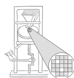


Eingebettete Prozessorarchitekturen

6. Befehlssatzerweiterung mit TIE

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Wintersemester 2006/2007

Material und Literatur

- 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

TIE-Beispiel: Führende Null-Bits zählen

A. Koch

- Nützlich z.B. bei Gleitkommaberechnungen
    - Normalisierung

lz1.tie

```
operation lz1 {out AR leading_zeros, in AR input } {} {
        assign leading_zeros = input[31] == 1'b1 ? 32'd0
                              : input[30] == 1'b1 ? 32'd1
                              : input[29] == 1'b1 ? 32'd2
                              ...
                              : input[0]  == 1'b1 ? 32'd31
                              : 32'd32;
}
```

Hätte sehr langsame Realisierung in Software

TIE-Beispiel: Fehlerkorrektur mit CRC

CRC8 für ATM Header Error Correction

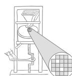
A. Koch

atmhec.tie

```
operation ATM_HEC_BYTE {out AR result, in AR myA, in AR myB} {} {
   wire [7:0] Y = myA[7:0];
   wire [7:0] X = myB[7:0];
   wire [7:0] X_next =
                 {Y[7] ^ X[7] ^ X[6] ^ X[5], Y[6] ^ X[6] ^ X[5] ^ X[4],
                  Y[5] ^ X[5] ^ X[4] ^ X[3], Y[4] ^ X[4] ^ X[3] ^ X[2],
                  Y[3] ^ X[3] ^ X[2] ^ X[1] ^ X[7],
                  Y[2] ^ X[2] ^ X[1] ^ X[0] ^ X[6],
                  Y[1] ^ X[1] ^ X[0] ^ X[6], Y[0] ^ X[0] ^ X[7] ^ X[6]};
       assign result = {24'h000000, X_next};
}
```

Beispiel für `wire`

Speichern von Zustand in TIE



A. Koch

```
#define SIZE 1024
unsigned short A[SIZE];
unsigned short B[SIZE];
unsigned int sum; …

for (i = 0; i < SIZE; i++) {
    sum += A[i] * B[i];
}
```

Diskussion: **`sum += a * b`**

- Drei Eingänge (**`a`**, **`b`**, **`sum`**), ein Ausgang (**`sum`**)
- Normaler LX-Assembler: Zwei Eingänge (*read-ports*), ein Ausgang (*write-port*)
- Aber in TIE möglich: Write-Port als Read-Write-Port benutzen

TIE-Beispiel: 16b Multiply-Accumulate (MAC)

Demonstriert **inout**-Parameter

A. Koch

my_mac16.tie

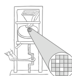
```
operation MYMAC16{inout AR accum, in AR A, in AR B}
  {}{
   assign accum = (A[15:0] * B[15:0]) + accum;
}
```

my_mac16.c

```
unsigned int sum;

sum =0;      //very important to init. sum first

for (i = 0; i < SIZE; i++) {
    MYMAC16(sum, A[i], B[i]);
}
```

Was, wenn nun noch ein weiterer Parameter hinzukommt?

Skalierter 16b MAC



A. Koch

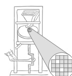
scaled_mac16.c

```
#define SIZE 1024
unsigned short A[SIZE];
unsigned short B[SIZE];
unsigned int sum;
unsigned int scale; …

for (i = 0; i < SIZE; i++)
    sum += ((A[i] * B[i]) >> scale);
```

- `sum += (a * b) >> scale`
  - Drei Eingänge, einen Ausgang
- Zuviel selbst für einfache TIE-Instruktion
  - Wäre im allerdings im FLIX-Modus möglich
- Lösung: Daten ausserhalb der Instruktion speichern
- Dediziertes Register in Prozessor anlegen: `state`

Beispiel für dedizierte Register

TIE-Instruktionen halten so *Zustand*



A. Koch

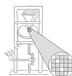
my_scaled_mac16.tie

```
state  scale  16

operation MYSCALEDMAC16 {inout AR accum, in AR A, in AR B}
 {in scale} {
  assign accum = ((A[15:0] * B[15:0]) >> scale) + accum;
}
```

State Usage

- Zusätzliches 16b Register namens `scale` angelegt
- Wird von Operation `MYSCALEDMAC16` referenziert
  - In zweitem Teil der Schnittstelle aufgeführt

Benutzung von TIE-Instruktionen mit Zustand
Einfache Verwendung



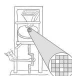
A. Koch

my_scaled_mac16.c

```
unsigned short * A;
unsigned short * B;
unsigned int sum;
int i;
…
sum = 0;
//We need to init. the "scale" state to a value.

for (i = 0; i < SIZE; i++) {
    MYSCALEDMAC16(sum, A[i], B[i]);
}
```

Problem: Wie Zustand initialisieren?

