Rekonfigurierbare Rechensysteme (Skript Teil 1)

Prof. Dr. Martin Middendorf
Schwarmintelligenz und Komplexe Systeme
middendorf@informatik.uni-leipzig.de

Literatur

Bücher

C. Bobda: Introduction to Reconfigurable Computing: Architectures, algorithms and applications. Springer 2007.

R. Vaidyanathan, J. L. Trahan: Dynamic Reconfiguration: Architectures and Algorithms. Kluwer, 2004.

M. Platzner, N. Wehn: Dynamically Reconfigurable Systems. Springer, 2010.

J. Cardoso, M. Hübner (Eds.)Reconfigurable Computing - From FPGAs to Hardware/Software Codesign. Springer, 2011.

S. Churiwala (Ed.): Designing with Xilinx® FPGAs. Springer, 2017.

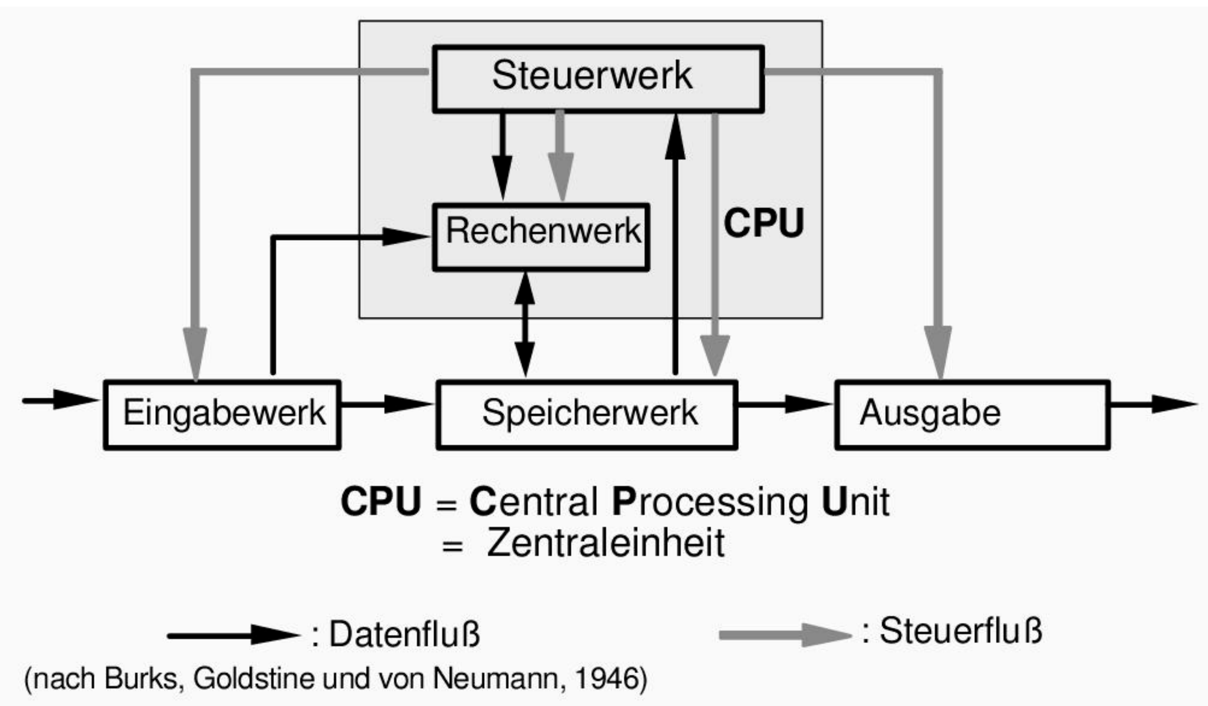
I. Skliarova, V. Sklyarov: FPGA-BASED Hardware Accelerators. Springer, 2019.

Überblick

1. **Einführung**
2. **Coarse-grained und fine-grained rekonfigurierbare Systeme**
3. **Field Programmable Gate Arrays (FPGAs)**
4. **Hyperrekonfigurierbare und multi-level rekonfigurierbare Architekturen**
5. **Rekonfigurierbare Gitter**
6. **Prozessorfelder mit optisch rekonfigurierbaren Bussen**

Von-Neumann-Architektur

- **Der von-Neumann-Rechner**
  - **Speicher:** Speicherung von Progamm und Daten
  - **Rechenwerk:** Ausführung arithmetischer/logischer Operationen
  - **Steuerwerk:** Steuerung des Programmablaufs
  - **Ein- und Ausgabewerk:** Eingabe von Daten/Programmen, Ausgabe von Ergebnissen „nach außen“



Von-Neumann-Architektur

- **Klassischer Universalrechner** mit fest vorgegebener Struktur, die unabhängig vom bearbeiteten Problem ist.

- Zentrale Steuerung durch das Steuerwerk

- Programme werden von außen eingegeben und **im gleichen Speicher** wie die Daten abgelegt. Daten und Programm sind **binär codiert**.

  Interpretation eines Speicherinhalts hängt nur vom **aktuellen Kontext** des laufenden Programms ab.

- Speicher: in Einheiten (Speicherzellen) unterteilt und ansonsten unstrukturiert.

  Zugriff auf Speicherzellen: **direkt** über ihre Adresse.
  - ⇨ Befehle des Programms stehen in aufeinanderfolgenden Speicherzellen

Von-Neumann-Architektur

- ⇨ Nächster Befehl wird durch Erhöhen der Befehlsadresse um Eins angesprochen - außer bei **Sprungbefehlen**:

  - **Unbedingter Sprung**: Befehlsadresse wird auf label gesetzt

    GO TO label

  - **Bedingter Sprung**: Befehlsadresse wird auf label gesetzt, falls test, sonst um Eins erhöht

    IF test THEN label

- **Befehlsabarbeitung nach 2-Phasen Konzept:**

  - ⇨ **Interpretations-Phase**: Der entsprechend dem Stand des Befehlszählers geholte Inhalt einer Speicheradresse wird als Befehl interpretiert

  - ⇨ **Ausführungs-Phase**: Aufgrund der im Befehl enthaltenen Adresse wird ein Speicherwort geholt und als Datenwert verarbeitet

Von-Neumann-Architektur

- Die meisten heutigen Rechner beruhen auf dem von-Neumann-Prinzip besitzen jedoch Erweiterungen (allgemein: **General Purpose Processor (GPP)**)

- **Funktionseinheiten heutiger Rechner:**
  - ⇨ Arbeitsspeicher
    - → normalerweise Speicherhierarchie
    - → im Befehl codierte Adressen werden durch das Betriebssystem modifiziert
  - ⇨ Central Processing Unit (CPU): Steuerwerk + Rechenwerk
    - → meist mehrere Rechen- und Steuerwerke
    - → externe Signal können den Programmablauf unterbrechen
  - ⇨ Ein- /Ausgabeeinheit
  - ⇨ Datenwege zum Austausch von Informationen zwischen den Funktionseinheiten

Von-Neumann-Architektur

- **Vorteile der von-Neumann-Architektur**
  - Geringer Speicheraufwand
  - Geringer Hardwareaufwand
- **Nachteile**
  - Befehle werden nacheinander über die Verbindung zwischen Speicher und Steuerwerk geholt („von-Neumann-Flaschenhals“)
  - Festlegung einer sequentiellen Bearbeitungsreihenfolge wird gefordert (intellektueller „von-Neumann-Flaschenhals“)
  - Geringe Strukturierung der Daten
  - Maschinenbefehl bestimmt den Operandentyp (semantische Lücke)

Alternative Konzepte

Beispiel: **Befehlsphasen-Pipelining**

Befehlsausführung sei in folgende Schritt aufgeteilt:

1. Befehl holen (F)
2. Befehl dekodieren (D)
3. Operanden holen (O)
4. Befehl ausführen (E)

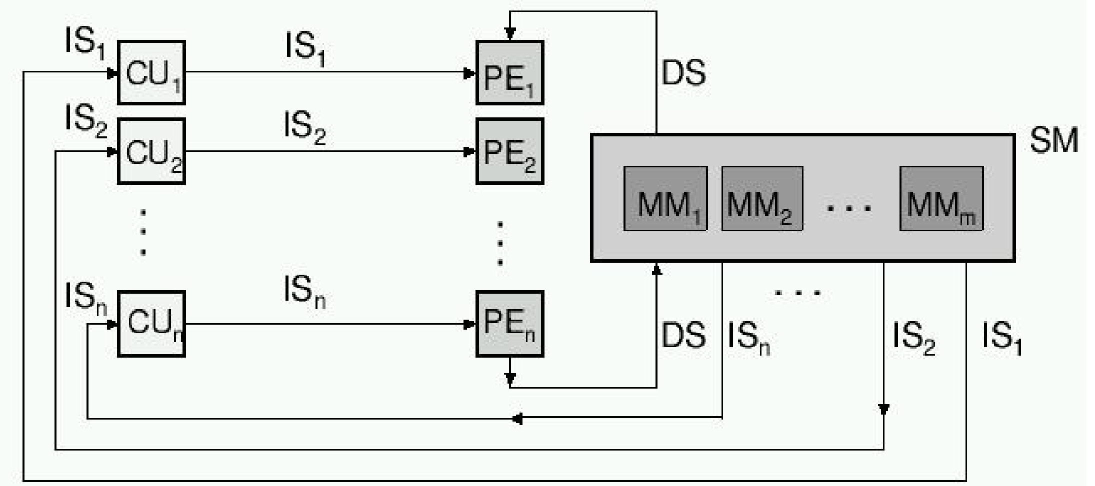
Ziel: Die Schritte beim Pipelining sollten möglichst etwa gleich viel Zeit benötigen

Takt 1	F_1			
Takt 2	F_2	D_1		
Takt 3	F_3	D_2	O_1	
Takt 4	F_4	D_3	O_2	E_1
Takt 5		D_4	O_3	E_2
Takt 6			O_4	E_3
Takt 7				E_4

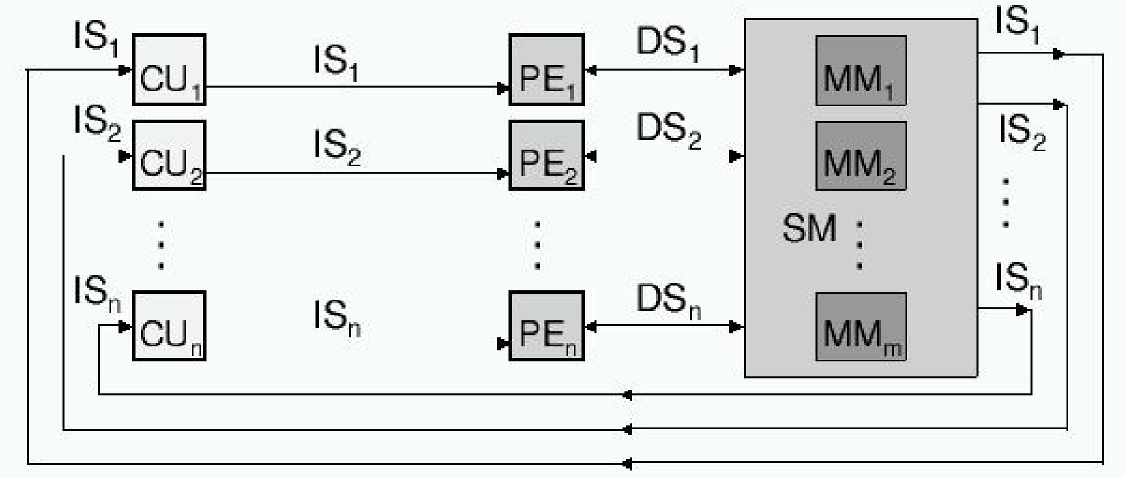
Ausführen von N Befehlen benötigt $N+3$ Takte statt $4N$ bei Ausführung ohne Pipelining

Alternative Konzepte

MISD: Multiple Instruction Single Data: nur Spezialanwendungen



MIMD: Multiple Instruction Multiple Data



Application Specific Instruction Set Processors

Application Specific Instruction Set Processor (ASIP):
Spezialisierung des Instruktionssatzes für eine gegebene Klasse von Anwendungen (häufig als Ergänzung einer Basisarchitektur)

Beispiele:
- ⇨ Operatorverkettung (multiply-accumulate Instruktion)
- ⇨ Vektoroperationen
- ⇨ Anwendungen: Multimedia, kryptographische Aufgaben, Mikro-Controller, als Teilblöcke in heterogenen Systemen on Chip

Goossens et al.

Zusätzlich oft spezielle Funktionseinheiten, die bestimmte Instruktionen unterstützen
Beispiele:
- ⇨ Pixel-Operationen,
- ⇨ 1/sqrt(x)

Vorteile gegenüber Universalprozessoren
- -höhere Performance
- -niedrigere Kosten (kleinere Chipfläche, weniger Pins)
- -geringere Codegröße
- -geringere Leistungsaufnahme

Domain Specific Processors

Ziel: Überwinde den von-Neumann Flaschenhals zum Speicher durch Optimierung des Datenpfades für eine gegebenen Klasse von Anwendungen

Digitale Signalprozessoren (**DSPs**, Digital Signal Processors):

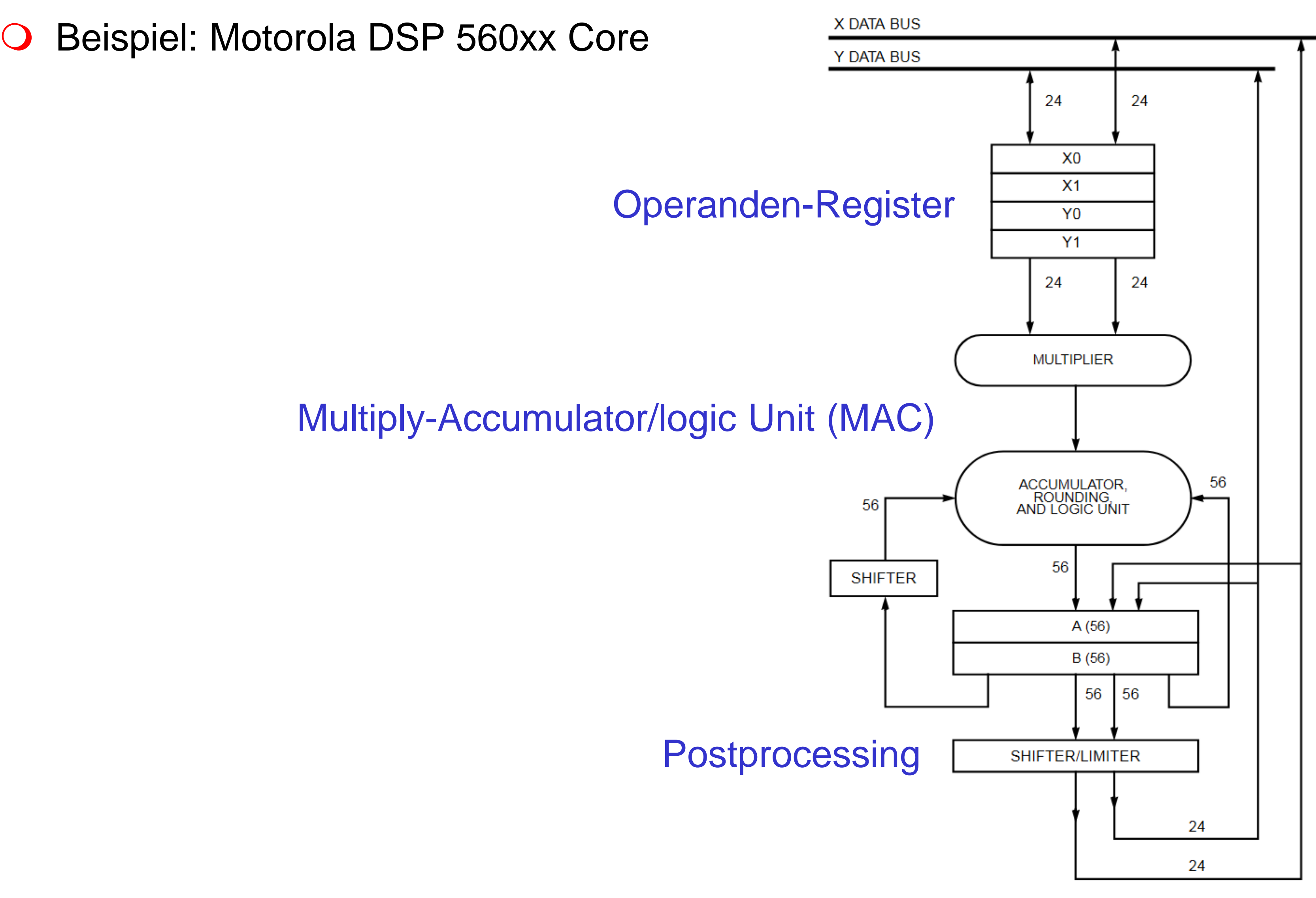
- Aufgabe: Verarbeitung digitaler Signale
- Architektur ist optimiert, um bestimmte Signalverarbeitungsalgorithmen (Fast Fourier Transformation, Filter) schnell ausführen zu können
- Typische Operation: Multiplikation-Akkumulation-Operationen (MAC)
- Der Datenpfad ist so ausgelegt, dass eine oder mehrere MACs in einen Zyklus durchgeführt werden können
  - ⇨ Speicherzugriff ist reduziert indem direkt auf den Eingabedatenfluss zugegriffen wird

Domain Specific Processors

- Spezielle Datenformate, die hardwaremäßig unterstützt werden: Festkomma, geringe Datenwortbreite (16 Bit)
- **Hardwareeigenschaften:**
  - schnelle I/O-Einheiten
  - ein oder mehrere spezielle MAC-Einheiten
  - schneller Speicherzugriff: mehrere Speicherbänke mit jeweils eigenem Bus
  - mehrere Busse auf dem Chip
  - spezielle Adressverarbeitung
    - Spezialhardware zur Berechnung von Speicheradressen (arbeitet parallel zu den Verarbeitungseinheiten)
    - besonders schnelle Adressgenerierung für typische Zugriffsmuster auf Daten: Beispiel Circular Addressing (wiederholter sequentieller Zugriff auf Block von Daten)
  - spezielle Schleifenbefehle ermöglichen Ausführung ohne Extrazyklus für Erhöhung/Testen des Schleifenzählers (Zero-Overhead Looping)

Domain Specific Processors

- Beispiel: Motorola DSP 560xx Core



Application Specific Integrated Circuits

Ziel: Entwerfe eine gesamte Schaltung speziell für eine gegebene Anwendung

Application Specific Integrated Circuit (ASIC)

Der Anwendungsalgorithmus wird direkt als Schaltung auf dem Chip realisiert

- ⇨ Datenpfad ist nur für eine Anwendung hin optimiert
- ⇨ es müssen keine Instruktionen geholt und dekodiert werden
- ⇨ Eingabedaten werden möglichst direkt verwendet
- ⇨ möglichst viele Funktionen sollen gleichzeitig auf dem Chip ablaufen

Application Specific Integrated Circuits

Beispiel:

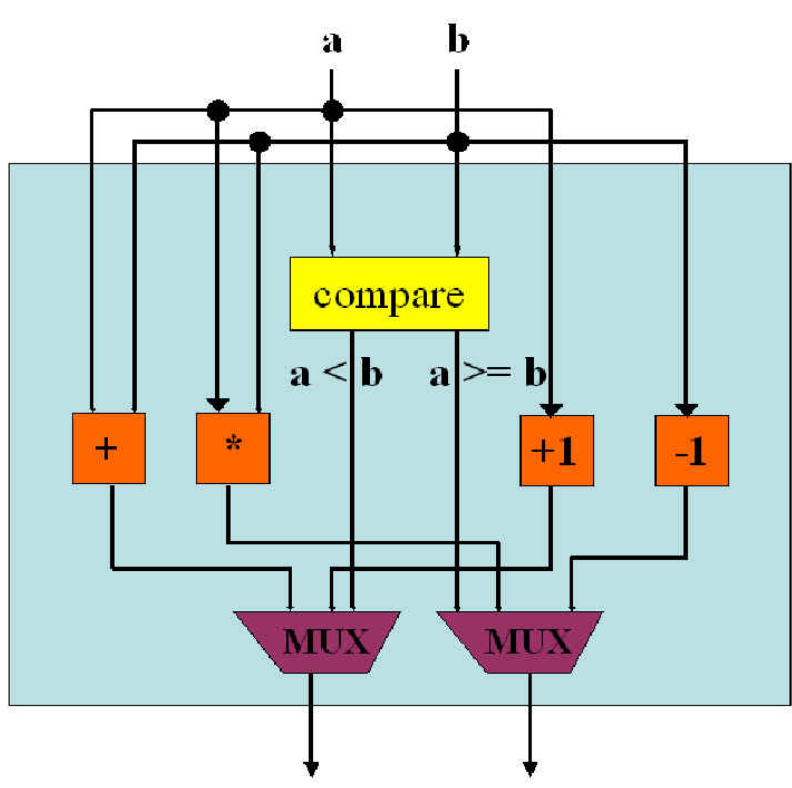
Implementation auf Universalrechner

if (a < b) then
{d = a+b;
c = a*b;}
else
{d = a+1;
c = b-1;}

z.B. 5 Befehle nacheinander

Laufzeit $\geq 5 * t_{instruction}$

Implementation auf ASIC:



Ausführung in einem Takt

Laufzeit = Delay auf längstem Pfad

Zusammenfassung

1. **GPPs (z.B. von-Neumann-Rechner):**
   - flexibel und für unterschiedliche Anwendungen geeignet
   - viele Einschränkungen durch Programmkodierung und die Ausführungsreihenfolge
     - Programm muss an die Maschine angepasst werden
   - sehr hoher Stromverbrauch

2. **ASIPs:**
   - flexibel
   - Einschränkungen durch Programmkodierung und die Ausführungsreihenfolge
     - Programm muss an die Maschine angepasst werden
   - spezielle Instruktionen für eine Anwendungsklasse mildern die Einschränkungen
   - hoher Stromverbrauch

3. **DSPs:**
   - flexibel und effizient nur für die gegebene Anwendungsklasse

4. **ASICs:**
   - für eine spezielle Anwendung gemacht
     - sehr effizient und geringer Strom- und Ressourcenverbrauch
   - nicht flexibel, kann nicht an andere Anwendung angepasst werden

Rekonfigurierbare Architekturen

Ziel: Verbinde die Flexibilität der von-Neumann Architektur mit einer effizienten Ausführung unter Nutzung der inhärenten Parallelität von Hardware wie bei ASICs.

Prinzip: eines solchen Devices

- zu einem gegebenen Zeitpunkt wird hardwaremäßig die möglichst optimale Implementierung für die aktuelle Anwendung realisiert
- wenn sich die Anwendung ändert wird die Hardware adaptiert

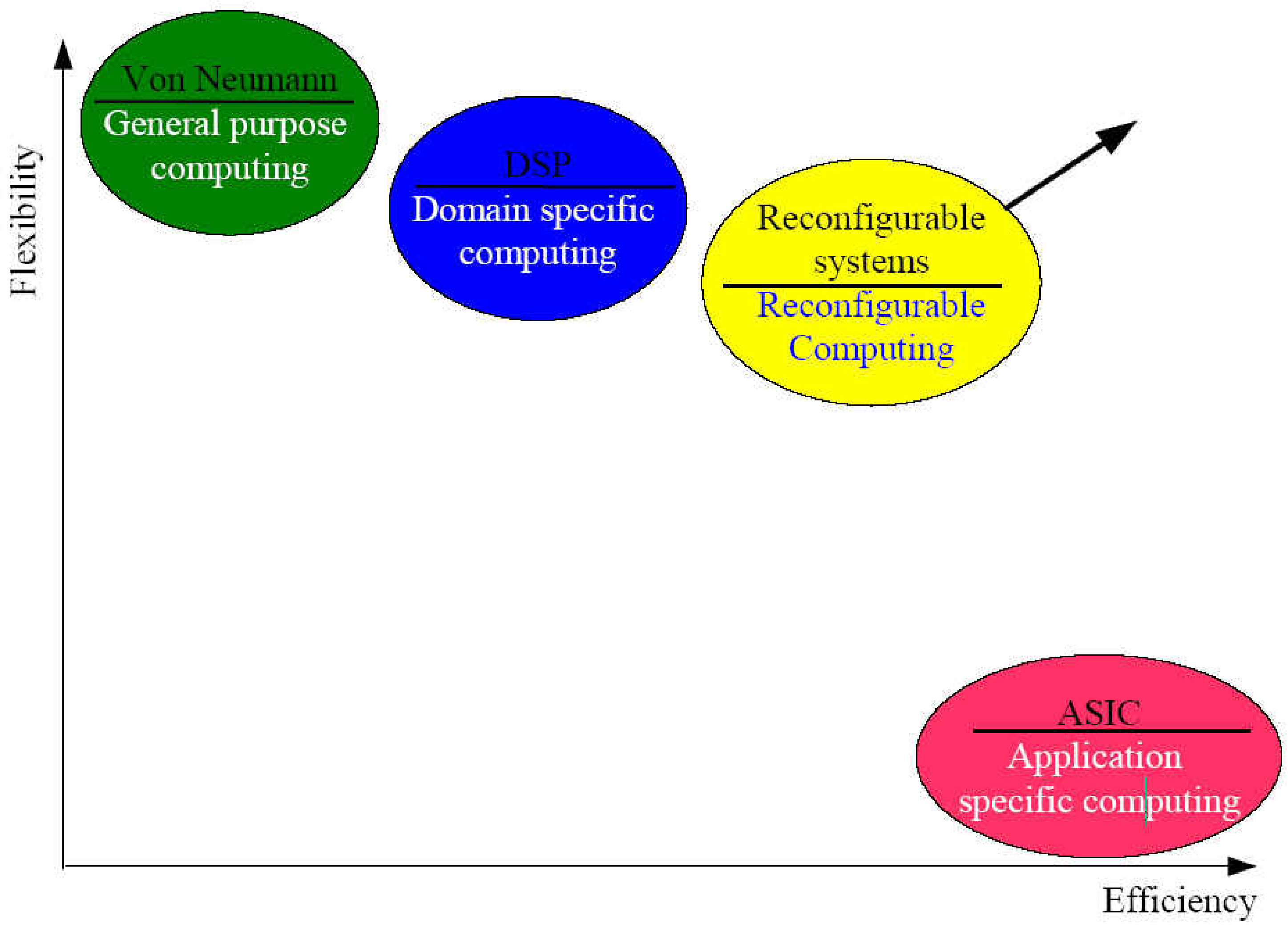
Solche Hardware wird **rekonfigurierbar** (**reconfigurable**) genannt

Reconfigurable computing beschäftigt sich mit rekonfigurierbaren Devices:

⇨ Architektur, Algorithmen, Betriebssysteme, Tools, Anwendungen

Rekonfigurierbare Architekturen

Trade-off zwischen Flexibilität und Effizienz



Rekonfigurierbare Architekturen

Anwendungsgebiete:

- Rapid Prototyping
- Post Fabrication Customization
- Multi-modal Computing Tasks
- Adaptive Rechnersysteme
- Fehlertoleranz
- High Performance Parallel Computing

Rekonfigurierbare Architekturen

Rapid Prototyping: Test von Hardware unter realen Bedingungen bevor die Produktion beginnt

- Simulation in Software
  - ⇨ relativ preiswert
  - ⇨ langsam
  - ⇨ ist nur eine Annäherung an die Realität, Genauigkeit oft schwer abschätzbar

- Emulation (Nachbildung) in Hardware: Hardwaretest unter realen Bedingungen
  - ⇨ schnell
  - ⇨ hohe Genauigkeit
  - ⇨ mehrere Wiederholungen von Testläufen sind möglich

FPGA-basierte Emulationssysteme
z.B. Synopsis Zebu Server

Rekonfigurierbare Architekturen

Post Fabrication Customization

- Vorteil durch geringe Time-to-Market-Zeiten
  - ⇨ erste Version eines Produkts wird sehr frühzeitig ausgeliefert
  - ⇨ spätere Updates durch neue Versionen werden später vor Ort beim Kunden vorgenommen
- Anpassen an die Bedürfnisse/Verhältnisse beim Kunden vor Ort
  - ⇨ Kunde kann unter mehreren Produktversionen auswählen
- Reparatur kann vor Ort beim Kunden durchgeführt werden

Rekonfigurierbare Architekturen

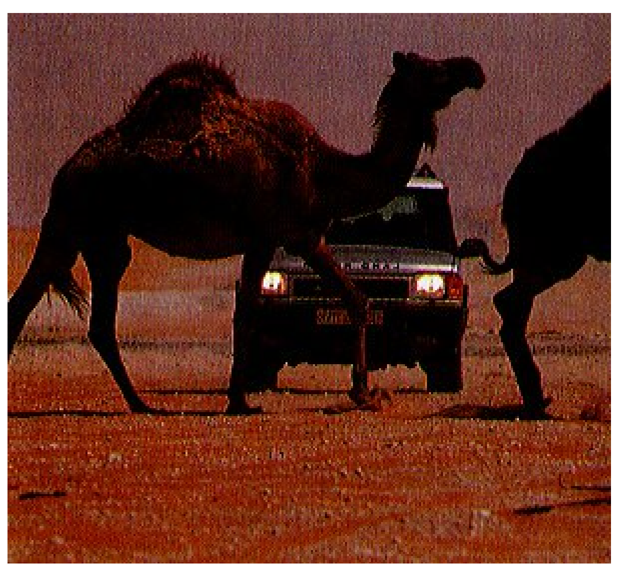
Multi-modal-Computing-Tasks sind typisch für viele rekonfigurierbare Geräte, wie Fahrzeuge, Handys, …

- ⇨ insbesondere, wenn Gewicht oder Hardwareressourcen minimiert werden müssen (und die Verwendung einer großen Anzahl von ASICs zu aufwendig wäre)
- ⇨ Echtzeitanforderungen vorhanden sind (und ein normaler GPP zu langsam ist)

- Digitale Camera
- Video phone
- Navigationssystem
- Diagnose
- Monitoring
- Spiele

Rekonfigurierbare Architekturen

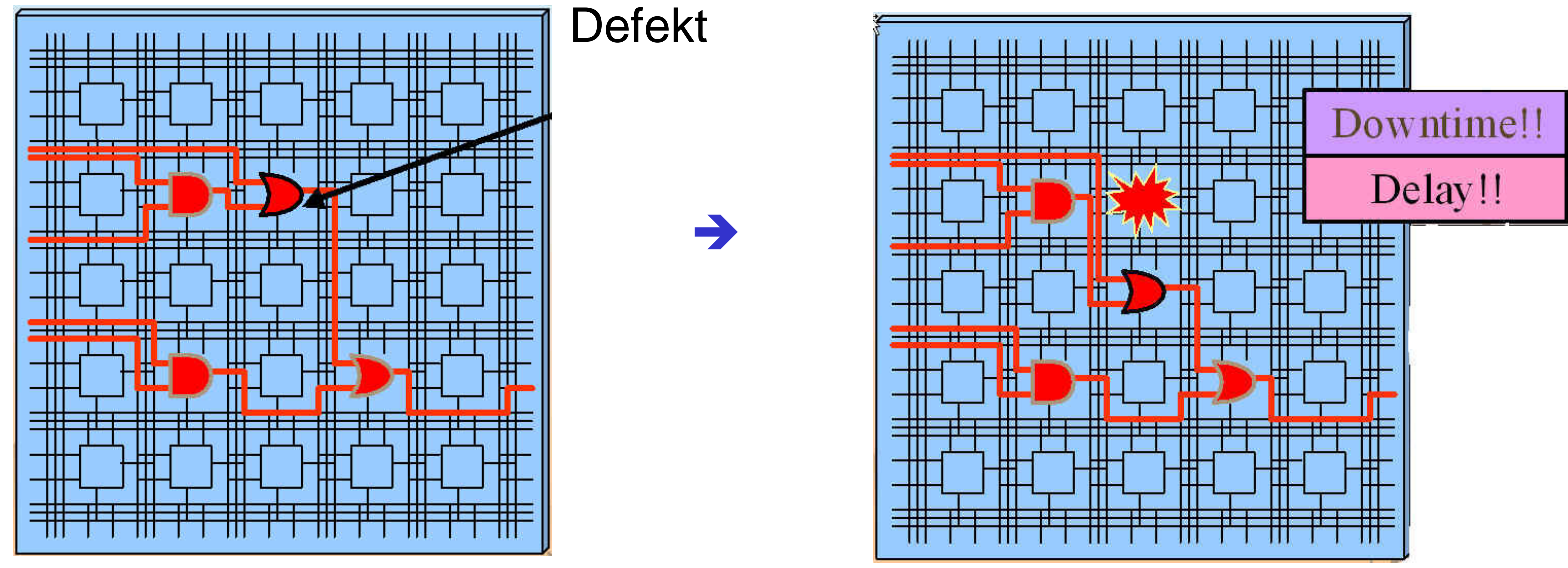
Adaptive Rechensysteme sollen ihr Verhalten und ihre Struktur selbstständig und dynamisch an den wechselnden Gebrauch durch den Benutzer, an sich ändernde Bedingungen der Außenwelt, an neue Anforderungen des Benutzers und an neue Performanzkriterien, oder wechselnde Hard- und Softwareressourcen anpassen.



Rekonfigurierbare Architekturen

Fehlertoleranz:

- Durch Ändern der Konfiguration können fehlerhafte Elemente auf dem Chip umgangen werden



Rekonfigurierbare Architekturen

High Performance Parallel Computing:

- Beispiel: Computational Gene Analysis
  - ⇨ Algorithmen zur Analyse von Genom- und Proteinsequenzen
  - ⇨ Nur der jeweils zu benutzende Algorithmus wird geladen
  - ⇨ Neue Algorithmen können auf vorhandene Hardware eingespielt werden

Time Logic: Similarity Search Engine

- Beispiel: Cloud Computing (z.B. Machine Learnig)
  - ⇨ Microsoft (Azure: Configurable cloud), Baidu, Huawei, ….

