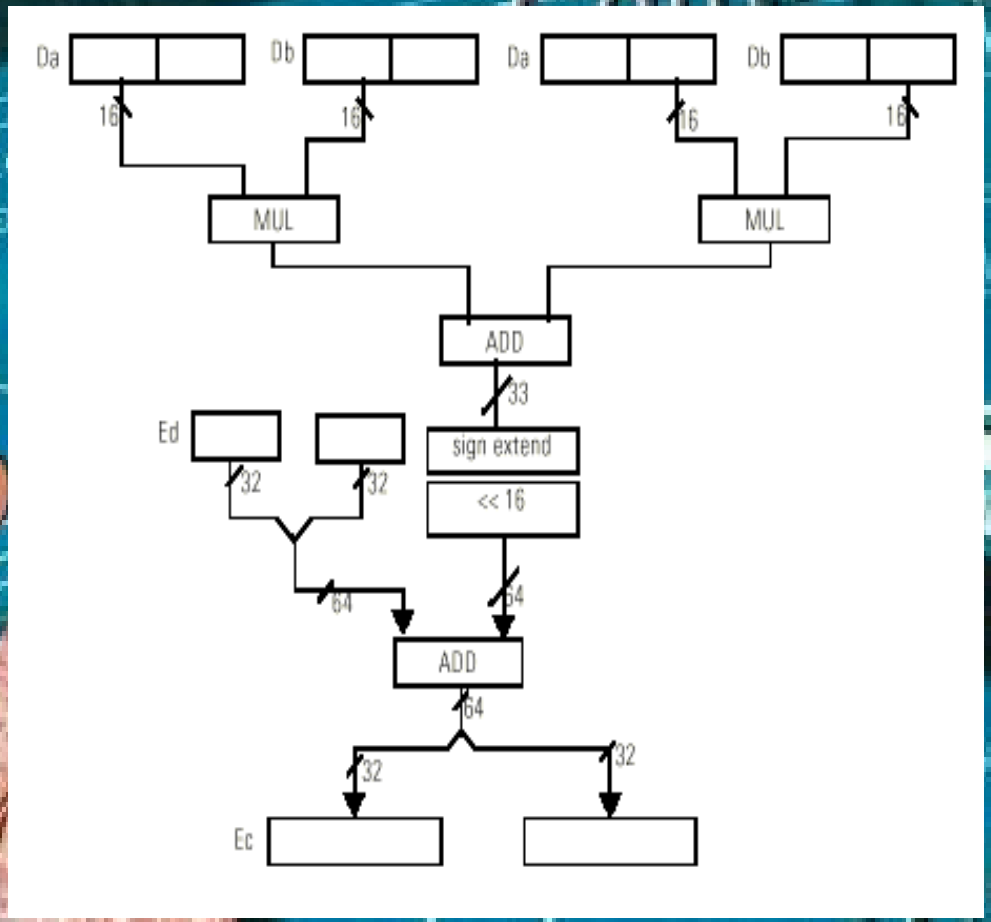
# MADDR.H


Da
x
Db
x
16
16
MUL
32
left-justify
Ed
ϕ MUX 1
n
32
32
ADD
32
round
+
0x8000
16
saturate
16
16
Dc
Da
x
Db
x
16
16
MUL
32
left-justify
Ed
ϕ MUX 1
n
32
32
ADD
32
round
+
0x8000
16
saturate