Zugriff auf dedizierte Zustandsregister



A. Koch

my_scaled_mac16_new.tie

```
state scale 16 add_read_write

operation MYSCALEDMAC16 {inout accum, in AR A, in AR B}
  {in scale} {

  assign accum = ((A[15:0] * B[15:0]) >> scale) + accum;

}
```

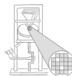
my_scaled_mac16_new.c

```
…
sum = 0;

WUR_scale(2);


for (i = 0; i < SIZE; i++) {
    MYSCALEDMAC16(sum, A[i], B[i]);
}
```

- **add_read_write** Option in **state**-Deklaration
- **RUR_***Name*: Zum Lesen des User Registers *Name*
- **WUR_***Name*: Zum Schreiben des User Registers *Name*
- Was bei breiten Registern ($> 32b$)?

Zugriff auf breite Zustandsregister

In 32b-Abschnitten



A. Koch

widestate.tie

```
state  widestate    40  add_read_write
…
```

widestate.c

```
…
unsigned long long sum;
…
WUR_widestate_0(0);  WUR_widestate_1(0);  //Initialize to 0
…
sum = ( (long long) RUR_widestate_1()<<32) +
        (long long) RUR_widestate_0() );
```

- **RUR_***Name***_0()**, **WUR_***Name***_0()**: *Name*`[31:0]`
- **RUR_***Name***_1()**, **WUR_***Name***_1()**: *Name*`[63:32]`

Details der `state`-Deklaration

Definition von Speicherelementen

A. Koch

state *name width* [**add_read_write**]

- Maximale Breite *width* ist 1024b
- Maximal 1024 Zustandsregister in Prozessor
  - Benutzerdefinierte und konfigurationsspezifische Register zusammen!

Entwicklungsmethodik

1. Ansatzpunkte für TIE-Instruktionen finden
   - Software-Sicht: Kritische Stellen finden und beschleunigen
   - Hardware-Sicht: Externen Hardware-Beschleuniger via TIE anschliessen
2. Verschiedene TIE-Entwurfstechniken
   - Instruktionssequenzen verschmelzen (*fusion*)
   - Parallele Datenverarbeitung (*SIMD*)
   - FLIX (VLIW-artiges Vorgehen, hier nur am Rande behandelt)
3. Prozessorarchitektur (Konfiguration und Erweiterungen) evaluieren
   - Flächenaufwand
   - Taktfrequenz

➥Teilweise auch automatisiert möglich (➜XPRES Compiler)

Software-Entwicklersicht

Finde zeitkritische Bereiche und beschleunige Ausführung

A. Koch

- Algorithm in C/C++
- Profile and Analyze
- Identify Hot Spots
- ID Application Specific Inst.
- Develop TIE Descriptions
- Compile TIE
- Re-compile Source with TIE Intrinsic Functions
- Run Cycle Accurate ISS
- Acceptable ?
  - N → Identify Hot Spots
  - Y → Synthesize and Build Processor

Hardware-Entwicklersicht

Entwerfe Hardware-Beschleuniger und binde sie als TIE ein

A. Koch

Analyze Application

Partition into Blocks

ID Computational Bottlenecks

Design HW Assist Data Path

Encapsulate Data Path in TIE

Compile TIE

Verify TIE w/ Software Kernel on ISS

Acceptable ?

