

Eingebettete Prozessorarchitekturen

5. Konfigurierbare Prozessorarchitekturen

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Wintersemester 2010/2011

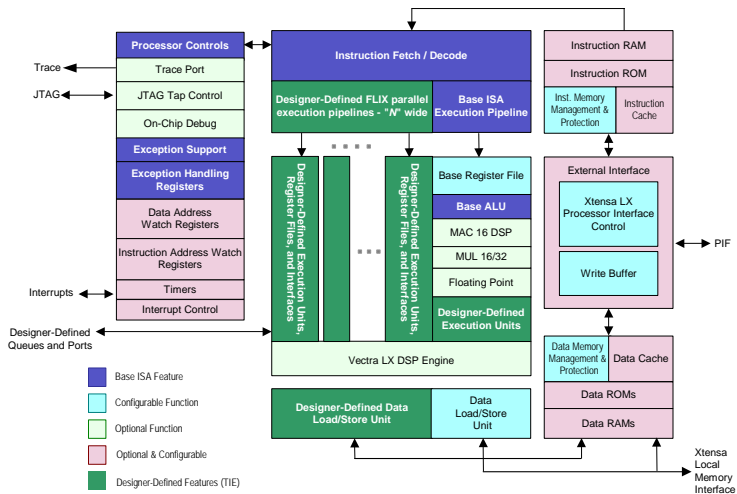
- Arbeitsplatzrechner werden demnächst für 1. Aufgabe freigeschaltet
 - Wahrscheinlich nächste Woche
- VLIW-Teil beendet
- Nächstes Thema: Konfigurierbare Prozessoren
 - ... gelegentliche Ähnlichkeiten zu VLIW

- Auszüge aus Trainingsmaterial der Fa. Tensilica
 - Insbesondere alle Zeichnungen und technischen Daten
- Material ist **vertraulich**
 - Nur für die Lehre zur Verfügung gestellt
 - Darf **nicht** weiterverbreitet werden
- Weiterführende Informationen
 - Tensilica Dokumentation (in FG ESA installiert)
 - Buch “Engineering the Complex SOC” von Chris Rowen

- Können an Erfordernisse der Anwendung(en) angepasst werden
- Genauer: Solche Prozessoren sind
 - Konfigurierbar** Komponieren **vordefinierter** Funktionen
 - Erweiterbar** Hinzufügen **neuer** Funktionen
- Verschiedene Anbieter
 - Tensilica (Marktführer)
 - ARC
 - ARM (in Grenzen durch Hersteller konfigurierbar)

Tensilica LX Architektur

A. Koch



- RISC-artig
- Harvard-Architektur (getrennte I+D Speicher)
- Pipeline mit 5 oder 7 Stufen
 - An Zieltaktfrequenz anpassbar
- Normale Instruktionslänge 24b
- Viele Instruktionen auch platzsparend als 16b
- VLIW-Betriebsart mit 32b oder 64b Instruktionen
- Registerfenster von 16 logischen Registern
- 16 oder 32 physikalische Register
- Bis zu zwei Load-Store-Units

- Zusätzliche Instruktionen
 - Kommt später noch im Detail!
- Zusätzliche Schnittstellen
 - I/O
 - Interprozessorkommunikation
 - Schneller Speicher
 - Koprozessoren

- Load/Store
- Move
- Shift
- Arithmetik
- Kontrolltransfer
- Steuerung

➔ Ca. 80 Instruktionen **immer** verfügbar

Konfigurierbare optionale Instruktionen

Vom Benutzer selektiv zum Basissatz hinzufügar

- Schleifen ohne Zeitverlust (zero-overhead loops)
- Multiplikation
- Normalisierung von Zahlen
- Minimum, Maximum
- Gesättigte Arithmetik
- Vorzeichenerweiterung
- Bedingungsregister und -befehle (ähnlich Lx/Vex!)
- Gleitkommakoprozessor
- 128b Vektor- und DSP-Einheit
- Multiprozessorsynchronisation