Catapult Mezzanine Card v2

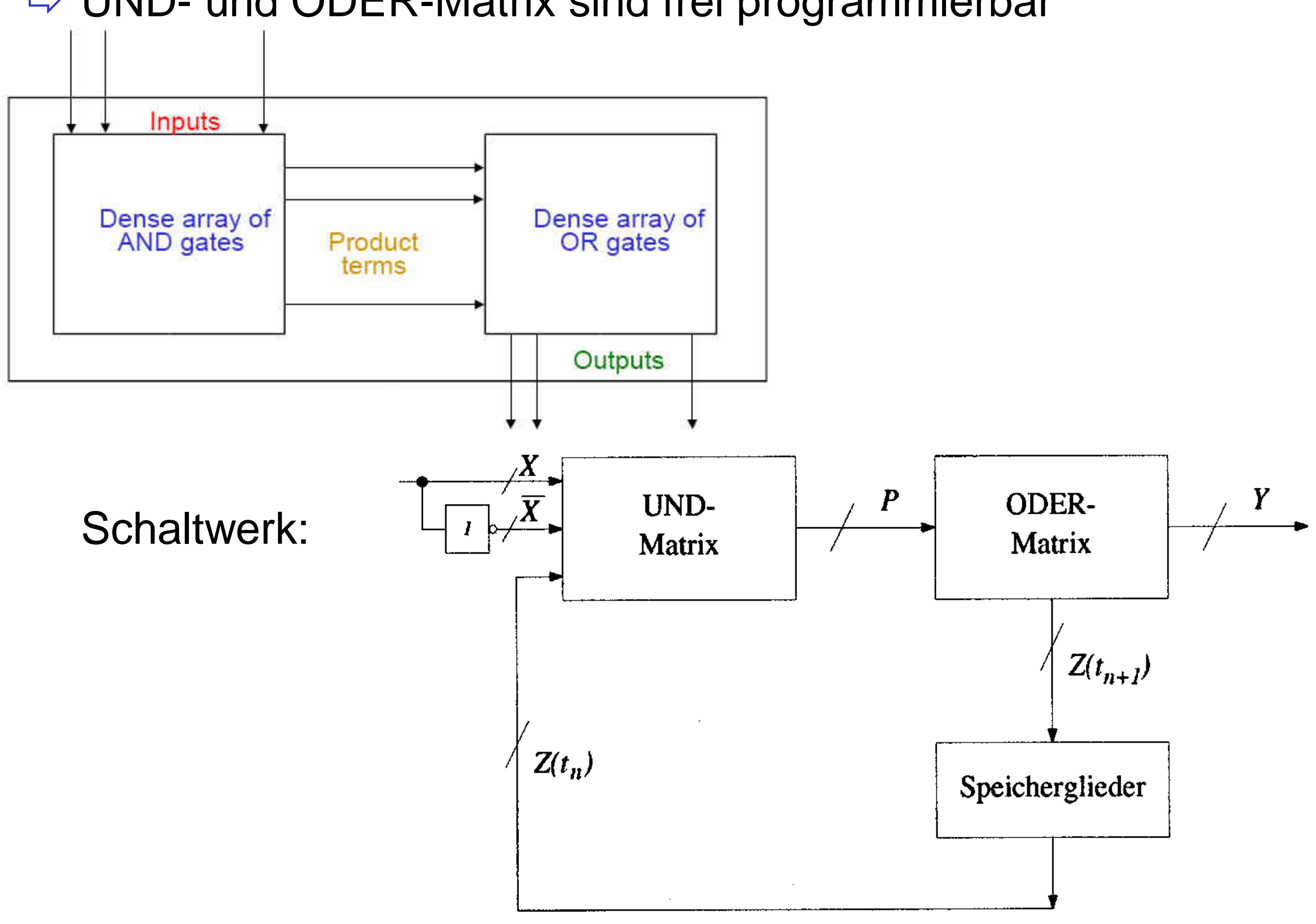
Feingranulare Rekonfigurierbare Architekturen

PLA

Programmable Logic Array (PLA)

⇨ UND- und ODER-Matrix sind frei programmierbar

Schaltwerk:



PLA

Alternative representation for high fan-in structures

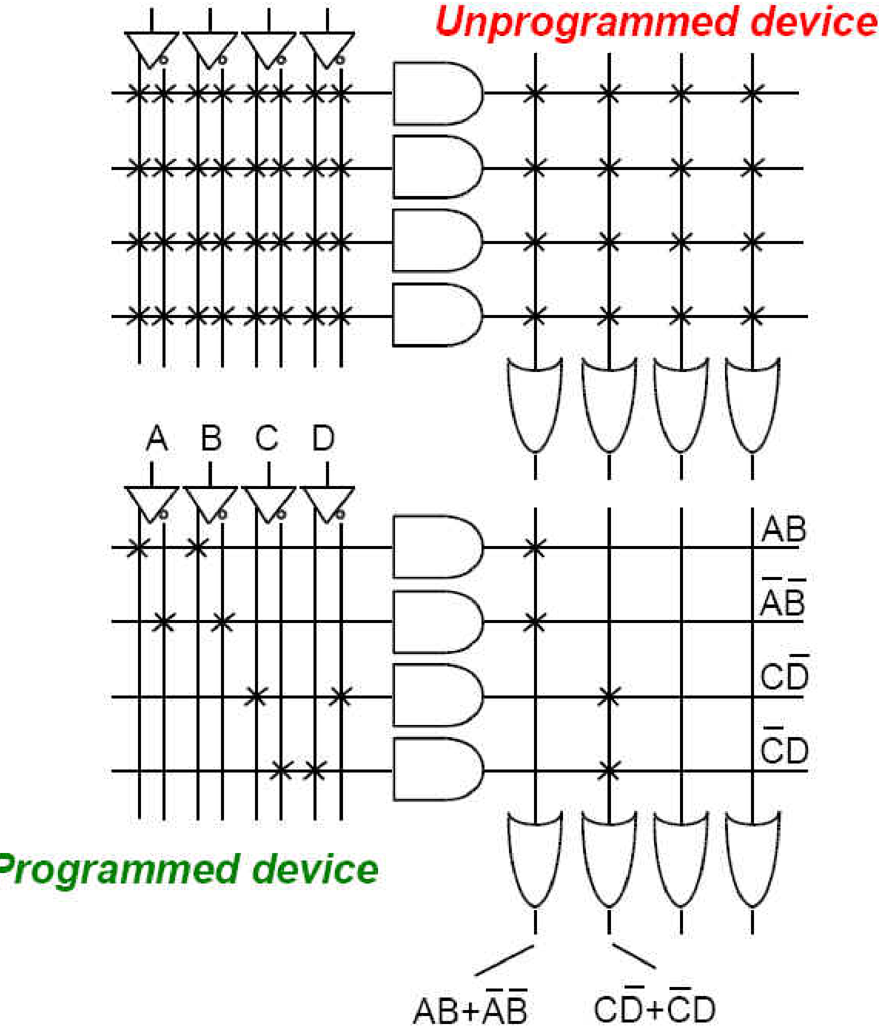
Short-hand notation so we don't have to draw all the wires!

X at junction indicates a connection

Notation for implementing

$F0 = A\,B + \overline{A}\,\overline{B}$

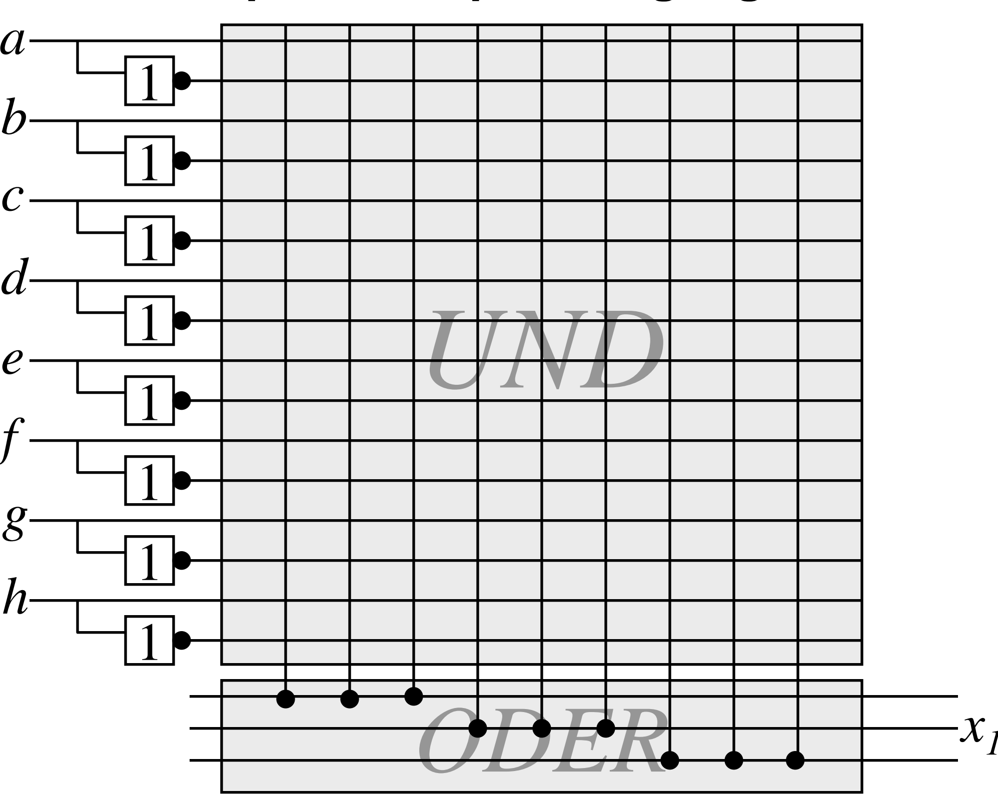
$F1 = C\,\overline{D} + \overline{C}\,D$



PAL

Programmable Array Logic (PAL)

- ⇨ **die ODER-Matrix ist vorgegeben**
- ⇨ **es steht eine feste Anzahl von Implikanten pro Ausgang zur Verfügung**
- ⇨ **die UND-Matrix ist programmierbar**



Nachteil im Vergleich zu PLA: kein „sharing“ von Produkttermen

Complex Programmable Logic Devices

Complex PLDs (CPLD)

- Verbinden meist PAL-Logik mit Flip Flops
- Logische Blöcke sind durch Verbindungsmatrix gekoppelt
- Kombinatorischer oder Register-Output
- Logik reicht für einfache Zähler, endliche Automaten, Dekodierer, usw.
  Anwendungen: Controller, Interfaces, glue logic
- CPLDs Logik reicht nicht für komplexe Operationen
- FPGAs haben deutlich mehr Logikfunktionen als CPLDs

Xilinx Coolrunner

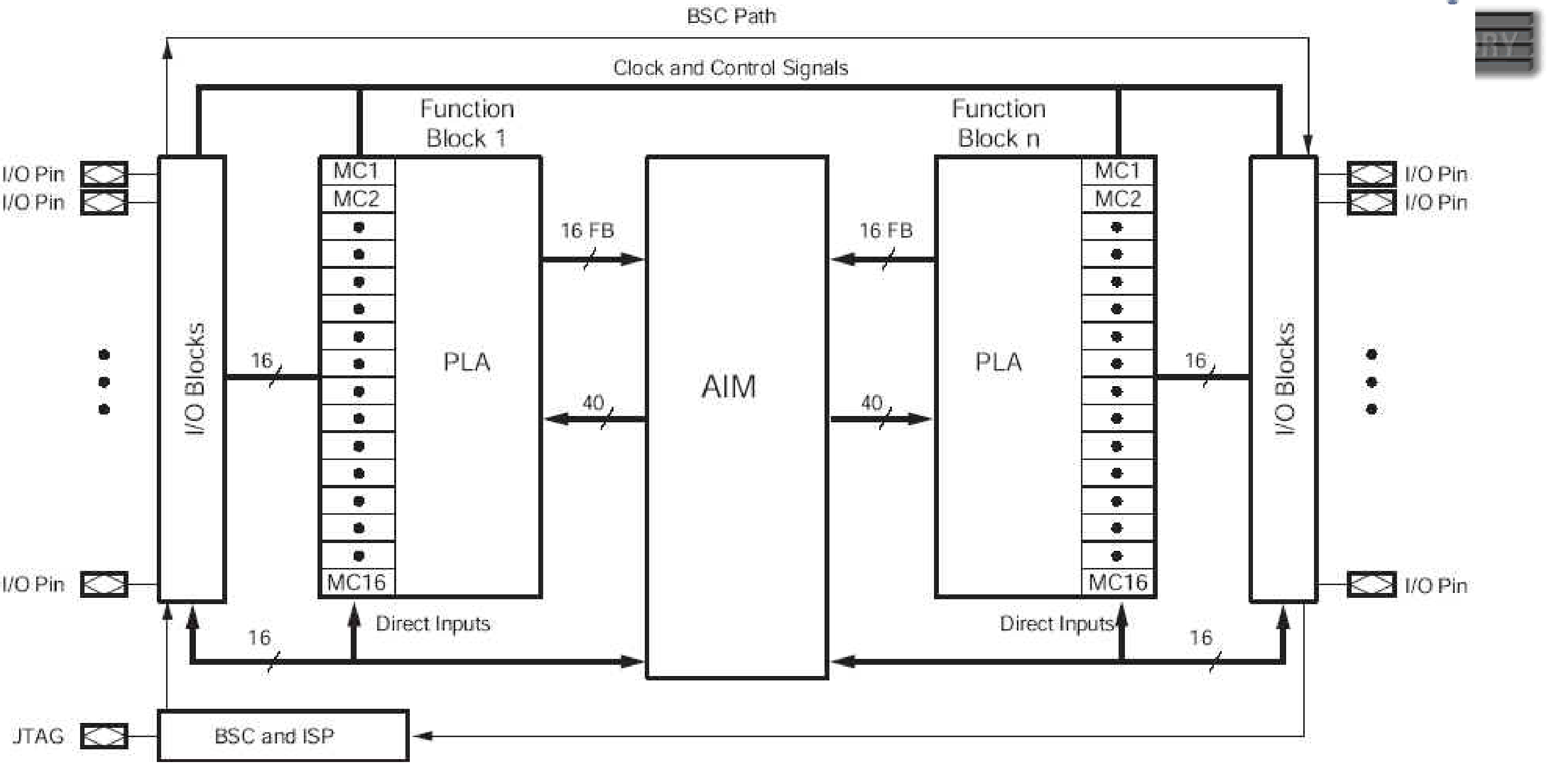
Beispiel CPLD: Xilinx Coolrunner II

low power, variable clock (low power vs. performance)

AIM: Advanced Interconnect Matrix

MC: Makrozellen (zwischen 32 – 512 erhältlich)

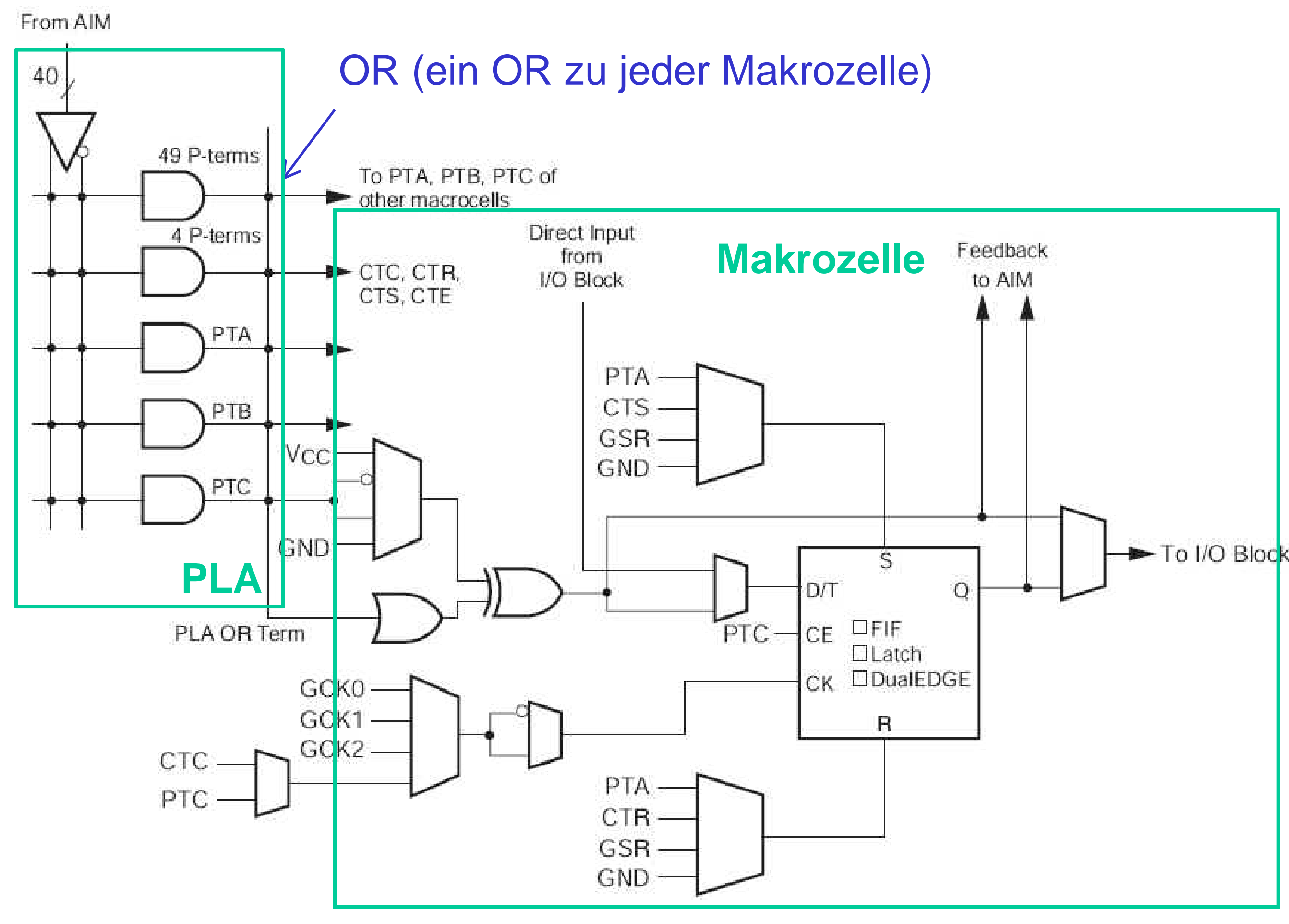
PLA: Programmable Logic Array



Xilinx Coolrunner

Makrozelle:

- Summen (OR) von Produkten (SOP) möglich mit bis zu 40 Inputs und 56 Produkttermen (P-term)
- SOP kann mit weiterem P-term in XOR Gatter kombiniert werden



Ausgabe in AIM, I/O-Block oder in Register (T- oder D-Flip-Flop, Dual EDGE getriggered möglich)

Xilinx Coolrunner

Makrozelle:

- PTA, PTB, …, PTE sind Produktterme von anderen Funktionsblöcken (FB)
- The CT (Control Terms) product terms are available for FB clocking (CTC), FB asynchronous set (CTS), FB asynchronous reset (CTR), and FB output enable (CTE)
- Global clocks (GCK), global sets/resets (GSR)
- GCK2: global clock (Frequenz geteilt durch 2,4,8, …16 möglich)
- On-The-Fly Rekonfiguration möglich, d.h. Teile des Chips können rekonfiguriert werden, während andere in Benutzung sind

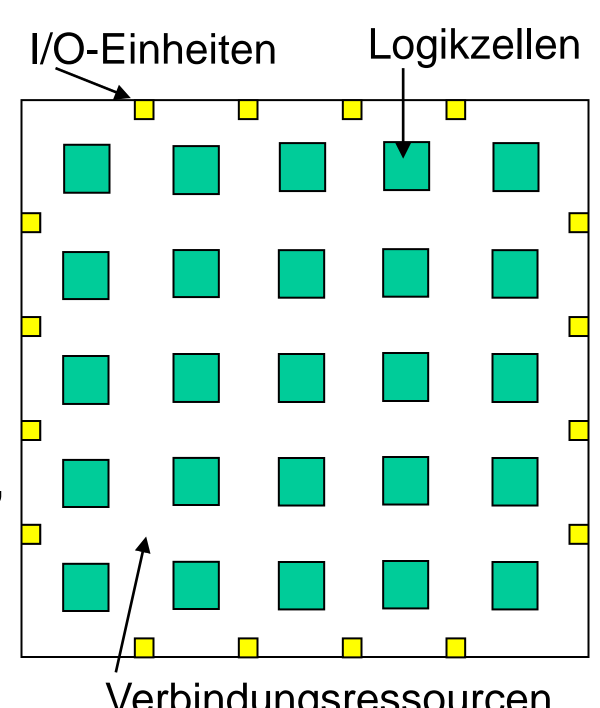
FPGAs

Field Programmable Gate Arrays (**FPGAs**)

1985 von Xilinx eingeführt

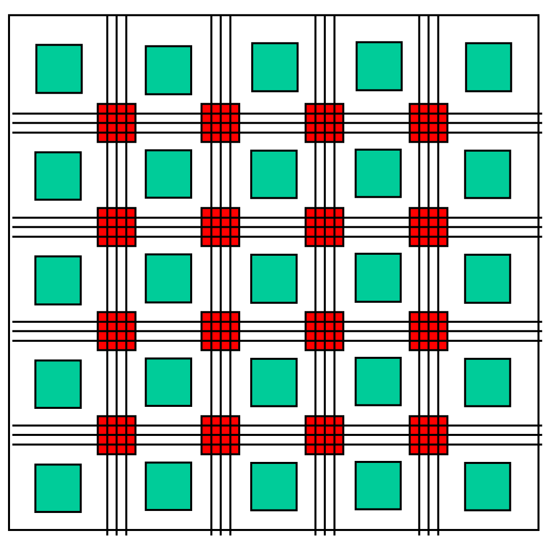
Bestandteile eins FPGA:

- Programmierbare **Logikzellen** (Makrozellen)
- Programmierbares **Verbindungsnetzwerk**
- Programmierbare **Input/Output-Blöcke**
- Die Bestandteile einer (komplexen) Funktion werden in den Makrozellen implementiert, die dann geeignet verbunden werden
- Die I/O-Blöcke können so programmiert, werden, dass sie die Eingabe oder Ausgabe der Makrozellen bedienen
- Im Unterschied zu ASICs wird die realisierte Funktion vom Benutzer nach der Herstellung festgelegt
- Physikalische Struktur und Programmierung eines FPGAs sind herstellerabhängig

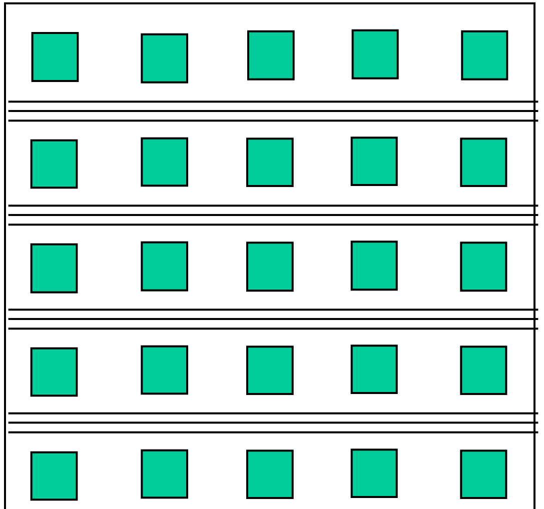


FPGAs

Organisationsformen von FPGAs



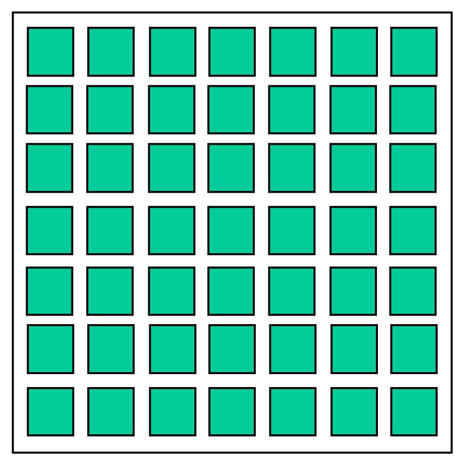
- **Symmetrischer Array**
  - ⇨ 2-D Array von Logigzellen/Prozessorelementen (PE) mit Verbindungsnetzwerk
  - ⇨ Verbindungsstellen (Switch-Matrix) an Schnittpunkten



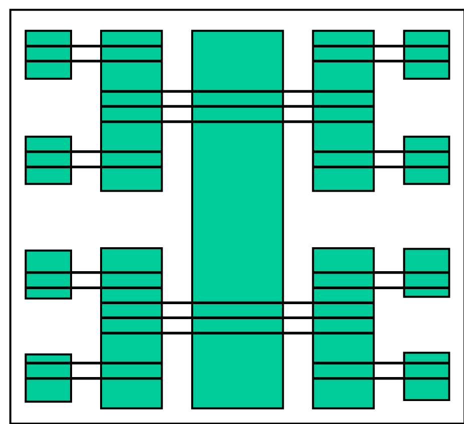
- **Zeilen-basierter Array**
  - ⇨ Zeilen von Logikzellen/Prozessorelementen
  - ⇨ Horizontales Routing über horizontale Kanäle
  - ⇨ Kanäle sind in Segmente geteilt
  - ⇨ Vertikale Verbindungen über vertikale Leitungen (nicht eingezeichnet)

FPGAs

- **Sea of Gates**
  - ⇨ 2-D Array von Logikzellen/Prozessorelementen
  - ⇨ Kein Platz für Verbindungen zwischen den PEs
  - ⇨ Verbindungen werden in eigenem Layer über den PEs realisiert

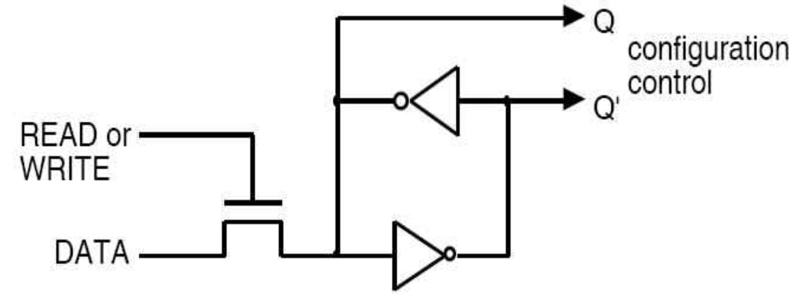


- **Hierarchisch**
  - ⇨ Hierarchisch organisierte Makrozellen
  - ⇨ Gruppe von low-level Makrozellen bildet eine Makrozelle auf höherer Ebene

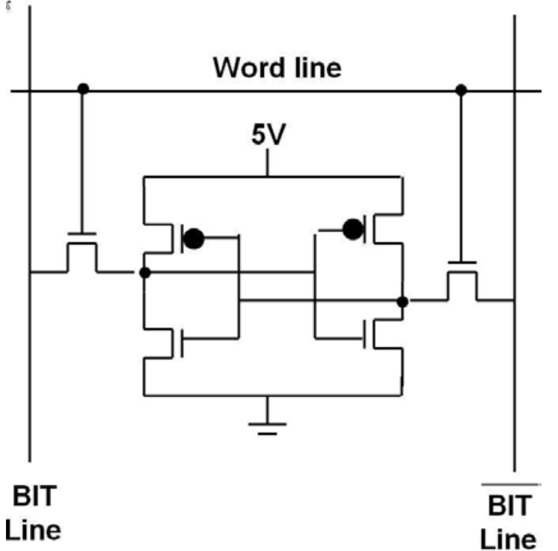


Programmiertechnologien

SRAM: (LUT-basiert)



- SRAM-Zellen werden verwendet um die Werte einer Funktion zu speichern
- Den Wert der Funktion erhält man durch Eingabe der entsprechenden SRAM-Adresse
- **Look-up table** (**LUT**): Funktion die mittels SRAM implementiert ist
- Eine neue Funktion wird implementiert indem neue Werte in die LUT geschrieben werden
  - ⇨ SRAM-basierte FPGAs können während der Laufzeit umprogrammiert (rekonfiguriert) werden
  - ⇨ Durch die Realisierung mittels SRAM, geht eine LUT Konfiguration verloren, wenn das System ausgeschaltet wird (volatile)



Programmiertechnologien

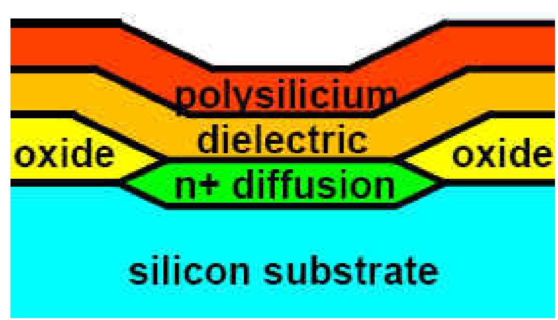
Anti-fuse:

- programmierbare Schaltelemente, bei denen die Endpunkte im Ursprungszustand nicht leitend verbunden sind
- Die Endpunkte können durch Anlegen einer hohen Spannung verbunden ("**fused**") werden
- Anti-fuse-Elemente, die in FPGAs verwendet werden, unterscheiden sich in der Konstruktion je nach Hersteller
- geringer Platzverbrauch
- niedriger Widerstand und geringe parasitäre Kapazität (im Unterschied zu Transistoren)
    - reduzierte Delays beim Routing
- nicht reprogrammierbar
- Zustand bleibt nach Ausschalten der Spannungsversorgung erhalten (non-volatile)

Programmiertechnologien

Poly-diffusion Anti-fuse: ACTEL PLICE

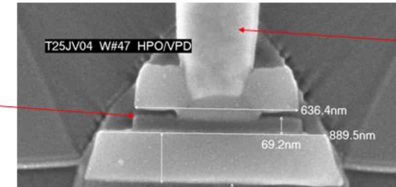
PLICE Anti-Fuse



- programmierbares Verbindungselement (im Ursprungszustand nicht leitend)
- Endpunkte: Polysilizium
- Oxyd-Nitrit-Oxyd-Isolierschicht
- Durch Anlegen einer Spannung schmilzt die Isolierschicht und schafft eine Verbindung

Unprogrammed Antifuse

Antifuse



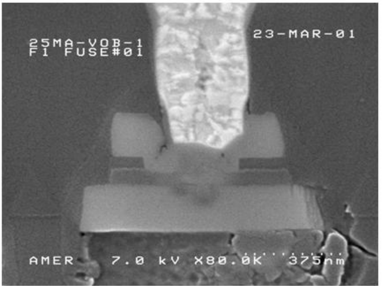
V

Metal Anti-fuse: Quick-Logic Vialink

ViaLink Anti-Fuse

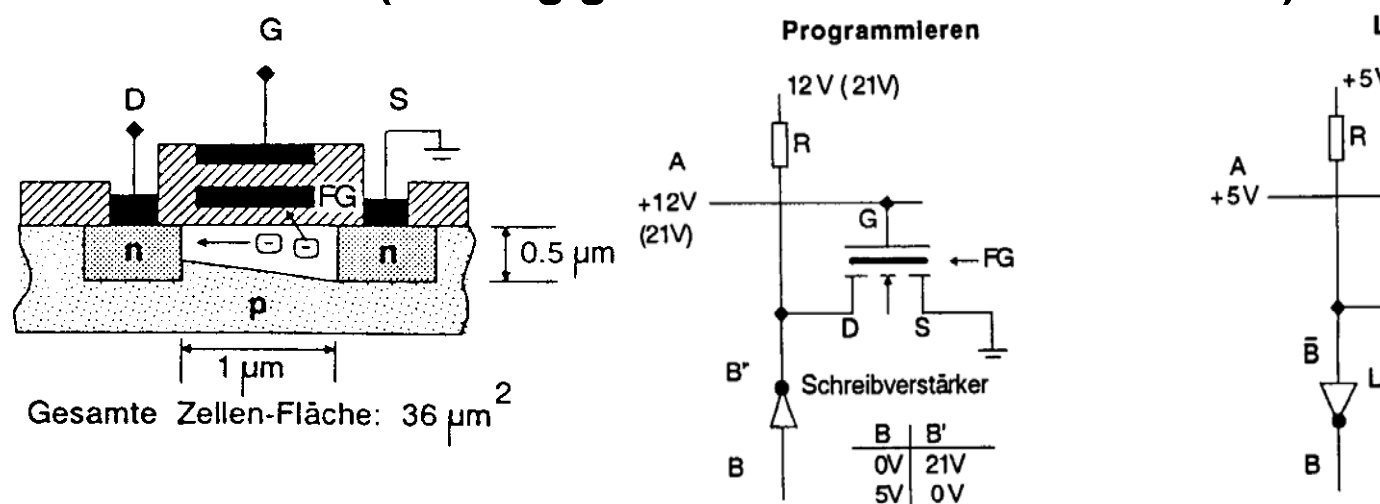
- Endpunkte: 2 Metal Layer (Titanium-Tungsten)
- Endpunkte sind durch amorphes Silizium getrennt

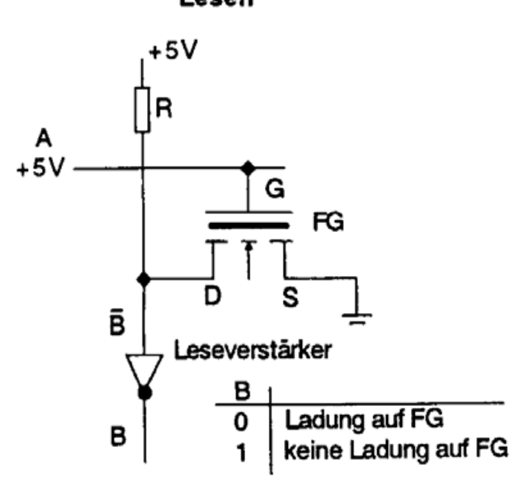
Programmed Antifuse



Programmiertechnologien

EPROM: FAMOS (floating gate avalanche MOS-transistor)





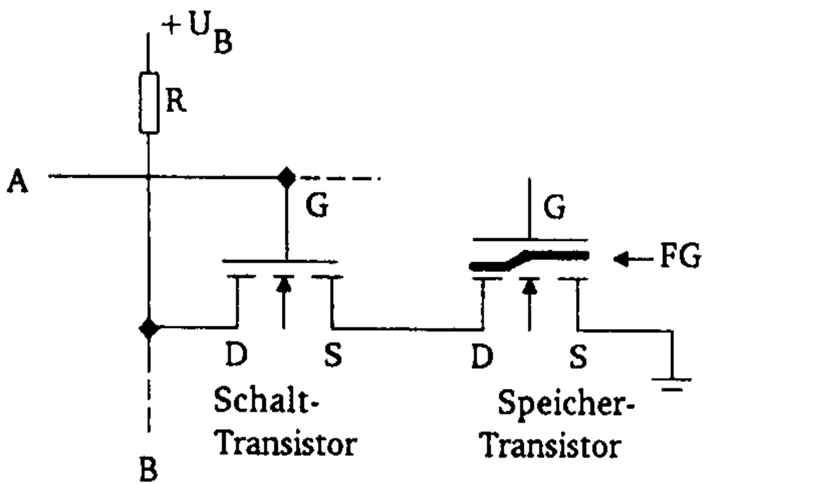
Programmieren und Lesen einer EPROM-Zelle

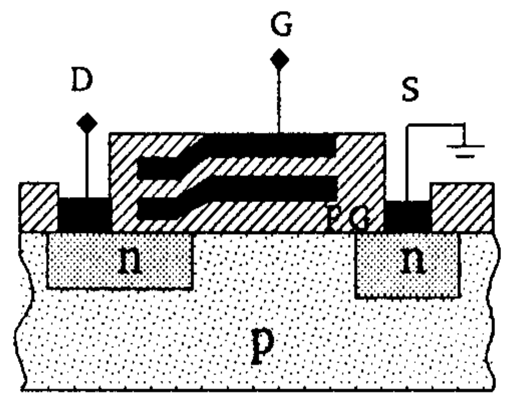
- Besitzt zweites Gate, das vollständig isoliert ist
  - ⇨ Speicherung der Ladung über 30 Jahre
- Löschen durch UV-Licht (senkt Widerstand der Isolierschicht
- Programmierung durch hohe Spannung (12-21 V)
  - ⇨ Elektronen werden angezogen auf das Floating Gate
- Lesen durch Anlegen einer niederen Spannung (5 V)
  - ⇨ ist das Floating-Gate geladen, schaltet der Transistor nicht

Programmiertechnologien

EEPROM:

- Dünne Isolierschicht des Floating Gates
  - ⇨ Lesen: Wenn das Floating Gate des Transistors (negativ) geladen ist, sperrt dieser
  - ⇨ Programmieren : Hohe Spannung (+21 V) am Gate des Transistors zieht Elektronen auf das Floating Gate (U_B = 0V)
  - ⇨ Löschen: 0 V am Gate und eine hohe Spannung am Drain des Transistors entlädt Floating Gate (logisch 0)





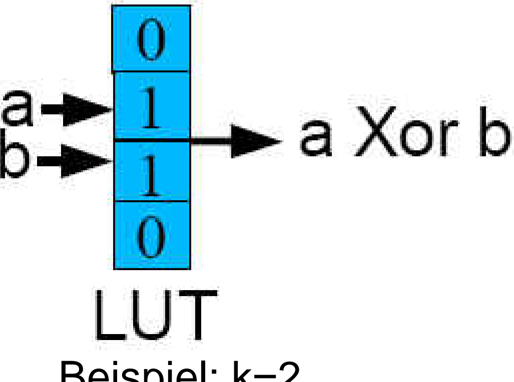
- Spezielle Weiterentwicklung sind Flash-Speicher, die mehrere Bits pro Zelle speichern können (durch mehrere Ladungslevel)

Logische Zellen

LUT: (Look-Up-Tables)

- LUTs werden als Funktionsgeneratoren in SRAM-basierten FPGAs verwendet
- eine k-Input LUT kann bis zu 2^{2^k} verschiedene Funktionen realisieren
- eine k-Input LUT besitzt 2^k SRAM-Zellen
- eine Funktion wird realisiert, indem ihre möglichen Werte in die entsprechenden SRAM-Zellen geschrieben werden
- die Eingabeleitungen werden verwendet um die entsprechenden SRAM-Zellen zu adressieren und so den Funktionswert auszugeben
- In der Praxis: meist 4- bis 6-Input LUTs

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0



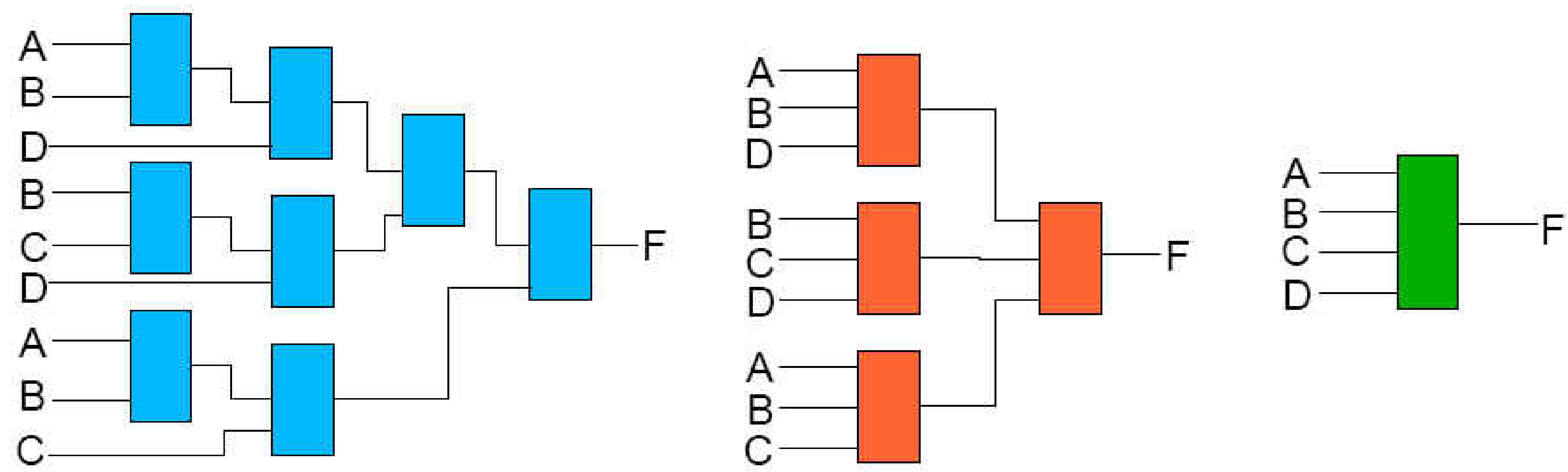
Beispiel: k=2

Logische Zellen

Beispiel:

Realisiere die Funktion $f = ABD + BC\overline{D} + \overline{A}\,\overline{B}\,\overline{C}$
mittels

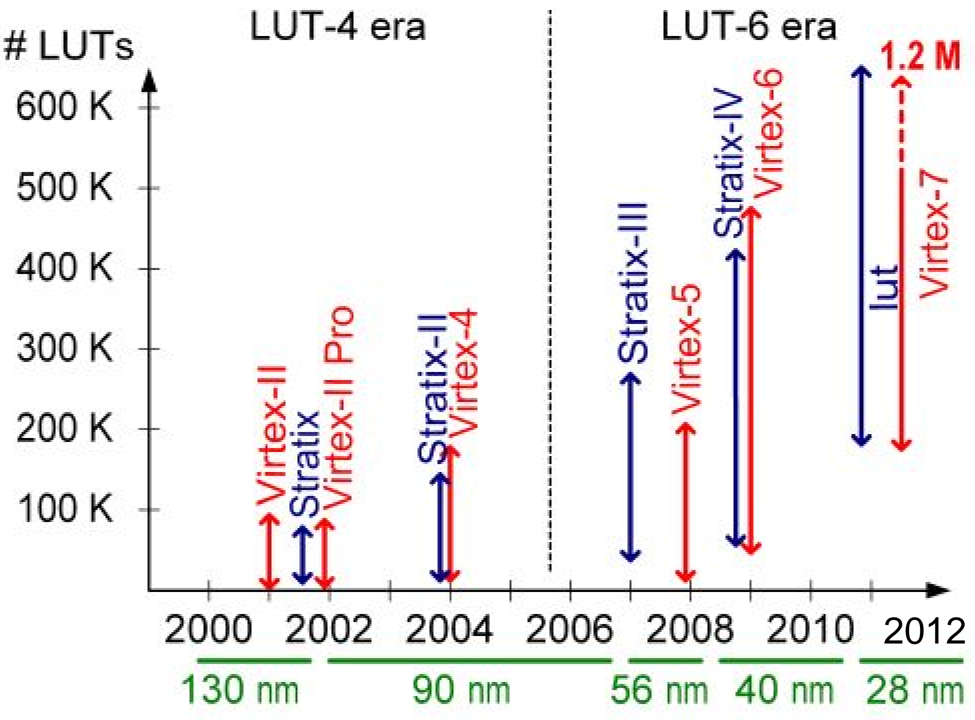
- 2-input LUTs
- 3-input LUTs
- 4-input LUTs



- Beachte: Größere LUTs reduzieren die Tiefe einer Schaltung

Logische Zellen

Größe und Anzahl von LUTs in FPGAs:

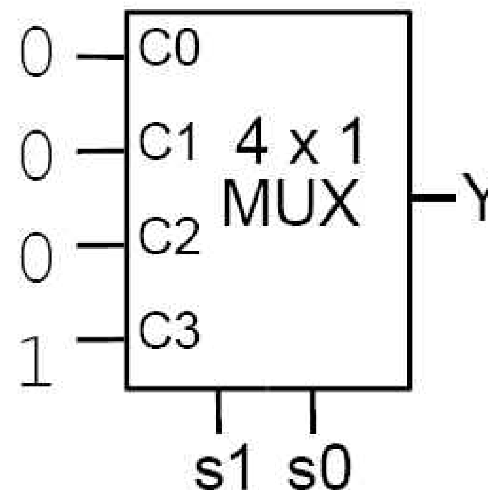


Quelle: Koch, Torrensen

Logische Zellen

Multiplexer (MUX):

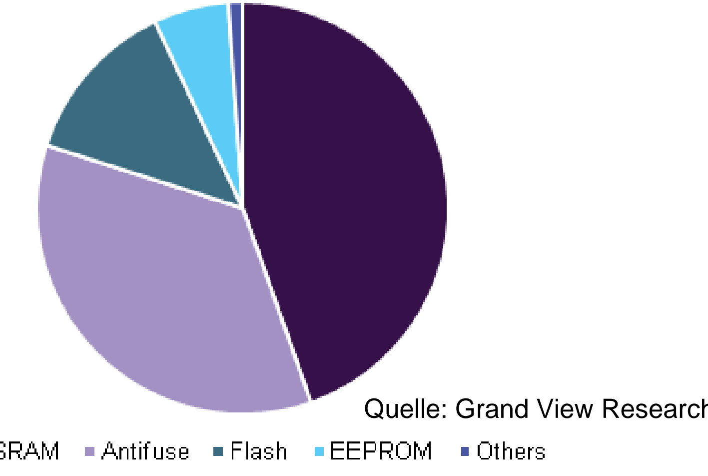
- ein 2^kx1 MUX kann bis zu 2^{2^k} verschiedene Funktionen realisieren
- eine Funktion wird realisiert, indem die Funktionswerte für alle möglichen Eingaben, an die MUX-Eingaben gelegt werden
- die Selektor-Eingänge werden verwendet, um den entsprechenden Eingang an den MUX-Ausgang zu legen
- Funktion mit vielen Eingabevariablen können mittels des Shannonschen Entwicklungssatzes in Funktionen mit weniger Eingabevariablen zerlegt werden und durch mehrere MUX realisiert werden



s1	s0	Y =AND	
0	0	C0	0
0	1	C1	0
1	0	C2	0
1	1	C3	1

Actel ACT3

Marktanteile FPGA Technologien 2015:



Quelle: Grand View Research

Marktanteile FPGA Technologien:

	2015		2016		
Vendor	FPGA Total	Market share	FPGA Total	Market share	Growth CY15-CY16
Xilinx	$2,044	53%	$2,167	53%	6%
Intel (Altera)	$1,389	36%	$1,486	36%	7%
Microsemi	$301	8%	$297	7%	-1%
Lattice	$124	3%	$144	3%	16%
QuickLogic	$19	0%	$11	0%	-40%
Others	$2	0%	$2	0%	0%
TOTAL	$3,879	100%	$4,112	100%	6%

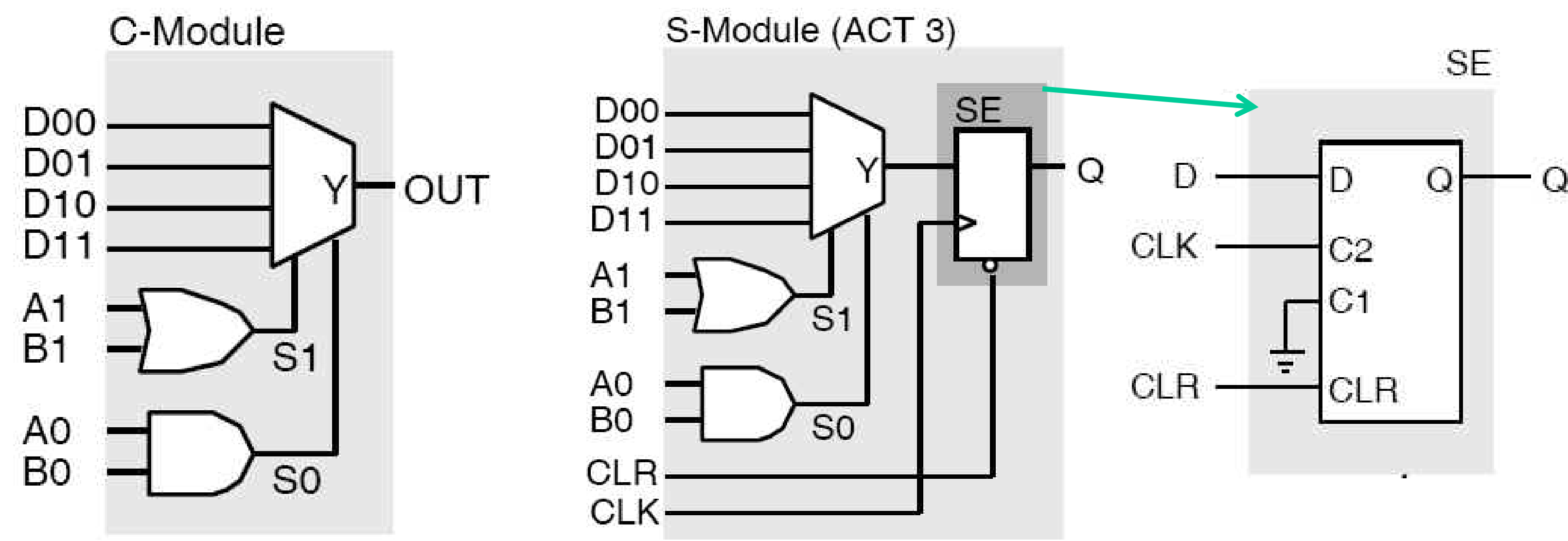
Guesstimate of final numbers x 1,000,000 (Source: Paul Dillien)

Quelle: EE Times

Actel ACT3

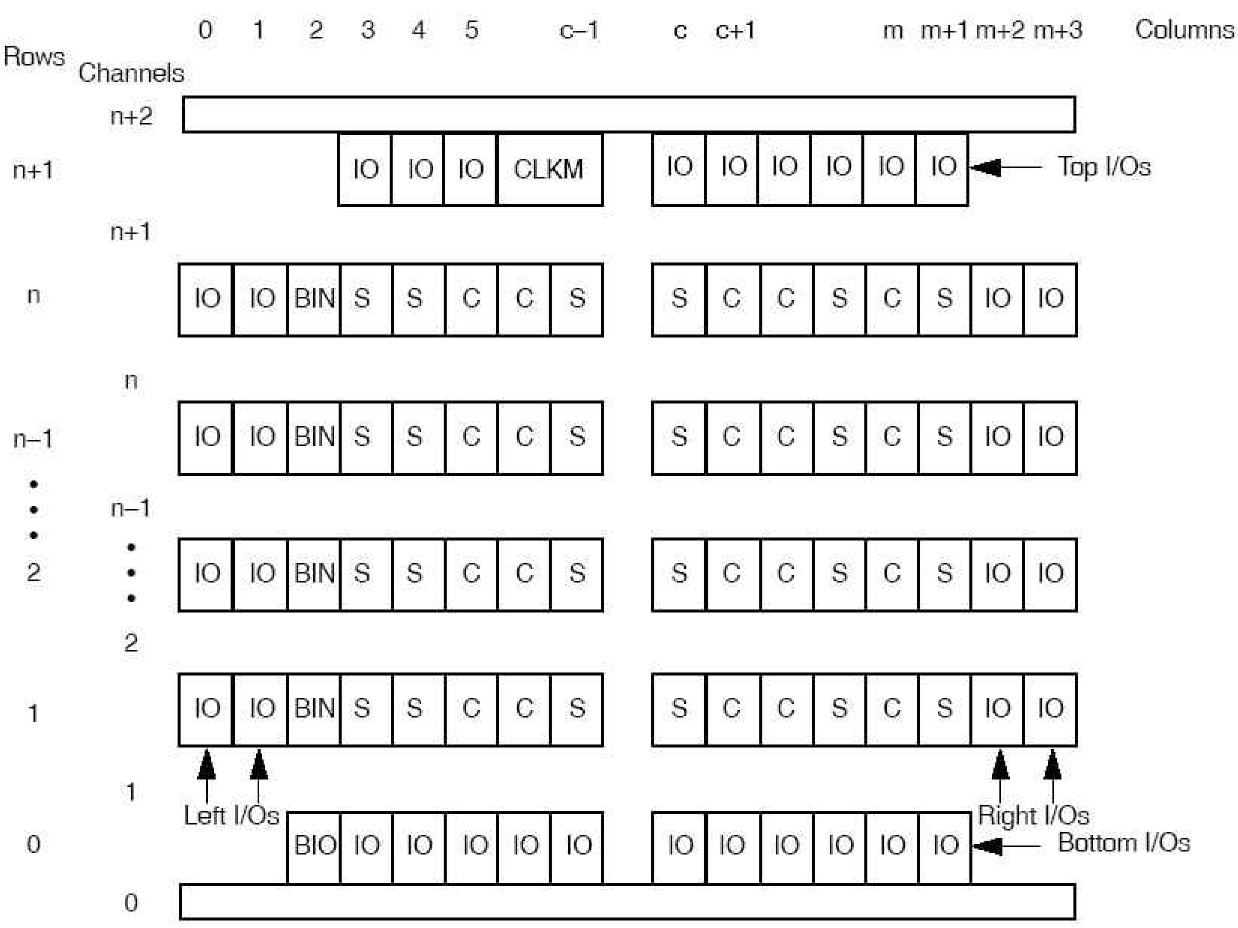
Zeilen-basierte FPGAs: Actel ACT3 (Nachfolger: MX Serie)

- Module in Zeilen zwischen denen Routingkanäle liegen
- Makrozellen (C-Module, S-Module) sind MUX-basiert (>1000 Module Chip)
- C-Module: 4x1 MUX + 1 OR + 1 AND
- S-Module: 4x1 MUX + 1 OR + 1 AND + 1 Flip Flop
- I/O-Blöcke sind am Rand platziert



Actel ACT3

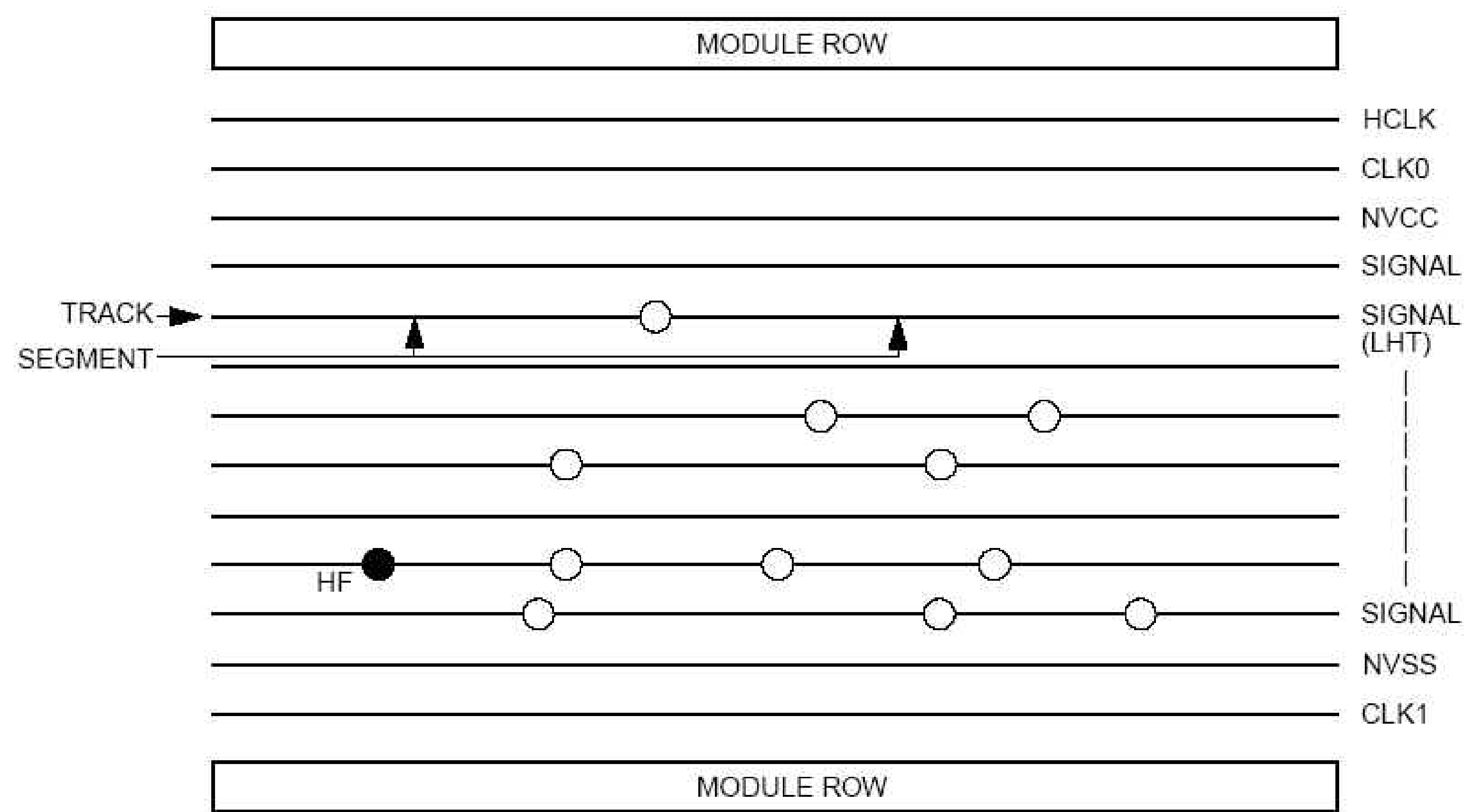
Modulanordnung:



Actel ACT3

Routing-Tracks bestehen aus mehreren Segmenten

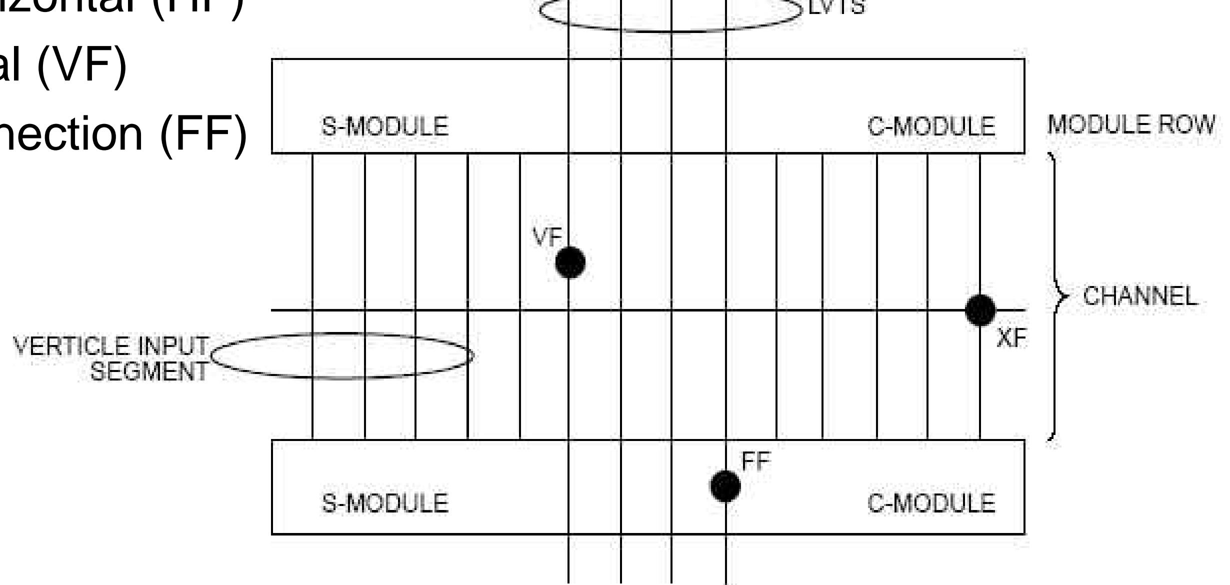
Verbindungen sind anti-fuse basiert



Clock: CLK0 + CLK1 (routbar), HCLK (hardwired clock, speed independent of number of modules)

Actel ACT3

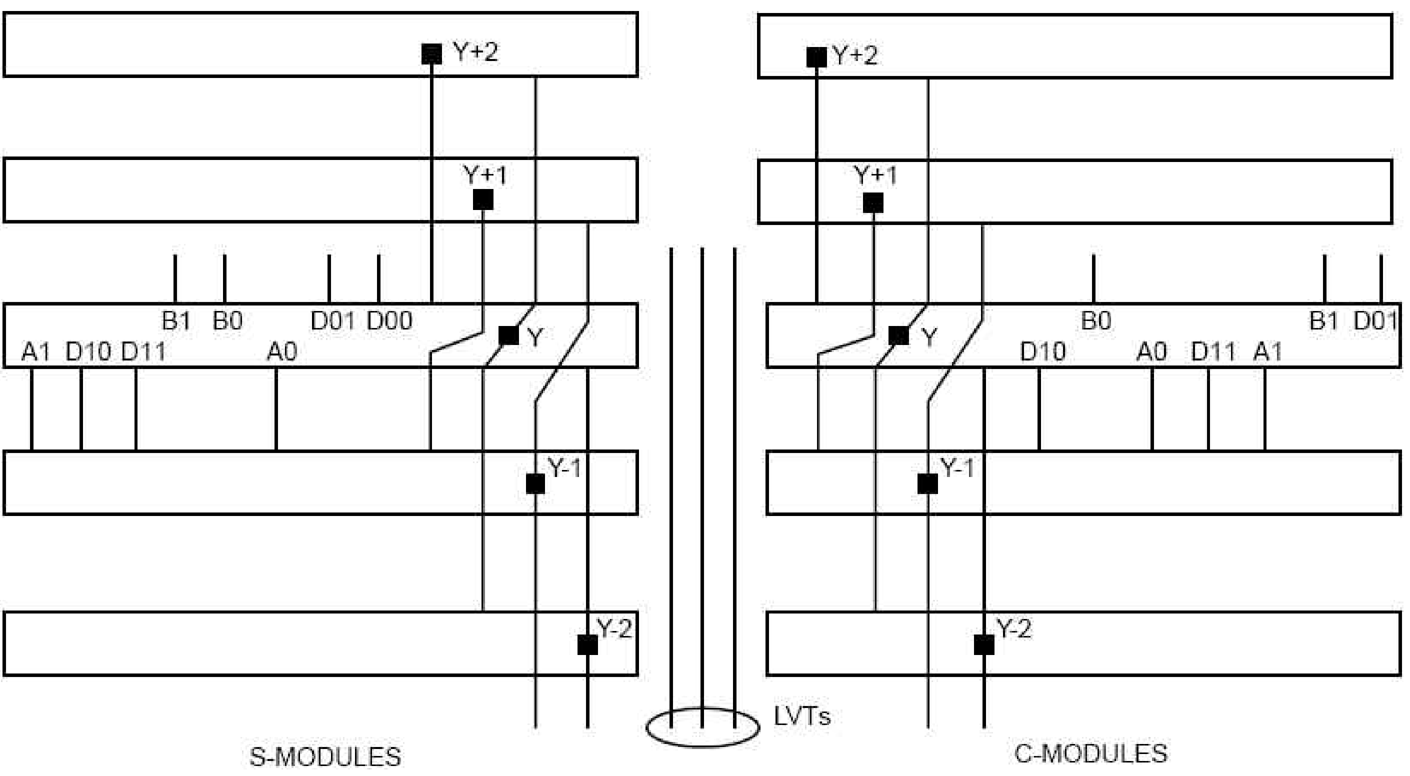
- Anti-fuse Verbindungen
  - ⇨ Horizontal-to-vertical (XF)
  - ⇨ Horizontal-to-horizontal (HF)
  - ⇨ Vertical-to-vertical (VF)
  - ⇨ Fast vertical connection (FF)



- Vertikale Tracks: 1 für jeden Input (Input-track Segmente bis zur Zeile darüber und darunter) und Output (Output-track Segmente bis zu 2 Zeilen darüber und darunter), long (nicht festgelegt) (LVTS)
- Für Testmöglichkeiten erlauben spezielle Pass-Transitoren Leitungen wieder komplett zu verbinden (auch wenn sie durch Programmierung von Anti-fuse getrennt wurden)

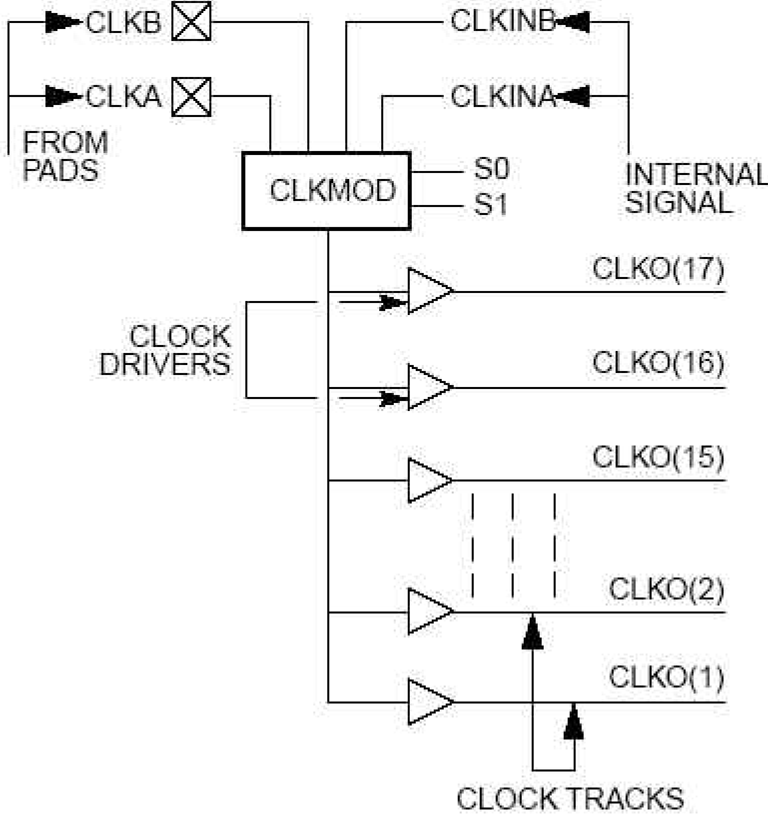
Actel ACT3

- Die Modul-Outputs haben eigene Verbindungskanäle, die bis zwei Zeilen oberhalb und zwei Zeilen unterhalb reichen (außer in den oberen und unteren Zeilen)



Actel ACT3

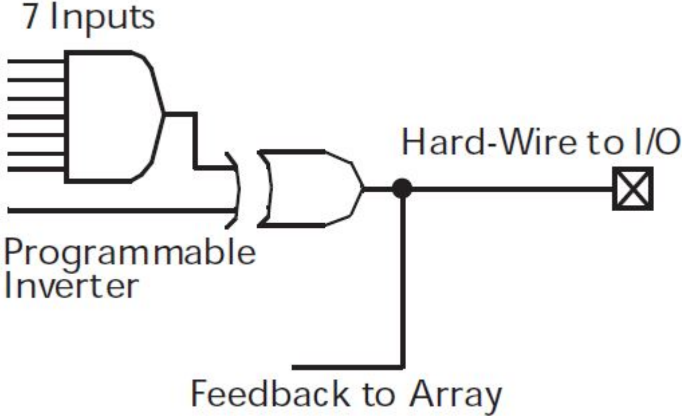
- 2 routable clocks (CLK0, CLK1):
  externe clock (CLKA, CLKB) vom Clock-Input-Buffer, oder interne clock (CLKINA, CLKINB))



Actel MX

- 3 Arten von Logik-Modulen: S-Module, C-Module, D-Module

D-Module (Decode-Module, UND-Funktion) liegen am Rand des Chips



MX-Serie: bis zu 1230 S-Module, 1184 C-Module, 24 D-Module
dazu auch Speichermodule (SRAM-Module) möglich

Xilinx Virtex

Symmetrische Array FPGAs: Xilinx Virtex-7

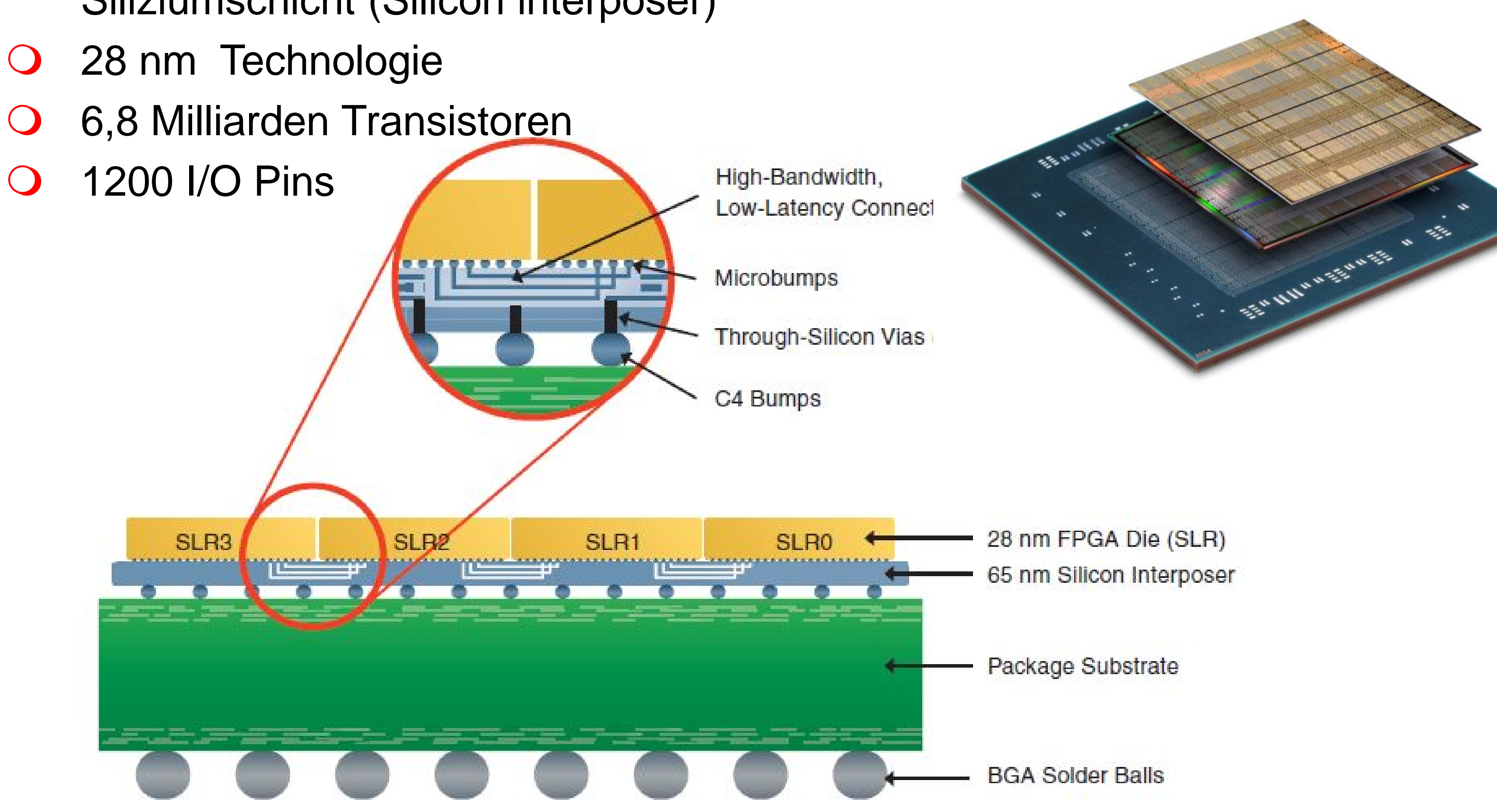
(kein rein symmetrischer Array, eher spaltenbasiert; Vorgänger waren symmetrisch)

- Makrozellen = konfigurierbare logische Blöcke (Configurable Logic Block, CLB) (bis zu 1955K CLBs)
- Spezielle DSP-Slices (bis zu 3600) mit: 25x18 Bit Multiplier (cascadable) und 48 Bit Adder/Subtractor/ Accumulator (z.B. für DSP-Algorithmen)
- Block RAM für internen Gebrauch (bis zu 68 Mb)
- Clock Manager: für Clock mit vom Benutzer spezifizierter Frequenz
- Input/Output Blöcke (IOB) für off-chip Kommunikation
- Anzahl der Rekonfigurationsbits (bis) 450 Mbit
  Übertragung:
  - bitseriell oder wortseriell (bis 32 Bit Wortlänge)
  - werden geladen vom FPGA aus Speicher oder extern gesteuert
- Virtex-7 in 28nm Technologie: UltraScale-Varianten in 20nm und 16 nm Technologie

Xilinx Virtex

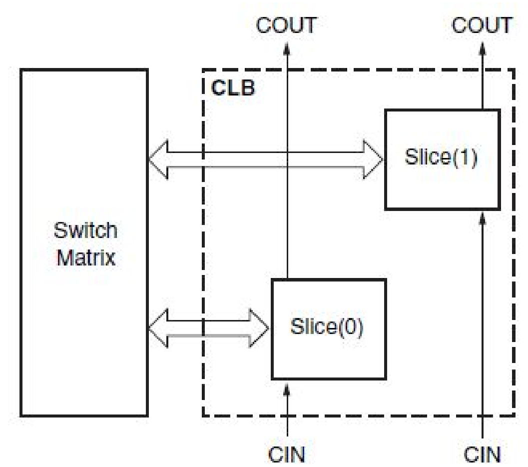
Chipaufbau:

- SSI-Technologie (Stacked Silicon Interconnect): mehrere Dies (super logic regions, SLR) sind verbunden durch mehr als 10000 Verbindungen in extra Siliziumschicht (Silicon interposer)
- 28 nm Technologie
- 6,8 Milliarden Transistoren
- 1200 I/O Pins



Xilinx Virtex

- Ein **CLB** enthält 2 identische Slices auf dem Virtex-7



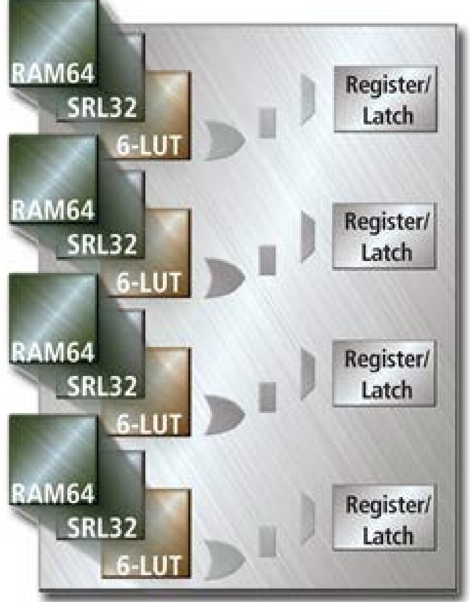
- 1 Slice enthält:
  - ⇨ 4 6-input LUTs

    (jede LUT alternativ als 2 5-input LUTs mit gemeinsamen Inputs nutzbar)

    mind. 25% der LUTs alternativ konfigurierbar als:

    64-Bit distributed SelectRAM oder

    32 Bit Shift Register (oder 2 mit jeweils 16 Bit)
  - ⇨ 8 FF zum Speichern der LUT-Resultate
  - ⇨ MUX um die LUT entweder mit FF oder dem Output zu verbinden

