



A. Koch

# Eingebettete Prozessorarchitekturen

## 8. Programmierung mit Stretch-C

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt

Wintersemester 2006/2007



- Auszüge aus Trainingsmaterial der Fa. Stretch
  - Insbesondere alle Zeichnungen und technischen Daten
- Material ist **vertraulich**
  - Nur für die Lehre zur Verfügung gestellt
  - Darf **nicht** weiterverbreitet werden



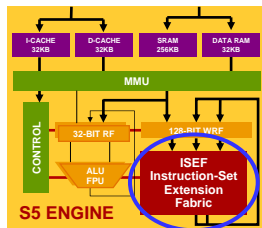
- Rekonfigurierbare Prozessoren
- Stretch S5000-Architektur
- Erstes Beispiel in Stretch-C
- Schwerpunkt: Kommunikation mit ISEF
  - Wide Registers ausgerichtet lesen/schreiben
  - Streaming von unausgerichteten Daten

# Instruction-Set Extension Fabric

Kann als konfigurierbare ALU angesehen werden



A. Koch



- Maximal 48 Bytes Operanden (drei 128b WR-Register)
- Maximal 32 Bytes Ergebnis (zwei 128b WR-Register)
- Pipelined, multi-zyklen Operationen realisierbar
- S5 Engine hat **zwei** ISEF RAs
  - Teilen sich dieselben Register
  - Nur ein ISEF gleichzeitig aktiv
  - Vorteile: Mehr Platz und schnellere Rekonfiguration

# Erweiterungsinstruktionen

## Extension Instructions (EI)



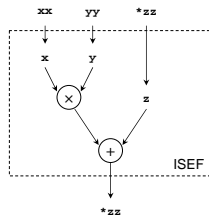
- Werden in Stretch-C als C-Funktionen definiert
  - C-artige Syntax, aber mit limitiertem Sprachumfang
- Funktionsparameter beschreiben Schnittstelle
- Funktionsrumpf beschreibt Hardware der neuen Instruktion

A. Koch

```
SE_FUNC
void mul_add(WR xx, WR yy, WR *zz)
{
    short x = xx;
    short y = yy;
    int z = *zz;

    z = z + x * y;

    *zz = z;
}
```



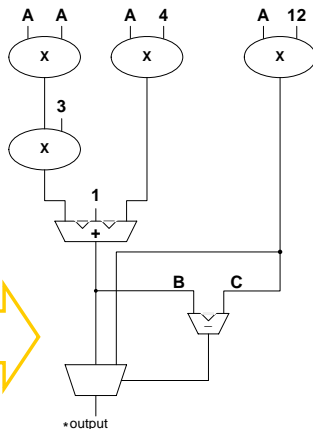
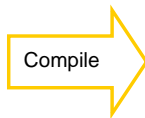
# Beispiel 1 für erzeugte Hardware

Compiler nutzt ILP aus



A. Koch

```
void poly(WR input, WR *output)
{
    short A = input;
    int B, C;
    B = 3*A*A + 4*A + 1;
    C = 12*A;
    *output = (B>C) ? B : C;
}
```

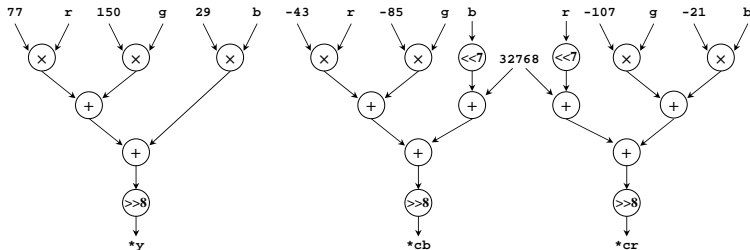


# Beispiel 2 für erzeugte Hardware



A. Koch

```
void RGB2YCC( ... )
{
    *y = ( 77*r + 150*g + 29*b          ) >> 8;
    *cb = ( -43*r - 85*g + 128*b + 32768 ) >> 8;
    *cr = ( 128*r - 107*g - 21*b + 32768 ) >> 8;
}
```