N

Y

Build Processor

Verify with Test Bench

TIE-Entwurfstechniken

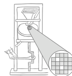
A. Koch

- Fusion
  - Verschmelze Folge von Instruktionen zu Einzelinstruktion
- Single Instruction Multiple Data (SIMD)
  - Eine Instruktion wird gleichzeitig auf mehreren Datenelementen ausgeführt
  - Idee: Vektorprozessor
- FLIX
  - Ähnlich VLIW
  - Hier aber genau auf Anwendung abstimmbar
  - Nicht nur Anzahl, sondern auch Art der Operationen in Instruktion

➥Auch Kombinationen sind möglich

Idee: Fusion

Verschmelzen von Instruktionsfolgen in einzelne TIE-Instruktion



A. Koch

Original C Code

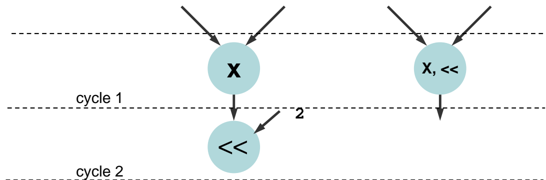
```
for(i=0;i<SIZE;i++){
 sum =(A[i]*B[i])<< 2;
}
```

Compiled Assembly

```
…
mul a13,a10,a8;
slli a12,a13,2;
…
```

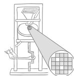
Compiled Assembly with a Fusion Instruction (merging mul and srl)

```
…
mulshift  a12,a10,a8;
…
```



Idee: SIMD

Eine Instruktion bearbeitet mehrere Datenelemente gleichzeitig

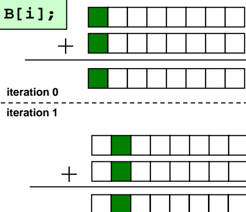


A. Koch

Original C Code

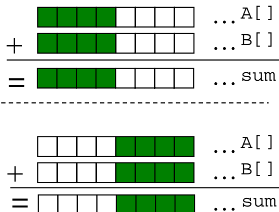
```
for(i=0;i<SIZE;i++)
  sum[i] = A[i] + B[i];
```

Typical Processor



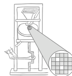
Xtensa Processor with a SIMD Instruction

(add operation on 4 data)



Idee: FLIX

Mehrere Operationen in einer Instruktion ausführen



A. Koch

Original C Code

```
for (i=0; i<n; i++)
c[i]= (a[i]+b[i])>>2;
```

Compiled Assembly

```
loop:
 …
 addi  a9,   a9,    4;
 addi  a11, a11,  4;
 l32i  a8,   a9,    0;
 l32i  a10, a11,  0;
 add   a12, a10,  a8;
 srai  a12, a12,  2 ;
 addi  a13, a13,  4;
 s32i  a12, a13,  0;
…
```

cycle 3

cycle 8

Compiled Assembly with a 64bit FLIX
(bundling 3 operations in 64bit FLIX inst.)

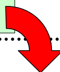
```
loop:
 { addi   a9,a9,4;      add  a12,a10,a8 ;  l32i   a8,a9,0 }
 { addi   a11,a11,4 ;  srai  a12,a12,2 ;   l32i   a10,a11,0  }
 { addi a13,a13,4;     nop;                 s32i   a12,a13,0 }
```

Beispiel: Fusion

A. Koch

orig.c

```
for (i=0; i<SIZE; i++)
    sum += A[i] * B[i];
```



mymac16.tie

```
operation MYMAC16 {inout accum, in AR A, in AR B} {} {
  assign accum = (A[15:0] * B[15:0]) + accum;
}
```

use_mymac16.c

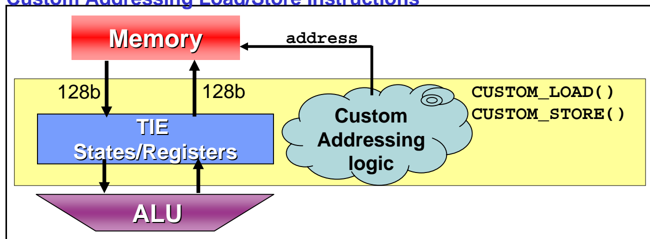
```
#include <xtensa/tie/mymac16.h>
sum=0;      //Initialize to 0

for (i = 0; i < SIZE; i++) {
    MYMAC16(sum, A[i], B[i]);
}
```

- Multiplikation/Aufsummieren in einer TIE-Instruktion
- Hier noch mehr verschmelzbar?

