

# Eingebettete Prozessorarchitekturen

## 13. Hardware-Entwurf aus Hochsprachen

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt

Wintersemester 2009/2010

# Organisatorisches

Zeiten für zweite mündliche Teilprüfung am 12.04.2010

10.00 – 10.20 David de la Chevallerie

10.20 – 10.40 Martin Tsarev

10.40 – 11.00 Nicolas Weber

11.00 – 11.20 Raad Bahmani

11.20 – 11.40 Stefan Zügel

11.40 – 12.00 David Kreitschmann

14.00 – 14.20 Steffen Jäger

14.20 – 14.40 Tobias Rückelt

14.40 – 15.00 David Meier

15.00 – 15.20 Daniel Demmler

15.20 – 15.40 Jonas Schölnichen

15.40 – 16.00 Patrick Neugebauer

16.00 – 16.20 Leqiao Peng

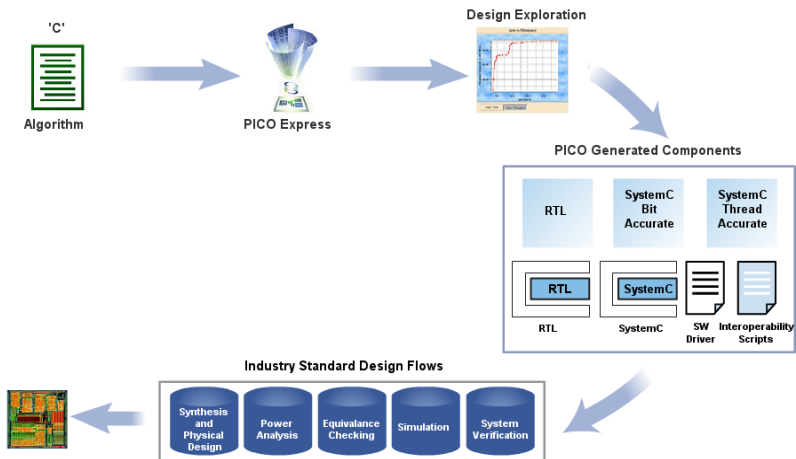
# Material

- ▶ Ab hier Auszüge aus Trainingsunterlagen und Dokumentation von Synfora Inc.
- ▶ Weitere Dokumentation unter `/opt/synfora/pico_express_fpga-09.03-3/docs`
- ▶ Auf Anfrage auch interaktives Computer-based Training Material verfügbar
- ▶ **Alles nur für die Arbeit innerhalb der Vorlesung bestimmt!**