# Anatomie einer Stretch-C Funktion



A. Koch

## Extension Instruction:

```
#include <stretch.h>
```

```
SE_FUNC void RGB2YCC(WR A, WR *B) {  
    se_uint<8> r, g, b, y, cb, cr;  
    r = A(7,0); g = A(15,8); b = A(23,16);  
  
    y = ( 77*r + 150*g + 29*b          ) >> 8;  
    cb = (-43*r - 85*g + 128*b + 32768) >> 8;  
    cr = (128*r - 107*g - 21*b + 32768) >> 8;  
  
    *B = (y,cb,cr);  
}
```

## Program Loop:

```
for (...) {  
    WRGETOI(&A, 3);  
    RGB2YCC(A, &B);  
    WRPUTI(B, 3);  
}
```

## Extension Instruction:

```
#include <stretch.h>
```

MUST INCLUDE <stretch.h> HEADER

```
SE_FUNC void RGB2YCC(WR A, WR *B) {  
    se_uint<8> r, g, b, y, cb, cr;  
    r = A(7,0); g = A(15,8); b = A(23,16);
```

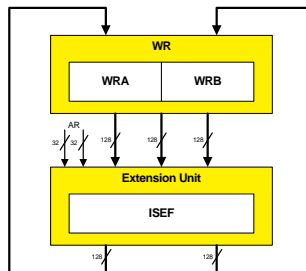


# Parametertypen



A. Koch

- Drei Eingaben aus
  - **WRA** (=WR), **WRB**,
  - AR Typen **char**, **short**, **int**, **long**
- Zwei Ausgaben nach
  - **WRA\*** (=WR\*), **WRB\***
  - In unterschiedliche Bänke
- Einschränkung
  - Keine **direkte** AR/WRF-Kommunikation
  - Gehe über Speicher oder ISEF-Ergebnis





- Compiler bestimmt Ein-/Ausgaben automatisch
  - Aus Parameterbenutzung

## Examples:

```
SE_FUNC void
FOO(int c1, WR v1, WRB *vOut)
{ . . . }
```

```
SE_FUNC void
BAR(WR v1, WRA *vOut1, WRB *vOut2)
{ . . . }
```

```
SE_FUNC void
BAZ(WR v1, WRA *vInOut1, WRB *vOut2)
{ . . . }
```

# Stretch-C Datentypen

Im Funktionsrumpf



A. Koch

- Standarddatentypen
  - **char, short, int, long, long long**
- Keine Gleitkommatypen im ISEF
- Wide-Register-Typen
  - **WR, WRA, WRB**
- Integer-Typen mit beliebiger Breite
  - **se\_sint<n>**:  $n$ -bit, vorzeichenbehaftet
  - **se\_uint<n>**:  $n$ -bit, vorzeichenlos

# Zustandsregister als statische Variablen

Analog zu **state** in Tensilica TIE



- ISEF-interne Register
- **Nicht** aus Xtensa V-Pipeline zugänglich
- Halten Daten über **mehrere** ISEF-Aufrufe
  - Können auch über Rekonfigurationen hinweg erhalten werden
- Initialisiert und modifiziert in Els
- Deklaration als **static** im Stretch-C File (**.xc**)

A. Koch

```
static se_sint<40> accumulator;

SE_FUNC void initialize()
{
    accumulator = 0;
}

SE_FUNC void runningSum(short x, WR *total)
{
    accumulator += x;
    *total = ((se_uint<96>)0, (se_sint<32>)accumulator);
}
```

# Konstante Tabellen



A. Koch

- Werden mit `static const` deklariert
- Verbrauchen *keine* ISEF-Register

```
/* gamma[i] = round( ((i/256.) ^ (1/2.5)) * 256 )
*/
static const se_uint<8> gamma[256] = {
    0, 28, 37, 43, 49, 53, 57, 61,
    64, 67, 70, 73, 75, 78, 80, 82,
    . . .
};

SE_FUNC void doGamma(char in, WR *out)
{
    *out = gamma[in];
}
```