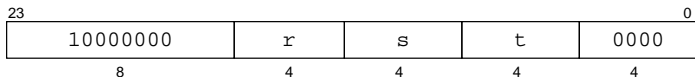
A. Koch

➔ Übersicht in *Xtensa LX Microprocessor Overview*

Basisbefehlssatz kodiert in 24b

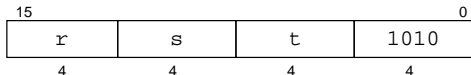
ADD ar, as, at $AR[r] \leftarrow AR[s] + AR[t]$

A. Koch



Häufige Instruktionen auch kompakt in 16b darstellbar (.N)

ADD.N ar, as, at $AR[r] \leftarrow AR[s] + AR[t]$



Kleine konstante Werte (*immediates*) können direkt im Befehl untergebracht werden

Flexible Length Instruction eXtensions

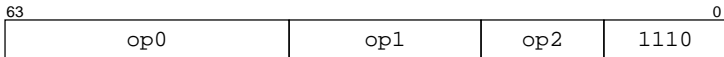
VLIW-artige Betriebsart

- 32b oder 64b Instruktionen
- Können Mix aus 24b oder 16b Operationen enthalten

A. Koch

Beispiel:

```
60000520 { l8ui a10, a11, 0; addi a9, a9, 2; abs a12, a10 }
60000528 { l8ui a8, a9, 0; sub a14, a14, a13; add a13, a8, a12 }
60000530 { l8ui a14, a11, 1; addi a11, a11, 2; abs a12, a14 }
60000538 { l8ui a13, a9, 1; sub a10, a10, a8; add a8, a13, a12 } sad+0x70
```



- 32 oder 64 allgemeine Register (AR), 32b breit
 - “AR” steht laut Tensilica für Address Register
- Optionale Registerfelder
 - 16 Bedingungsregister (1b)
 - 16 Gleitkommaregister (32b)
 - 4 MAC-Zwischenregister (32b)
 - MAC = Multiply Accumulate
 - 16 Vektorregister (160b)
 - Benutzerdefinierte Registerfelder
- Spezialregister
 - Z.B. für Schleifengrenzen, Rundungsmodi, etc.
 - Benutzerdefinierte Speicherelemente

- In der Regel zwei Lese-Ports und ein Schreib-Port
- FLIX-Instruktionen können aber mehr Ports anfordern
- Physikalische ARs organisiert in Fenster von 16 **logischen** Registern `a0 . . . a15`
- Benutzt für Parameterübergabe bei Prozeduraufruf

Registerfenster

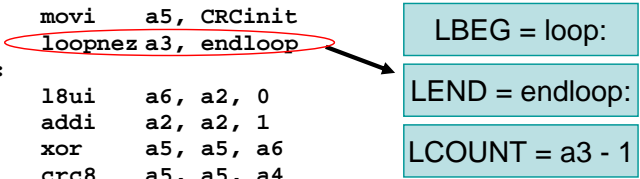
**Aufrufer
Register
Window**

	AR31	
	AR30	
	AR29	
	AR28	
	AR27	
	AR26	
	A625	
	AR24	
	AR23	a15
	AR22	a14
	AR21	a13
	AR20	a12
	AR19	a11
	AR18	a10
	AR17	a9
	AR16	a8
a15	arg5	a7
a14	arg4	a6
a13	arg3	a5
a12	arg2	a4
a11	arg1	a3
a10	arg0 - return value	a2
a9	Caller - SP	a1
a8	Return Addr	a0
a7	AR7	
a6	AR6	
a5	AR5	
a4	AR4	
a3	AR3	
a2	AR2	
a1	Caller - SP	
a0	AR0	

**Aufgerufene P.
Register
Window**

- Ersetzt “normale” Schleifeninstruktionen
 - add, cmp, branch
- Ohne Pipelining-Zeitverlust von bedingten Sprüngen
- **Nicht** verschachtelbar

```
    movi    a5, CRCinit
loop: loopnez a3, endloop
    l8ui    a6, a2, 0
    addi    a2, a2, 1
    xor     a5, a5, a6
    crc8    a5, a5, a4
endloop:
```