Verschmelzen von Speicherzugriff und Berechnung

A. Koch

- Speicherzugriffe häufig auf fortlaufende Adressen
  - LOAD - Laden der Daten, max. 32b breite Worte
  - ADD - Erhöhen der Adresse für nächstes Datum
- Verschmelze Laden und Adressberechnung
  - Nun auch breitere Worte ladbar
  - Variationsmöglichkeiten der Adressberechnung
    - Pre, post, Schrittweite $\neq 1$, Ringpuffer, ...



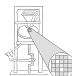
Eigene Speicherschnittstellen in TIE

A. Koch

- Maximale Breite ist 128b
  - Limitiert durch Breite des Prozessorbusses (PIF)
- Vorgehen beim Entwurf der TIE
  - Beschreibe Speicherschnittstelle in **`operation`**-Kopf
  - Formuliere Adressberechung in **`operation`**-Rumpf
  - Spezielle Schlüsselwörter für eigentlichen Speicherzugriff

TIE-Beispiel: 128b STORE-Instruktion

Schreibt Zustandsregister mit Autoinkrement der Adresse



A. Koch

mystore128.tie

```
state STATE128  128  add_read_write
operation STORE128 {inout AR *address}
      {in STATE128, out VAddr, out MemDataOut128} {
      assign VAddr = address;
      assign MemDataOut128 = STATE128;
      assign address = address + 16;
   }
```

Schnittstelle

- Zieladresse in **AR**-Register ***address**, autoinkrement
- Zustandsregister: Schreibdatum **STATE128**
- Speicherschnittstelle
  - Adresse **VAddr** (vordefiniert, Schlüsselwort!)
  - Daten **MemDataOut128** (vordefiniert, Schlüsselwort!)

TIE-Beispiel: 128b LOAD-Instruktion

Liest Zustandsregister mit Autoinkrement der Adresse

A. Koch

myload128.tie

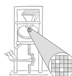
```
state STATE128  128  add_read_write
state POINTER   32   add_read_write
operation LOAD128 {} {out STATE128, inout POINTER,
                      out VAddr, in MemDataIn128 }{
      assign VAddr = POINTER;
      assign STATE128 = MemDataIn128;
      assign POINTER = POINTER + 16;
   }
```

- Hier auch Zieladresse in internem Zustandsregister
- Instruktion braucht keine **AR**-Register mehr
  - Erster Teil der Schnittstelle ist leer

Grundsätzliches zur Speicherschnittstelle

- Zugegriffene Adresse mit **`assign`** an **`VAddr`** zuweisen
  - Virtuelle (bei optionaler MMU) 32b Adresse
- Datenbus ist 8b, 16b, 32b, 64b, 128b breit
  - Konfigurierbar (*Processor Interface, PIF*)
  - Lesezugriffe über **`MemDataIn8`** ...**`MemDataIn128`**
  - Schreibzugriffe über **`MemDataOut8`** ...**`MemData128`**

Konstante Werte als unmittelbare Operanden
Immediate Operands



- Werden direkt in Instruktion untergebracht
- Ersparen explizites Laden
  - Aus Speicher
  - In Zwischenregister
- ➜Nützlich!
- In der Regel kleiner Wertebereich
  - Beispiel SPARC: 13b Konstanten
  - 8192 verschiedene Werte
- Direkt kodiert in Instruktion
- Unflexibel
  - Schrittweiten > 1? (0,4,8,12,16,...)
  - Nichtaufeinanderfolgende Werte? (1,3,5,7,11,13,...)

Unmittelbare Operanden mit Schrittweite > 1

A. Koch

mystore64i.S
```
store64i a3, 16
```

mystore64i.c
```
STORE64I(address, 16);
```

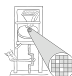
mystore64i.tie
```
immediate_range immr8  -256  248  8
state STATE64   64  add_read_write

operation STORE64I {in AR *address, in immr8 offset}
                   {in STATE64, out VAddr, out MemDataOut64} {
     assign VAddr = address + offset;
     assign MemDataOut64 = STATE64;
  }
```

immediate_range name untergrenze obergrenze schrittweite

- Hier 64 verschiedene Werte ➜Darstellung in 6b
- Bei direkter Kodierung (Zweierkomplement): 9b