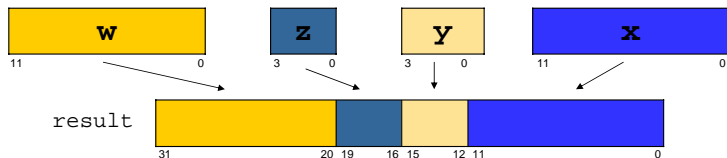
# Operatoren in Stretch-C



A. Koch

- Alle C-Operatoren **außer** Division und Modulus
  - Ausnahmen: Beide Operanden konstant
  - Ausnahme: Division durch  $2^n$
- Zwei neue Operatoren für Bit-Manipulation
- Konkatination durch  $(a, b, c, d)$ 
  - Entspricht  $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$  in Verilog
- Extraktion durch  $x(i, j)$ 
  - Entspricht  $\mathbf{x}[\mathbf{i}, \mathbf{j}]$  in Verilog

# Konkatenation mit $(w, z, y, x)$



A. Koch

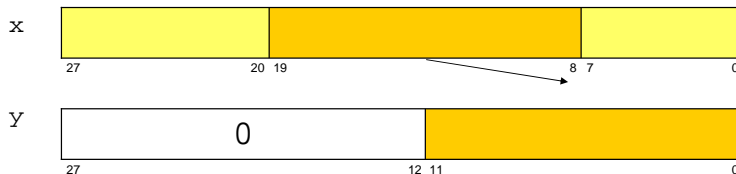
```
se_sint<4> z,y; se_sint<12> w,x;  
se_uint<32> result = (w,z,y,x);
```

- Ergebnis ist vom Typ `se_uint<n>`
  - $n$  ist summierte Breite aller konkatenierten Variablen
- Mindestens eine Variable **muss** einen Stretch-C Typ haben
- ... sonst wird Komma-Operator wie in **Standard-C** interpretiert
  - Gibt den am weitesten rechts stehenden Wert zurück
  - Im Beispiel oben also  $x$

# Extraktion mit $x(i, j)$



A. Koch



```
se_sint<28> x, y;
```

```
y = x(19, 8);
```

- Ergebnis ist vom Typ `se_uint<n>`
  - Wobei  $n$  die Breite des Typs von  $x$  ist
- Ergebnis wird in niederwertigen Bits abgelegt
- Und bis zur vollen Breite mit Nullbits aufgefüllt
- Falls  $i < j$  werden Bits im Ergebnis **verdreht**



# Typkonvertierung

Entsprechend den ANSI C-Regeln



A. Koch

- Von breiterem Typ in schmaleren: Abschneiden der höherwertigen Bits
  - `se_uint<12>` ← `unsigned short`: Verliert Bits 15:12
- Von schmalerem zu breiterem Typ: Auffüllen je nach Vorzeichenart
  - `se_sint<40>` ← `se_sint<32>`: Kopiert Bits 31:0, vervielfacht Bit 31 nach 39:32
- Von vorzeichenbehaftetem zu vorzeichenlosem Typ
  - Passe Breite gemäß der beiden vorigen Regeln an
  - Ändere nun Interpretation des MSB als Vorzeichen

## Beispiel

```
se_sint<3> a;           /* signed 3-bit integer */
se_uint<5> b;           /* unsigned 5-bit integer */
a = -2;                 /* binary = 110, decimal = -2 */
b = a;                  /* binary = 11110, decimal = 30 */
```

# Gründe für Typkonvertierungen



A. Koch

- Explizite Konvertierung: `(se_uint<5>) x`
- Zuweisung: `y = x`, konvertiert nach Typ von `y`
- Operatoren: `x + y`
  - Wohin konvertieren? Nach `x` oder nach `y`?
  - (ANSI C-Typ) `Op` (ANSI C-Typ)
    - ANSI C Konvertierungsregeln
  - (Stretch C-Typ) `Op` (Stretch C-Typ)
    - Konvertiere schmalere Typ in Typ des breiteren Operanden
    - Erweitere Breite ggf. um Overflow zu vermeiden (hängt von `Op` ab)
  - (Stretch C-Typ) `Op` (ANSI C-Typ), oder umgekehrt
    - Konvertiere ANSI C-Typ in äquivalenten Stretch C-Typ
    - Z.B. `int` → `se_sint<32>`
    - Wende dann vorige Regel an

# Typkonvertierung von Stretch C nach ANSI C



A. Koch

- Erforderlich für
  - Array Indices
  - Bit-Positionen von Extraktionsoperator
- Durch die Funktion `integer(n)`

```
se_uint<4> k, n, m;
```

```
. . .
```

```
x = array[integer(k)];
```

```
u = x(integer(n), 0);
```