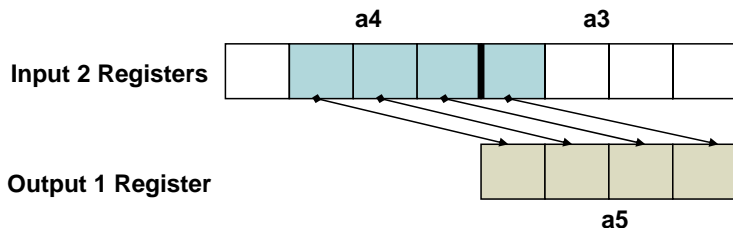
LBEG = loop:

LEND = endloop:

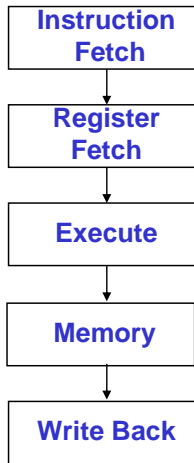
LCOUNT = a3 - 1

- Extraktion von 32b Datenfeldern aus bit-orientierten Datenströmen
 - Beispielsweise Netzwerkpaketen

A. Koch



```
ssai 24          //set shift amount to 24bits  
src a5, a4, a3  //shift right combined to a5
```

I: Local Instruction Memory Read

R: Instruction Decode
Register Access

M: Memory Access

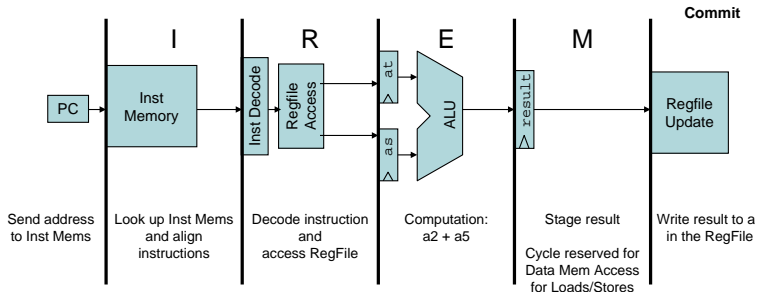
W: Register Write Back

LX Pipeline

Klassischer Ablauf → TGD12

A. Koch

```
6000117f: ...  
60001181: add.n a3, a5, a2  
60001183: ...
```



Zugriff auf geladene Daten

```
l32i a5,a4, 4  
add a3,a5,a2
```

Pipeline bubble due to load dependency



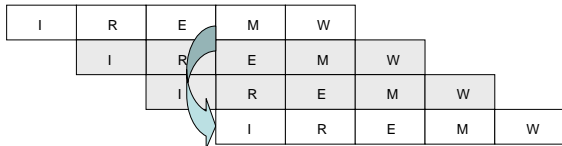
Load data
bypass

A. Koch

Bedingter Sprung

```
beq a5, a4, L0  
add ...  
abs ...  
L0: mov a7, a5
```

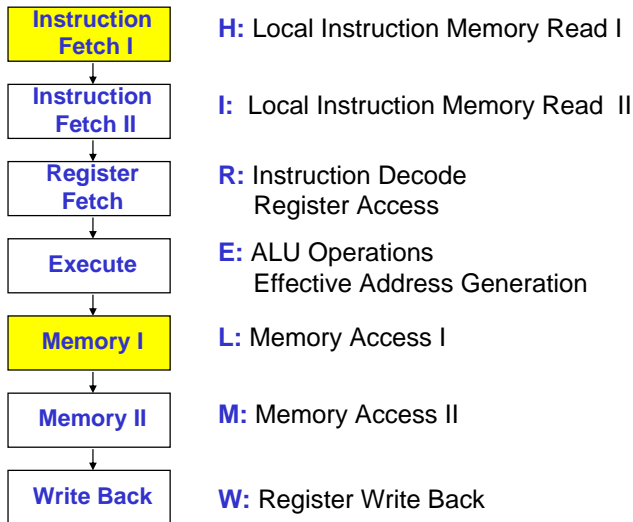
Jump/branch instruction causes 2 cycle penalty