Darstellung von nicht-fortlaufenden Werten

Als unmittelbare Operanden



A. Koch

mystore32C.S

```
store64C a3, a4, 128
```

mystore32C.c

```
STORE64C(data, address, 128);
```

mystore32C.tie

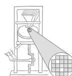
```
table crazy 17 4 {32, 64, 128, 256}

operation STORE64C {in AR data, in AR *address, in crazy p}
                   {out VAddr, out MemDataOut32} {
    assign VAddr = address + p;
    assign MemDataOut32 = data;
  }
```

`table` name elementbitbreite elementanzahl

- Hier: 2b (vier verschiedene Werte) statt 8b direkt
- Maximal 64 Einträge in **`table`**
- Vorzeichenbehaftete Werte immer in 32b darstellen!

Wertetabellen in TIE-Code

Nicht mehr in Instruktionscodierung sichtbar



A. Koch

tab.tie

```
table primes 16 4 {101, 103, 107, 109}
operation crypt {out AR cipher, in AR data, in AR i}
  {} {
   assign cipher = primes[i] ^ data;
}
```

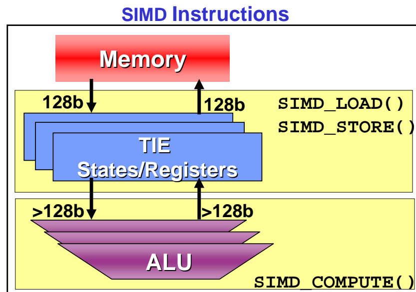
```
for (i=0; i<4; i++) {
   data[i] = crypt(data[i], i); //data[i] ^ primes[i]
}
```

- Nun explizite Indizierung
- Ungültige Indizes liefern den Wert 0

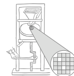
Spezielle LOAD/STORE-Instruktionen für SIMD-Anwendungen

A. Koch

- Breite LOAD/STORE-Instruktionen nützlich für SIMD-Vorgehensweise
- Nun mehr als 32b in TIE bearbeitbar



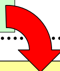
Beispiel: SIMD



A. Koch

orig.c

```
for (i=0; i<SIZE; i++)
    sum[i] = A[i] + B[i];
```



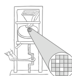
vec4_add16_state.tie

```
state Vec_S 64
state Vec_A 64
state Vec_B 64
operation ld_Vec_A{in AR* Addr} {out Vec_A, out VAddr, in MemDataIn64} {…}
operation ld_Vec_B{…} {…} {…}
operation st_Vec_S{in AR* Addr} {in Vec_S, out Vaddr, out MemDataOut64 } {…}
operation vec4_add16_state {} {out Vec_S, in Vec_A, in Vec_B} {
  assign Vec_S = {(A[15: 0] + B[15: 0]), (A[31:16] + B[31:16]),
                  (A[47:32] + B[47:32]), (A[63:48] + B[63:48])};}
```

use_vec4_add16_state.c

```
…… for (i = 0; i < SIZE/4; i++) {
    ld_Vec_A(&A[(i*4]); ld_Vec_B(&B[i*4]);
    vec4_add16_state();
    st_Vec_S(&sum[i*4]);;
}……
```

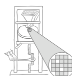
Diskussion der Realisierung



A. Koch

- Operanden und Ergebnis in TIE-internen Zustandsregistern
  - `state`
- Vorteile
  - Kompakte Hardware (=kleine Fläche)
  - Braucht keinen Platz im Instruktionswort
  - Instruktion kann mehr als drei Zustandsregister verwenden
  - Sehr leicht benutzbar
- Nachteile
  - Nicht allgemein verwendbar
  - Andere Instruktionen können Werte überschreiben
- Besseres Vorgehen
  - Dediziertes Registerfeld

Dedizierte Registerfelder



A. Koch

Vergleichbar dem allgemeinen **AR** Registerfeld

Eigenschaften

- Legt mehrere Speicherelemente gleichzeitig an
- Register von aussen direkt durch Software manipulierbar
  - Nicht mehr nur über **RUR_** und **WUR_**-Anweisungen
- Automatische Erzeugung von
  - LOAD/STORE/MOVE-Anweisungen
  - Passendem C-Datentyp
- …aber einige Einschränkungen (kommt noch)