# Schleifen in Stretch-C



- Nur mit **festen Grenzen** erlaubt
- Werden dann vollständig entrollt
  - Parallele **kombinatorische**, keine sequentielle Ausführung

A. Koch

```
for (i=sum=0; i<MAX; i++) {  
  v[i] = x(16*i+15, 16*i) ^ 3;  
  sum += v[i];  
}
```

If  $MAX=4$  at compile time, this unrolls to:

```
sum = 0;  
v[0] = x( 15, 0 ) ^ 3;  
sum += v [0];  
v[1] = x( 31, 16 ) ^ 3;  
sum += v[1];  
v[2] = x( 47, 32 ) ^ 3;  
sum += v[2];  
v[3] = x( 63, 48 ) ^ 3;  
sum += v[3];
```

Wird optimiert mit  $sum = ((v[0]+v[1]) + (v[2]+v[3]))$

# Kompilierung von EI-Beschreibungen



A. Koch

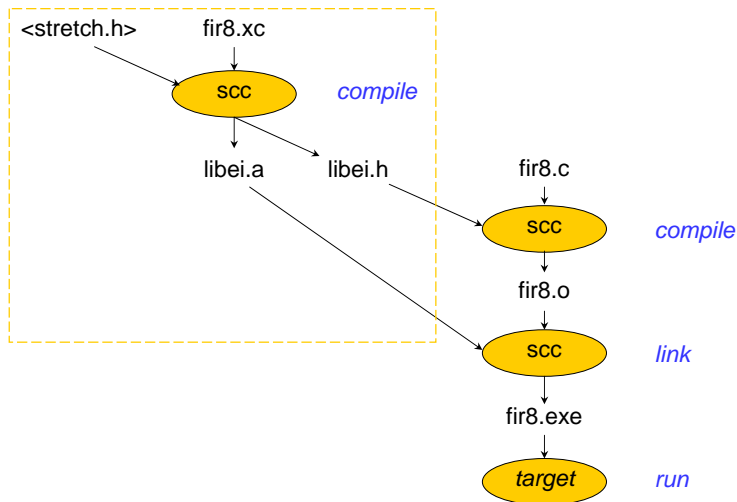
- Els definiert in `.xc`-Dateien
- `.xc`-Datei ↔ Konfiguration ↔ Bis zu 16 Els
- `.xc`-Datei muss `<stretch.h>` einbinden
- Compilen/linken einer `.xc`-Datei erzeugt
  - Header-Datei `libei.h`
    - Deklariert C-Pseudofunktionen für Els
    - C-Code, der Els benutzt muss diese Datei einbinden
  - Bibliotheksdatei `libei.a`
    - Enthält **Bitstream** zur Konfiguration des ISEF
    - Wird mit C/C++-Anwendung gelinked

# Generierungsschritte

Hinter den Kulissen der IDE



A. Koch



# Beispiel der Definition und Benutzung von Els



- **Stretch C** (*define Els*)

```
#include <stretch.h>
SE_FUNC void RGB2YCC(WR A, WR *B)
{ . . . }
```

- **Application C** (*use Els*)

```
#include "libei.h"
WRGET0I(&A, 3);
RGB2YCC(A, &B);
WRPUTI(B, 3);
```

- **Assembly** (*from compiler*)

```
wraget0i      wra8,3
se_rgb2ycc    wra8,wra5
wraputi       wra5,3
```

A. Koch

# Definition mehrerer EIs

In einer SE\_FUNC, erlaubt **Wiederverwendung** von Ressourcen



## Stretch C

```
SE_FUNC void fooFunc(  
    SE_INST FOO_INIT,  
    SE_INST F00, ...)  
{  
    static se_sint<20> statevar;  
  
    ...    /* code common to both EIs */  
  
    statevar = FOO_INIT ? 0 : (a + b);  
  
    ...    /* more code common to both EIs */  
}
```