Target address if branch is taken

Verlängerte Pipeline

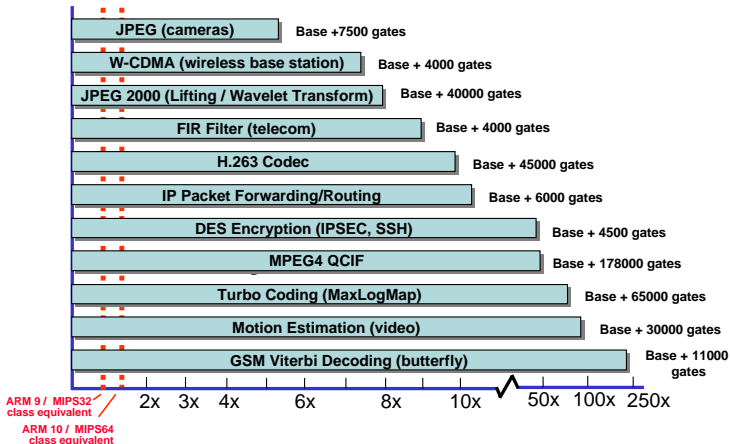
Erlaubt höhere Taktfrequenz bzw. langsameren Speicher



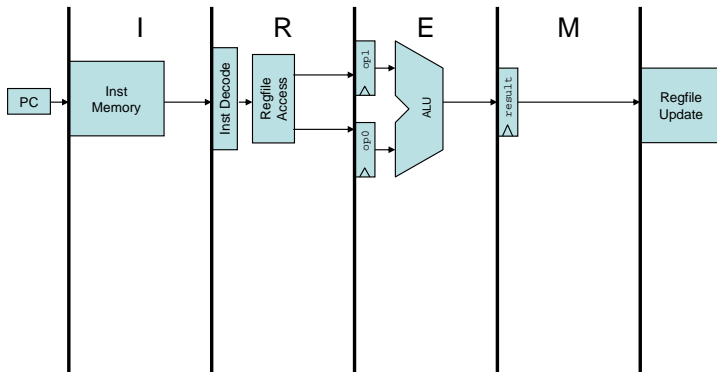
- Tensilica Instruction Extensions (TIE)
- **Neue** Befehle
- **Neue** Register und Registerfelder
- Spezialschnittstellen
- Formuliert in eigener Sprache
 - Stellenweise ähnlich zu Verilog