Anlegen von Registerfeldern mit `regfile`

`regfile` name bitbreite anzahlelemente kurzname

- Name wird im TIE-Code verwendet
- Kurzname im Assembler

Beispiel: Legt 8 je 128b breite Register an

```
regfile myregs 128 8 v
```

Passende Assembler-Instruktion

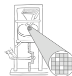
```
ADDV v1, v2, v3
```

Einschränkungen von dedizierten Registerfeldern

A. Koch

- Maximale Breite von Elementen ist 1024b
- Maximale Anzahl von Einträgen (Tiefe) je Registerfeld ist 1024
- Tiefe muß Zweierpotenz sein
- Maximale Anzahl von Registerfeldern auf ganzen Prozessor ist 24
  - Inklusive AR, Entscheidungsregister, VectraDSP-Vektorregister, ...
- Für automatisch erzeugte LOAD / STORE / MOVE-Instruktionen und Datentypen:
  - Registerbreite $\leq$ Datenbusbreite (PIF)

TIE-Beispiel: Registerfeld für SIMD Add16



A. Koch

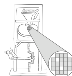
- **vec4_add16** rechnet auf 64b Registern mit 64b Ergebnis
- Konstruktion des entsprechenden Registerfeldes

vec4_add16.tie

```
regfile simd64 64 16 v     // 16 x 64bit wide registers

......
```

Anpassung von Add16 an Registerfeld

Erste Version, wird noch verfeinert



A. Koch

vec4_add16.tie

```
regfile simd64 64 16 v     // 16 x 64bit wide registers


operation vec4_add16 {out simd64 sum, in simd64 A, in simd64 B} {} {
    wire [15:0] result0 = (A[15: 0] + B[15: 0]);
    wire [15:0] result1 = (A[31:16] + B[31:16]);
    wire [15:0] result2 = (A[47:32] + B[47:32]);
    wire [15:0] result3 = (A[63:48] + B[63:48]);
    assign sum = {result3, result2, result1, result0};
}
```

- Nun kein Zustand, sondern Register in TIE-Schnittstelle
- Analog zu Verwendung von **AR**

Verwendung des Registerfeldes aus C

A. Koch

```
simd64 A[VECLEN];
simd64 B[VECLEN];
simd64 sum[VECLEN];

for (i=0; i<VECLEN; i++){
    sum[i] = vec4_add16(A[i],B[i]);
}
```

- Eigener C-Datentyp für Registerfeld, hier **`simd64`**
- Variablen dieses Types werden in Registern gehalten
- Automatische Registerallokation durch C-Compiler

Automatisch erzeugte Assembler-Instruktionen für Registerfelder

A. Koch

- Load:
  `ld_`typ zielregister, basisadressregister, offsetwert
- Store:
  `st_`typ zielregister, basisadressregister, offsetwert
- Move:
  `mv_`typ zielregister, ursprungsregister

C Code

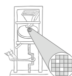
```
simd64 A[VECLEN];
simd64 B[VECLEN];
simd64 sum[VECLEN];

for (i=0; i<VECLEN; i++)
 sum[i] = vec4_add16(A[i],B[i]);
```

Xtensa Assembly Code

```
  loopnez a8, .L14
.L13:
    ld_simd64  v1,a10,0
    ld_simd64  v0, a9,0
    ….

    vec4_add16  v2,v0,v1
    …..
```

Beachte: Im Assembler Kurznamen des Registerfeldes!

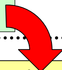
TIE-Beispiel: SIMD-Add16 mit Registerfeld und C-Code



A. Koch

orig.c

```
for (i=0; i<SIZE; i++)
    sum[i] = A[i] + B[i];
```



vec4_add16.tie

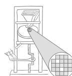
```
regfile simd64 64 16 v            // 16 x 64bit wide registers

operation  vec4_add16 {out sim64 sum, in simd64 A, in simd64 B} {}  {
   wire [15:0] result0 = (A[15: 0] + B[15: 0]);
   wire [15:0] result1 = (A[31:16] + B[31:16]);
   wire [15:0] result2 = (A[47:32] + B[47:32]);
   wire [15:0] result3 = (A[63:48] + B[63:48]);
   assign sum = {result3, result2, result1, result0};
}
```

use_vec4_add16.c

```
#include <xtensa/tie/vec4_add16.h>
simd64 A[VECLEN];
simd64 B[VECLEN];
simd64 sum[VECLEN];

for (i=0; i<VECLEN; i++)
 sum[i] = vec4_add16(A[i],B[i]);
```

Benutzt automatisch erzeugte LOAD/STORE-Anweisungen.