# Anwendungsgebiete

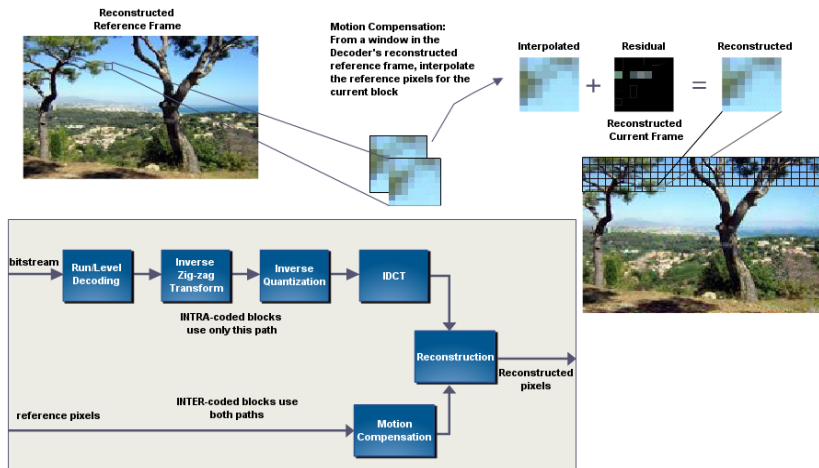
# Entwurfsfluss

Vom Hochsprachenprogramm zum Chip

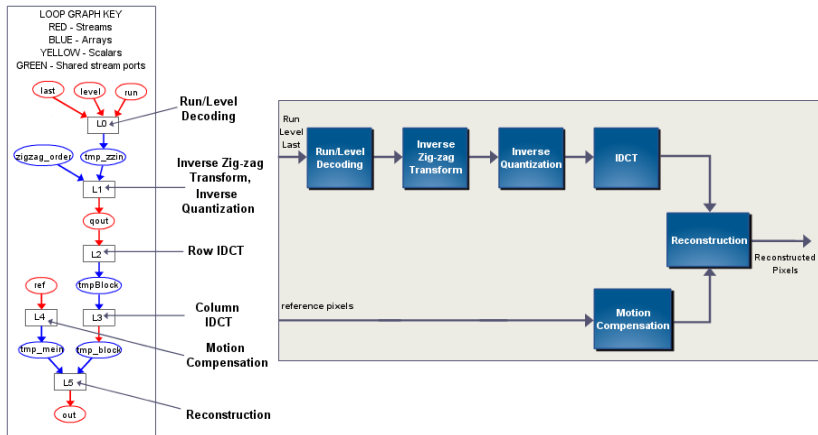


# Typische Anwendung: Videoverarbeitung

Hier: Dekompression



# Logische Struktur der Dekompressions-Pipeline



# Entwurfsfluss



# Ein- und Ausgaben

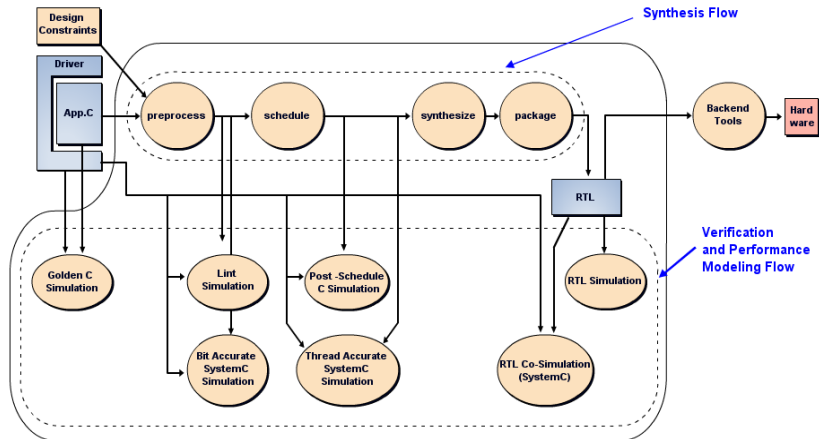
## Eingaben

- ▶ C Programm
- ▶ C Testrahmen und -eingabedaten
- ▶ Anforderungen (*constraints*)
  - ▶ Taktfrequenz
  - ▶ Laufzeitanforderungen

## Ausgaben

- ▶ Hardware-Modell als RTL-Verilog
- ▶ Testrahmen und -vektoren
  - ▶ Hardware RTL Simulation (*offline*)
  - ▶ Hardware/Software Co-Simulation (*online*)
- ▶ Code-Rahmen für Treiber-Software
- ▶ Skripte für andere CAD-Werkzeuge

# PICO Werkzeugfluss



# Verarbeitungsschritte 1

## Vorverarbeitung (*preprocess*)

- ▶ Programmtransformationen (Inlining)
- ▶ Statische Überprüfung auf Synthetisierbarkeit
- ▶ Erzeuge Lint und bit-genaue Simulationsmodelle

## Ablaufplanung (*schedule*)

- ▶ Schleifentransformation und -ablaufplanung
- ▶ Statische Überprüfung auf Synthetisierbarkeit
- ▶ Erzeuge Post-Schedule und Thread-genaue Simulationsmodelle

# Verarbeitungsschritte 2

## Hardware-Synthese (*synthesize*)

- ▶ Optimierung und Abaufplanung auf Instruktionsebene
- ▶ Ressourcenallokation
- ▶ Hardware-Erzeugung auf Register-Transferebene
- ▶ Erzeuge RTL Simulationsmodelle und Testrahmen

## Ausliefern (*package*)

- ▶ Stelle alle erforderlichen Dateien für nachfolgende Werkzeuge bereit

# Simulationsmodelle 1

Für automatische Regressionstests

## “Goldene” C Simulation

- ▶ Führt Programm nur in **Software** aus
- ▶ Vergleicht Ergebnisse mit Referenzdaten (*golden output*)
- ▶ Oder: Sammelt Referenzdaten für spätere Phasen

## Lint-Simulation

- ▶ Führt **dynamische Fehlersuche** durch
- ▶ Sucht uninitialisierte Variablen, Überläufe bei beschränkten Bitbreiten, Verletzung von Array-Grenzen