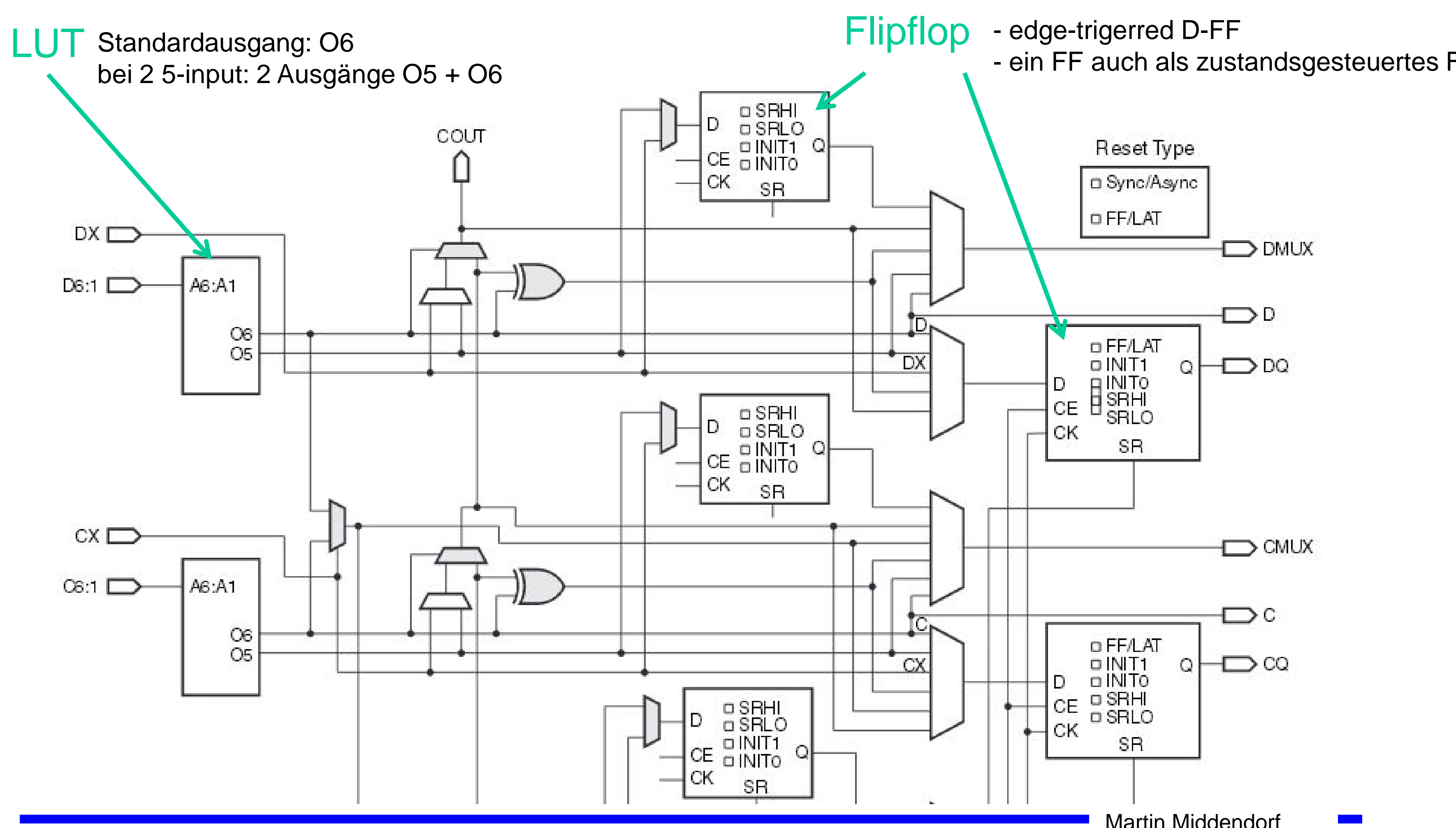


- Carry-in and Carry-out dienen zur Erzeugung schneller Addierer mit den benachbarten CLBs

Xilinx Virtex

Obere Hälfte eines Slices

(SLICEL ohne SelectRAM+ShiftRegister Möglichkeit)

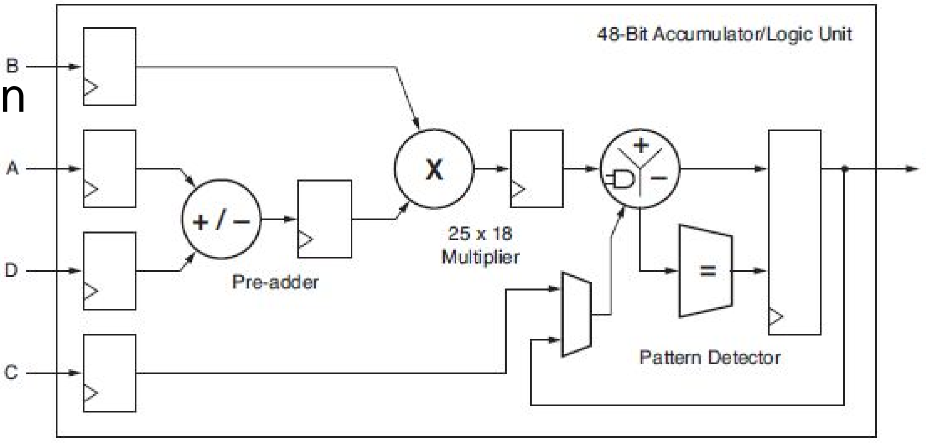


Xilinx Virtex

- **BlockRAM** auf dem Virtex-7:
  - ⇨ insgesamt bis 1880 Blöcke dual-port RAM
  - ⇨ pro Block 36Kb → konfiguriert als 36K x 1, 18K x 2, … 512 x 72
  - ⇨ zusätzliche FIFO Logik
  - ⇨ nicht verwendete Teilblöcke können ausgeschaltet werden (power saving)

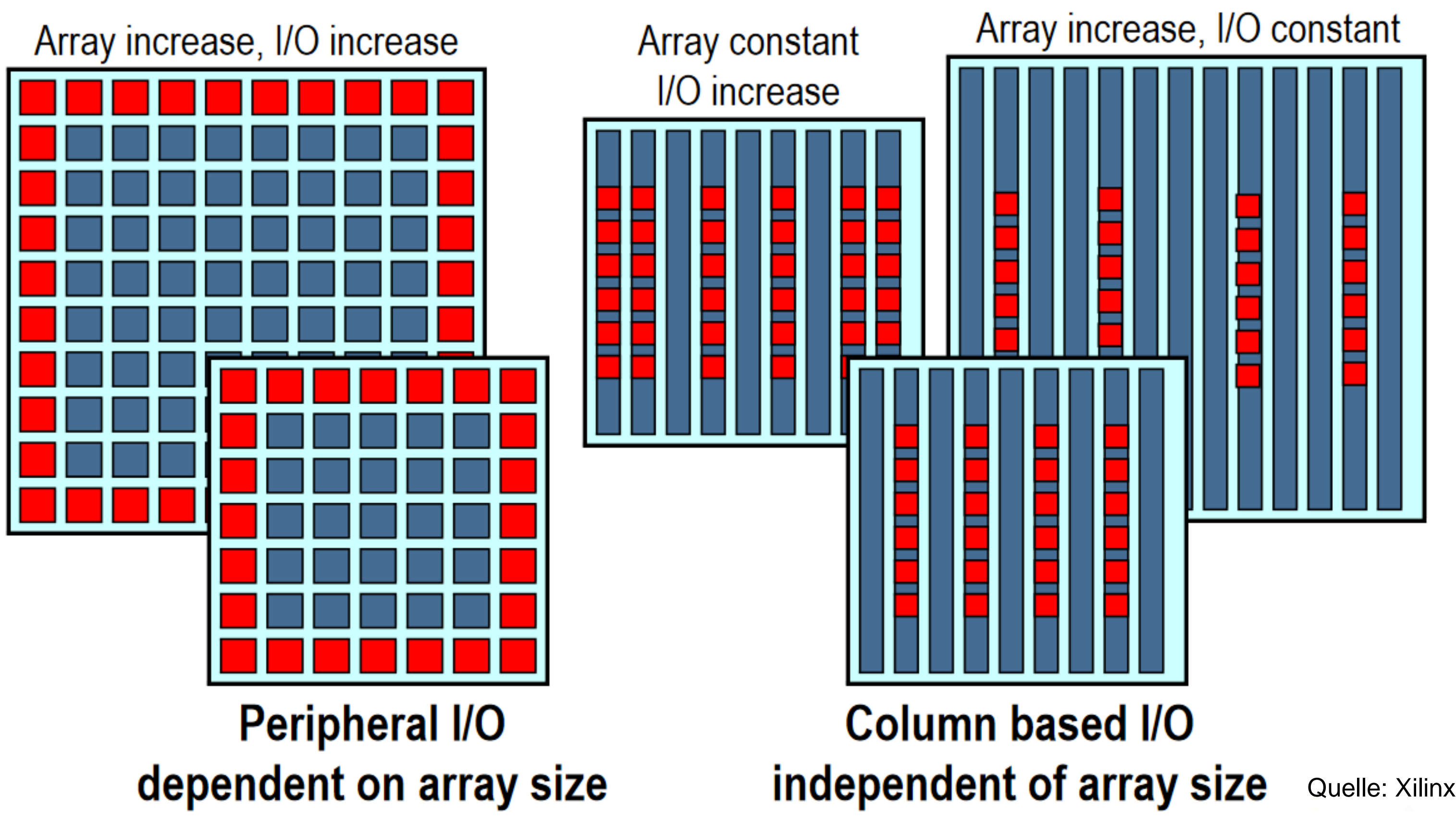
- **DSP slice** auf Virtex-7: 25x18 Bit Multiplier (cascadable), 48 Bit Adder/Subtractor/ Accumulator sowie logische Funktionen, Pattern Detector (z.B. underflow)

  Anwendung: z.B. DSP-Algorithmen



Xilinx Virtex

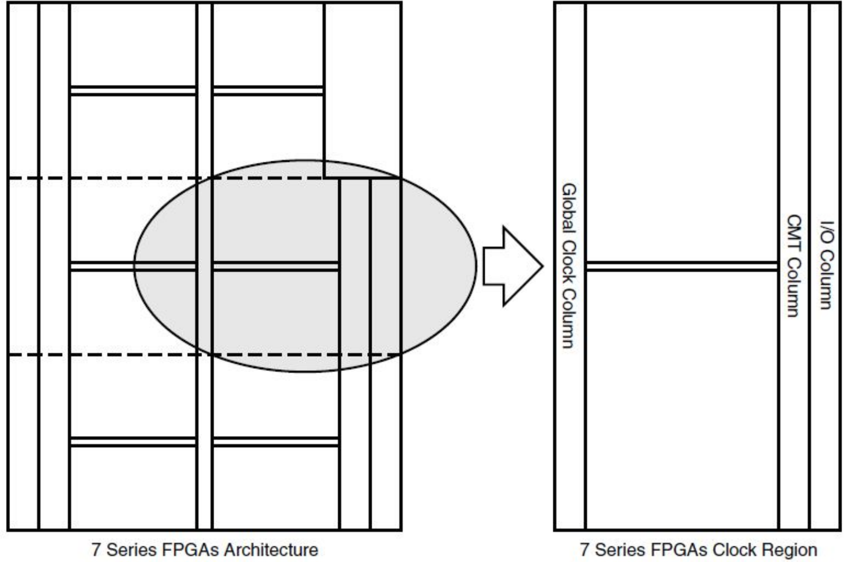
I/O in den Spalten (nicht mehr am Rand):



Quelle: Xilinx

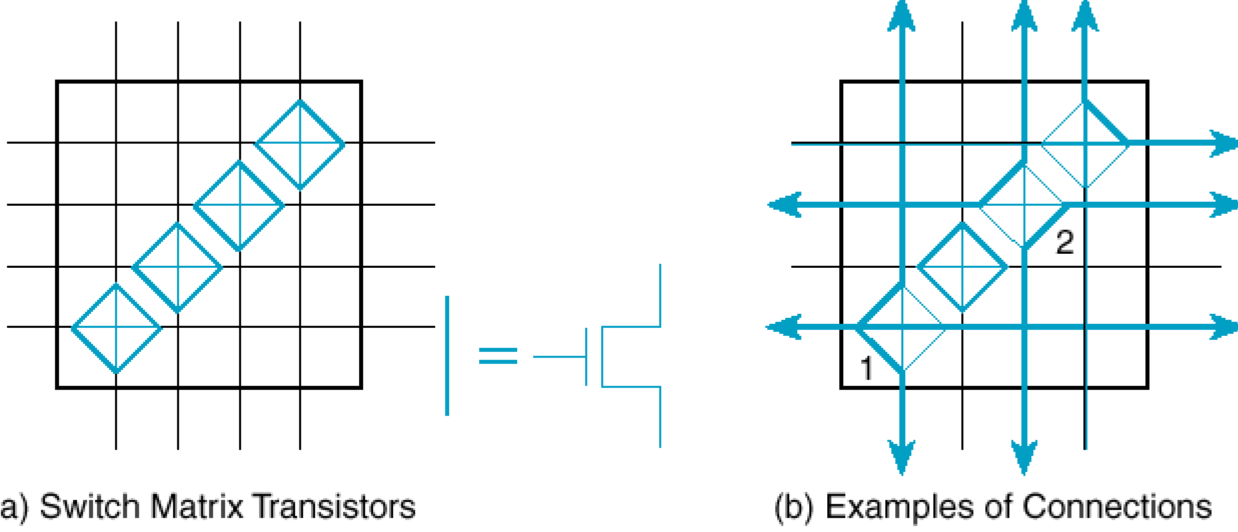
Xilinx Virtex

- **Clock lines**:
  - ⇨ **Global Clock Column** (Backbone):
    - ⇨ verläuft über den Die und teilt ihn in rechte und linke Hälfte
    - ⇨ 32 global clock lines: können alle synchronen Ressourcen auf dem Die takten (CLB, block RAM, DSPs, und I/O)
  - ⇨ **Clock Region** (bis 24): 50 CLBs hoch (50 IOBs) jeweils auf linker und rechter Seite des Die (clock regions können einzeln ausgeschaltet werden – power saving)
  - ⇨ **Clock Management Column** (CMT): erzeugt Clock-Signale, eliminate clock distribution delay, or to adjust delay relative to another clock
  - ⇨ **I/O-Column**: pro Region
    4 Clock-I/O-Pin-Paare



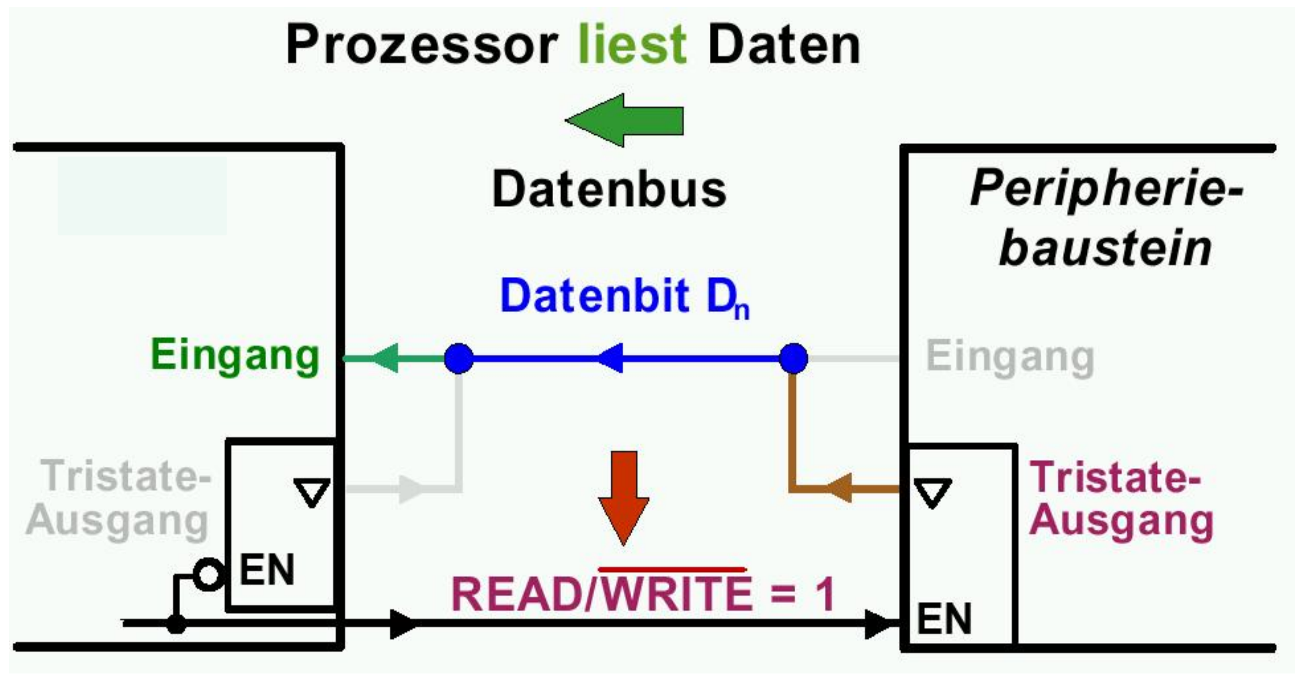
Xilinx Virtex

- Beispiel Switch-Matrix



Einschub: Tri-state Verbindung

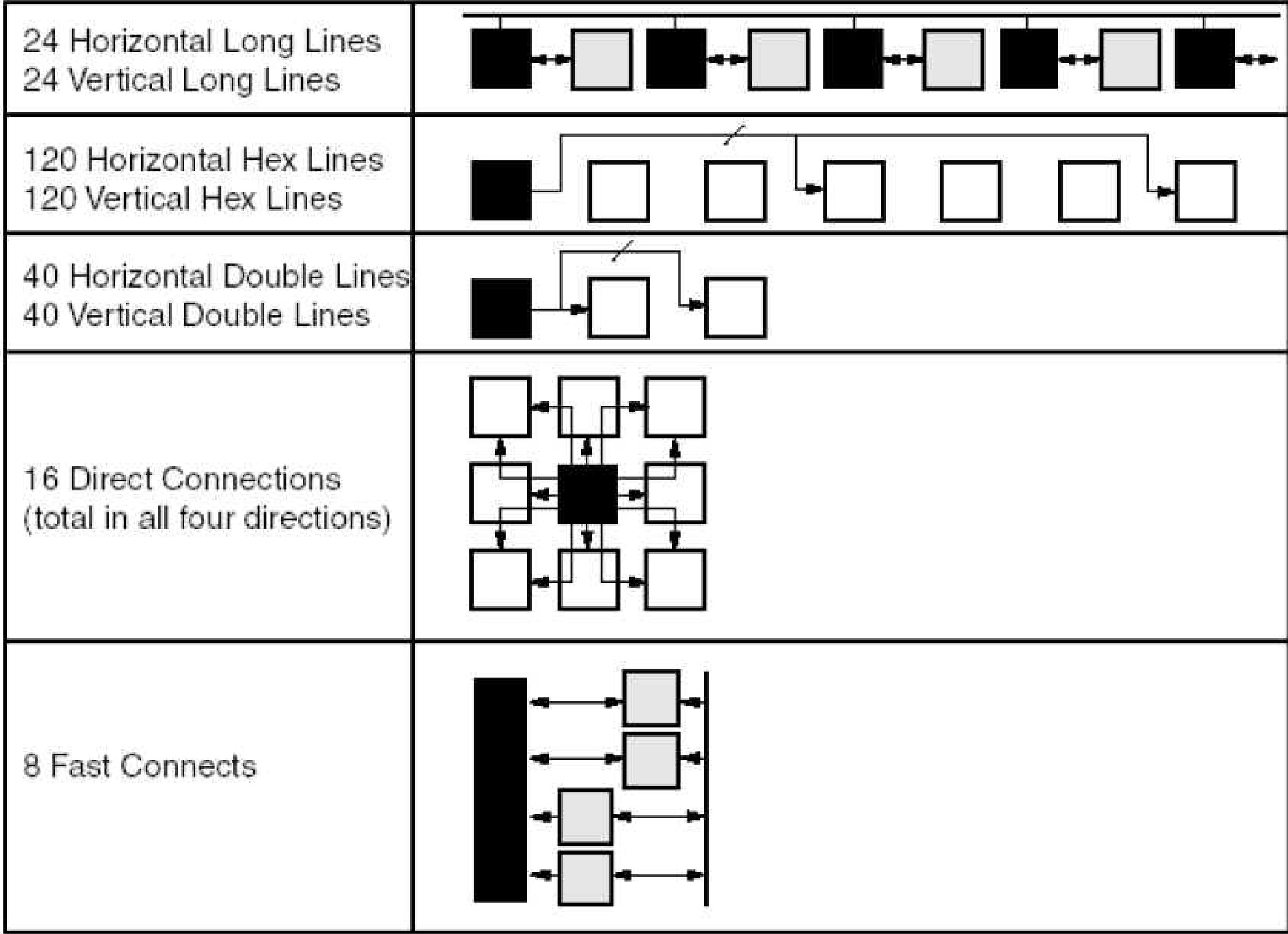
- Tri-state-Ausgänge, die durch Signal gesteuert werden.



- READ/$\overline{\text{WRITE}}$=1
  - ⇨ deaktiviert den Tri-state-Ausgang des Prozessors (legt ihn auf hochohmig), so dass er keinen Einfluss auf den Bus hat
  - ⇨ aktiviert den Tri-state-Ausgang des Speichers, der Low oder High anlegt

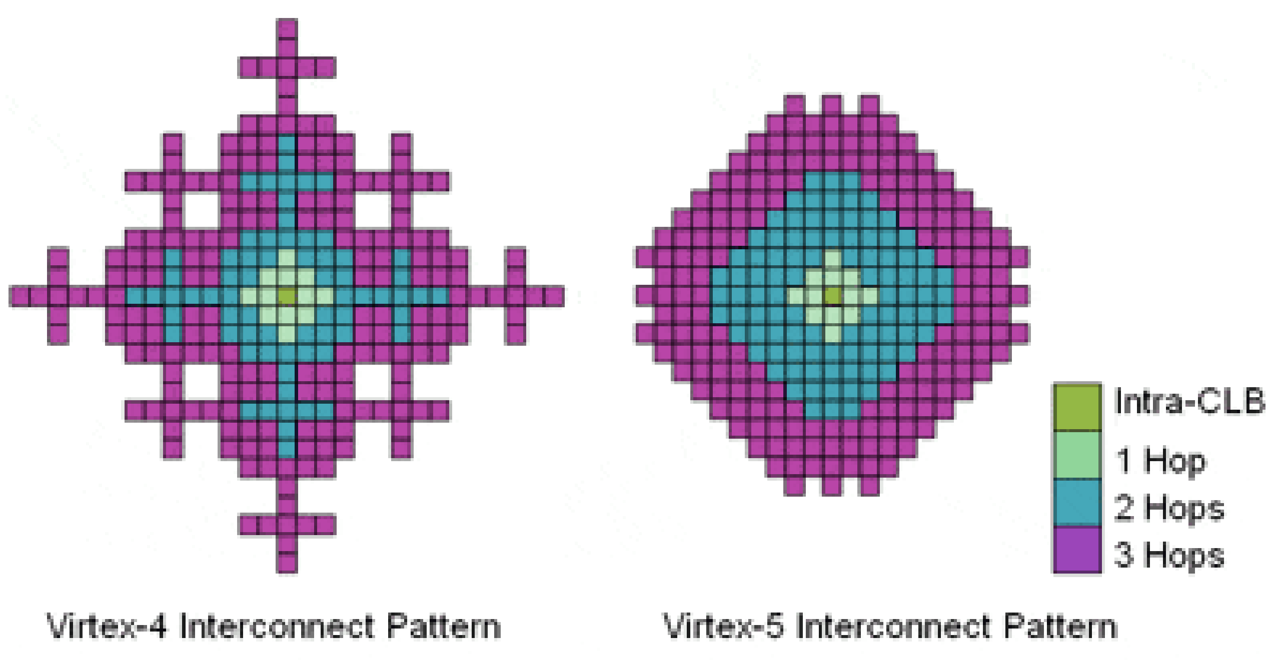
Xilinx Virtex

- Verbindungen (Virtex-II)



Xilinx Virtex

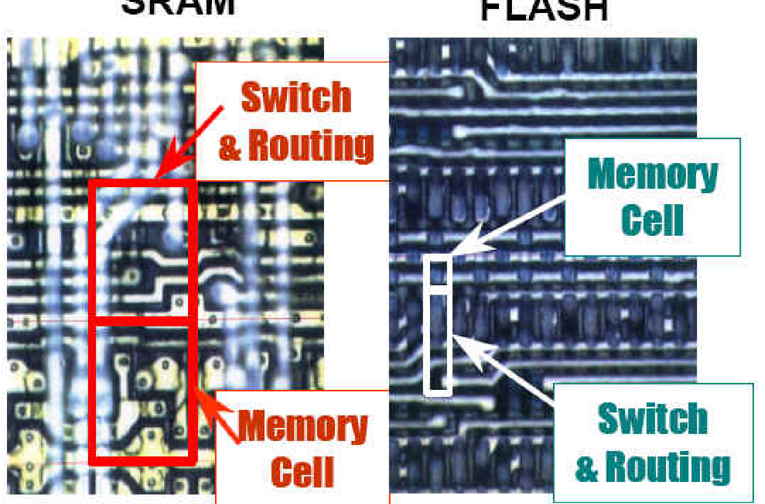
- Vergleich der Entfernung (Anzahl der Hops über Switch Boxen) um von einer CLB zu benachbarten CLBs zu kommen

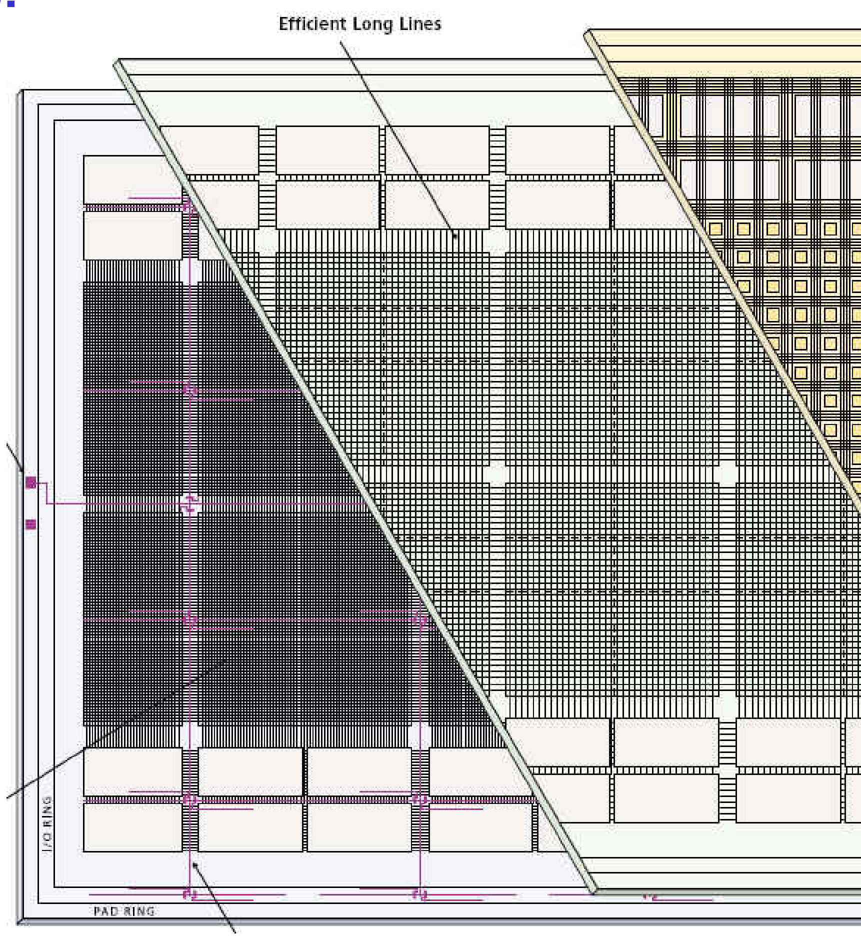


Actel (Mircrosemi) ProAsic

Sea-of-gates, Sea-of-tiles (Actel ProAsic):

- Makrozellen sind EEPROM basierte Kacheln (Tiles, bis zu 75000)
  → non-volatile Flash
- Nachteil: da es vorkommen kann, dass Ladung in die umgebende Oxidschicht geht, nicht beliebig oft rekonfigurierbar (ungefähr 500 mal)

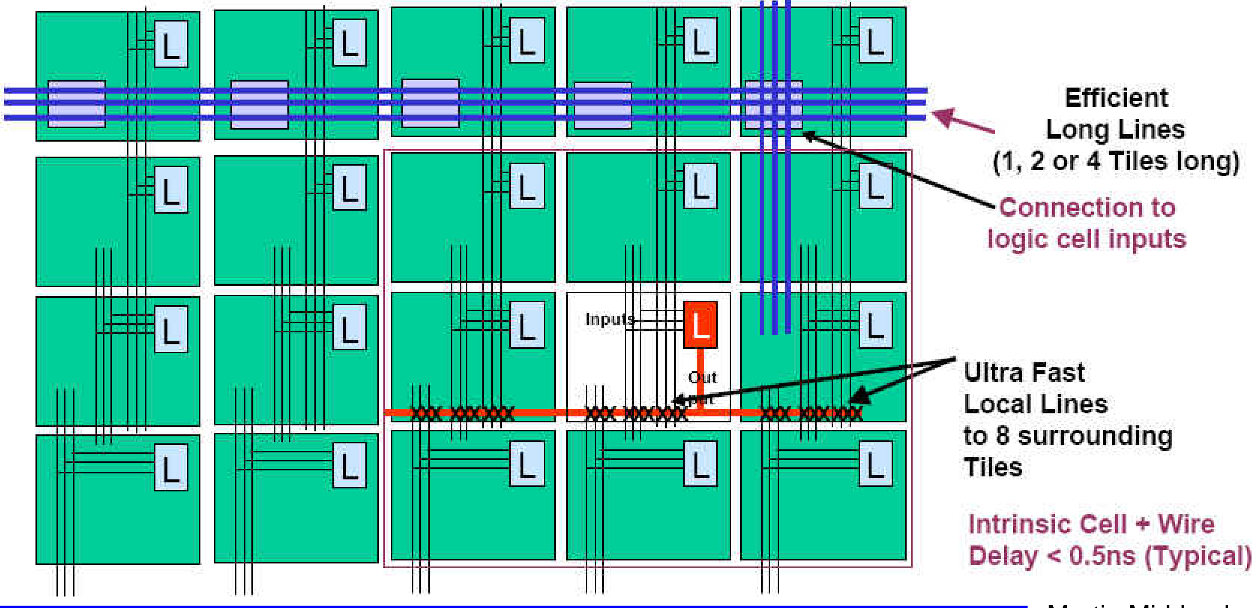




Actel ProAsic

Routing Ressourcen: vierstufige Hierarchie

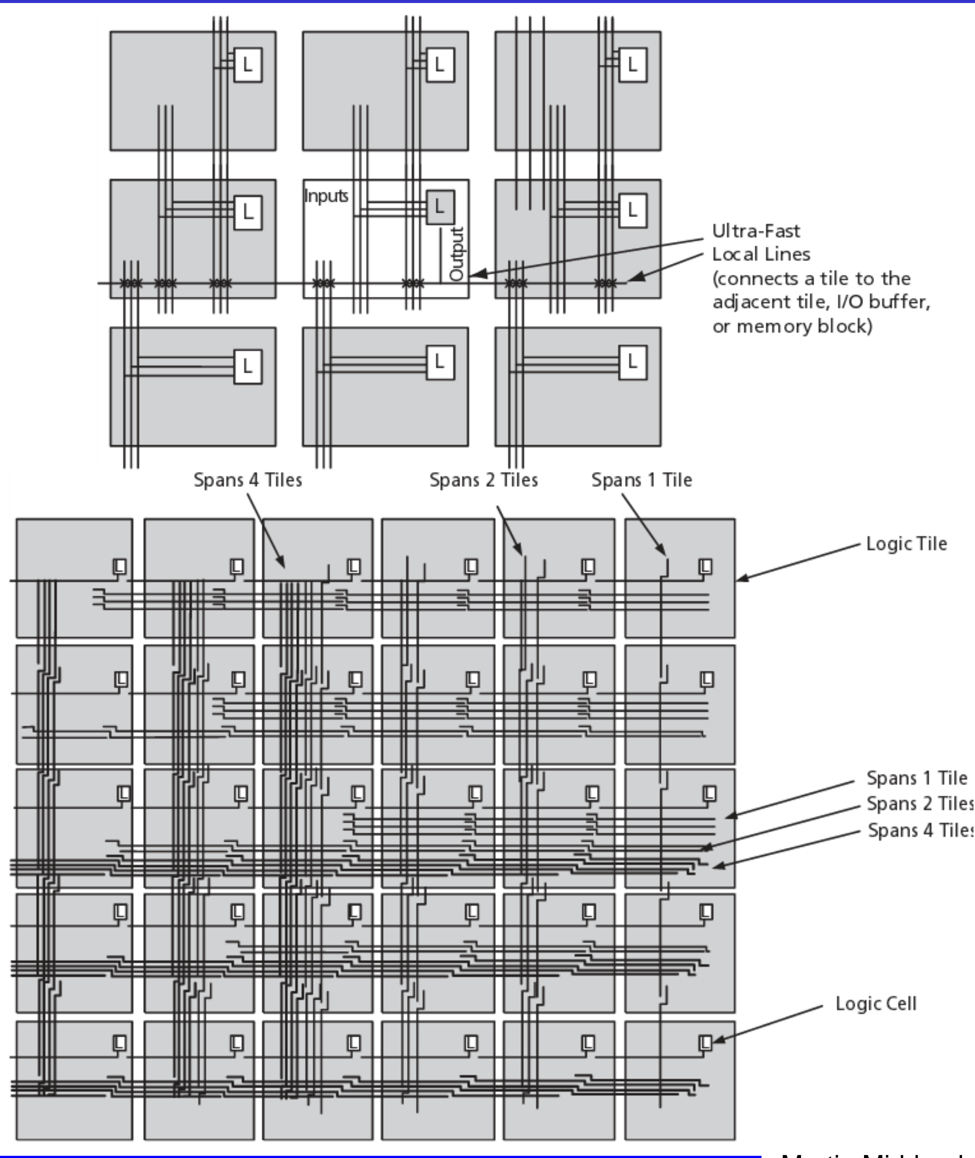
- ⇨ Lokale Leitungen verbinden Output einer Kachel mit Inputs ihrer 8 benachbarten Kacheln
- ⇨ Lange Leitungen (vertikal und horizontal) sind für größere Distanzen (überspannen 1, 2 oder 4 Kacheln) und großen Fan-out
- ⇨ Sehr lange Leitungen überspannen den gesamten Chip
- ⇨ Global Networks ziehen sich über den gesamten Chip und dienen für Signale wie Clock und Reset



Actel ProAsic

Lokale Leitungen

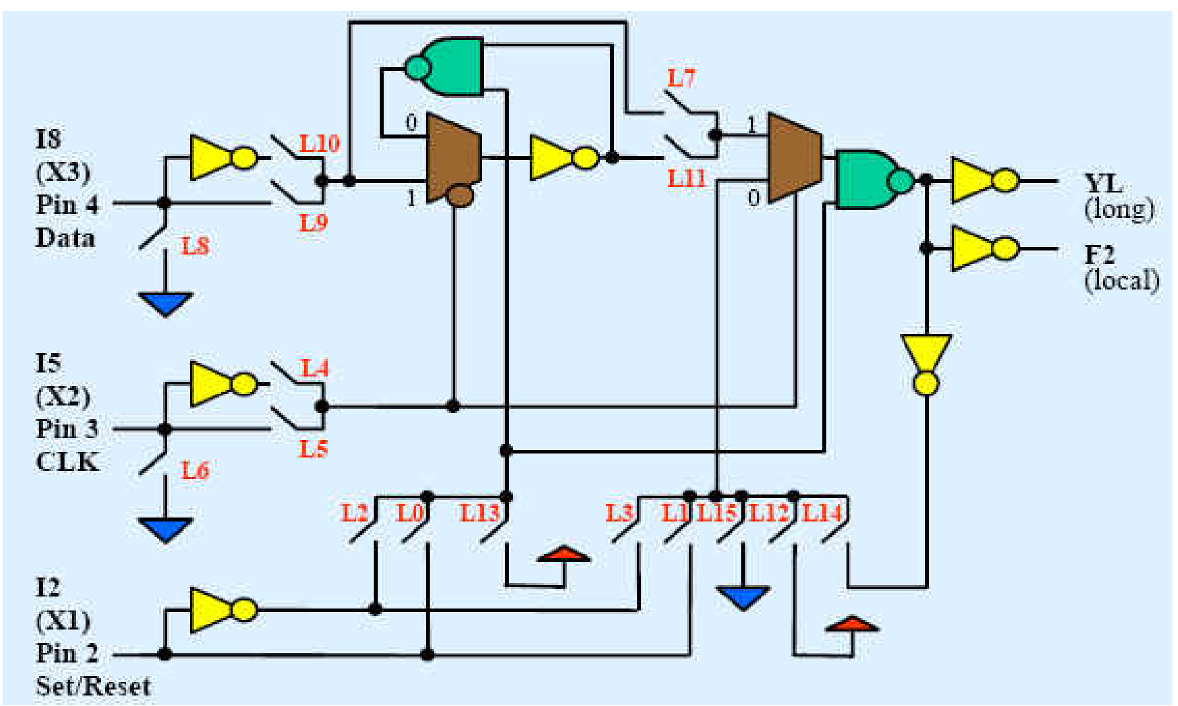
Lange Leitungen



Actel ProAsic

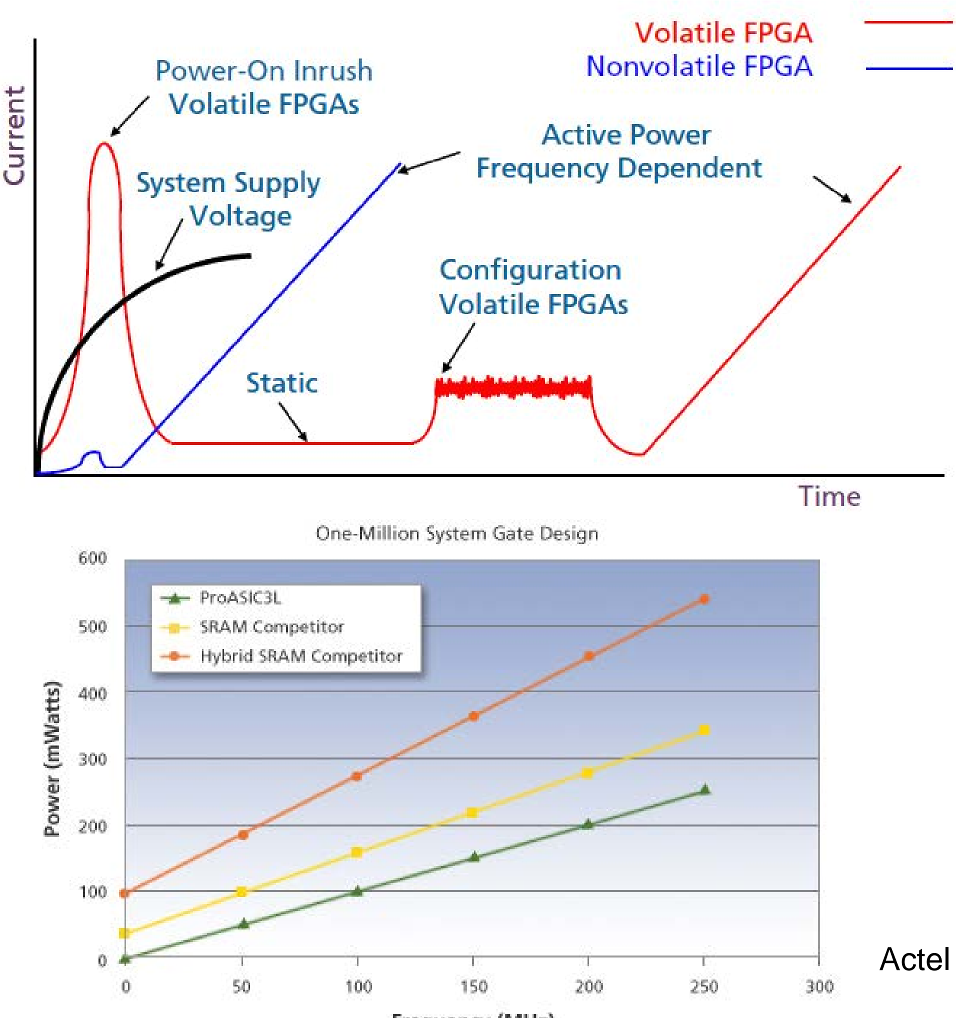
Logische Zelle

- 3 Inputs, 1 Output
- kann alle 3-stelligen Booleschen Funktionen realisieren (außer XOR)
- kann als D-Flipflop mit CLR und SET konfiguriert werden
- basiert auf Multiplexern