## Application C

```
FOO_INIT(...);  
...  
FOO(...);
```

A. Koch

Umschalten durch boolesche **SE\_INST** Parameter

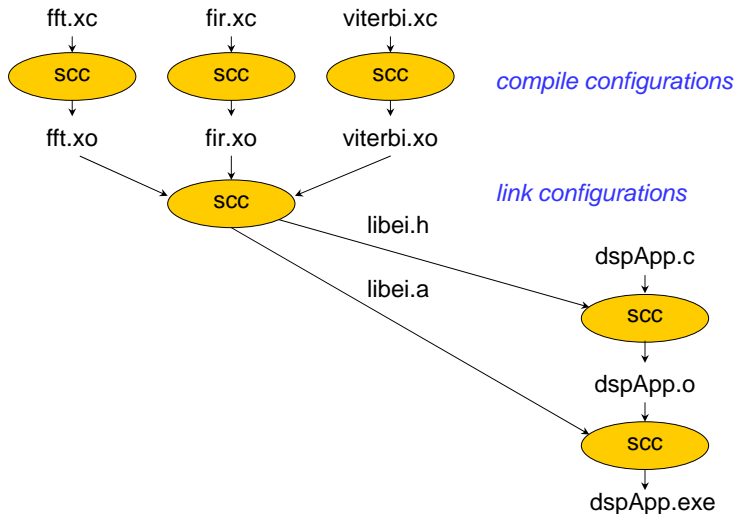
↳ Genau einer **wahr**



# Definition mehrerer Konfigurationen



A. Koch





- On-Demand
  - ISEF wird **automatisch** mit gerade benötigter Instruktion geladen
- Manuell
  - StretchBIOS-Aufrufe zum **expliziten** Laden der ISEFs
  - Vorgezogenes Laden von Instruktionen möglich
  - Double-Buffering beim Austausch von Instruktionen

➡ Später mehr Details



- Feste Obergrenzen auf Chip
  - Unterschied zu Tensilica, dort flexibler anpassbar
- Feingranulare 1b-Recheneinheiten
  - 4096 *arithmetic units* (AU)
  - 8192 *multiplier units* (MU)
- Zustandsregister
  - 1024 *extension registers* (ER)
- Verdrahtungsressourcen
- Pipelineressourcen

Je ISEF, aber **zwei** ISEFs auf S5000-Chips vorhanden  
➔ Bis zu 32 EI Instruktionen gleichzeitig vorhalten

# Flächenbedarf verschiedener Operationen

Abhängig von Bitbreiten der Operanden



A. Koch

C Operators	AUs	MUs
$A(+, -)B$	$\max( A ,  B )$	0
$A(\&, ^,  )B$	$\min( A ,  B )$	0
$A*B$	0	$ A  *  B $
$A(\ll, \gg)B$ (variable B)	0	$ A  * 2^{ B }$
$A(\ll, \gg)B$ (constant B)	0	0

- Compiler kann zwischen AUs und MUs verschieben
  - $x*128 \leftrightarrow x \ll 7$
- Zusätzliche Ressourcen für
  - Pipelining
  - Interlocks
  - Sichern/Wiederherstellen von Zustand in ERs



- Generiert von `scc` Compiler nach Bearbeiten einer `.xc`-Datei
- Angaben über
  - Benötigte AUs, MUs und ERs
  - Insgesamt, je Funktion, je Programmzeile
  - Rechenzeit in Zyklen (Latenz)