## Post-Schedule C Simulation

- ▶ Erzeugt **Testvektoren** für RTL Simulation

## RTL Simulation

- ▶ Simuliert erzeugtes Verilog **taktgenau** mit oben gesammelten Testvektoren
- ▶ Erlaubt Simulation von veränderten Umgebungsparametern

# Simulationsmodelle 2

## Bit-genaue SystemC-Simulation

- ▶ Überprüft korrekte **Berechnung** auf Transaktionsebene

## Thread-genaue SystemC-Simulation

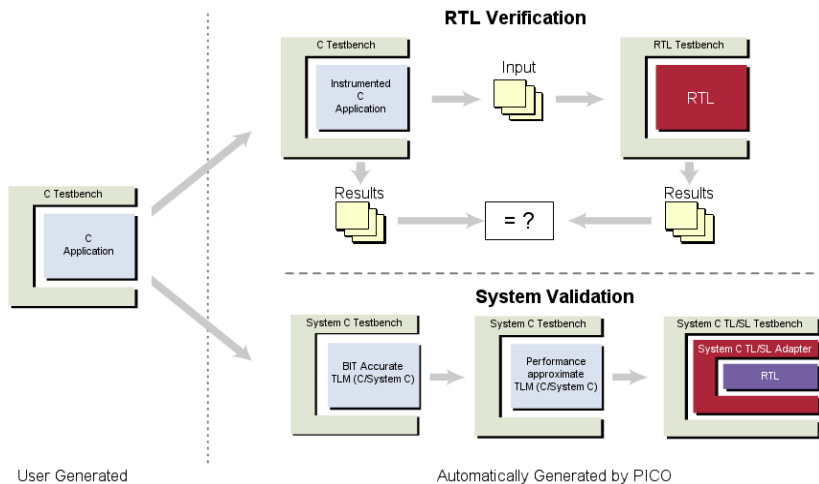
- ▶ Modelliert **parallele** Abläufe in Hardware auf Transaktionsebene
- ▶ Für erste Abschätzungen der Rechenleistung
- ▶ Läuft **deutlich** schneller als RTL-Simulation

## RTL Co-Simulation

- ▶ Simuliert Software und Hardware taktgenau **zusammen** auf RT-Ebene
- ▶ Kann sehr langsam sein

# RTL und System-Simulation

*offline* und *online* Simulation

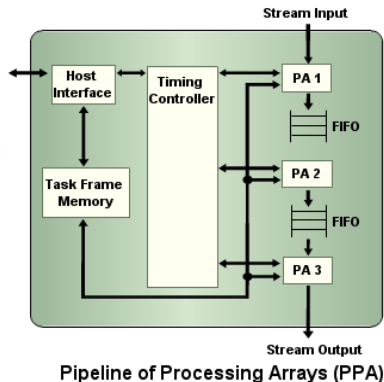


# Mikroarchitektur

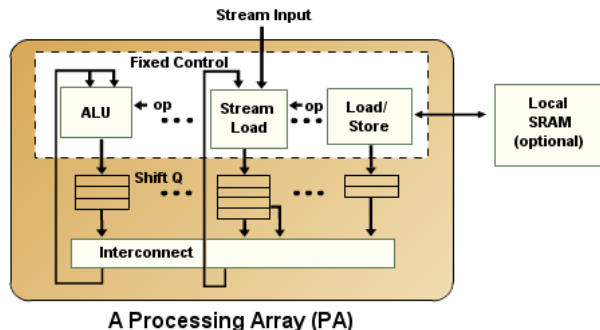


# Pipeline aus Processing Arrays (PPA)

- ▶ Processing Array (PA) entspricht ungefähr einem Schleifennest
- ▶ Mehrere PAs können parallel laufen
- ▶ Kommunizieren über
  - ▶ Datenströme
  - ▶ gemeinsame Speicher
  - ▶ gemeinsame Register
- ▶ Jedes PA hat eigene Flußkontrolle