Actel ProAsic

Non-volatile Flash FPGAs sind geeignet für low-power Anwendungen:



Actel

Intel Stratix (vormals Altera)

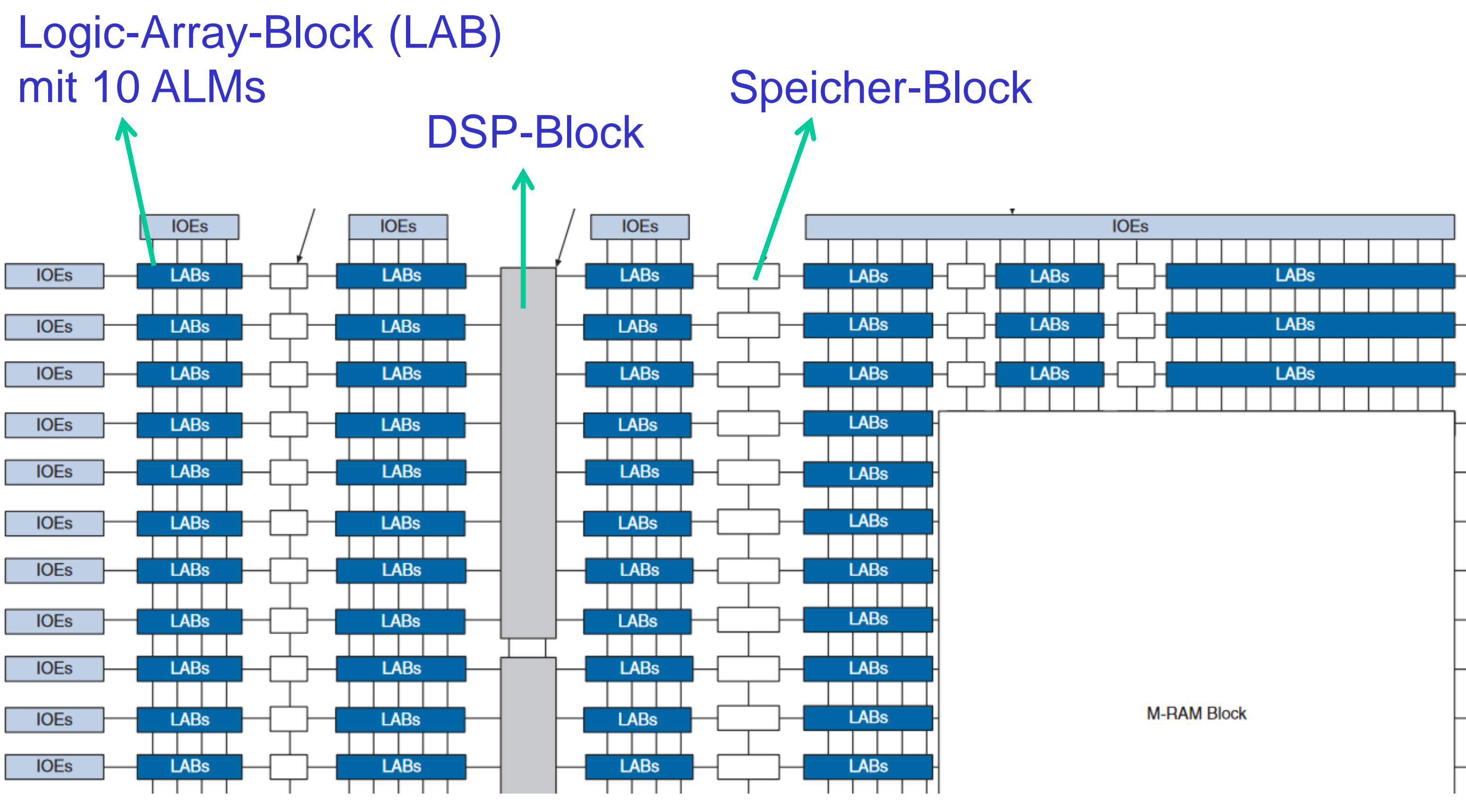
Stratix 10 GX:

Adaptive logic modules (ALMs)	1867680
- ALM Register	7470720
Hyper-Register	Millionen
M20K Speicherblöcke	7033
Variable-precision DSP-Blöcke	1980
18 x 19 Multiplizierer	3960

- Prozessor: Quad-core 64 bit ARM Cortex-A53 MPCore

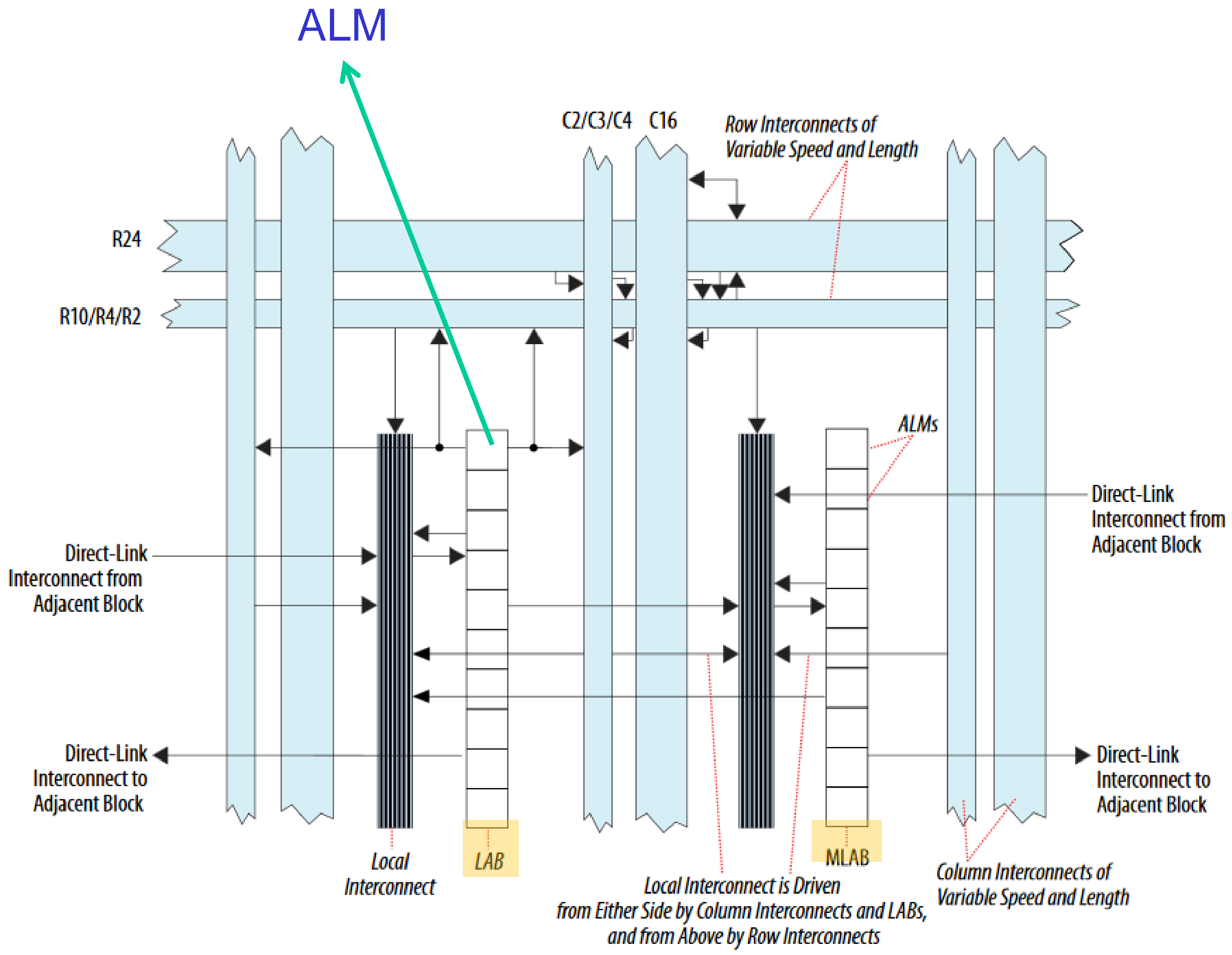
Intel Stratix

Aufbau:



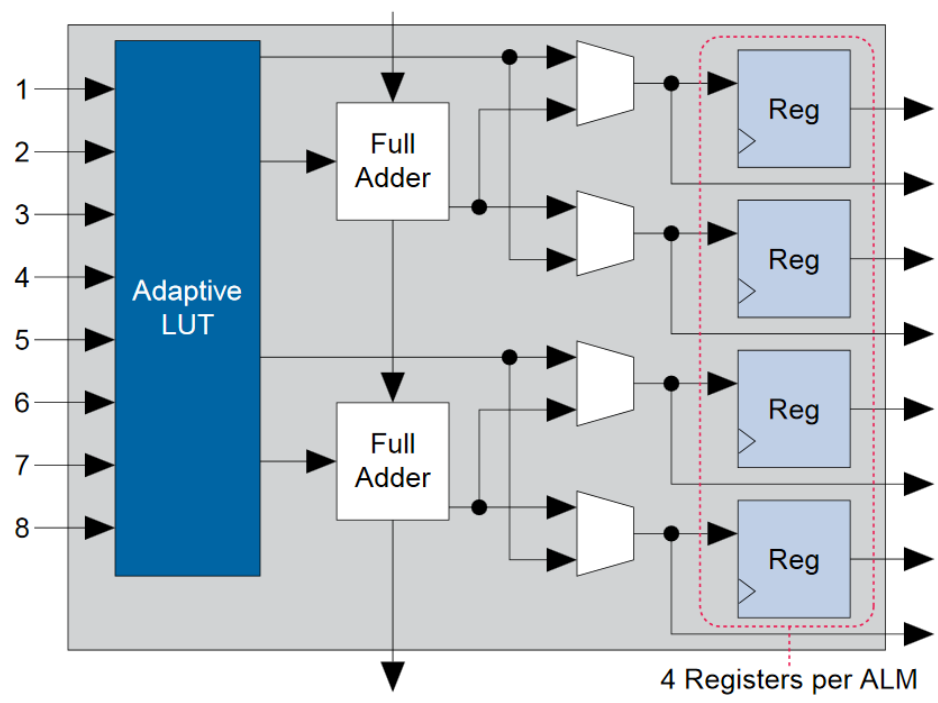
Intel Stratix

Verbindungen:



Intel Stratix

ALM:



Adaptive LUT:

- manche 7-Input Funktionen
- alle 6-Input Funktionen
- zwei 4-Input Funktionen (oder 3-Input Funktion + 5-Input Funktion)

Intel Stratix

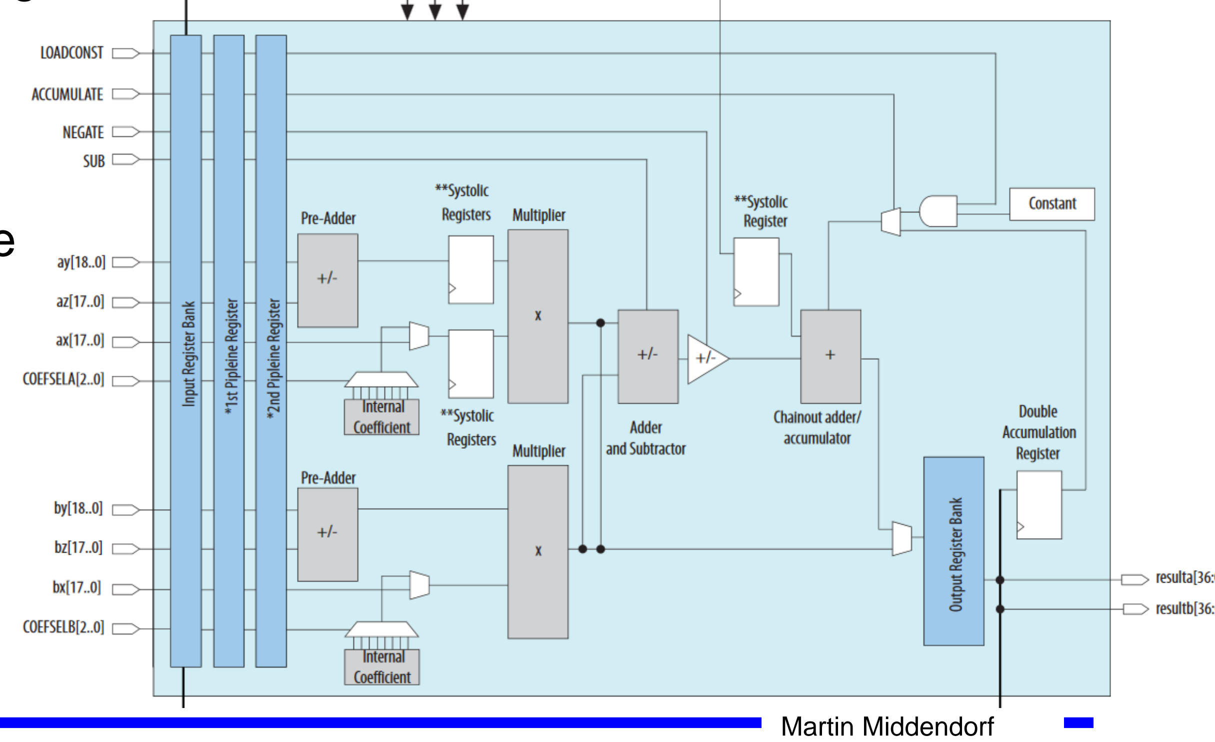
Variable-precision DSP-Block:

Festpunktarithmetik:

- 18-bit and 27-bit Wortlänge
- zwei 18 x 19 Multiplizierer oder ein 27 x 27 Multiplizierer
- Built-in addition, subtraction, and 64-bit double accumulation register
- Internal coefficient register bank in both 18-bit and 27-bit modes for filter implementation

Beispiel:

18-Bit Precision Mode



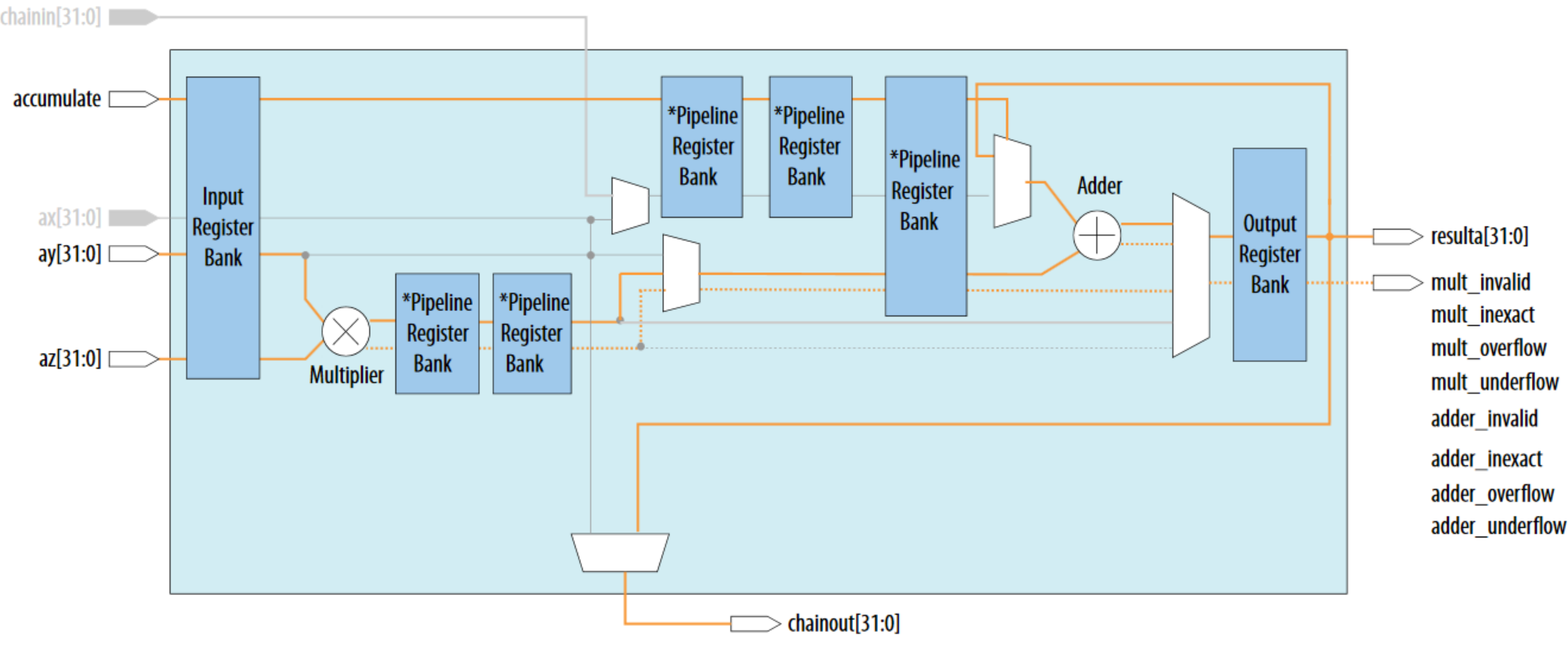
Intel Stratix

Variable-precision DSP-Block:

Fließpunktarithmetik:

- Multiplication, addition, subtraction, multiply-add, and multiply-subtract
- Complex multiplication (4 Blöcke)

Beispiel: Multiply Accumulate Mode



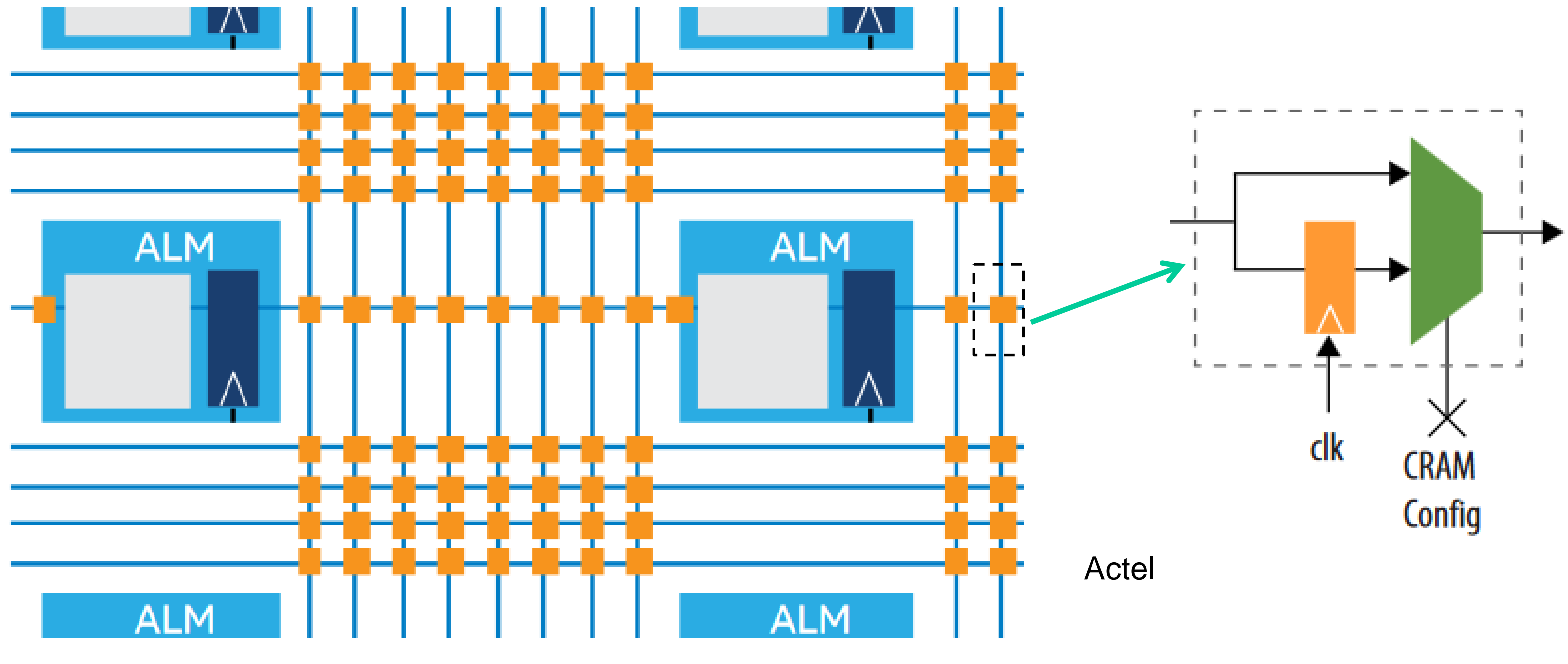
Intel Stratix

Hyperflex-Architecture:

- Prinzip ist "Registers Everywhere„: zusätzliche durch Rekonfiguration umgehbare Register (bypassable Hyper-Register) um lange kritische Pfade und Delays zu vermeiden

  → durch Retiming – s. Vorlesung Par. Alg. -und Pipelining

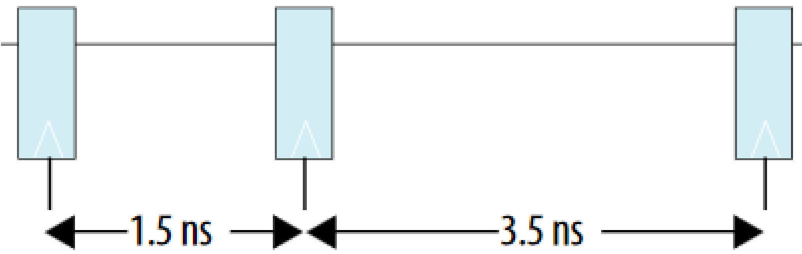
- Hyper-Register in jedem Routing-Segment und auf jedem Input eines funktionellen Blocks (ALM, M20K Speicherblöcke, DSP Blöcke, I/O cells)



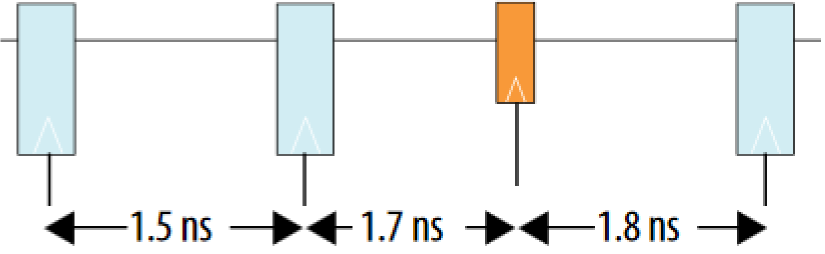
Intel Stratix

Retiming mit Hyper-Register: Beispiel

- bisher: 3 Taktzyklen mit 286 MH (wegen 3.5 ns Delay)→ 10,9 ns



- nach Retiming: 4 Taktzyklen mit 555 MH (wegen 1.8 ns Delay) → 7,3 ns



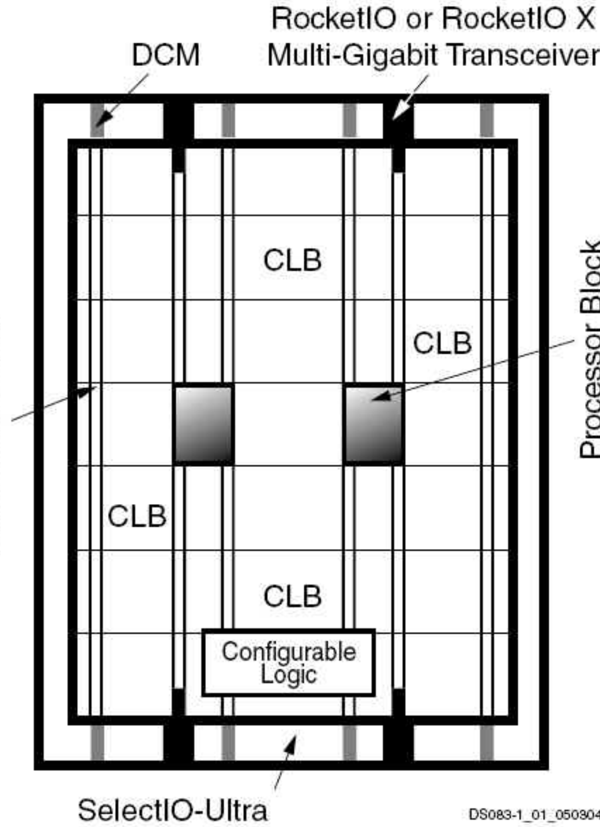
Actel

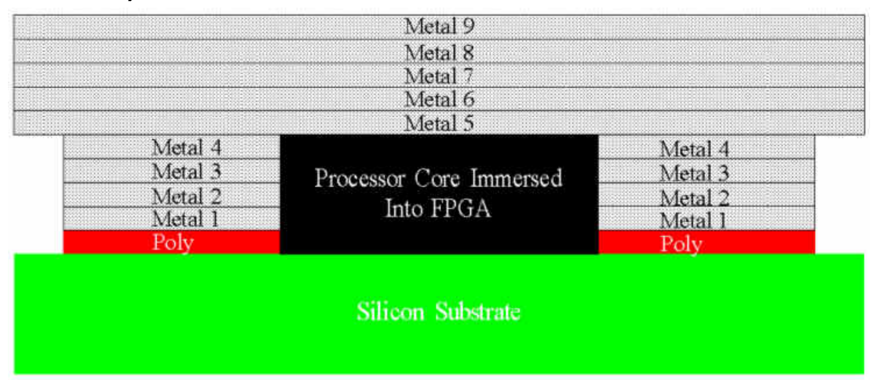
Hybride FPGAs

Xilinx VirtexII-Pro

´

- Grundlegende Struktur wie VirtexII
- Zusätzlich:
  - Bis zu 4 eingebettete IBM Power PC 405 RISC Prozessoren mit 400 MHz
- 18bit x 18bit eingebettete Multiplizierer
- Dual-ported RAM
- Eingebettete High-speed-serial Kommunikation durch RocketIO Multi-gigabit Transceivers (bis 3Gb/sec)

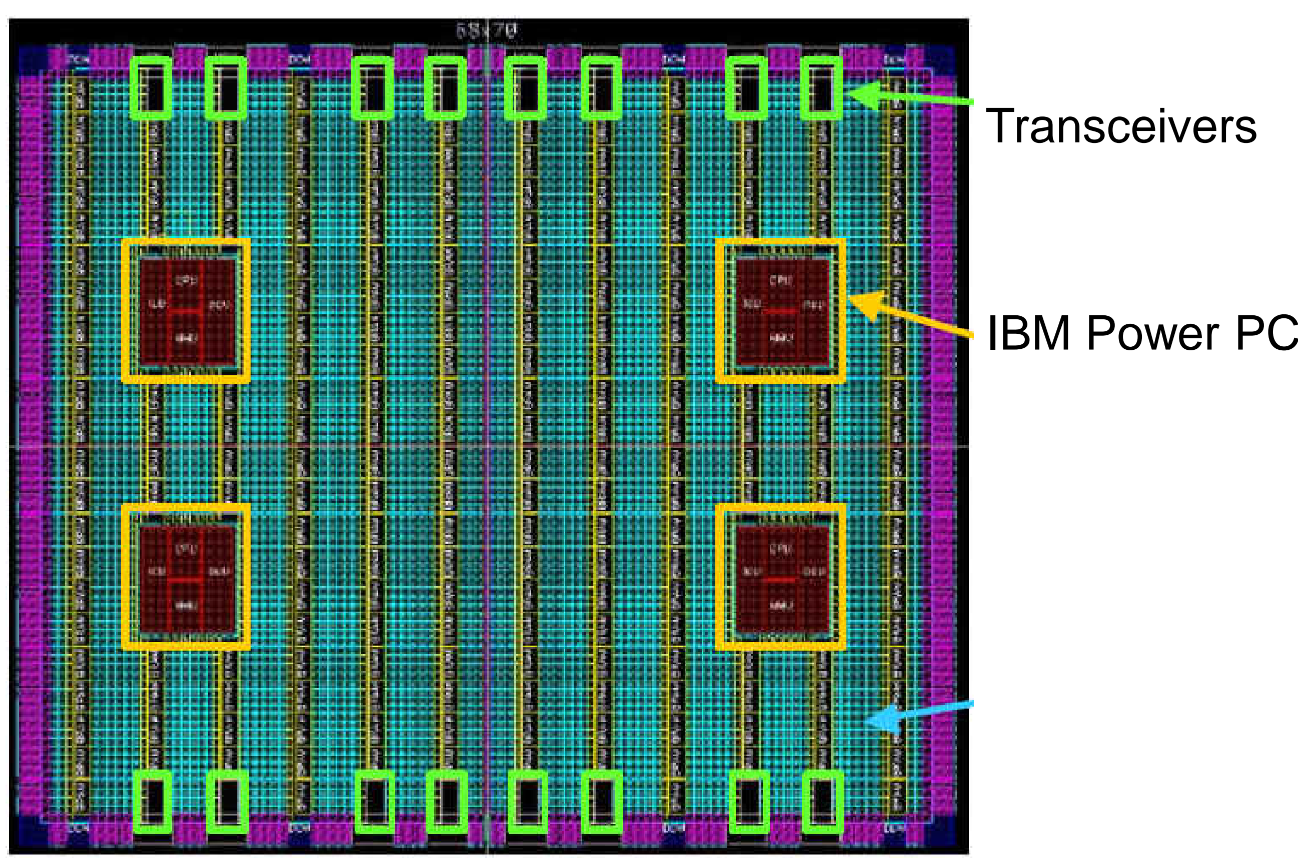




Hybride FPGAs

Xilinx VirtexII-Pro

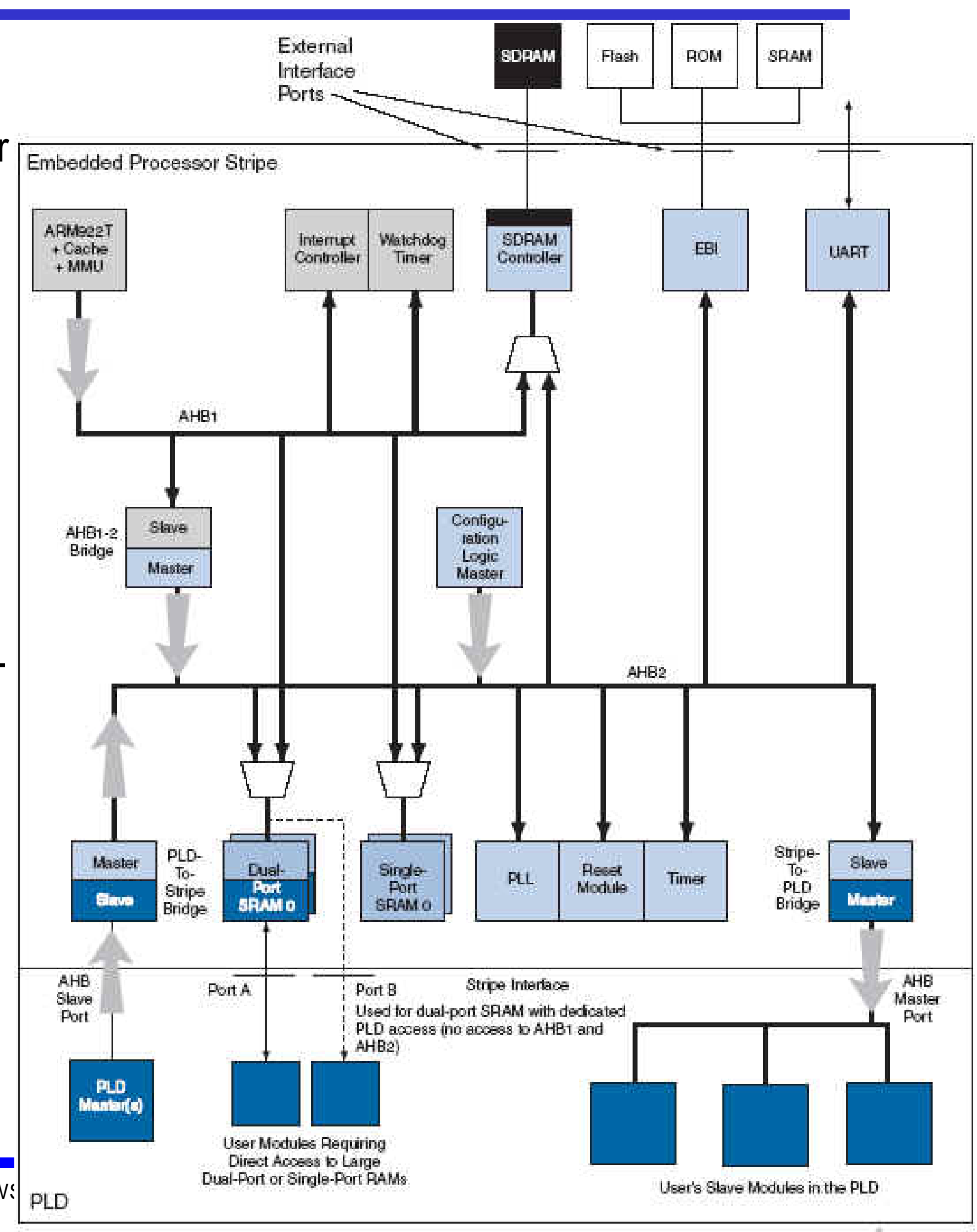
´



Hybride FPGAs

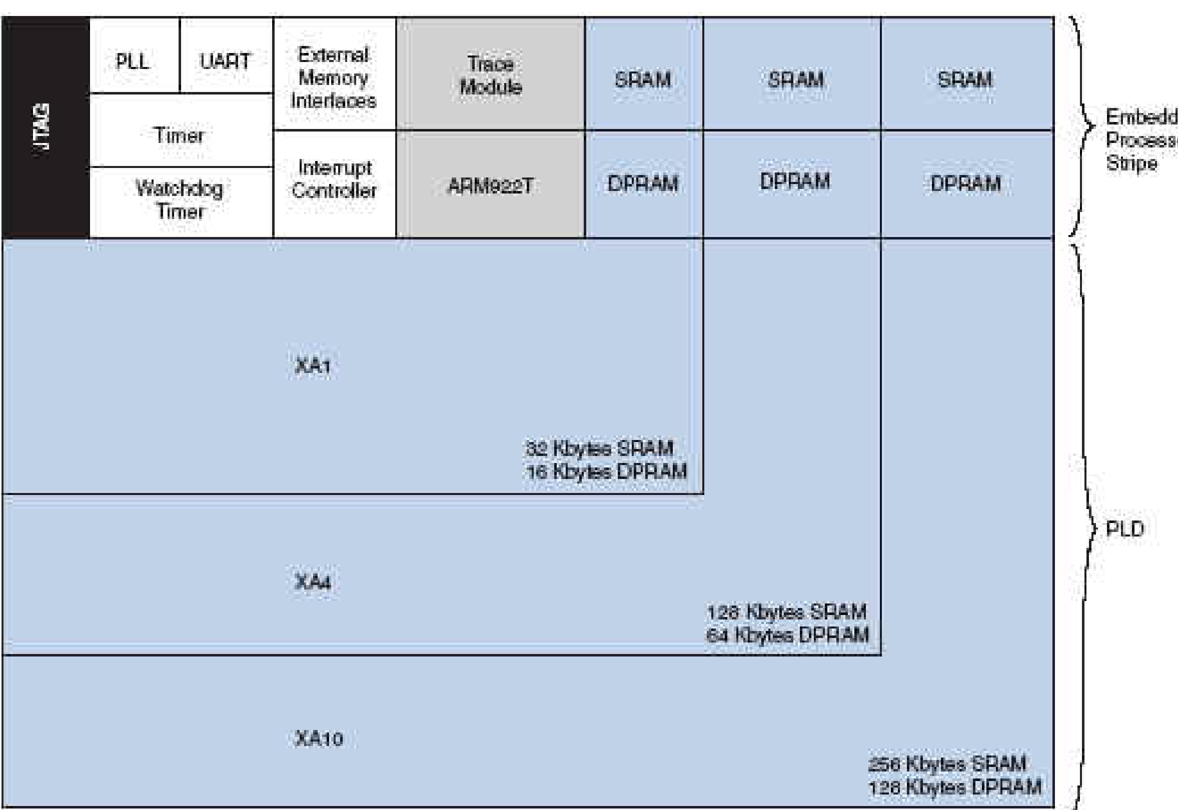
Altera Excalibur:

- Ein ARM922T 32-Bit RISC-Prozessor mit 200 Mhz kann
  - ⇨ unabhängig von PLD Betriebssystem fahren
  - ⇨ PLD konfigurieren
  - ⇨ extern kommunizieren
  - ⇨ System-Restart/-Reboot ausführen
- Interne Single Port SRAM (internal code space, data memory) und Dual-ported SRAM (Zugriff von Prozessor und PLD) und SDRAM Controller
- PLD ist ähnlich Altera Flex
- UART: Kommunikationsbaustein, Umsetzung parallele/serielle Daten
- Expansion Bus Interface (EBI): bidirektionales externes Speicherinterface



Hybride FPGAs

- enthält mehrere programmierbare Clock-domains
- UART: Receiver/Transmitter
- JTAG: Debug-Support für den Prozessor
- Watchdog Timer: kontrolliert, ob die Software funktioniert, indem er regelmäßig zurückgesetzt wird



Spezialisierungsgrad und Granularität

LUTs sind allgemeine Funktionsgeneratoren: jede n-stellige Boolesche Funktion kann mittels einer n-Input LUT realisiert werden

Problem: Da die Größe einer LUT exponentiell in der Anzahl der Eingaben wächst, lassen sich nur LUTs mit wenigen Eingängen realisieren

- ⇨ Größere Funktionen müssen dann durch Nutzung mehrerer LUTs zusammengeschaltet werden
  - Routing oft aufwendig
  - lange Wege durch viele Routing-Matrizen führen zu Delays
- ⇨ Für spezielle Funktionen ist eine direkte Realisierung in Hardware meist deutlich schneller und verbraucht weniger Platz

Idee: Verbessere die Effizienz von FPGAs, indem spezielle oft genutzte Funktionen durch spezialisierte und effiziente Funktionseinheiten realisiert werden (Addiere, Multiplizierer, Integrierer,…)

Beispiel: Multiplizierer im Xilinx Virtex

Grobgranulare Rekonfigurierbarkeit

FPGAs mit grobgranularen Einheiten verwenden

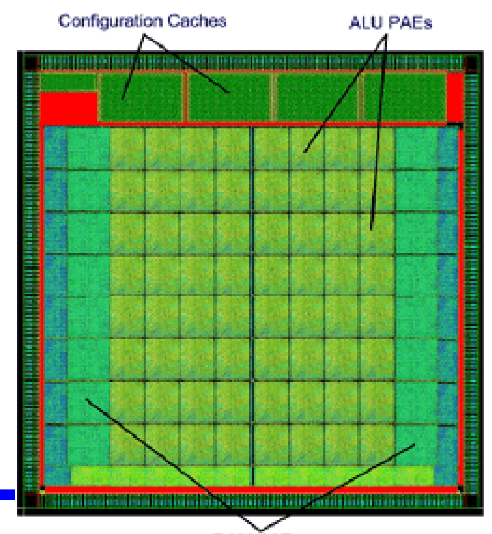
- Spezielle Funktionseinheiten (Addierer, Multiplizierer, Integrierer, etc...), welche
  - ⇨ effizient und direkt verdrahtet implementiert sind (spart Routing-Ressourcen für andere Einheiten)
- Dazu kommt typischerweise ein Array programmierbarer meist identischer Prozessorelemente (PE)
  - ⇨ die wenige und einfache Operationen ausführen können (Addition, …)
    - z.B. 8-bit, 16-bit oder 32-bit ALU (die pro Konfiguration nur eine Operation ausführen kann
  - ⇨ Kommunikation meist über Busse (Packet-Verbindung) oder direkte Verbindung über programmierbare Routing-Matrizen
- Speicherblöcke sind nahe den PEs vorhanden
- Beispielarchitekturen: PACT XPP, Quicksilver ACM, NEC DRP, IPflex DAP/DNA

PACT XPP

PACT XPP (Extreme Processing Plattform)

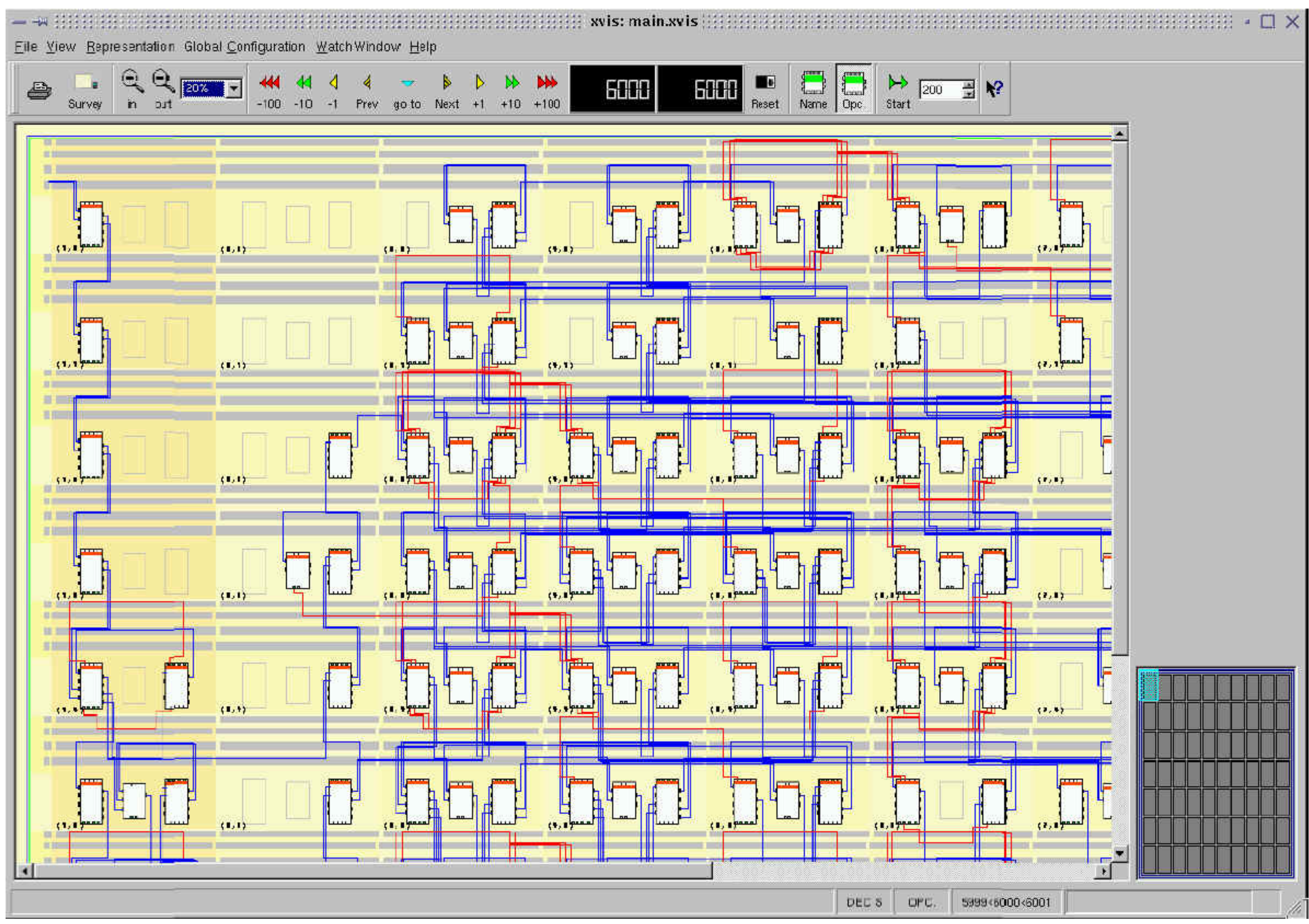
- Hierarchisch aufgebaut aus:
- Array of Processing Array Elements (PAE) die zu Clustern - sogenannten Processing Arrays (PA) - zusammengeschlossen sind
  - ⇨ ALU-PAEs und RAM-PAEs
- PAC = Processing Array Cluster (PA) + Configuration-Manager (CM)

- Hierarchischer Konfigurations Baum:
  - ⇨ Lokale CMs managen die Konfiguration auf PA-Ebene und greifen dabei auf den lokalen Konfigurationsspeicher zu
  - ⇨ Der Supervisor CM (SCM) greift dagegen auf den externen Speicher zu und steuert den gesamten Konfigurationsprozess auf dem Chip



PACT XPP

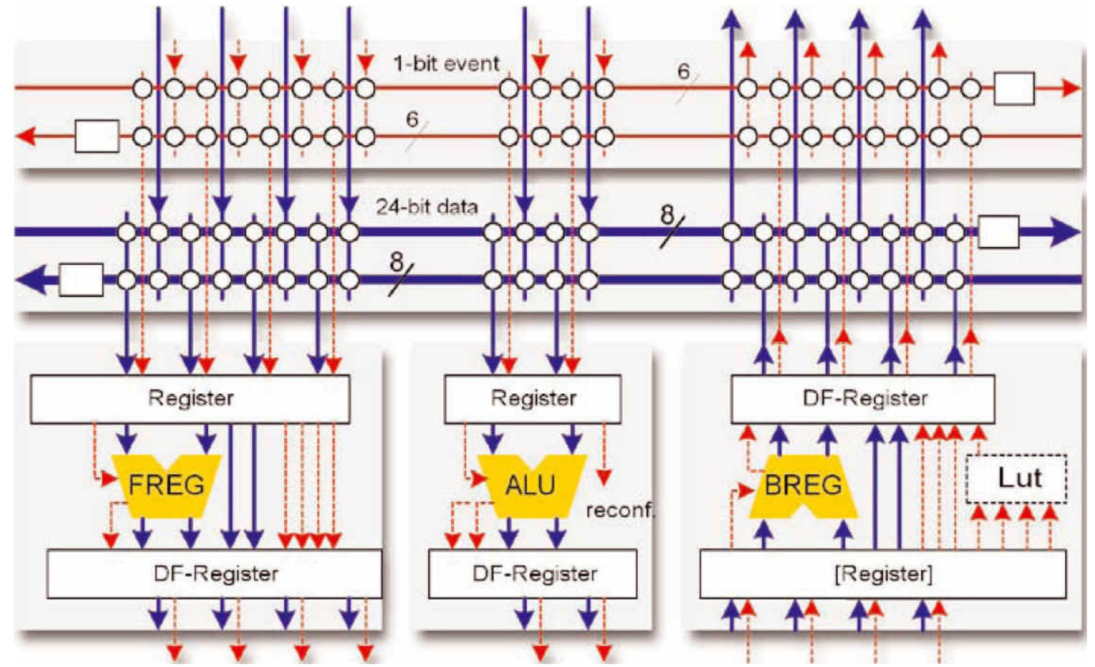
Beispiel: (Development-Tool)



PACT XPP

ALU-PAEs (64, 8x8 Array):

- Bestehen aus mehreren Objekten:
  - ⇨ ALU-Objekt führt arithmetische, logische und Shift-Operationen aus
  - ⇨ Back-Registerobjekt (BREG) stellt vertikalen bottom-to-top Routingkanal für Daten und Ereignisse, zusätzliche arithmetische Funktionen und einen Barrel-shifter zur Verfügung
  - ⇨ Forward-Registerobjekt (FREG) enthält vertikalen top-to-bottom Routingkanal für Daten und Ereignisse und dient zur Steuerung des Datenflusses mit Ereignissen
- Objekte sind mit horizontalen Routingkanälen verbunden (Switch-Objekte dienen der direkten Verbindung zur Nachbar PAE
- Alle Objekte unterstützen XPP's Packet-Transfer-Mechanismus



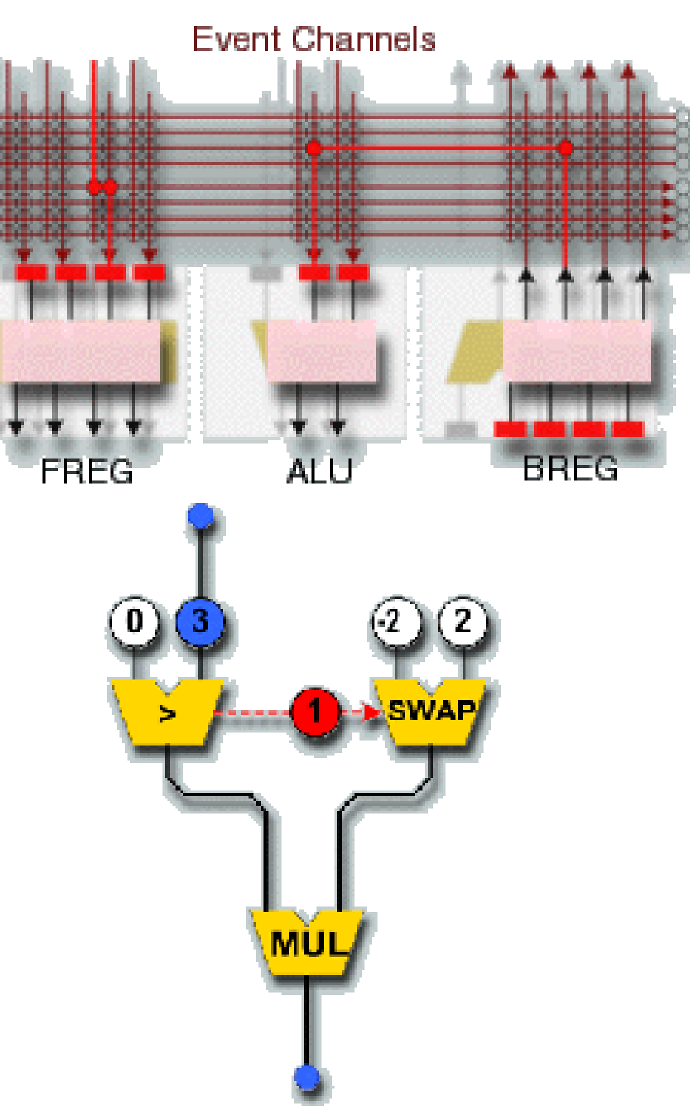
PACT XPP

Ereignisse (Events):

- Ereignisse unterstützen Entscheidungen, die auf dem Ergebnis von Berechnungen beruhen
- Ereignisse werden als Pakete behandelt, die nur ein Bit enthalten
- Ereignisse stammen aus ALU Operationen (z.B. Vergleich) oder von externen Ports
- Ereignisse können als ALU-Input dienen (z.B. Carry oder Shift-Input) oder als Enable/Disable für ALU-Operationen
- Ereignisse können den Datenfluss ändern oder Pakete löschen
- Mehrere Ereignisse können kombiniert werden in LUTs (im BREG)

Beispiel: If value of packet < 0 then multiply by 2, else multiply by -2

→Resultat des Komparators ist Event-Paket, welches 2 or -2 selektiert (Swap Opcode in FREG), die dann mit dem Input-Stream multipliziert wird.



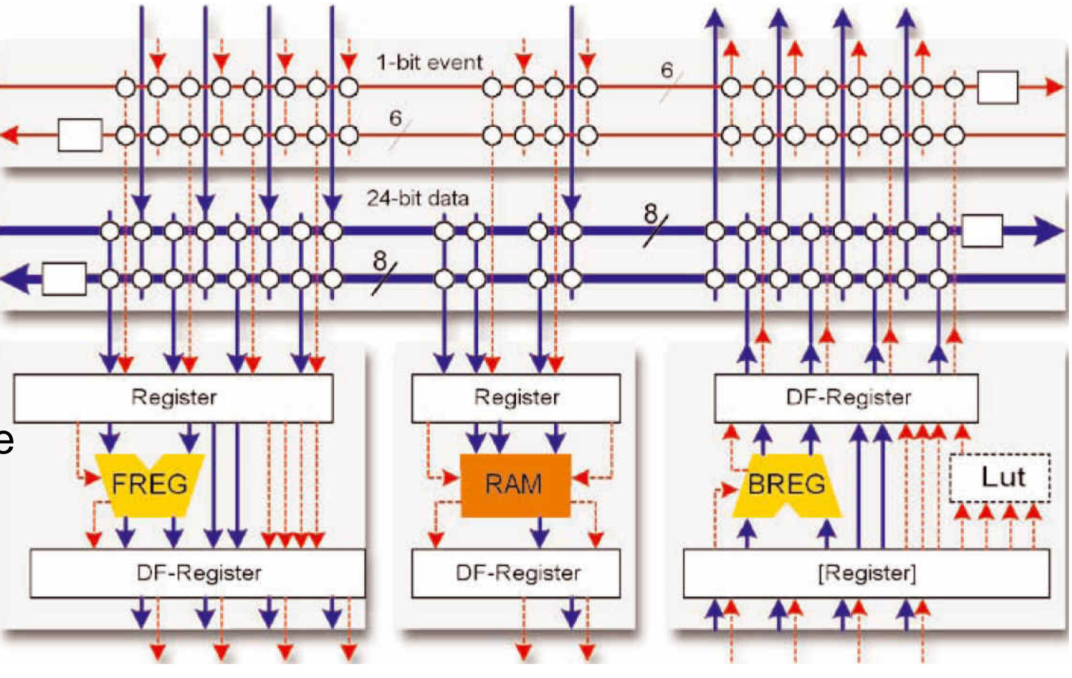
PACT XPP

RAM-PAEs (16):

- ähnlich wie ALU-PAEs - anstelle des ALU-Objekts- two-port RAM mit Kapazität von 512 x 24 Bit als Speicher oder für LUTs

- das RAM-PAE kann im FIFO-Modus arbeiten (keine Adressen für jedes einzelne Datum nötig)

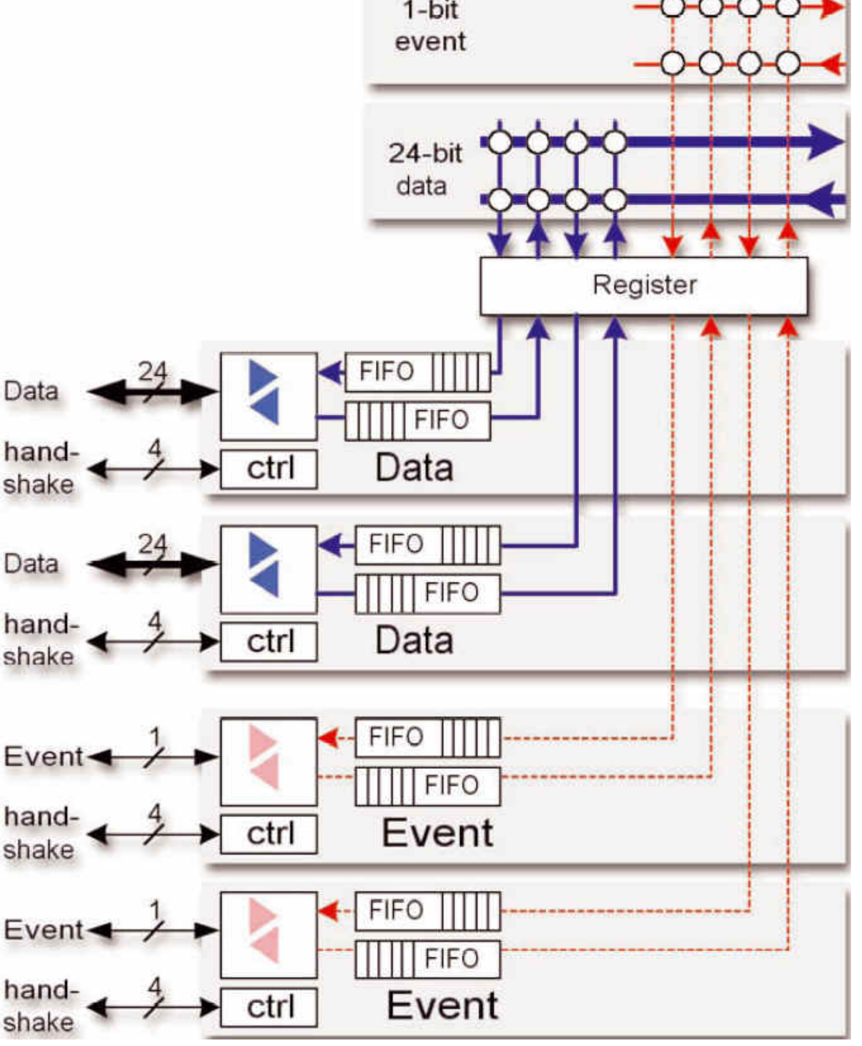
- XPP's Paket-orientierte Kommunikation bewirkt:
  - RAM generiert ein Daten-Paket nachdem ein Adress-Paket am READ-Eingang erhalten wurde
  - Schreiben erfordert zwei Pakete: eins mit der Adresse und eins mit dem zu schreibenden Datenwort



PACT XPP

Input-Element:

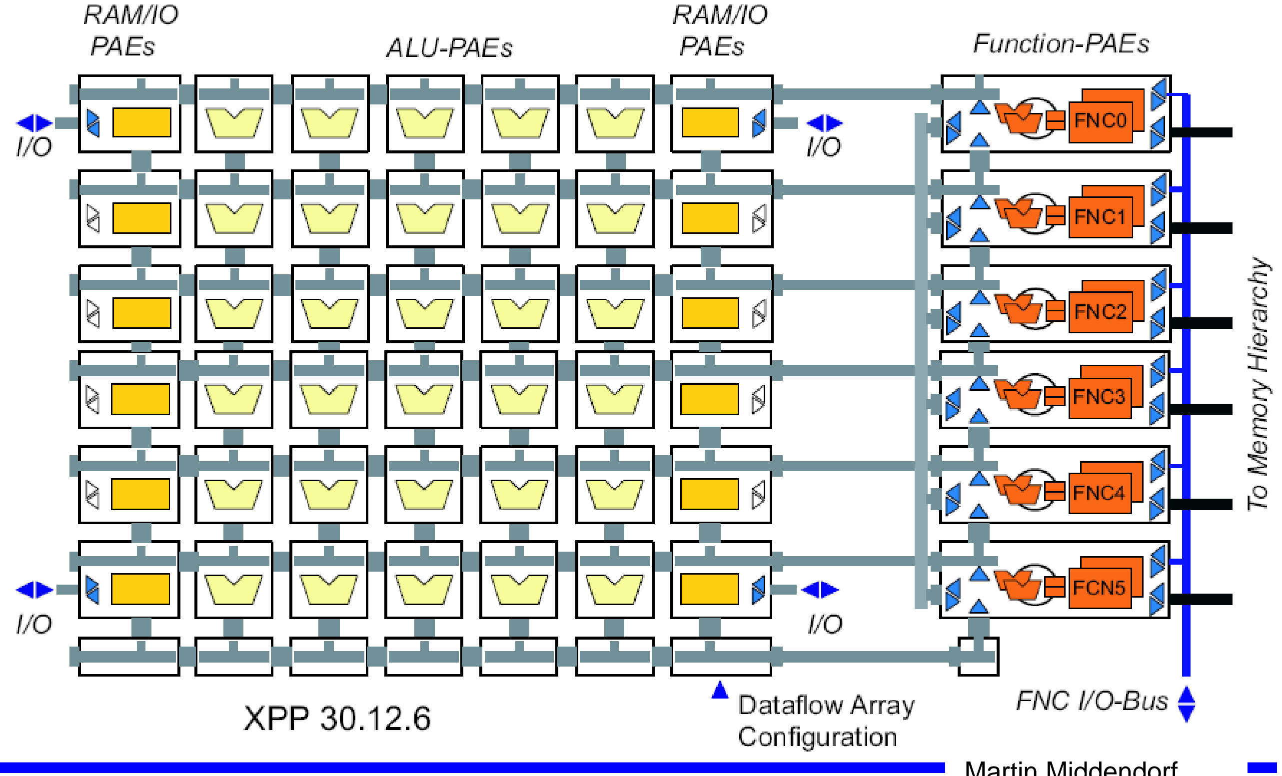
- 4 I/O Elemente arbeiten unabhängig voneinander
- Zwei Operations-Modi
  - ⇨ RAM-Modus und Streaming-Modus
- Streaming-Modus:
  - ⇨ 2 bidirektionale 24-bit Ports; Handshake-Signale für Busprotokoll und Synchronisation der Daten Pakete und externen Ports
  - ⇨ 2 1-bit Events können mit externen Kontroller ausgetauscht werden
- RAM-Modus:
  - ⇨ Jeder Port kann auf externes SRAM zugreifen.
  - ⇨ Steuersignale für SRAM Transaktionen



PACT XPP

XPP III:

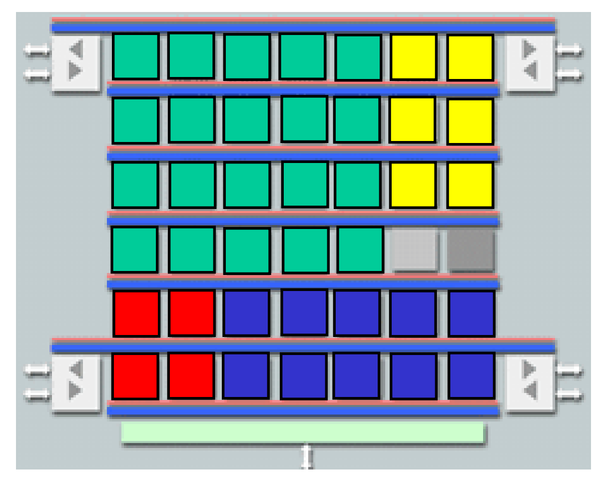
- Zusätzlich Function-PAEs (FNC-PAEs):
  - ⇨ 16-bit Processor Kernel (has 8 ALUs)
  - ⇨ Optimiert für sequentielle Algorithmen, die viele bedingte Sprünge ausführen (Bit-Stream Dekodierung, Verschlüsselung)



PACT XPP

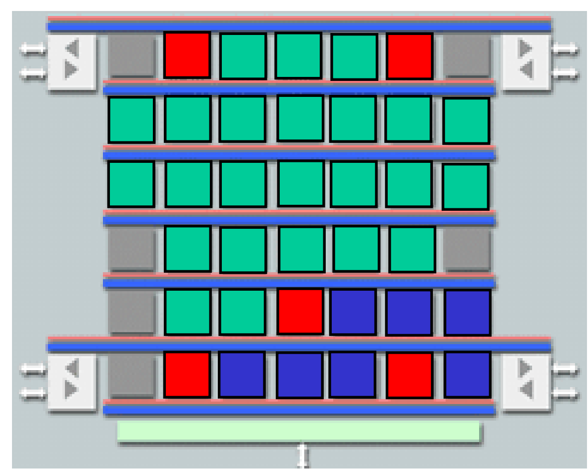
Rekonfiguration:

- Menge der Rekonfigurationsdaten: kBits
  (mehrere MBits bei vielen FPGAs)



- Teile des Arrays können rekonfiguriert werden während auf anderen Teilen Berechnungen ausgeführt werden

- Konfigurationsfolge kann durch Resultate von Berechnungen über Ereignisse gesteuert werden



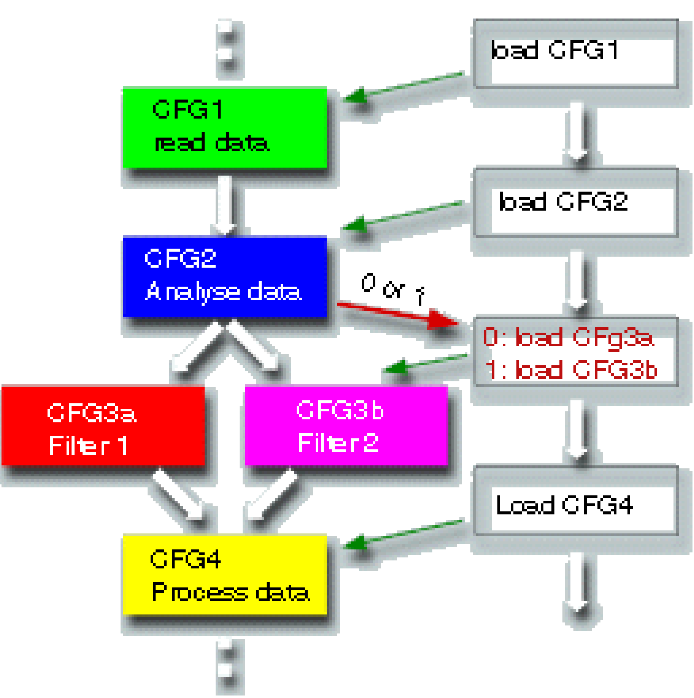
- zwei Modi pro PAE: „konfiguriert" „nicht konfiguriert"
  → falls „nicht konfiguriert", akzeptiert sie eine Konfiguration mit ihrer Adresse und fängt sofort an zu arbeiten, falls Eingaben da sind und Ausgaberegister frei sind (datenflußorientiert)

PACT XPP

Dynamische Rekonfiguration:

- Das Beispiel zeigt eine bedingungsabhängige Konfigurationsfolge

- Dynamische Rekonfiguration erweitert den Instruktionsfluss herkömmlicher Architekturen durch einen Konfigurationsfluss von komplexeren Algorithmen



PACT XPP

Programmierung:

Paralleler oder sequentieller Algorithmus wird als Flow Graph mittels XPP's Primitiven für die Knoten beschrieben

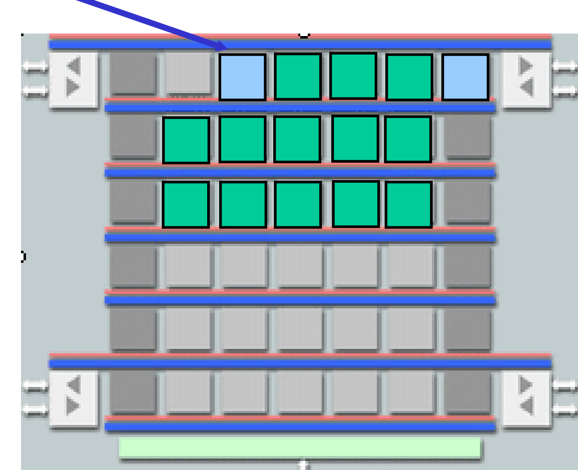
Zerlege den Flow Graphs in Partitionen

Mappe die erste Partition des Flow Graphen auf den Array (Konfiguration)

Eingabendaten werden in den Array geladen und die Berechnungen ausgeführt.
Zwischenresultate werden lokal gespeichert.
Ein Ereignis zeigt das Ende der Berechnungen an.