# Beispiel Ressourcenbericht



A. Koch

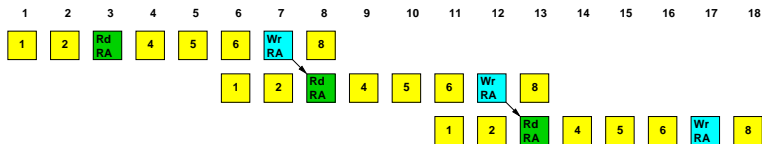
```
// Total Number of configurations: 1
//
// Configuration 'reedsolomon_enc', implementing instructions RS_ENC_INIT,RS_ENC:
//
// INSTRUCTION=ReedSolomonEncode:RS_ENC_INIT Opcode: 0: Assembly 'se_rs_enc_init in:in, out:out'
//   Input : variable 'in' read from unsigned on ISEF read port 0 at beginning of cycle 1
//   Output: variable 'out' written to WRA on ISEF write port 0 at end of cycle 3
//   ISEF Local State variable 'bb' (128 bits) written at end of cycle 2
//
// INSTRUCTION=ReedSolomonEncode:RS_ENC Opcode: 1: Assembly 'se_rs_enc in:in, out:out'
//   Input : variable 'in' read from unsigned on ISEF read port 0 at beginning of cycle 1
//   Output: variable 'out' written to WRA on ISEF write port 0 at end of cycle 3
//   ISEF Local State variable 'bb' (128 bits) read at beginning of cycle 1
//   ISEF Local State variable 'bb' (128 bits) written at end of cycle 2 (Write - Read = 1)
//
// Class Assignments for configuration 'reedsolomon_enc'
//
//
//          Read Class   Write Class
// INSTRUCTION      0 1 2 3   0 1 2 3
// -----
// ReedSolomonEncode:RS_ENC_INIT                x
// ReedSolomonEncode:RS_ENC      x          x
//
//
// Class 0: 1 stall(s)
// Class 1: 0 stall(s)
// Class 2: 0 stall(s)
// Class 3: 0 stall(s)
//
//
// Computational Resources
//   Arithmetic bits.....0
//   Logic bits.....551
//   Mux bits.....137
//   Register bits.....24
//   Pipeline bits.....13
// AU total.....725 out of 4096
//   Multiply bits.....0
// MU total.....0 out of 8192
// Extension registers.....128 out of 4096
```

# Interlock-Verzögerungen durch ERs

Beispiel: Aufeinanderfolgende Instruktionen Lesen/Schreiben ein ER **RA**



A. Koch



- Durchsatz =  $t(\text{WrRA}) - t(\text{RdRA}) + 1 = 7 - 3 + 1 = 5$
- Eine Instruktion nur **alle 5 Takte** starten

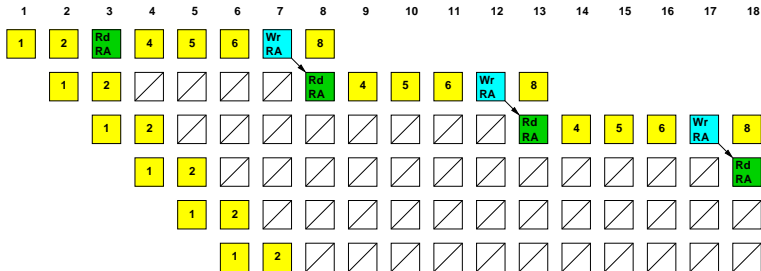
# Interlock-Verzögerungen durch ERs

## Alternative Sicht



A. Koch

Je 4 Stall-Zyklen je Instruktion in Pipeline





# Beispiel Annotierte Programmzeilen

Einfacheres Beispiel als im Ressourcenbericht eben



```
#include <stretch.h>
static se_sint<32> acc;
```

```
/* Performs 8 parallel MACs */
SE_FUNC void firFunc(SE_INST FIR_MUL, SE_INST FIR_MAC, WR X, WR H, WR *Y)
{
    se_sint<16> x;           // 16-bit input data samples
    se_sint<16> h;           // 16-bit filter coefficients
    se_sint<32> sum ;        // sum of all 8 32-bit products
    int i ;
    sum = 0;
    for(i = 0; i < 128; i += 16) { // loop 8 times
        h= H(i + 15, i);       // extract 16-bit coefficient/* CYCLE=1 */
        x= X(127-i, 112-i);    // extract 16-bit data sample/* CYCLE=1 */
        sum += x * h ;         // multiply and add to running sum/* AU=224 MU=2048 CYCLE=4 */
    }
    acc = FIR_MUL ? sum : se_sint<32> (sum + acc) ;/* PIPELINE=6 MUX=32 CYCLE=5 */
    *Y = acc >> 14 ;          // shift result down to Q15/* CYCLE=5 */
}
```

A. Koch

- 7 32b Additionen = 224 AUs
  - Erste Addition hat **sum=0** und entfällt damit!
- 8 16b x 16b Multiplikationen = 2048 MUs