TIE-Module



- Können in Rumpf einer `operation`-Definition verwendet werden
  - Wie Verilog-Module
- Realisieren schlecht formulierbare Operationen
  - Z.B. vorzeichenbehafteter Vergleich
- Erzeugen effizientere Hardware als reine Verilog-Ausdrücke

Verfügbare TIE-Module

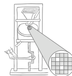
A. Koch

`TIEmux`	N-way multiplexor
`TIEpsel`	N-way priority selector
`TIEsel`	N-way one hot selector
`TIEadd`	Add with carry-in
`TIEaddn`	N-number addition
`TIEcmp`	Signed and unsigned comparison
`TIEcsa`	Carry-save adder
`TIEmac`	Multiply-accumulate
`TIEmul`	Signed and unsigned multiplication
`TIEmulpp`	Partial-product multiply

Weitere Informationen im *Tensilica Instruction Extension Language Reference Manual*

Quelle: /opt/cad/tensilica/tensilica-2006.5/XtDevTools/downloads/RA-2006.5/docs/tie_rm.pdf

TIE-Beispiel: SIMD Add16 mit TIE-Modul



A. Koch

orig.c

```
for (i=0; i<SIZE; i++)
    sum[i] = A[i] + B[i];
```

vec4_add16.tie

```
regfile simd64 64 16 v              // 16 x 64bit wide registers

operation  vec4_add16 {out sim64 sum, in simd64 A, in simd64 B} {}  {
   wire [31:0] result0 = TIEadd(A[15: 0], B[15: 0], 1'b0);
   wire [31:0] result1 = TIEadd(A[31:16], B[31:16], 1'b0);
   wire [31:0] result2 = TIEadd(A[47:32], B[47:32], 1'b0);
   wire [31:0] result3 = TIEadd(A[63:48], B[63:48], 1'b0);
   assign sum = {result3, result2, result1, result0};
}
```

use_vec4_add16.c

```
#include <xtensa/tie/vec4_add16.h>
simd64 A[VECLEN];
simd64 B[VECLEN];
simd64 sum[VECLEN];

for (i=0; i<VECLEN; i++)
 sum[i] = vec4_add16(A[i],B[i]);
```

TIE-Beispiel: Führende Null-Bits mit TIE-Modul

Verbesserung des ursprünglichen Ansatzes

Alter Ansatz

A. Koch

lz2.tie

```
state counter 32 add_read_write
operation lz2 {out AR leading_zeros, in AR input } {inout counter} {
        assign leading_zeros =    input[31] == 1'b1 ? 32'd0
                                          : input[30] == 1'b1 ? 32'd1
                                          ...
                                          : input[0] == 1'b1 ? 32'd31
                                          : 32'd32;
        assign counter = counter + 1;}
```

lz3.tie

```
state counter 32 add_read_write
operation lz3 {out AR leading_zeros, in AR input } {inout counter} {
        assign leading_zeros = TIEpsel(    input[31], 32'd0,
                                           input[30], 32'd1,
                                           ...
                                           input[0],  32'd31);
        assign counter = TIEadd(counter, 32’d1, 1’b0);}
```

Neuer Ansatz

Flächenbedarf und Zeitverhalten

A. Koch

- Entwurfsentscheidungen haben physikalische Konsequenzen
  - Wird gerne übersehen (auch von E-Technikern!)
  - Chip-Fläche
  - Taktfrequenz
  - Leistungsaufnahme
  - ...
- Chip-Fläche hat auch finanzielle Auswirkungen
  - Sonderangebote für EU Forschung und Lehre
    - UMC L130E: EUR 32.000/5mm x 5mm
    - UMC L90N: EUR 50.000/4mm x 4mm
    - Für ca. 20 Chips, Gehäuse kosten extra