Die nächste Partition wird geladen bis alle Partitionen berechnet wurden

Definiere gemeinsame ALU-PAEs und RAM-PAEs



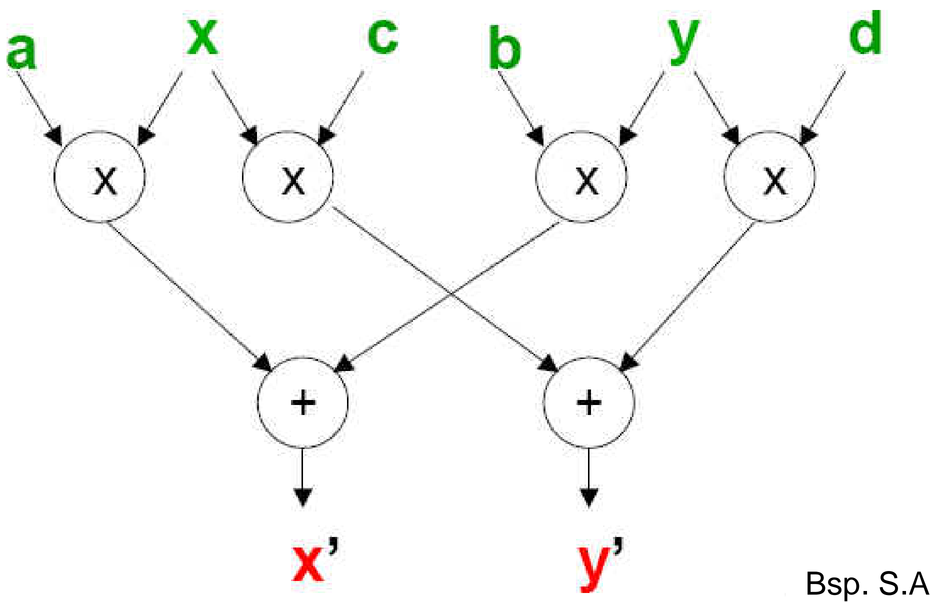
PACT XPP

Matrix Multiplication

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} \times \begin{vmatrix} x \\ y \end{vmatrix} = \begin{vmatrix} ax+by \\ cx+dy \end{vmatrix} = \begin{vmatrix} x' \\ y' \end{vmatrix}$$

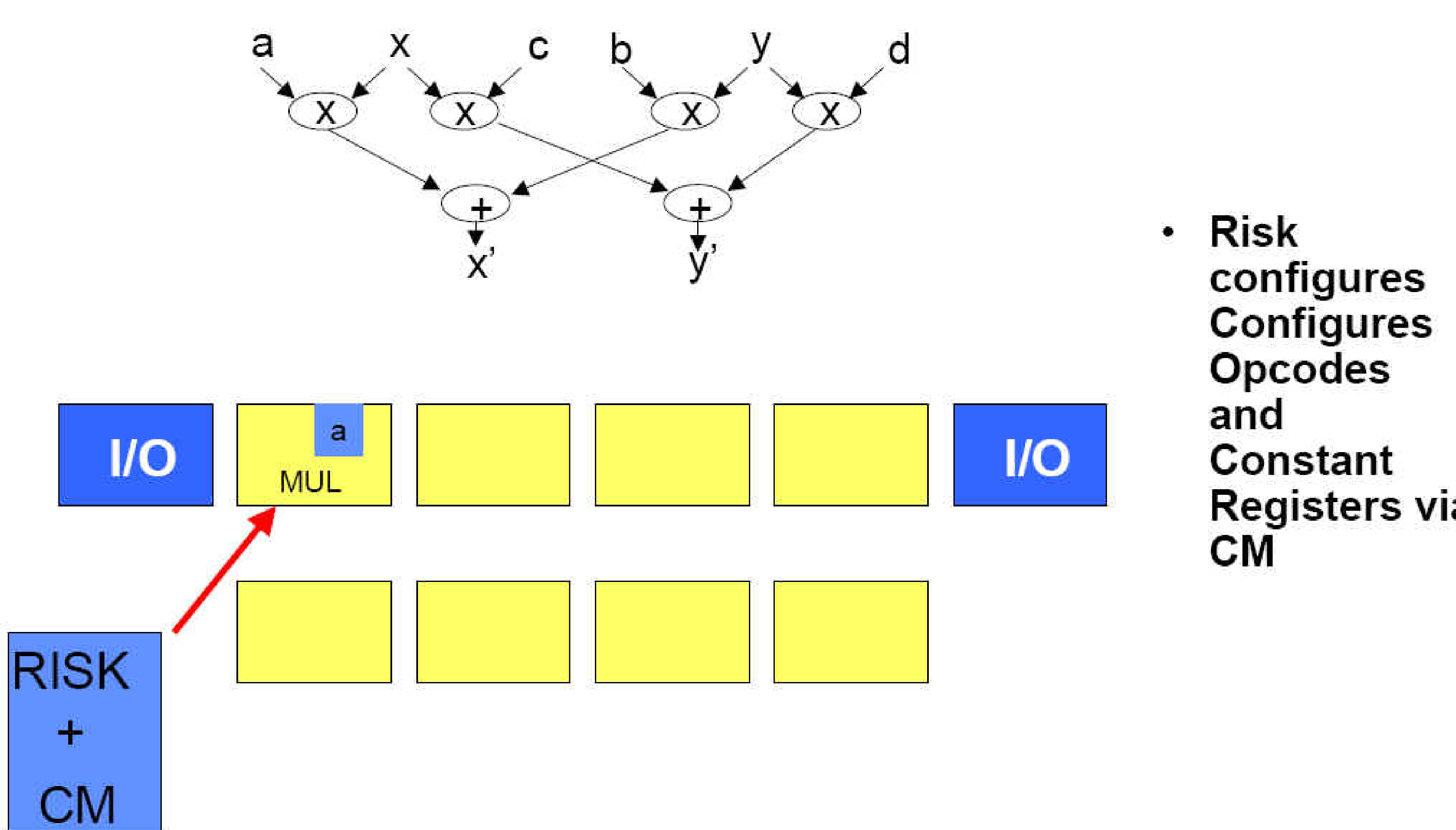
(matrix is constant)

Flow Graph



Bsp. S.A. Huss

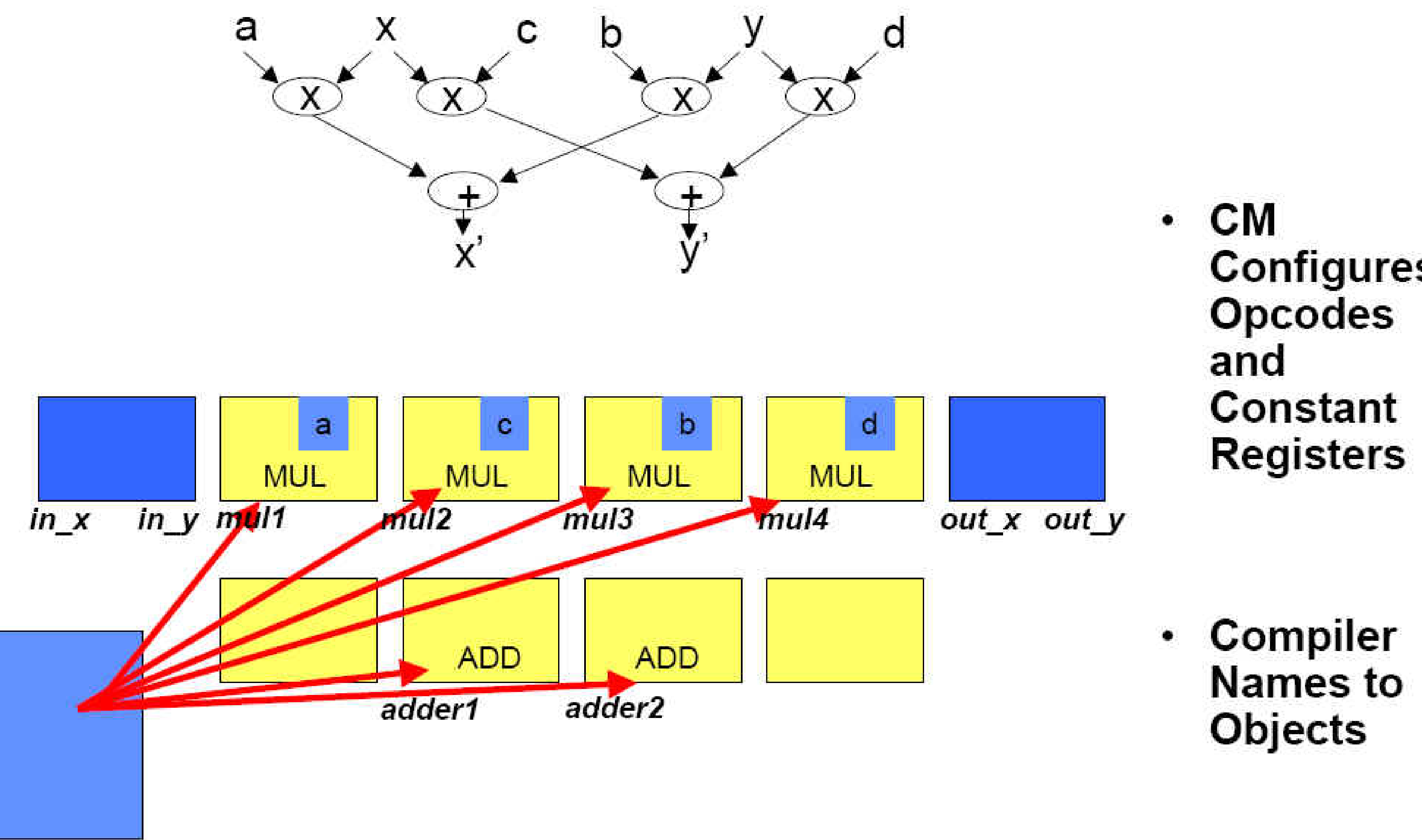
PACT XPP



- Risk configures Configures Opcodes and Constant Registers via CM

Bsp. S.A. Huss

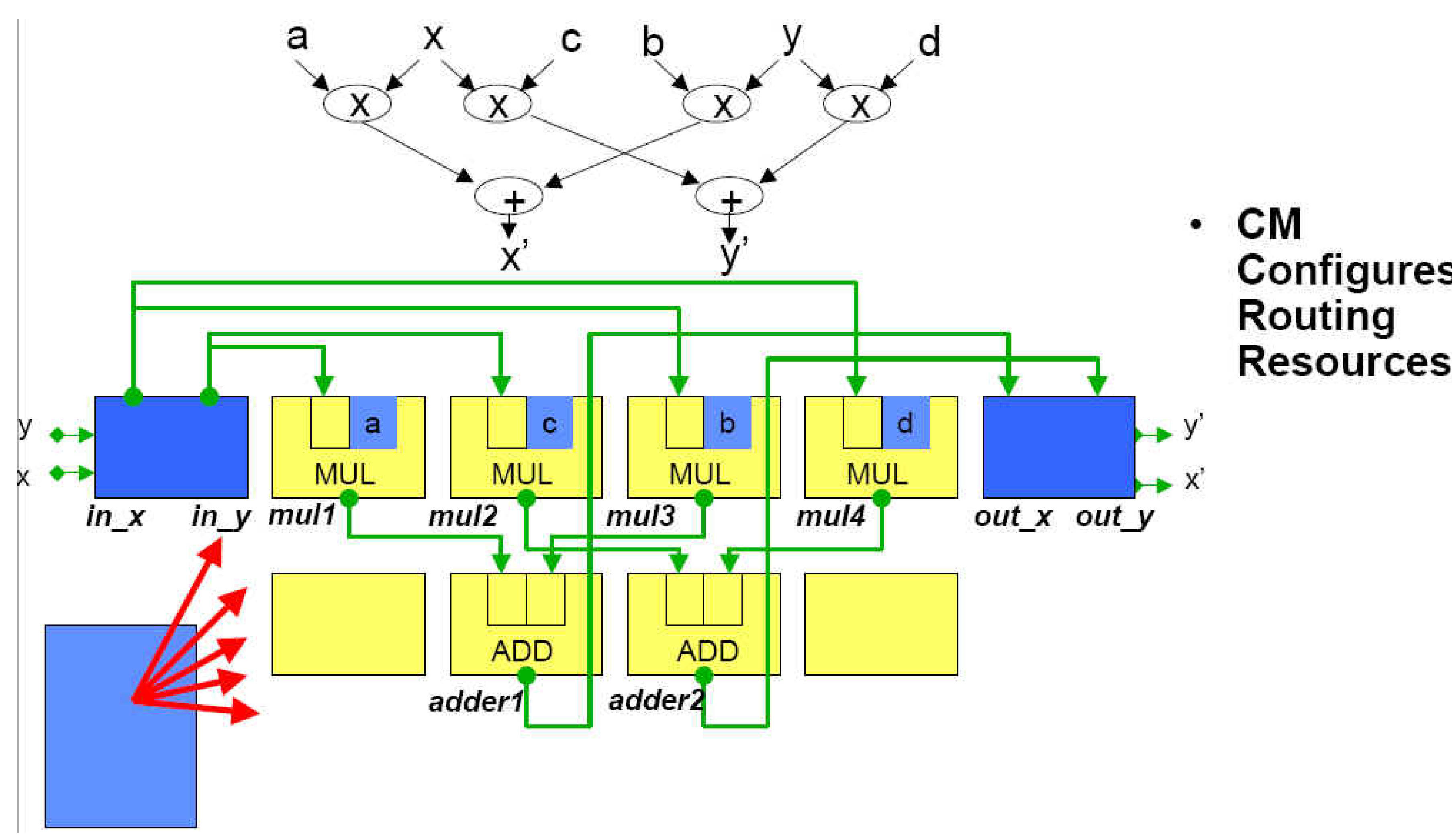
PACT XPP



- **CM Configures Opcodes and Constant Registers**
- **Compiler Names to Objects**

Bsp. S.A. Huss

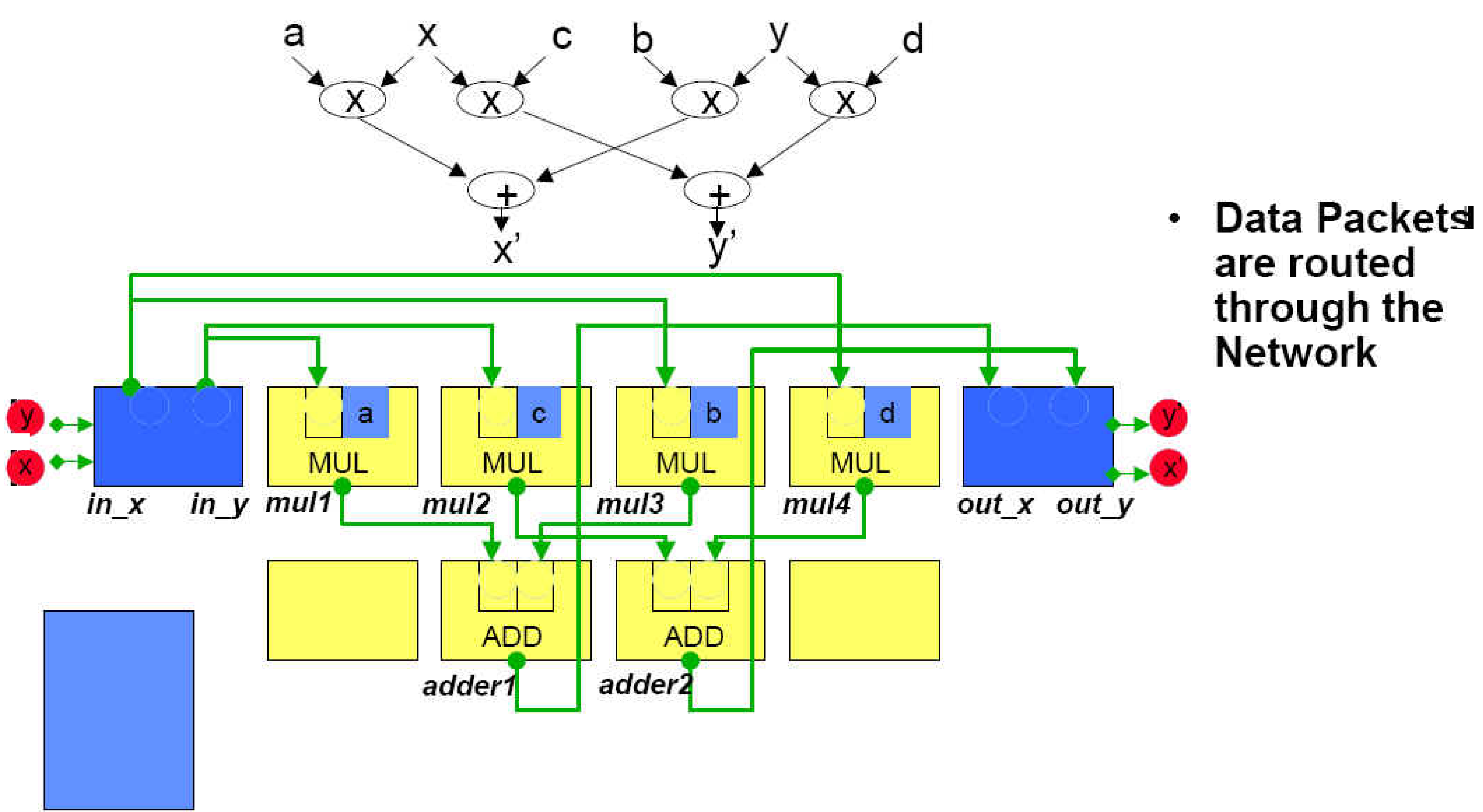
PACT XPP



- **CM Configures Routing Resources**

Bsp. S.A. Huss

PACT XPP

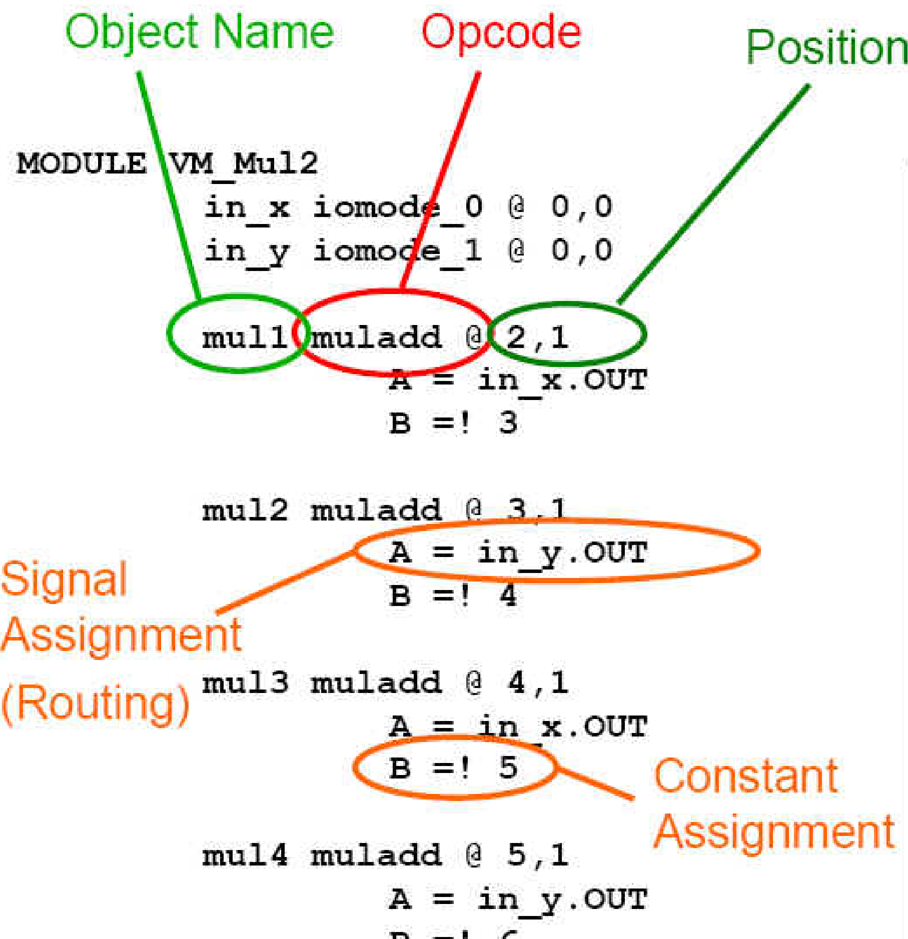


- Data Packets are routed through the Network

Bsp. S.A. Huss

PACT XPP

PACT Native Mapping Language (NML):




```
adder1 add @ 3,2
       A = mul1.L
       B = mul3.L

adder2 add @ 4,2
       A = mul2.L
       B = mul4.L

out_y iomode_0 @ 11,0
      IN = adder1.L

out_x iomode_1 @ 11,0
      IN = adder2.L

END VM_Mul2
```

Bsp. S.A. Huss

Note	ALU inputs:	A, B, C
	ALU outputs:	L, H

Einschätzung der Architektur

Vorteile der Architektur

- ⇨ Vector Aufbau erlaubt effiziente Codierung
- ⇨ Eignung für typische Stream Applikationen
- ⇨ Schnelle Rekonfiguration durch Prefetching
- ⇨ Sehr gute Skalierbarkeit
- ⇨ Hohe Performanz im Vergleich zu ähnlichen Systemen

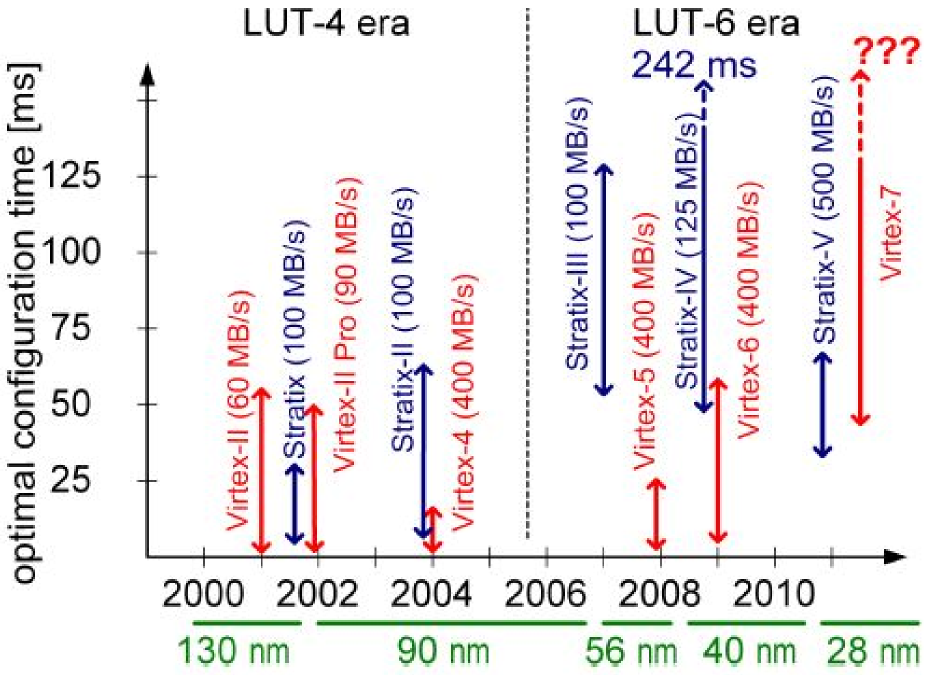
Problematisch

- ⇨ Verwischt stark die Grenzen von Hardware und Software
- ⇨ Hoher Stromverbrauch
- ⇨ Applikationen müssen reguläre Strukturen sein
- ⇨ Verminderte Parallelität in Algorithmen da grobgranular

Hyperreconfigurable Architectures [S. Lange, M. Middendorf]

Context: Dynamically reconfigurable architectures offer new abilities for flexibility so that computations can change their requirements to the architecture during run time.

Problem: Extensive reconfiguration potential causes a great amount of reconfiguration data for applications with frequent use of dynamic reconfiguration.

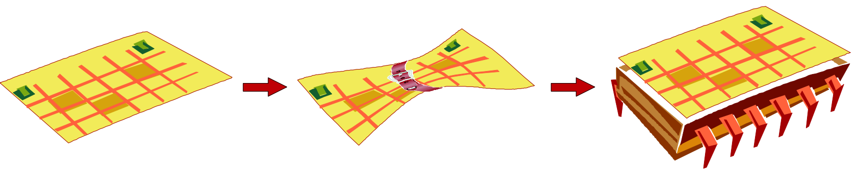


Quelle: Koch, Torrensen

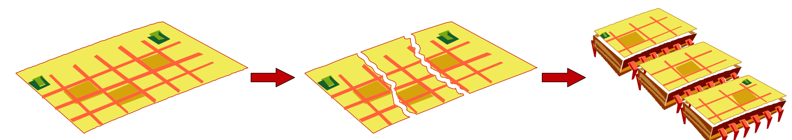
Previous Solutions

Reduction of the reconfiguration costs

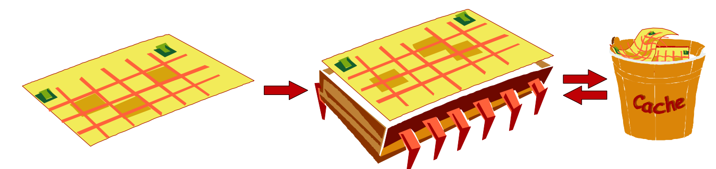
1. Bit stream compression
   - Off-line Methods [Dandalis,Prasanna,2001]
   - Don’t-care Bits [Hauck et al.,1999,2001]



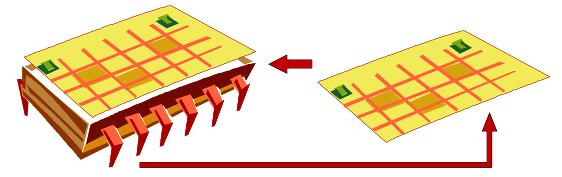
2. Incremental Reconfigurability
   - Multi-FPGA Systems [Lee,Wong,2002]



3. Configuration Caching
   - Multi Context FPGAs
     [Hauck et al.,1999,2001;Li et al.,2000]



4. Self Reconfigurability
   [Köster,Teich,2002;Sidhu et al.,2000]
   [Wadhwa,Dandalis,2000]



Hyperreconfiguration

Observation: Computations show phases with differing resource utilization

→ Time-variant resource demands

Basic Idea: Dynamically adapt reconfiguration capabilities so that reconfiguration data contain ideally only information about active resources

"Reconfiguration of Reconfigurability"

Realization: 2-layered model of reconfiguration

- Hypercontexts define the limits of reconfiguration
- Contexts specify the behavior of active resources within a given hypercontext through reconfiguration

Note: Granularity of (hyper)reconfiguration not specified

Hyperreconfiguration

Formal Description:

- Machine defines
  - ⇨ Set of feasible **hypercontexts** $\mathcal{H} = \{h_i\}$
  - ⇨ Set of admissible **resource demands** $\mathcal{C} = \{c_j\}$
  - ⇨ For each hypercontext h the set of resource demands it fulfills

$$h(\mathcal{C}) \subset \mathcal{C}$$

- For a sequence $c_1, \ldots, c_k$ of context requirements and a hypercontext h let $c_1 \ldots c_k \subset h(\mathcal{C})$ denote the fact that for each context requirement c_i, $i=1, \ldots, k$ holds:

$$c_i \in h(\mathcal{C})$$

- An **algorithm** defines sequence of resource demands $c_1 \ldots c_n$
- An algorithm is **feasible** if hypercontexts $h_1, \ldots, h_r$ exist such that the following operations can be executed

$$h_1 c_1 \ldots c_{i_1} \, h_2 c_{i_1+1} \ldots c_{i_2} \;\; \ldots \;\; h_r c_{i_{r-1}+1} \ldots c_n$$

i.e. $c_{i_j} c_{i_j+1} \ldots c_{i_{j+1}} \subseteq h_{j_{j+1}}$

Hyperreconfiguration

Cost Model:

- For each hypercontext h
  - ⇨ **cost for hyperreconfiguration** $init(h)$
  - ⇨ **cost for a single reconfiguration** in h $cost(h) = |h|$
- **Total costs** of an algorithm/computation

$$h_1 c_1 \ldots c_{i_1} \; h_2 c_{i_1+1} \ldots c_{i_2} \; \ldots \; h_r c_{i_{r-1}+1} \ldots c_n$$

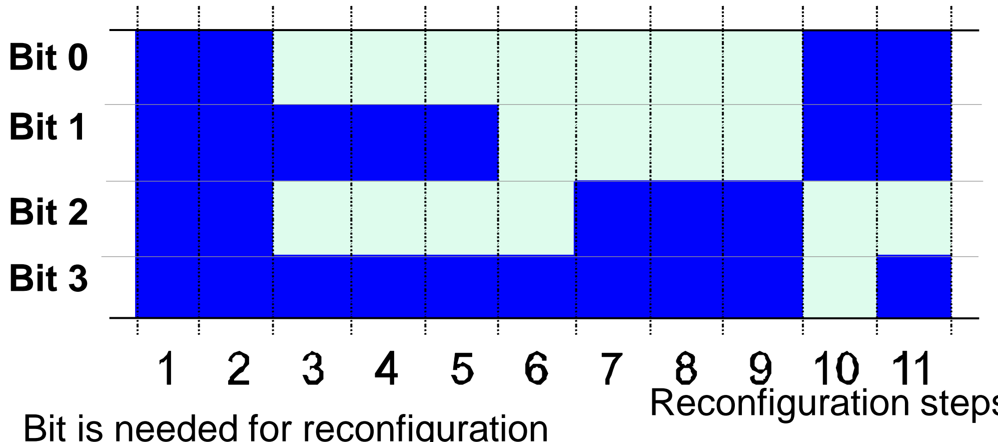
are defined as

$$\sum_{i=1}^{r} init(h_i) + \sum_{j=1}^{r} |h_j| \cdot (i_j - i_{j-1})$$

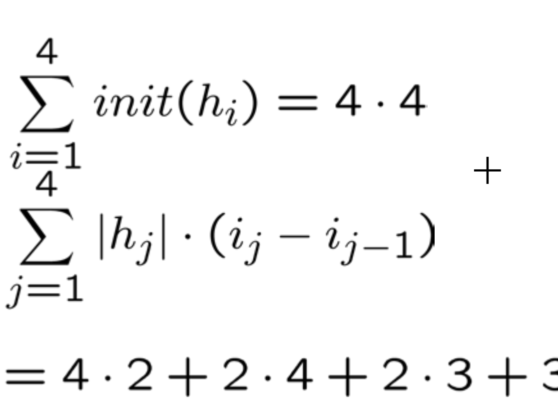
Hyperreconfiguration

Example:

- 11 reconfigurations
- 4 reconfiguration bits = HR-costs $init(h) = 4$
- 4 hyperreconfigurations



- Total costs are



$$\sum_{i=1}^{4} init(h_i) = 4 \cdot 4 \quad + \quad \sum_{j=1}^{4} |h_j| \cdot (i_j - i_{j-1})$$

$$= 4 \cdot 2 + 2 \cdot 4 + 2 \cdot 3 + 3 \cdot 2$$

Bit 0
Bit 1
Bit 2
Bit 3

hyperreconfiguration

Bit is not available for reconfiguration

Bit is available and needed for reconfiguration

Bit is available for reconfiguration (but not needed)

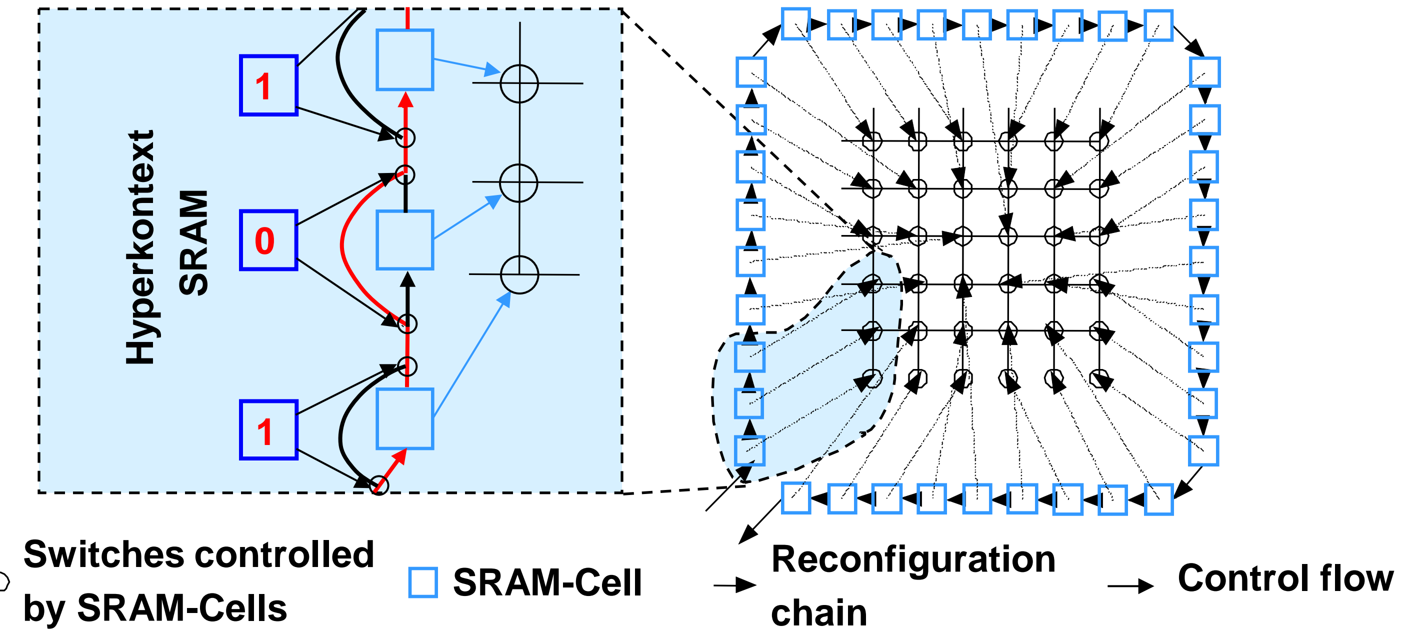
Hyperreconfiguration

Switch Model of Hyperreconfiguration:

All reconfigurable resources are treated as simple switches and each switch can be manipulated directly in the bit stream

Example: Switch-Matrix

- SRAM-cells control switches
- 1 SRAM-cell per switch (shift register)
- bypass of SRAM-cells through hypercontext



PHC Problem

Partition into Hypercontexts (PHC) Problem:

- ⇨ **Given**: Task T and a sequence of context requirements $c_1 \ldots c_n$
- ⇨ **Find**: The best time steps when to perform hyperreconfigurations and define corresponding hypercontexts so that the total time for (hyper)reconfigurations is minimal.

Theorem: The **PHC Problem** is NP-complete.

PHC Problem

Switch Model of hyperreconfiguration:

- ⇨ Given a set of switches $X = \{x_1, \ldots, x_n\}$
- ⇨ Set of feasible **hypercontexts** $\mathcal{H}$ and the set of admissible **resource demands** $\mathcal{C}$

$$\mathcal{H} = \mathcal{C} = 2^X$$

- ⇨ For each hypercontext the set of resource demands it fulfills are given by:

$$\forall x \in X: \quad x \in h(\mathcal{C}) \;\Leftrightarrow\; x \in h$$

- ⇨ Costs are defined by: $cost(h) = |h|$

PHC-Problem for the Switch Model:

- ⇨ **Given**: Task T and a sequence of context requirements $C = c_1, \ldots, c_m$
- ⇨ **Find**: Partition of C into substrings $S_1, \ldots, S_r$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$ such that $S_i \subset h_i$ and the following costs are minimal

$$r \cdot n + \sum_{i=1}^{r} |h_i| \cdot |S_i|$$

PHC Problem

DAG Model of hypereconfiguration: (DAG= Directed Acyclic Graph)

- ⇨ Set of feasible **hypercontexts** $\mathcal{H}$ and the set of admissible **resource demands** $\mathcal{C}$
- ⇨ Given a DAG G=(V,E) with $V = \mathcal{H}$ and for each $h \in \mathcal{H}$ a set $h(\mathcal{C})$ such that for each edge $(h_1, h_2) \in E$ the following relation holds:

$$h_1(\mathcal{C}) \subset h_2(\mathcal{C})$$

- ⇨ It is assumed that a hypercontext h exists with $h(\mathcal{C}) = \mathcal{C}$
- ⇨ Let cost(h)>0 and init(h)=w for each $h \in \mathcal{H}$ and for each edge $(h_1, h_2) \in E$ it is assumed that $cost(h_1) \leq cost(h_2)$

PHC-Problem for the DAG Model:

- ⇨ **Given**: Task T and a sequence of context requirements $C=c_1,\ldots,c_m$
- ⇨ **Find**: Partition of C into substrings $S_1,\ldots,S_r$ (i.e. $C=S_1\ldots S_r$) and hypercontexts $h_1,\ldots,h_r$ such that $S_i \subset h_i$ and the following costs are minimal

$$r \cdot w + \sum_{i=1}^{r} cost(h_i) \cdot |S_i|$$

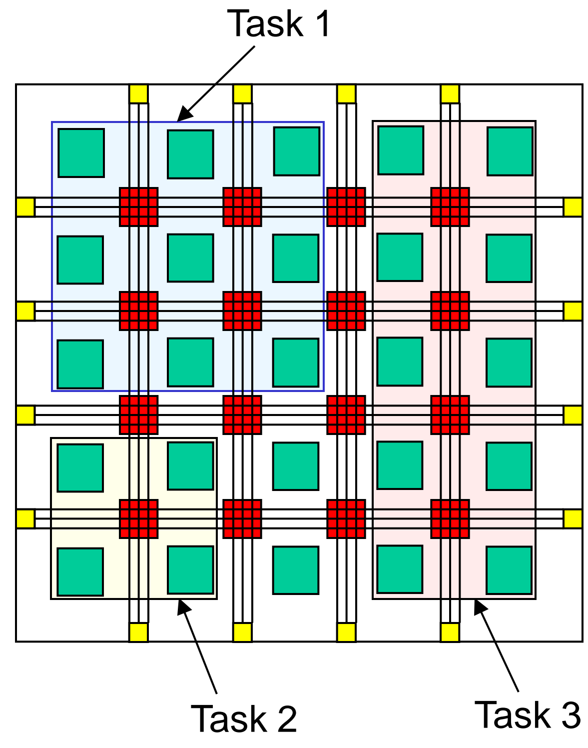
PHC Problem

For the special cases of the Switch Model and the DAG Model the PHC Problem is solvable in polynomial time:

Theorem : The **PHC-Switch Problem** (i.e., the PHC Problem for the Switch Model) can be solved in time $O(nm^2)$ where m is the number configurations of the given algorithm and n the number of switches.