# Beispiel Annotierte Programmzeilen

## Jetzt die zum Ressourcenbericht passende kompliziertere El



A. Koch

```
#include <stretch.h>

static se_uint<8> bb[17]; // Shift register (17th element is for convenience)
static const se_uint<8> gg[16] = { // Field polynomial
    0x76, 0x34, 0x67, 0x1f, 0x68, 0x7e, 0xbb, 0xe8, 0x11, 0x38, 0xb7, 0x31, 0x64, 0x51, 0x2c, 0x4f
};
se_uint<8> const POLY = 0x1d; // Generator polynomial

/* Reed Solomon Encoder for 802.16 (WiMax) */
/* Take one input byte and use it to update all 16 state registers */
/* All math is GF(2^8) -- ie: "+" = XOR and "*" is Galois Field with POLY = 0x1d */
SE_FUNC void ReedSolomonEncode( SE_INST RS_ENC_INIT, SE_INST RS_ENC, unsigned char in, WRA *out )
{
    int j, k;
    se_uint<8> fb, ggg[16][8];

    for (j = 0; j < 16; j++) { // Compute gg[j]^k mod POLY for all j and k
        ggg[j][0] = gg[j]; // Note this loop operates on CONSTANTS only, no AUs used!
        for (k = 1; k < 8; k++) {
            ggg[j][k] = (ggg[j][k-1] << 1) ^ ((ggg[j][k-1] & 0x80) ? POLY : 0);
        }
    }

    fb = RS_ENC_INIT ? in : (in ^ bb[0]); // (feedback) = (in) + (state 0)/* PIPELINE=10 LOGIC=437 MUX=4 CYCLE=3 */
    bb[16] = 0; // Special case: (state 15) = (feedback) * (genpoly 15)
    for (j = 0; j < 16; j++) { // (state j) = (state j+1) + (feedback) * (genpoly j)
        bb[j] = RS_ENC_INIT ? 0 : bb[j+1]; // LOGIC=5 CYCLE=2 */
        for (k = 0; k < 8; k++) { // Product = fb*gg^0 + fb1*gg^1 ...
            if (fb & (1 << k)) // LOGIC=82 MUX=125 CYCLE=3 */
                bb[j] ^= ggg[j][k]; // REG=24 PIPELINE=3 LOGIC=27 MUX=8 CYCLE=3 */
        }
    }

    *out = (bb[15], bb[14], bb[13], bb[12], bb[11], bb[10], bb[ 9], bb[ 8],
           bb[ 7], bb[ 6], bb[ 5], bb[ 4], bb[ 3], bb[ 2], bb[ 1], bb[ 0]);
}
```

# Abschlussbericht

Bis hier Schätzungen, nun genaue Angaben



A. Koch

```
/*-----*\
*               Final resource usage report               *
*-----*\
* Configuration reedsolomon_enc:
* Total AUs = 664 out of 4096
* Total MUs = 0 out of 8192
* ISEF can run at the required frequency.
*
\*****/
```

# Unterschiedliche Taktbereiche

*clock domains*



- Feste Xtensa V Pipeline läuft mit max. 300 MHz
- Rekonfigurierbare ISEFs laufen mit max. 100 MHz
- Starte eine ISEF EI **je zwei** Xtensa V Instruktionen
- Xtensa Speichertransfers **während** EI-Ausführungszeit

A. Koch

## Beispiel

Schleife entrollt

```
WRGETOI (&A, 15)  
rgb2ycc (A, &B)  
WRPUTI (B, 15)  
WRGETOI (&A, 15)  
rgb2ycc (A, &B)  
WRPUTI (B, 15)
```