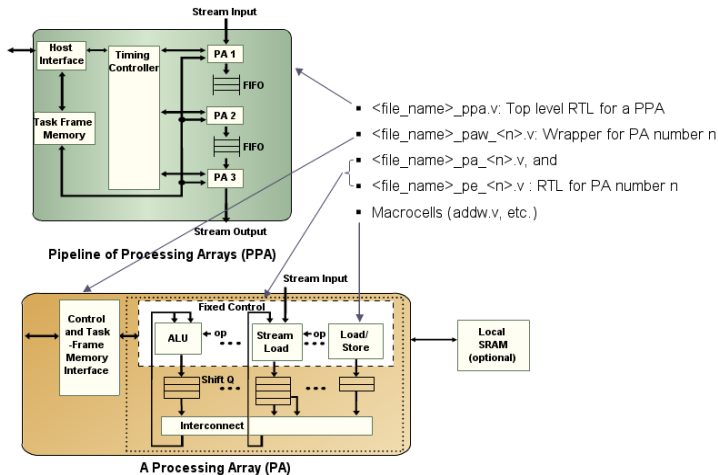


# Processing Array (PA)



- ▶ Parallele Basisoperationen
  - ▶ Add, Sub, Mul, Load, Store, ...
- ▶ Ähnlichkeiten zu VLIW-Architektur
- ▶ Kann nur im ganzen angehalten oder wieder gestartet werden

# Strukturierung der Hardware-Beschreibung



# Abbildung von C auf eine PPA

## Beispielprogramm

```
char x[64], z[64];
int sum;

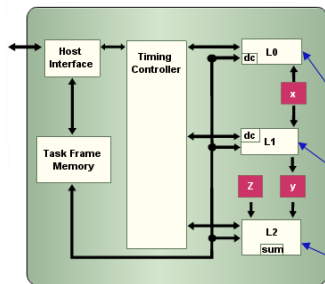
void ppa(void) {
    int i, dc=0;
    char y[64];
    for (i=0; i<64; i++) dc += x[i];
    dc = dc >> 6;

    for(i=0; i<64; i++) y[i] = x[i]-dc;

    sum = 0;
    for (i=0; i<64; i++) sum += y[i]*z[i];
}
```

# Abbildung von C auf eine PPA

## PPA-Architektur: Code



```
char x[64], z[64];  
int sum;
```

```
void ppa(void) {  
    int i, dc=0;  
    char y[64];  
    for (i=0; i<64; i++) dc += x[i];  
    dc = dc >> 6;
```

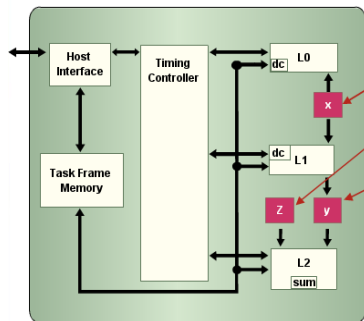
```
    for(i=0; i<64; i++) y[i] = x[i]-dc;
```

```
    sum = 0;  
    for (i=0; i<64; i++) sum += y[i]*z[i];
```

```
}
```

# Abbildung von C auf eine PPA

## PPA-Architektur: Daten

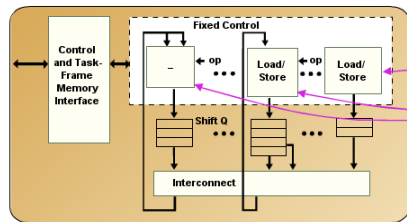


```
char x[64], z[64];  
int sum;
```

```
void ppa(void) {  
    int i, dc=0;  
    char y[64];  
    for (i=0; i<64; i++) dc += x[i];  
    dc = dc >> 6;  
  
    for(i=0; i<64; i++) y[i] = x[i]-dc;  
  
    sum = 0;  
    for (i=0; i<64; i++) sum += y[i]*z[i];  
}
```

# Abbildung von C auf eine PPA

## PA-Architektur: Operatoren



A Processing Array (PA)

```
char x[64], z[64];  
int sum;
```

```
void ppa(void) {  
    int i, dc=0;  
    char y[64];  
    for (i=0; i<64; i++) dc += x[i];  
    dc = dc >> 6;
```

```
    for(i=0; i<64; i++) y[i] = x[i]-dc;
```

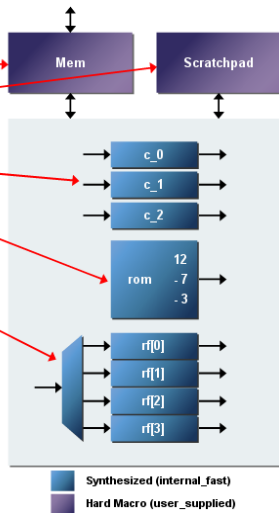
```
    sum = 0;  
    for (i=0; i<64; i++) sum += y[i]*z[i];  
}
```