Abschätzung

A. Koch

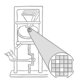
Flächenbedarf

- TIE-Compiler liefert schnell grobe Schätzung
- Genauere Schätzung erfordert Hardware-Synthese des Prozessors
  - Diese Software-Lizenz haben wir aber nicht

Zeitverhalten

- Geschätzt für Basis-Prozessorkonfiguration
- Zeitverhalten von Erweiterungen durch TIEs kann nicht abgeschätzt werden
- Erfordert Synthese der TIE-Blöcke
  - Problem: siehe oben

Beispiel für Abschätzung



A. Koch

byteswap.tie

Overview of TIE Source

Xtensa configuration (without TIE) is approx. 76,000 gates estimated at 320 MHz (130lv, Worst)
TIE area approx. 1,451 gates, of which 470 is decode, muxing etc. and 981 is instructions, states, regfiles etc.

Operations : 212 (14%)

Name	Area	Slots
BYTESWAP	198	
RUR.COUNT	7	
WUR.COUNT	7	

States : 769 (52%)

Name	Bits	Area
COUNT	32	769

Register Files : 0 (0%)

Name	W / D	Area

Ports : 0 (0%)

Name	Bits	Area

Functions : 0 (0%)

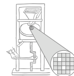
Name	Area	Slots

Check | Compile | Area | Select active Xtensa Configuration: example_core1

TIE Overview | TIE source | Construct Details

Verifikation

Überprüfung der Funktionsfähigkeit, hier nicht formal



A. Koch

Algorithm Development

C/C++ Algorithm

```
int main()
{
 int i;
 short c[100];
 for (i=0;i<N;i++)
 {
```

C/C++ testbench

```
int Test_data[100]
= {0x1, 0xa, …};
int
expected_data[100]
= {0xa, 0x1, …};
```

TIE Development

C/C++ Algorithm uses TIE

```
int main()
{
 int i;
 short c[100];
 for (i=0;i<N;i++)
 { myTIE();
}
```

User-Defined TIE

```
state foo 32
state bar 32
operation mytie{}{}{
}
```

C/C++ Testbench

```
int Test_data[100]
= {0x1, 0xa, …};
int
expected_data[100]
= {0xa, 0x1, …};
```

Hardware Implementation

C/C++ Algorithm uses TIE

```
int main()
{
 int i;
 short c[100];
 for (i=0;i<N;i++)
 { myTIE();
}
```

Xtensa + TIE RTL

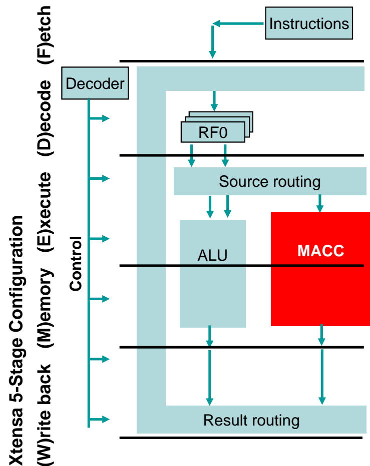
C/C++ Testbench

```
int Test_data[100]
= {0x1, 0xa, …};
int
expected_data[100]
= {0xa, 0x1, …};
```

Strategy 1:
Run full SW on X2000 FPGA board

Strategy 2:
Run simplified SW on RTL simulator inside our hardware testbench architecture

Instruktionen mit mehreren Zyklen



- TIEs dürfen auch länger als ein Takt sein
- Beispiel: 2 Takte
  - Register lesen: Anfang E-Stufe
  - Register schreiben: Ende E+1-Stufe
- Erlaubt komplexere TIEs bei gleicher Taktfrequenz
  - Mehr Pipeline-Stufen
- Kann aber feste Instruktionen verlangsamen

A. Koch

Zusammenfassung

A. Koch

- Speicherelemente für Zustand
  - **`state`** und **`regfile`**
- Entwurfstechniken
  - Fusion, SIMD, FLIX
- Speicherschnittstelle
- Unmittelbare Operanden
- TIE-Module
- Flächenbedarf und Zeitverhalten