Spannender Teil

```
rgb2ycc (A, &B)  
WRPUTI (B, 15)  
WRGETOI (&A, 15)
```

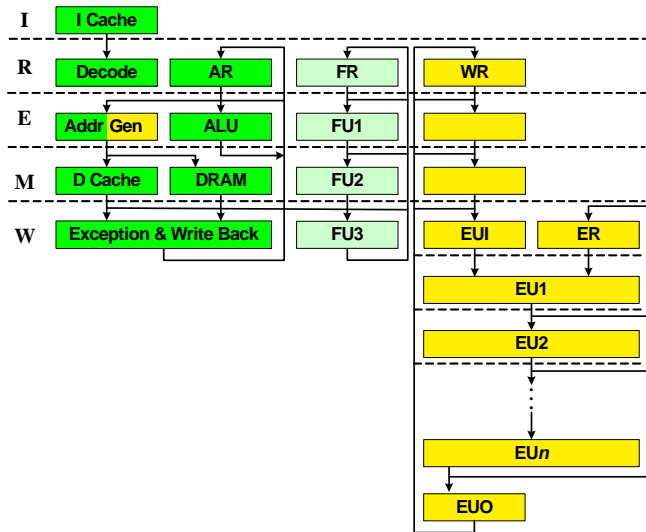
**Während** Ausführungszeit von `rgb2ycc` werden alte Daten geschrieben und neue geholt.

# Pipeline-Struktur

Bis zu 31 Takte lang



A. Koch



# Pipeline-Ablauf im ISEF



A. Koch

- Operanden werden **am Anfang** von EUI-Stufe gelesen (*input*)
- Ergebnisse werden **am Ende** von EUO-Stufe ausgegeben (*output*)
- EI mit  $n$ -Takten hat Stufen EUI, EU1, ..., EU $n$ , EUO
- $n$  ist hier in Xtensa-Takten, i.d.R. ISEF-Takte =  $n/3$
- Extension Registers dürfen ...
  - am Anfang jeder EU $i$ -Stufe gelesen werden
  - am Ende jeder EU $j$ -Stufe mit  $j \geq i$  geschrieben werden
- Alle Register (AR, WR, ER) sind
  - **Interlocked**: ISEF-Pipeline hält an, wenn RAW verletzt
  - **Bypassed**: Register sofort nach Definition verwendbar

# Beispiel Pipelining



A. Koch

```

for (i = 0; i < NP/5; i++) {
    WRGETOI(&A, 15); // load available after M
    rgb2ycc(A, &B); // rgb2ycc's latency is 10 cycles (2*3 + 4)
    WRPUTI(B, 15); // B must be available before E
}
    
```

Compute-store stall for B

Note Zero Overhead Loop  
Each iteration takes 12 cycles

n		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
0	WRGETOI	I	R	E	M	W																	
0	rgb2ycc		I	R	E	M	EU1	EU2	EU3	EU4	EU5	EU6	EU0										
0	WRPUTI			I	R	-	-	-	-	-	-	-	-	-	E	M	W						
1	WRGETOI				I	-	-	-	-	-	-	-	-	-	R	E	M	W					
1	rgb2ycc														I	R	E	M	EU1	EU2	EU3		

# Komplizierteres Szenario mit Loop Unrolling



A. Koch

## Unrolling im Compiler abgeschaltet:

Stalls highlighted

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
movi a8, 128	I	R	E	M	W																
vrpbitn0, a3		I	R	E	M	W															
vrget0init0, a2			I	R	E	M	W														
vrget0init1				I	R	E	M	W													
loopgtz a8, d00015c					I	R	E	M	W												
vrget0 vrna0, 15						I	R	E	M	W											
se_rfb2ycyc vrna0, n							I	R	E	M	W										
vrputi vrna0, 15								I	R	E	M	W									
									I	R	E	M	W								

3 instructions = 12 cycles (9 stalls)

Zero-overhead loop instruction

## Unrolling erlaubt:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
se_rfb2ycyc vrna3, vrna3	I	R	E	M	W																			
loopgtz a8, d00015bc		I	R	E	M	W																		
vrget0 vrna0, 15			I	R	E	M	W																	
se_rfb2ycyc vrna4, vrna5				I	R	E	M	W																
vrget0 vrna2, 15					I	R	E	M	W															
vrputi vrna7, 15						I	R	E	M	W														
se_rfb2ycyc vrna6, vrna7							I	R	E	M	W													
vrputi vrna1, 15								I	R	E	M	W												
vrget0 vrna4, 15									I	R	E	M	W											
se_rfb2ycyc vrna0, vrna1										I	R	E	M	W										
vrputi vrna3, 15											I	R	E	M	W									
vrget0 vrna6, 15												I	R	E	M	W								
se_rfb2ycyc vrna2, vrna3													I	R	E	M	W							
vrputi vrna5, 15														I	R	E	M	W						

12 instructions = 12 cycles (0 stalls)