# Abbildung von Arrays auf Speicher

```
int mem[200];
```

```
void ppa(void) {  
    int i;  
    short scratchpad[16][16];  
    unsigned char c[3];  
    const char rom[3] = {12, -7, -3};  
    unsigned char rf[4];  
    ...  
}
```

- File scope → external access
- Local scope + constant index usage → scalarization
- “const” modifier → ROMs
- #words, #bits → External vs. synthesized






# Parallelität

# Tasks

```
void ppa(void); // PPA procedure

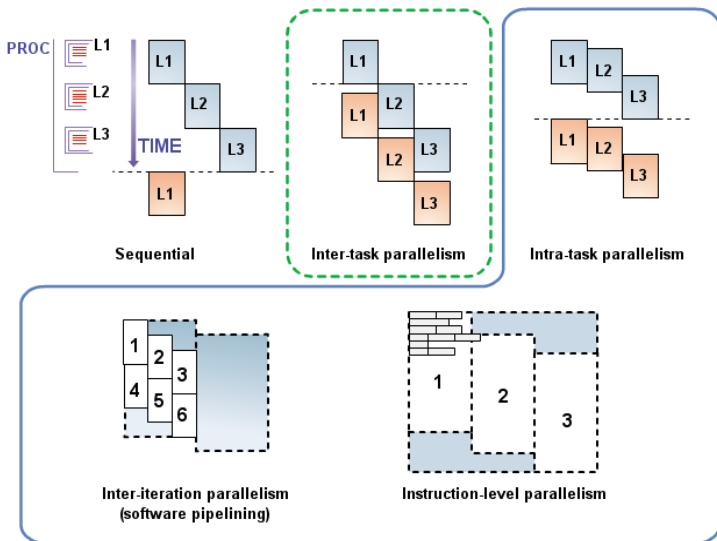
int main (int argc, char **argv) {
    int ppaid,i;

    ppaid = PICO_initialize_NPA(ppa);
    for(i=0;i<N;i++)
        ppa();
    PICO_finalize_NPA(ppaid);
}
```



- ▶ **Task:** Ein Aufruf einer in Hardware übersetzten Prozedur
- ▶ Im Beispiel:  $N$  Tasks, je einer pro Aufruf

# Arten von Parallelität im PICO Modell



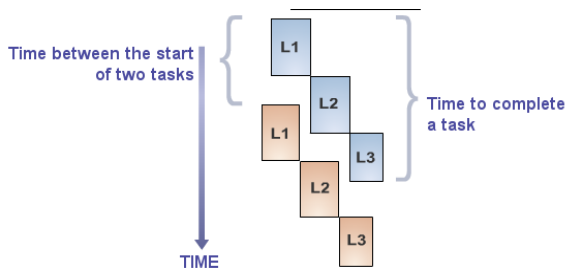
# Parallelität auf Task-Ebene

Wie schnell können Tasks gestartet werden?

Bestimmt **Durchsatz** des Systems

Wie schnell kann ein Task bearbeitet werden?

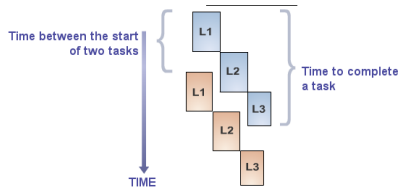
Bestimmt **Latenz** eines Tasks



**Blue and Red represent two different tasks**

# PICO Terminologie

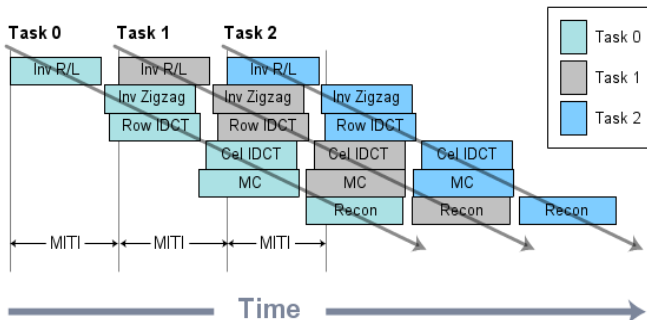
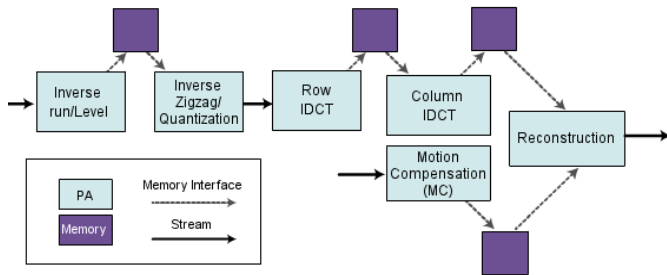
- ▶ **Minimum Inter-Task Interval (MITI)**
  - ▶ Minimale Zeit zwischen zwei Task-Starts
- ▶ **Task-Latenz**
  - ▶ Zeit zur Bearbeitung eines Tasks
- ▶ **Überlappende** Ausführung von Tasks
  - ▶  $MITI < Latenz$
- ▶ **Nichtüberlappende** Ausführung von Tasks
  - ▶  $MITI \geq Latenz$