# MADDM.H

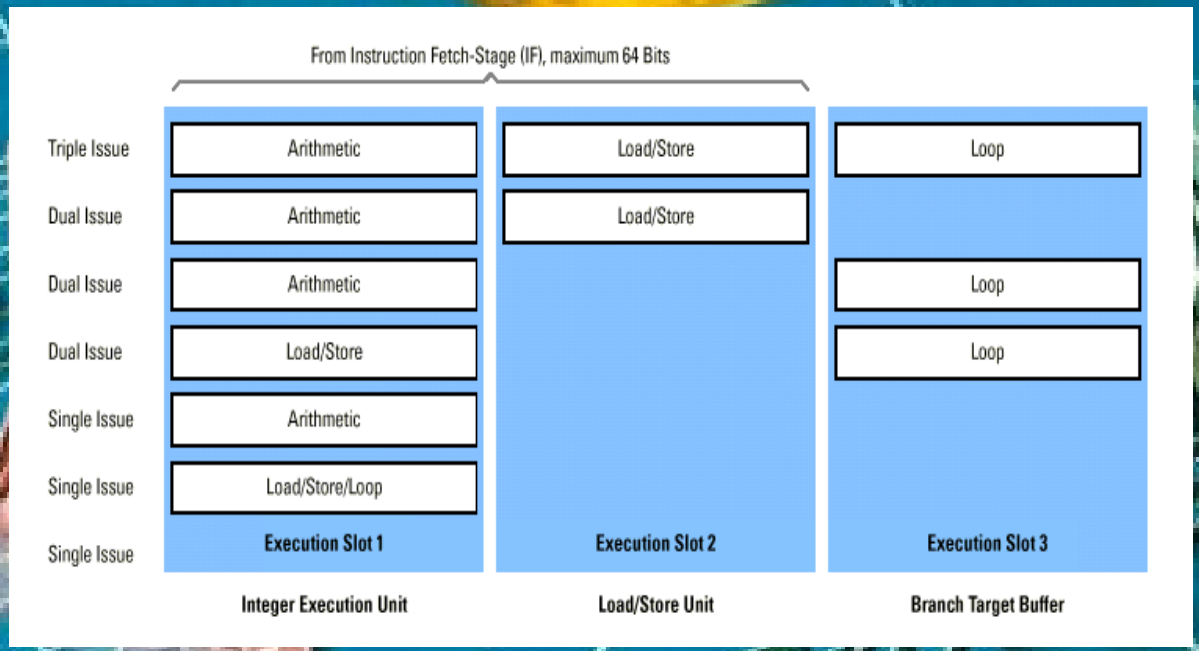
- Packed Multiply-Add with Multiword Result
  - (16x16)+(16x16)+64=>64 Bit
- syntax:
  - maddm.h Ec, Ed, Da, Db(RRR)
- z.B.: maddm.h e0, e4, d2, d7
- Multipliziert das höchste signifikante Halbwort des Daten Registers Da mit dem höchsten signifikanten Halbwort von Daten Register Db
- Multipliziert das niedrigste signifikante Halbwort des Daten Registers Da mit dem niedrigsten signifikanten Halbwort vom Daten Register Db
- Die Produkte von beiden Multiplikationen werden mit Vorzeichen behaftet auf 64 Bit, dann nach links 16 Bit geschoben.
- Addiert es mit Inhalt von Ed und speichert das Ergebnis in Ec in 64 Bit-Form.
- Die Operanden werden als Werte mit Vorzeichen behandelt