Gewinn an Rechenleistung durch TIEs

A. Koch



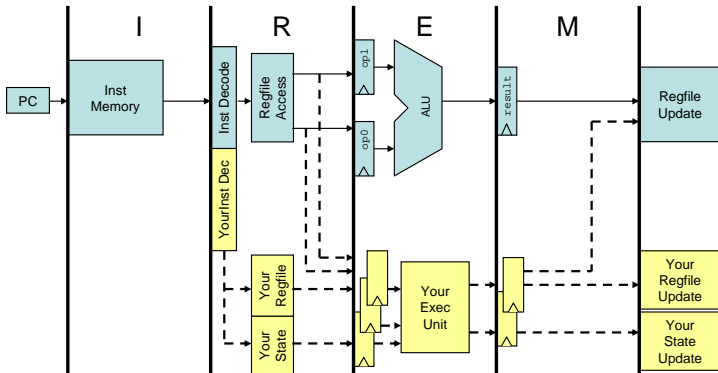
Erweiterung der Pipeline



A. Koch

Erweiterung der Pipeline

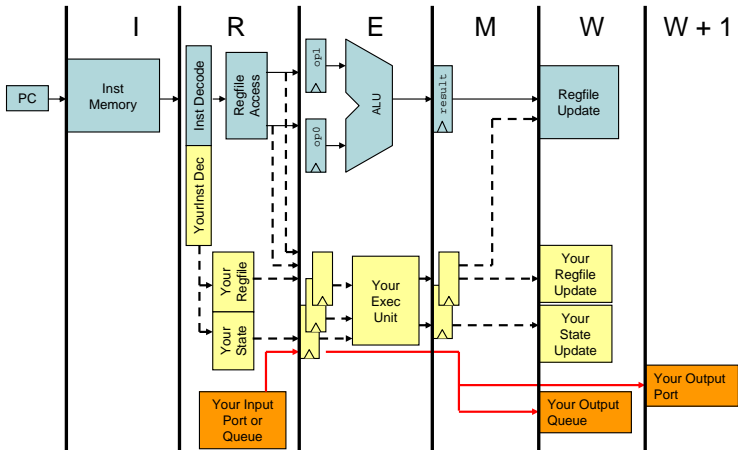
Neue Ausführungseinheiten



A. Koch

Erweiterung der Pipeline

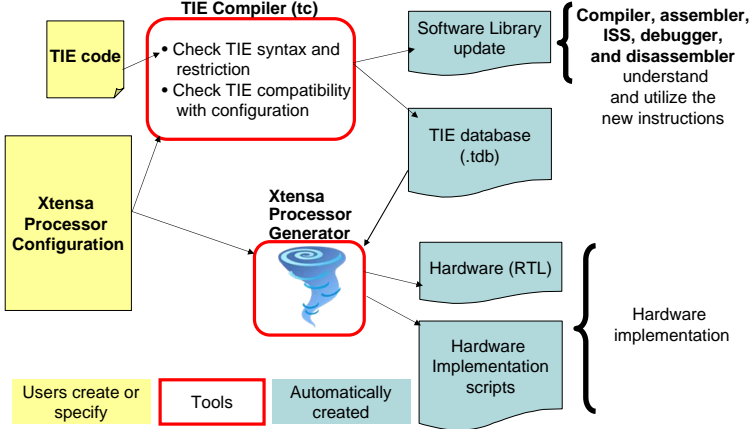
Neue Schnittstellen



A. Koch

Entwurfsfluß

A. Koch



TIE: Ein erstes Beispiel

Ursprüngliches C-Programm

myprogram.c

```
unsigned int add_array(unsigned char * data) {  
    int i;  
    unsigned int sum = 0;  
    for (i = 0; i < SIZE; i++) {  
        sum += data[i];  
    }  
    return sum;  
}
```

Idee: Betrachte ein 32b AR-Register als 4 einzelne 8b-Werte und addiere diese auf.

TIE: Ein erstes Beispiel

TIE-Code der neuen Instruktion

mytiefile.tie

```
operation ADD_BYTES {out AR sum, in AR fourbytes } {} {  
  
    assign sum = fourbytes[7:0] + fourbytes[15:8] +  
                fourbytes[23:16] + fourbytes[31:24];  
}
```

- `operation` beschreibt Name, Format und Schnittstelle
- Ist so schon ausreichend für viele Anwendungen

mytiefile.tie

```
operation ADD_BYTES {out AR sum, in AR fourbytes } {} {  
  
    assign sum = fourbytes[7:0] + fourbytes[15:8] +  
                fourbytes[23:16] + fourbytes[31:24];  
}
```

- **ADD_BYTES** ist der Name der neuen Instruktion
- TIE-Compiler erzeugt Opcode für neue Instruktion
- Alle Software-Werkzeuge werden aktualisiert und kennen nun neue Instruktion
 - Aber: C-Compiler benutzt sie **nicht** automatisch
 - Dafür andere Vorgehensweise → XPRES Compiler

mytiefile.tie

```
operation ADD_BYTES {out AR sum, in AR fourbytes } {}  
  
    assign sum = fourbytes[7:0] + fourbytes[15:8] +  
                fourbytes[23:16] + fourbytes[31:24];  
}
```