Blue and Red represent two different tasks

# Beispiel: Task-Parallelität

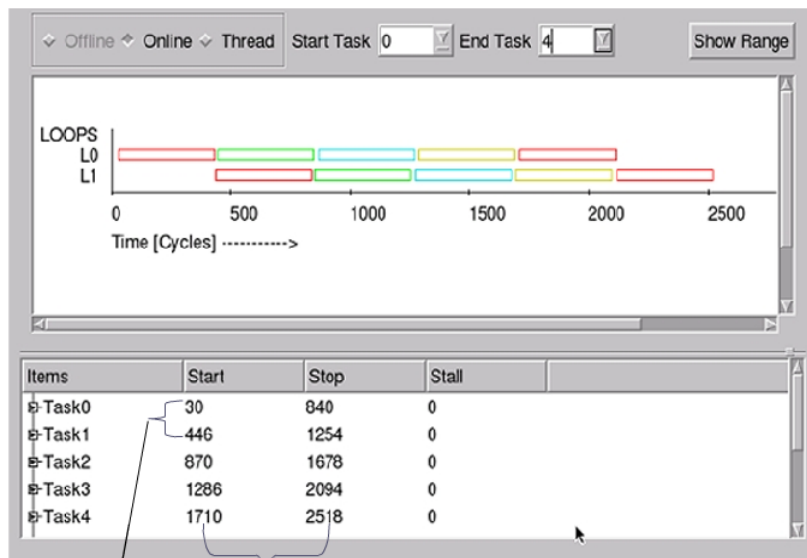
Videodekompression



# Anforderungen an Rechenleistung

- ▶ Vorgaben für Compiler
- ▶ Werden mit minimalem Hardware-Aufwand zu erreichen versucht
- ▶ **Zieltaktfrequenz**
- ▶ **MITI** in Takten
- ▶ Erreichen wird aber nicht garantiert!
- ▶ Tatsächliche Werte werden durch Simulation genau bestimmt

# Beispiel: Analyse der Rechenleistung durch Simulation



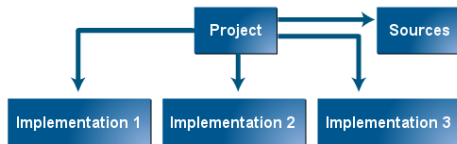
Difference is MITI

Difference is Task Latency



# Benutzung der PICO-Werkzeuge

# Organisation eines Projektes



## ▶ Quellen

- ▶ In Hardware abzubildende Prozedur
- ▶ Hauptprogramm mit Aufruf der Prozedur
- ▶ Testdaten

## ▶ Implementierungen

- ▶ Verschiedene Hardware-Realisierungen
- ▶ Ergebnis jeweils einer Compiler-Laufes
- ▶ Ggf. mit unterschiedlichen Constraints

# Nutzung verschiedener Modelle/Programme

Referenzmodell (C, C++ oder MATLAB)

- ▶ Direkte Umsetzung der Spezifikation
- ▶ I.d.R. ungeeignet für Hardware-Umsetzung

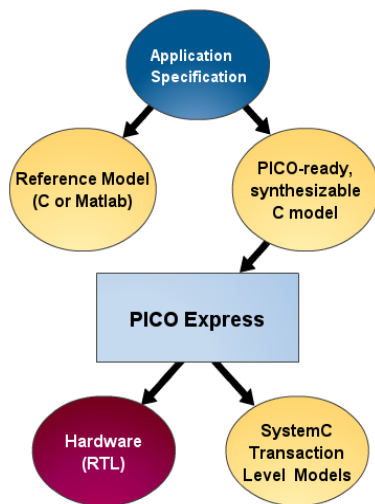
Synthetisierbares C Modell

- ▶ Optimiert für Hardware-Synthese
- ▶ I.d.R. andere Algorithmen und Datenstrukturen