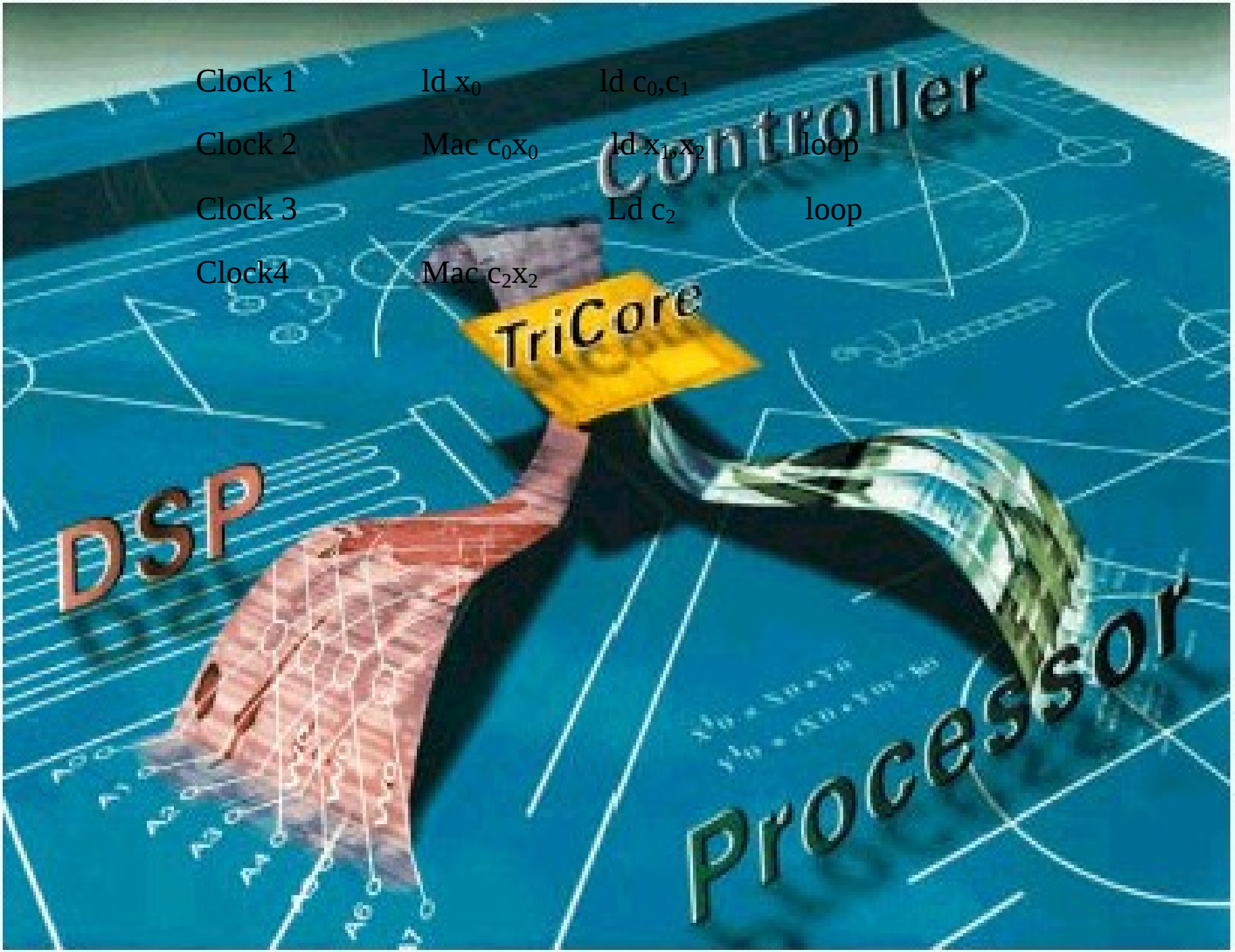
# MADDM.H


Da
Db
Da
Db
16
16
16
16
MUL
MUL
ADD
33
Ed
sign extend
<< 16
32
32
64
64
ADD
64
32
32
Ec

# DSP-Beispiele

$$\sum_{i=0}^{2} c_i x_i = c_0 x_0 + c_1 x_1 + c_2 x_2$$




Clock 1 ld $x_0$ ld $c_0$,$c_1$
Clock 2 Mac $c_0x_0$ ld $x_1$,$x_2$ loop
Clock 3 Ld $c_2$ loop
Clock4 Mac $c_2x_2$
Controller
TriCore
DSP
Processor

# Zusammenfassung

- „Unified Instruction Set“ ermöglicht die Entwicklung eines tools für kombinierte µC und DSP Aufgaben
- „fast task switch“ und Interrupt Behandlung unterstützen RTOS, HLL-Programmierung und debug
- Höhere Leistung, zuverlässige Operation und weniger Energieverbrauch wird durch größere Speicherkapazität auf dem Chip erzielt
- Tricore ermöglicht direkte Kontrolle von Peripherie on-chip ohne zusätzliche control logic für Peripherie.
- Code wird um ca. 30% bis 40% verringert durch gemischte 16-Bit und32-Bit Instruktion
- Tricore Architektur verbindet Microcontroller- und DSP-Eigenschaften ohne, daß sich diese gegenseitig behindern
- ermöglicht eine sauberes, schnelles und effizientes Multi-tasking
- ermöglicht preisgünstige Programmdurchführung mit Hilfe von Superskalar
- SAB-TC10