- Zwei Arten von Schnittstellen zur Instruktion
 - 1 Register (**AR**) und konstante Werte (immediates)
 - 2 Zustand und I/O → hier leer, kommt später ...
- Hier: Instruktion liest ein AR und schreibt ein AR
 - Auch möglich: **inout** für Operation im selben AR

Innenleben der TIE-Instruktion

Beschreibung der Semantik

mytiefile.tie

```
operation ADD_BYTES {out AR sum, in AR fourbytes } {} {  
    assign sum = fourbytes[7:0] + fourbytes[15:8] +  
                fourbytes[23:16] + fourbytes[31:24];  
}
```

A. Koch

- Ähnlich zu Verilog
- Hier also einfache kombinatorische Logik
 - Liest ein AR-Register (als einzelne Bytes)
 - Addiert diese auf
 - Schreibt Ergebnis in weiteres AR-Register
- Schlüsselworte
 - wire** Definiert lokale Variable, kann einmal zugewiesen werden
 - assign** Zuweisung an Wire oder Register, (Speicherelement, Output-Port)

Beispiele für TIE-Operatoren

Sehr ähnlich zu Verilog

Bit-Auswahl **a** [**n:m**]

Arithmetik **+**, **-**, *****

Logik **!**, **&&**, **||**

Vergleich **>**, **<**, **>=**, **<=**, **==**, **!=**

Bit **~**, **&**, **|**, **^**, **~&**, **~|**, **~^**

Reduktion **&**, **|**, **^**, **~&**, **~|**, **~^**

Schieben **<<**, **>>**

Konkatenation **{, }**

Replikation **{n{...}}**

Bedingter Ausdruck **?:**

Konstante **32'd12** (=12), **8'h10** (=16), **4'b0101** (=5)

Nullerweiterung **assign myreg[31:0] = 8'hff**

setzt **myreg** auf **0x000000ff**

➡ *Tensilica TIE Language Reference Manual*

- TIE-Code an sich ist sensitiv für Groß/Kleinschreibung
- Eindeutige Namen für neue Instruktionen vergeben
- Aber TIE-Assembler beachtet **nicht** Groß/Kleinschreibung für Instruktionsnamen
 - **Add_Bytes** ist z.B. gleich **ADD_BYTES**
- Namen dürfen nicht mit “_” oder “**TIE_**” anfangen

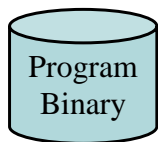
- TIE-Compiler `tc` übersetzt TIE-Programm
- Dann automatische Anpassung der Werkzeuge

Assembler

C/C++ Compiler Erlauben Benutzung der neuen
Instruktion

Debugger Zeigt neue Instruktion an, erlaubt
Einzelschritt, etc.

Disassembler Dekodiert und zeigt neue Instruktion



Disassembler
(`xt-objdump`)



`mynewprogram.S`

```
...  
132i.n a9, a2, 0  
add_bytes a5, a9  
...
```

mynewprogram.c

```
#include <xtensa/tie/mytiefile.h>
...
unsigned int add_array(unsigned char * cdata) {
    unsigned int *data = cdata;
    unsigned int sum = 0;

    for (i = 0; i < SIZE/4; i++) {
        sum += ADD_BYTES(data[i]);
    }
    return sum;
}
```

A. Koch

- TIE-Compiler hat .h Definitionsdatei erstellt
 - Kapselt neue Instruktion als C-Funktion
- Kapselungsfunktion nun normal aufrufen
- C-Compiler schreibt zugrundeliegende neue Instruktion in ausführbares Programm

- Konfigurierbare Prozessoren
 - Konfiguration
 - Erweiterung
- Tensilica LX
 - Instruction Set Architecture
 - Mikroarchitektur (Pipeline)
 - Erweiterte Pipeline
- Beispiel für neue Instruktion
- TIE-Code
 - ➡ davon nächste Woche mehr