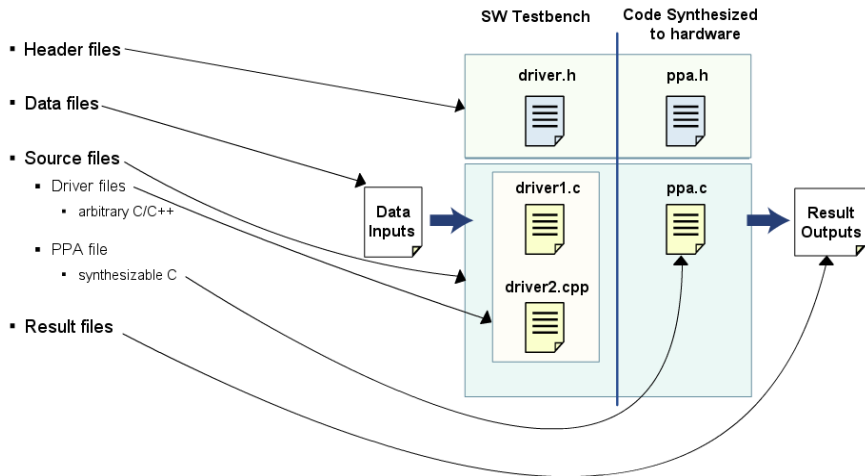
System-Validierungsmodell

- ▶ Zur Überprüfung Hardware-Schnittstellen und Protokollen

Automatisch von PICO erzeugt



# Dateistruktur Kompilierung mit PICO



# Beschreibung einer PPA-Prozedur

Im Beispiel eben: `ppa.c`

- ▶ Programm und Daten müssen in einer Datei stehen
  - ▶ ... kann über `#include` umgangen werden
- ▶ **Genau eine** Prozedur wird als PPA-Prozedur markiert
  - ▶ Beschreibt Hardware-Block
  - ▶ Unterprozeduren werden inlined
- ▶ Globale Variablen dürfen verwendet werden
  - ▶ Arrays, `const` Arrays, skalare Variablen
  - ▶ Konstante Arrays erlaubt, werden ROMs
- ▶ C Präprozessorkonstrukte erlaubt

# Deklaration der PPA-Prozedur

```
void ppa(void) { ... }
```

Alle Kommunikation über **globale** Variablen

- ▶ Live-Ins müssen im Hauptprogramm vor PPA-Aufruf gesetzt werden
- ▶ Live-Outs müssen in PPA vor Rückkehr gesetzt werden

Arten der Kommunikation

- ▶ Skalare Variablen: Werden Hardware-Register
- ▶ Arrays: Werden RAMs/ROMs in Hardware
- ▶ Datenströme: Werden FIFOs in Hardware (kommt noch ...)

```
Live Out →  
Live In →  
Local →  
int a, r;  
int x[10];  
void wsum(void) {  
    int k, tmp=0;  
    for (k=0;k<10;k++)  
        tmp += a*x[k];  
    r = tmp;  
}
```

# Beispiel: PPA-Prozedur

```
int cube (int x);  
int a,b;  
int c[10];
```

```
void ppa(void) {  
    int k;  
  
    b = cube(a);  
  
    for(k=0;k<10;k++) b += c[k];  
}
```

```
int cube(int x) {  
    int k,tmp = 1;  
    #pragma unroll  
    for(k=0;k<3;k++) {  
        tmp *= x;  
    }  
    return tmp;  
}
```

## ▪ Original code:

- PPA function
- Sub-functions
- Directives (pragmas)
- Loops

# Synthetisierbare vs. nicht-synthetisierbare Konstrukte

## Arbitrary C:

```
ptr = y;
for (i=0; i < N; i++) {
    sum = (*ptr++) + x[i];
    diff = (*ptr++) - x[i];
    ...
}
```

## PICO C:

```
for (i=0; i < N; i++) {
    sum = y[2*i] + x[i];
    diff = y[2*i+1] - x[i];
    ...
}
```

- ▶ PICO C läßt sich mit ANSI C Compiler übersetzen und ausführen
- ▶ ... aber nicht alles ANSI C läßt sich mit PICO übersetzen



# PICO C: Einschränkungen

- ▶ Nur Arrays erlaubt, keine Pointer
- ▶ Strukturierte Programme, kein `goto`
- ▶ Keine Gleitkommaoperationen
- ▶ Kein `struct`, `union`
- ▶ Kein `switch / case`
- ▶ Kein Variablen deklariert als `static` oder `volatile`
- ▶ Keine Schleife umschliesst eine Folge von Schleifen

... PICO entwickelt sich aber ständig weiter: Sprachumfang wächst!

# Explizite Bit-Angaben zu Variablenbreiten

## Zwei Möglichkeiten

- ▶ `#pragma bitsize variablenname bitbreite`
  - ▶ Betrifft nur **PICO-kompilierte** Programme
- ▶ `sc_int<bitbreite>, sc_uint<bitbreite> variablenname`
  - ▶ Verändert Semantik **aller** Programme (benötigt SystemC-Bibliotheken)

Empfehlung: Bit-Breiten für Schnittstellen und interne Zustandsvariablen setzen

- ▶ Alle anderen Bitbreiten werden dann **automatisch** inferiert

```
#pragma bitsize a 9
#pragma bitsize b 10
int a, b, x;
x = a + b;
/* inferiert: x braucht 11 Bits */
```

# Kommunikation mit Datenströmen

Ermöglichen effiziente parallele Ausführung

## Externe Datenströme

- ▶ Zur Kommunikation zwischen PPA und Software
- ▶ Benutzer definiert **außerhalb** der PPA-Datei:
  - ▶ `<Typ> pico_stream_input_<Name> (void)`
  - ▶ `void pico_stream_output_<Name> (<Typ>)`
- ▶ Können beliebigen Inhalt in vollem ANSI C haben
- ▶ Funktionen können in PPA aufgerufen werden
- ▶ Werden automatisch in Software/Hardware-Schnittstelle übersetzt

## Interne Datenströme

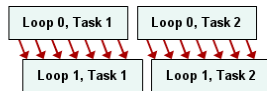
- ▶ Zur Kommunikation zwischen zwei Schleifen
- ▶ **In** PPA-Datei deklarieren: `FIFO (<Name>, <Typ>)`
- ▶ `pico_stream_input / _output_<Name>` werden automatisch erzeugt

# Beispiele für interne Datenströme

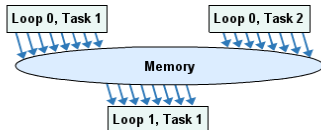
```
for() {  
    pico_stream_output_x(a);  
}  
for() {  
    b = pico_stream_input_x();  
}
```

```
for() {  
    x[i] = a;  
}  
for() {  
    b = x[j];  
}  
  
for() { //(same code as above  
        // with different MITI)  
    x[i] = a;  
}  
for() {  
    b = x[j];  
}
```

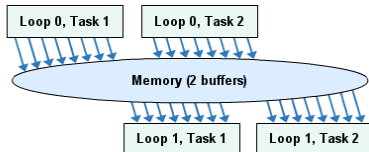
Streaming data:



Memory with single buffer:



Memory with double buffer:



# Software-Hauptprogramm

*driver code*

- ▶ Ruft in PPA kompilierte Prozedur auf
- ▶ Kann Testrahmen oder komplette Anwendung sein
- ▶ Muss Eingabedaten bereitstellen und Ausgabedaten entgegennehmen
- ▶ PICO stellt API für Kommunikation mit PPA bereit
  - ▶ I.d.R. memory-mapped Register

```
extern void ppa(void);

int main(int argc, char* argv[]){
    int i, id;
    ...
    /* initialize */
    id := PICO_initialize_NPA(ppa);

    /* call PPA */
    for (i=0; i<N; i++) {
        /* prepare inputs */
        ppa();
        /* process outputs */
    }

    /* finalize */
    PICO_finalize_NPA(id);

    return 0;
}
```

# Zusammenfassung

- ▶ Dieses Mal nur Sicht aus dem Orbit!
- ▶ Essenz: PICO erlaubt die Programmierung von ACS auf Hochsprachenebene
- ▶ Kein eigenes Übungsblatt zu PICO
- ▶ Bei Interesse an eigenen Experimenten
  - ▶ CBT nachfragen, selektiv Dokumentation lesen
  - ▶ Dann bekannte Aufgaben nochmal in PICO realisieren
- ▶ Mögliche studentische Arbeiten
- ▶ Demo!