Theorem : The **PHC-DAG Problem** (i.e., the PHC Problem for the DAG Model) can be solved in time $O(m^3+\alpha m^2)$ where m is the number configurations of the given algorithm, n the number of switches, and α is the time to find for two sets of hypercontexts in the DAG (or for one configuration c in $\mathcal{C}$) the set of minimal hypercontexts that are predecessor of at least one hypercontext in both sets (respectively, of c) + the time to compute the cheapest of these minimal hypercontexts

Multi Task Models



Private global resources:

⇨ have to be shared between tasks

⇨ ownership is defined by hypercontext

Example: I/O units

Public global resources:

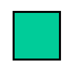
⇨ can be used by all tasks at the same time and according to quality as defined by hypercontext

Example: Switch boxes

Local resources:

⇨ available reconfiguration properties can be defined during hyperreconfiguration for each task separately

⇨ ownership is defined at initialization of a task

Example: functional units within the private area of a task 

Partial Hyperreconfigurability

Partially hyperreconfigurable machines:

- only a subset of the tasks can reconfigure
- use two types of hyperreconfiguration operations:

1. **Global Hyperreconfigurations**:
   - determine all public global resources and the assignment of private global resources to tasks
2. **Local Hyperreconfigurations**:
   - determine all local resources and the private global resources that are assigned to a task

Different types of partially hyperreconfigurable machines:

1. **Partially reconfigurable**:
   - Reconfigurations can be done partially
   - Hyperreconfigurations are global operations (i.e. involve all tasks)
2. **Partially hyperreconfigurable**:
   - Reconfigurations and local hyperreconfigurations can be done partially
3. (**Restricted partially hyperreconfigurable**:
   - Reconfigurations are global operations (i.e. involve all tasks)
   - Local hyperreconfigurations can be done partially)

Synchronization

Assumption 1: During a global hyperreconfiguration no task can perform computations and the old local hypercontexts are no longer valid

→ **synchronization effect**

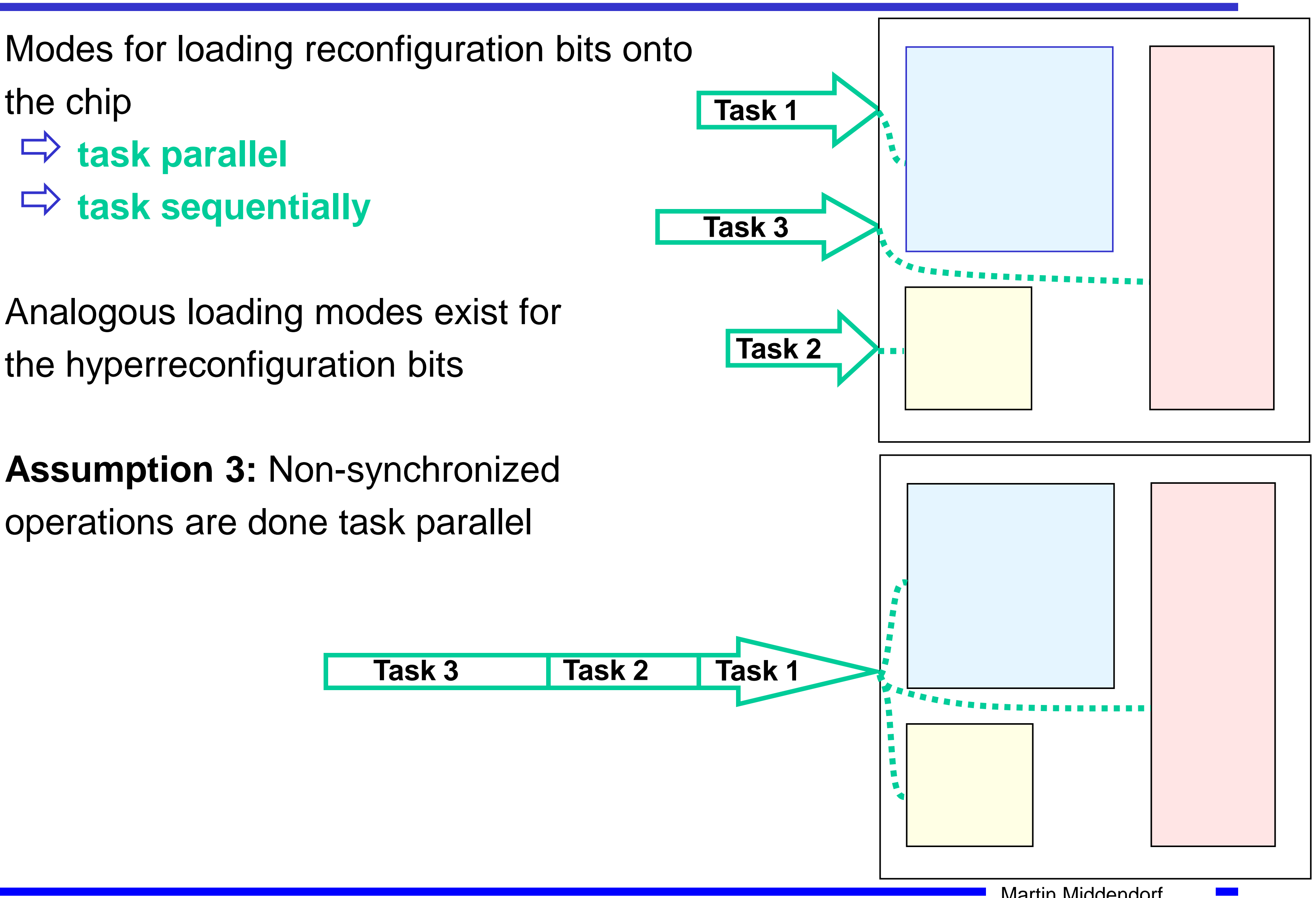
Models of synchronization

- **Hypercontext synchronized**:
  - Partial local hyperreconfigurations are synchronized between all tasks (no matter whether they perform a hyperreconfiguration or are idle)
- **Context synchronized**:
  - Reconfigurations are synchronized
- **Fully synchronized**:
  - Machine is hypercontext synchronized and context synchronized
- **Non-synchronized**:
  - Machine is neither local hypercontext synchronized nor context synchronized

Assumption 2: Barrier Synchronization

- Tasks that do not want to perform a (hyper)reconfiguration perform a no-(hyper)reconfiguration statement
- Tasks wait until all tasks have arrived at corresponding statements for (hyper)reconfigurations

Loading the Reconfiguration Bits



The Non-synchronous Case

Total reconfiguration costs for one global hyperreconfiguration + following local hyperreconfigurations and reconfigurations for the Switch-model in

⇨ **non-synchronous mode** where

⇨ local hyperreconfigurations and reconfigurations are executed **task parallel**

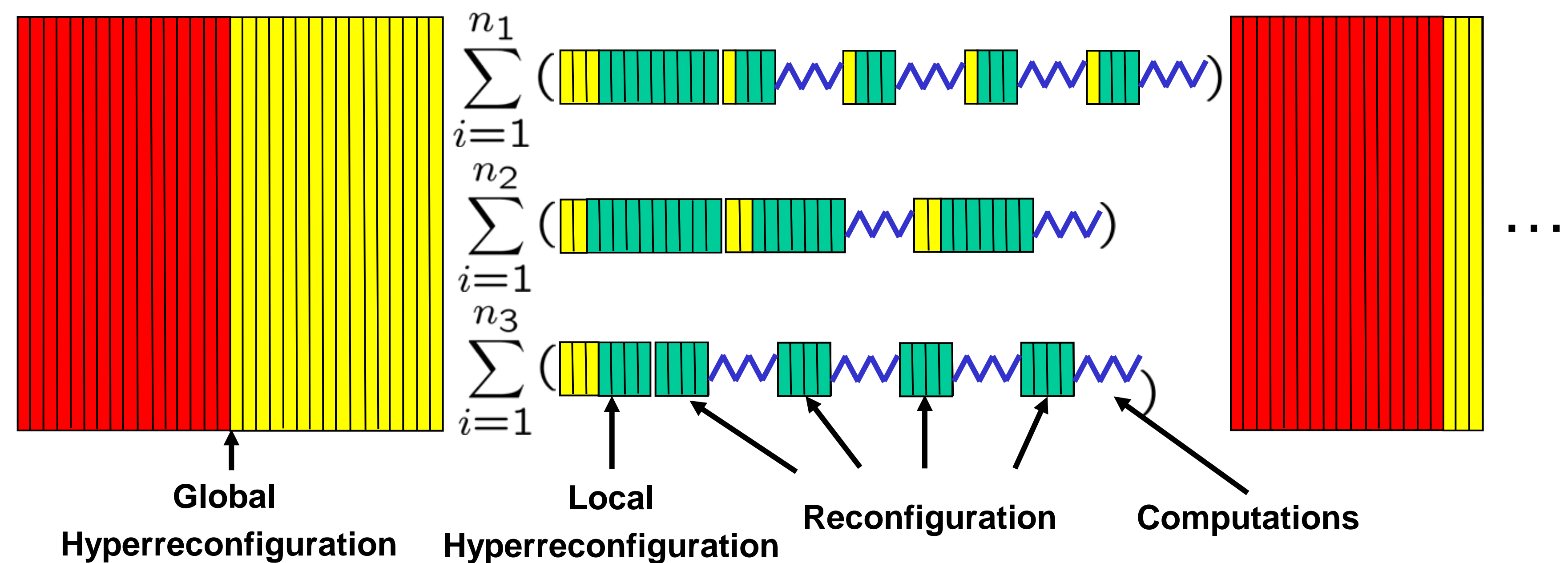
$$|X^{pub}|+|X^{priv}|+\max_{j=1}^{m}\{\sum_{i=1}^{n_j}(|h_j|+|f_j^{loc}|+(|h_{i,j}^{loc}|+|h_{i,j}^{priv}|)\cdot|S_{j,i}|)\}$$

- X^{pub} is the set of public global switches
- X^{priv} is the set of private global switches
- m = number of tasks
- n_j is the number of local hyperreconfigurations of task j
- h_j is the set of local switches of task j
- f_j^{loc} is the set of private global switches assigned to task j
- $|h_{i,j}^{loc}|$ and $|h_{i,j}^{priv}|$ are the costs for a local reconfiguration of task j
- $S_{j,i}$ is the sequence of reconfigurations of task j between its i-th and (i+1)th local hyperreconfiguration

The Non-synchronous Case

Example for Switch-model in **non-synchronous mode** where partial hyperreconfigurations and reconfigurations are executed **task parallel**

- 3 tasks (not included reconfiguration of ■)



The Synchronous Case

Total reconfiguration costs for one global hyperreconfiguration + following local hyperreconfigurations and reconfigurations for the Switch-model in

- ⇨ **fully synchronous** mode where
- ⇨ partial hyperreconfigurations and reconfigurations are executed **task parallel**

$$|X^{pub}|+|X^{priv}|+\sum_{i=1}^{n}(\{\max_{j=1}^{m}\{I_{j,i}\cdot(|h_j|+|f_l^{loc}|)\}+\max_{j=1}^{m}\{(|h_{f_j(l),j}^{loc}|+|h_{f_j(l),j}^{priv}|)|S_{ij}|\})$$

n is the number of local hyperreconfigurations

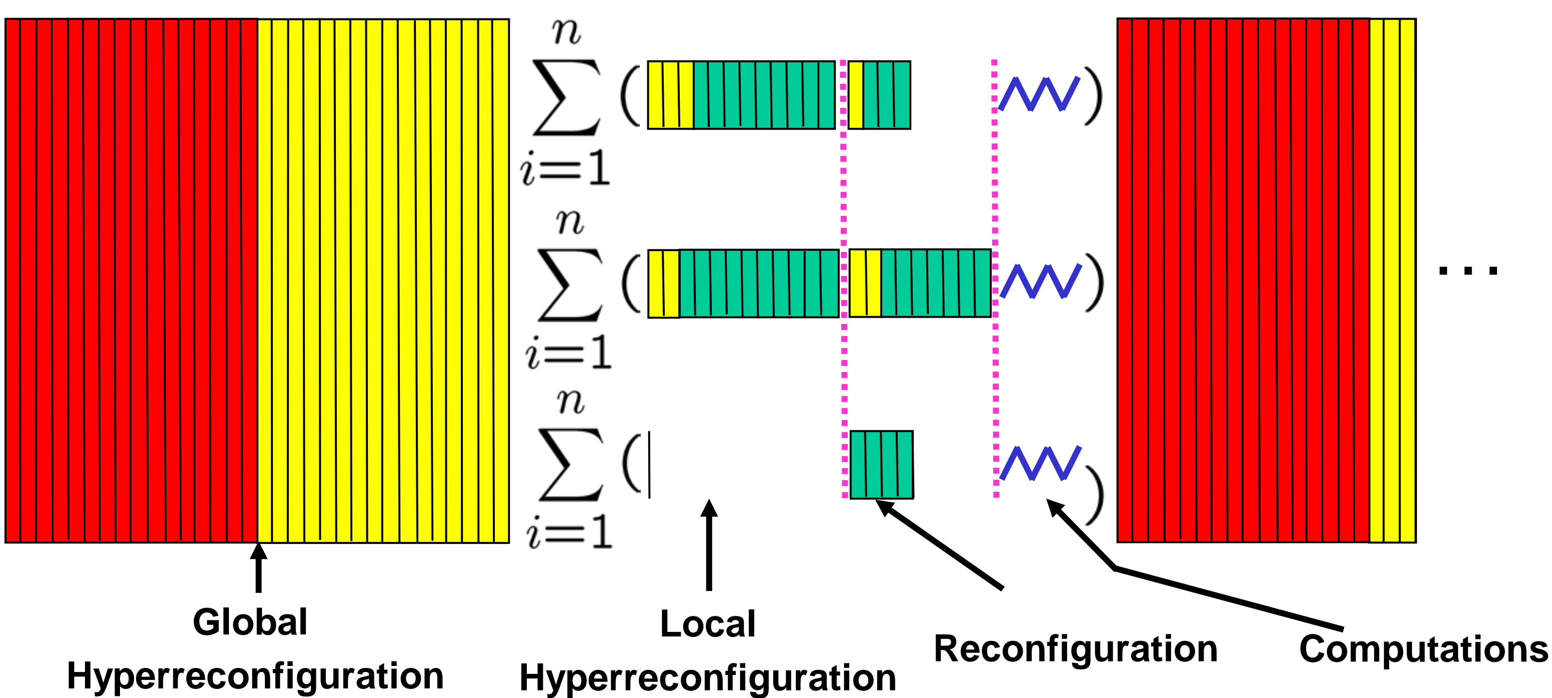
Let $I_{\{j,l\}}=0$ when the l-th local hyperreconfiguration operation of task T_j is a no-hyperreconfiguration operation and otherwise let $I_{\{j,l\}}=1$.

For task T_j let $f_j(l)$ be the number of the last local hyperreconfiguration operation in the sequence of its first l local (no-)hyperreconfiguration operations.

The Synchronous Case

Example for the Switch-model in **fully synchronous** mode where partial hyperreconfigurations and reconfigurations are executed **task parallel**

- 3 tasks (not included reconfiguration of ■)



The MTH Problem

Multi Task Hyperreconfiguration (MTH) problem:

- ⇨ **Given**: Tasks $T_j, j \in [1:m]$ and for each task a sequence of context requirements $c_{j,1} \cdots c_{j,n}$
- ⇨ **Find**: The best time steps when to perform (local and global) hyperreconfigurations and define corresponding hypercontexts so that the total costs for reconfigurations is minimal.

Theorem: The **MTH** problem for fully synchronized multi task hyperreconfigurable machines in the **Switch model** where partial hyperreconfigurations and reconfigurations are done task parallel can be solved

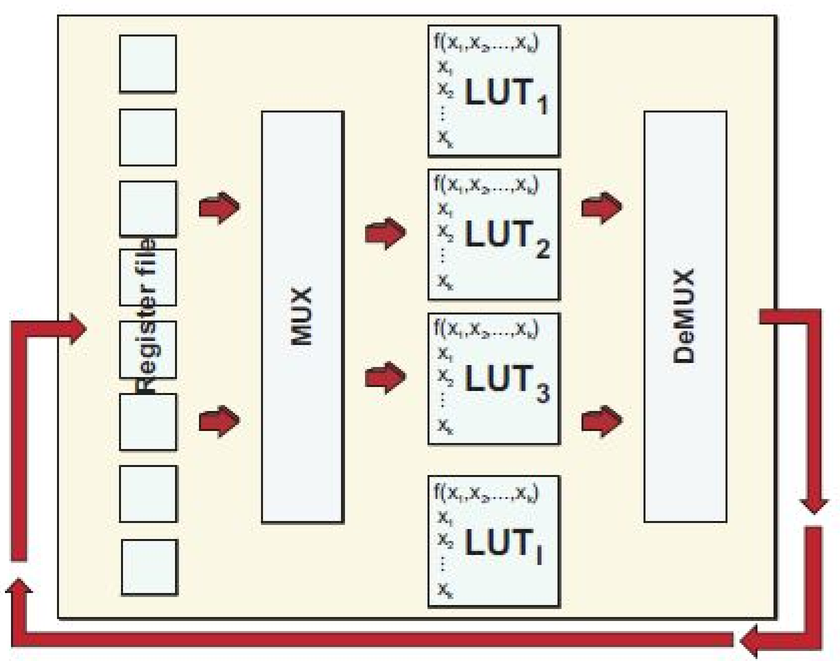
i) without private global resources in time $O(mn^4 l_{max}^2)$
where l_{max} is the maximal number of local switches assigned to a task.

ii) with private global resources in time $O(mn^7 (l_{max} + g)^2)$
where g is the number of private global switches.

Simulation Results for Test Architecture

Simple HyperReconfigurable Architecture

- consists of lookup tables, registers, and routing resources (MUX + DeMUX) which are all reconfigurable
- synchronous execution



- In the following examples:
  3-Input-LUTs
  1-Bit-Register
  Number of reconfiguration bits:
  - 8 for each LUT
  - for MUX and DEMUX it depends on the number of registers

Simulation Results for Test Architecture

4-bit Counter: SHyRA with 10 registers and 2 LUTs → 48 reconfiguration bits
Program run over 110 reconfigurations

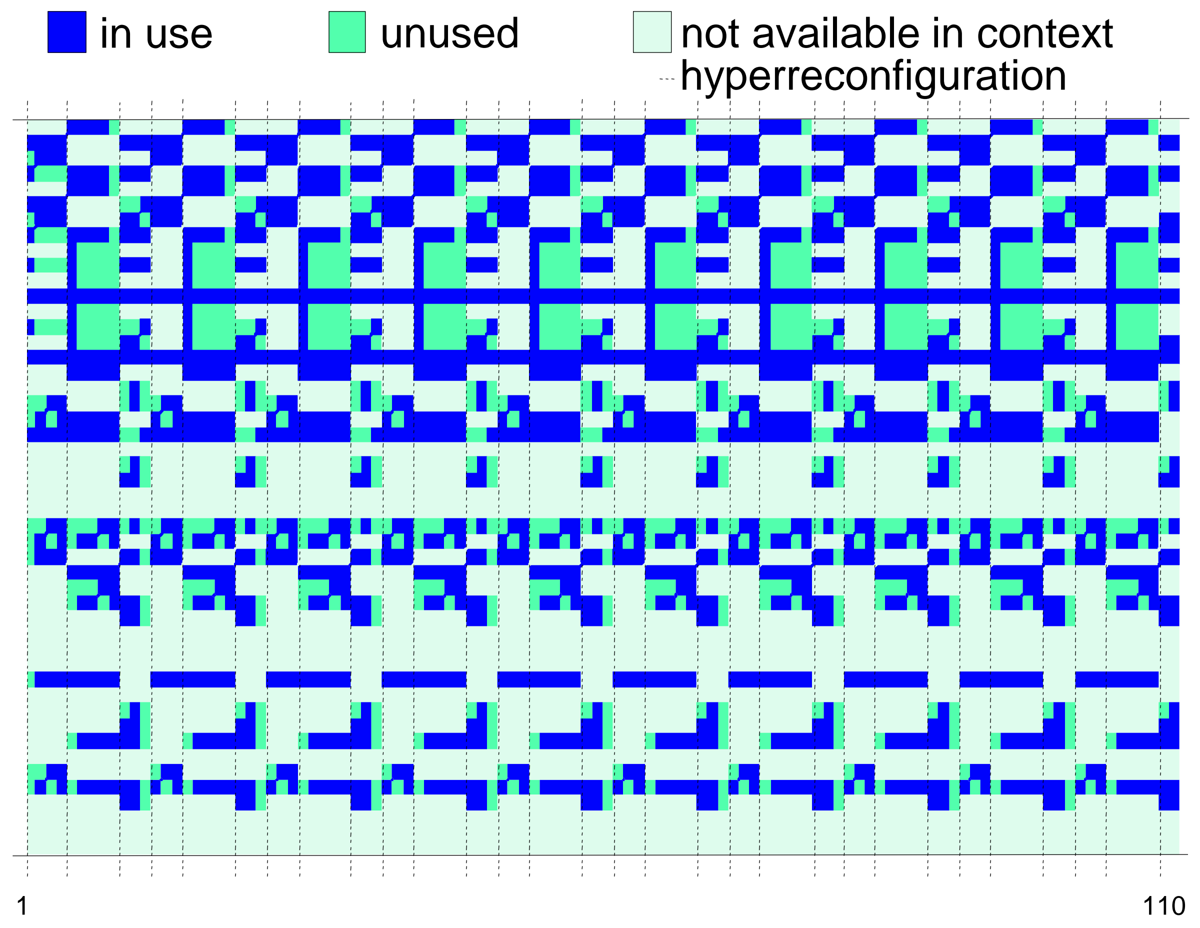
Pseudo Code		Reconfiguration Data	
LUT1	**LUT2**	**LUT1**	**LUT2**
NOT 0, 0, 0	NOT 8, 0, 0	01010101	01010101
XOR 1, 1, 8	AND 8, 8, 1	01100110	00010001
XOR 2, 2, 8	AND 8, 8, 2	01100110	00010001
XOR 3, 3, 8	AND 8, 8, 3	01100110	00010001
EQ 9, 0, 4	ONE 8, 8, 0	10011001	11111111
EQ 9, 1, 5	AND 8, 8, 9	10011001	00010001
EQ 9, 2, 6	AND 8, 8, 9	10011001	00010001
EQ 9, 3, 7	AND 8, 8, 9	10011001	00010001
NULL 9, 0, 0	AND 8, 8, 9	00000000	00010001
CMOV 0, 9, 8	CMOV 1, 9, 8	01010011	01010011
CMOV 2, 9, 8	CMOV 3, 9, 8	01010011	01010011

Single task model: 48 switches

4 task model: LUT1 (8 switches), LUT2 (8 switches), DeMUX (8 switches), MUX (24 switches)

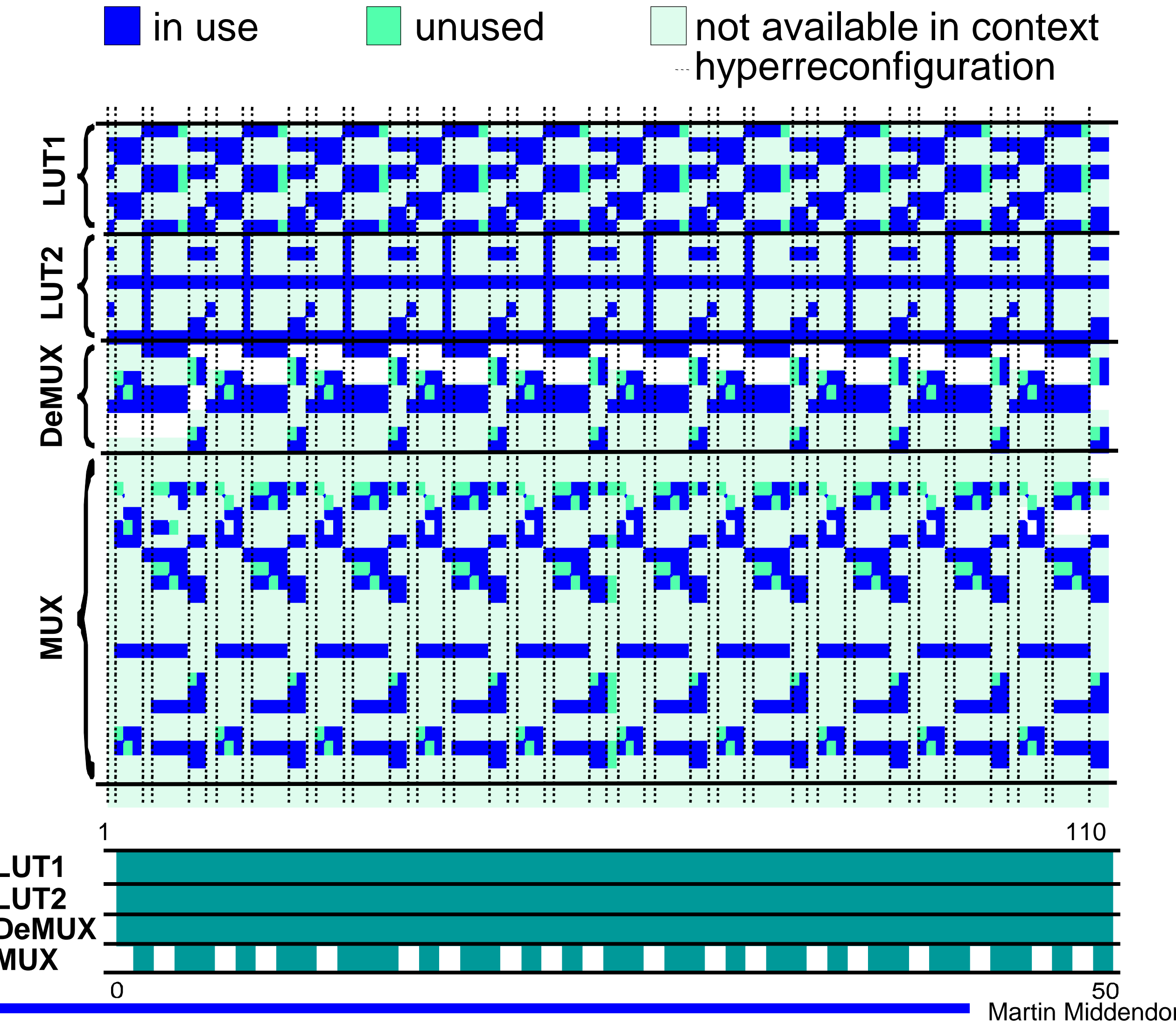
Simulation Results for Test Architecture

Single Task Model: optimal hyperreconfiguration schedule



Simulation Results for Test Architecture

4 Task Model: good hyperreconfiguration schedule (computed with a GA)



Simulation Results for Test Architecture

Comparison of total costs (reconfiguration + hyperreconfiguration)

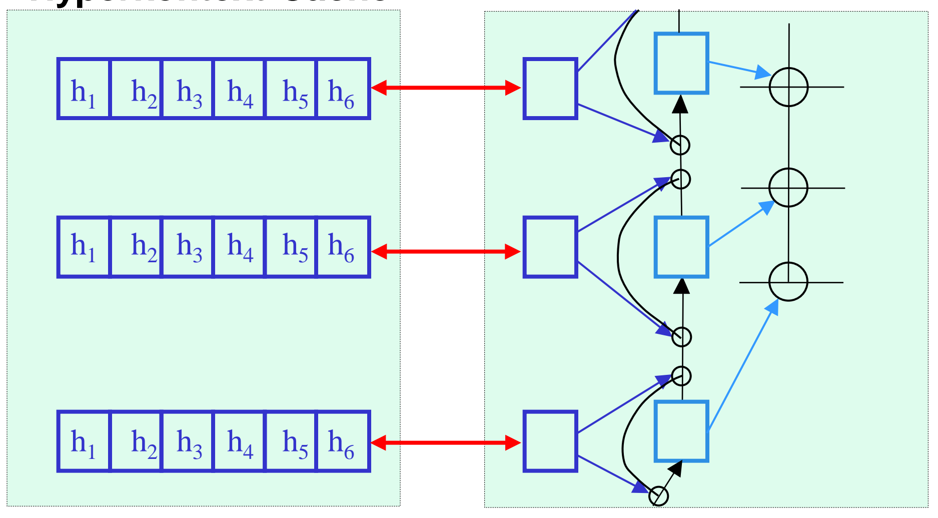
	# Hyperrecon-figurations	Costs	Relative Costs
not hyperrecon-figurable	**-**	**5280**	**100%**
Single Task Model	**30**	**3761**	**71.2%**
4 Task Model	**50**	**2813**	**53.3%**

Cache für Hyperkontexte

Hyperkontext-Cache: Annahmen

1. **Laden vom Cache in den Hyperkontextspeicher und umgekehrt:**
   - bitparallel
2. **Adressierung:**
   - Adresse eines benötigten Hyperkontexts im Cache ist bekannt
3. **Realisierung über MUX oder Schieberegister:**
   - Schieberegister führt zu unterschiedlichen Ladezeiten

Hyperkontext-Cache



○ **Schalter kontrolliert durch SRAM-Zellen** □ **SRAM-Zelle** → **Rekonfigurations-kette** → **Kontrollfluß**

Cache für Kontexte

Literatur: Multi Context FPGAs [Hauck et al.,1999; Li et al.,2000, Chong et al. 2005]

Beachte, ob Kontexte im Voraus bekannt sind oder erst zur Laufzeit
→ Algorithmus definiert Folge von Ressourcenanforderungen $c_1,\dots,c_m$ und Folge von Kontexten $k_{i_1},\dots,k_{i_n}$

Cache-Modell: Annahmen

1. **Gleichförmige Einträge**:
   - jeder Eintrag hat Länge n → nur maximal große Kontexte möglich
2. **Ungleichförmige Einträge sind möglich**:
   - Kontexte verschiedener Länge können eingetragen werden

Laden in den Cache:

1. **Vom Kontext-Speicher möglich**:
   - bei gleichförmigem Cache werden die Bits parallel übertragen
   - bei ungleichförmigem Cache werden die Bits sequentiell übertragen
2. **Von außen notwendig**:
   - nur maximale Kontexte können übertragen werden

Cache für Kontexte

Auswirkungen des Hyperkontexts auf das Laden/Auslesen des Kontextes vom/in den Cache: 2 Möglichkeiten

1. **Hyperkontext bestimmt:**
   - nur solche Bits werden übertragen, die auch im Hyperkontext sind

2. **Hyperkontext umgehend:**
   - sämtliche Bits werden übertragen

Erweitern von Kontexten (d.h. erweitert im Vergleich zum Hyperkontext) im gleichförmigen Cache-Modell: 2 Möglichkeiten

1. **nicht möglich**
   - neuer Kontext muss bereits maximal auf den Chip geladen werden

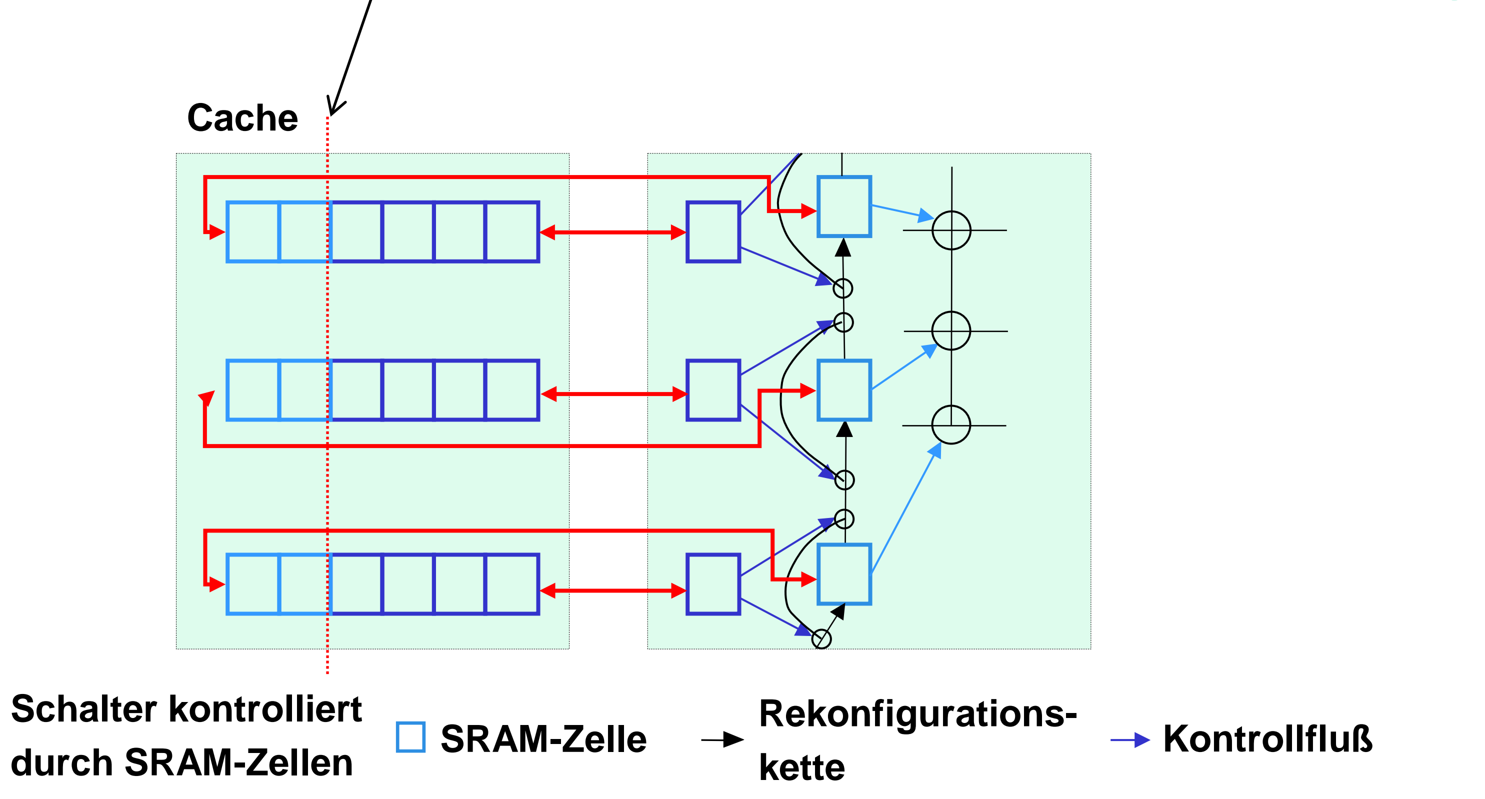
   Alternativen für die Realisierung:
     - Hyperkontext wird beim Laden von außen umgangen
     - Hyperkontext muss beim Laden von außen immer maximal sein

2. **möglich beim Einlesen in den Cache vom Kontext-Speicher**
     - Alternativen zur Behandlung der undefinierten Bits (d.h. der Bits, die nicht im Hyperkontext sind)
       - lade Default-Werte (Null oder Eins)
       - vorherige Werte an entsprechenden Stellen werden übernommen

Cache für Hyperkontexte + Kontexte

Möglicher Aufbau eines gemeinsamen Caches für Hyperkontexte und Kontexte:

1. **Gleichförmige Einträge**
2. **Verschiebbare Grenze zwischen Kontext- und Hyperkontexteinträgen**



Kostenmodell mit Cache und PHC Problem

Annahmen zu Kosten:

1. **Laden von Hyperkontexten und Kontexten aus dem Cache:**
   - konstante und jeweils gleiche Kosten oder (alternativ)
   - variable Kosten: maximale Kosten sind gleich der Anzahl der (Hyper)Kontexte im Cache (sinnvoll bei Shift-Register Organisation)

2. **Erstmaliges Laden von erweiterten Kontexten in den Cache :**
   - Kosten n (= Hyperrekonfiguration ohne Cache)

Satz: Das **PHC-Switch Problem** für hyperrekonfigurierbare Maschinen mit Hyperkontext-Cache ist NP-vollständig.

Bemerkung: Der Satz gilt für jede(!) mögliche Cache-Ersetzungsstrategie

Heuristik

Annahmen:

- Kosten für Laden des Hyperkontexts aus dem Cache sind Null
- Der Hypercache ist so groß, das er alle Hyperkontexte aufnehmen kann

Beobachtung: Für die Rekonfigurationskosten kommt es (unter obigen Annahmen) nur darauf an, welche Rekonfigurationen unter einem Hyperperkontext stattfinden, aber nicht wann diese stattfinden

$\Rightarrow$ PHC-Switch Problem ist äquivalent zu folgendem Problem

Permutation PHC-Problem for the Switch Model (Perm-PHC-Switch):

⇨ **Given**: Task T and a sequence of context requirements $C=c_1,\ldots,c_n$

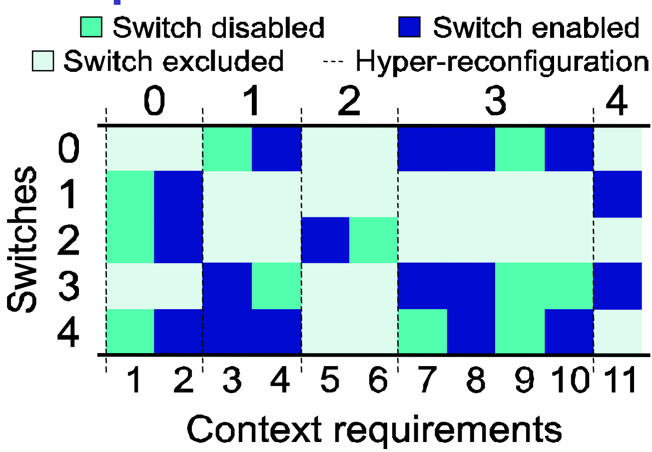
⇨ **Find**: Permutation π of (1 2 … n) and a Partition of $c_{\pi(1)},\ldots,c_{\pi(n)}$ into substrings $S_1,\ldots,S_r$ and hypercontexts $h_1,\ldots,h_r$ such that $S_i \subset h_i$ and the following costs are minimal

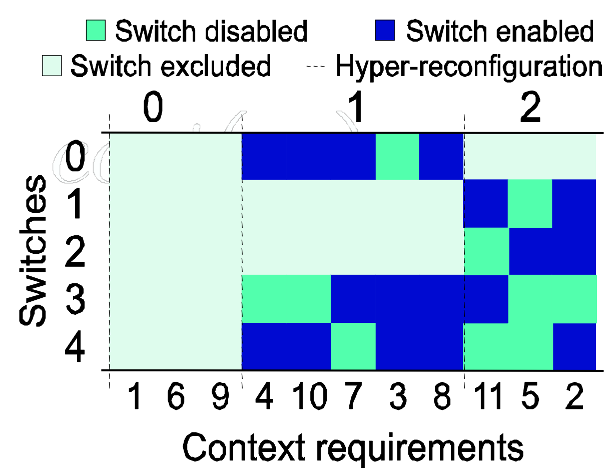
$$r \cdot n + \sum_{i=1}^{r} |h_i| \cdot |S_i|$$

Theorem: The Perm-PHC-Switch problem is NP-complete.

Heuristik

Beispiel:

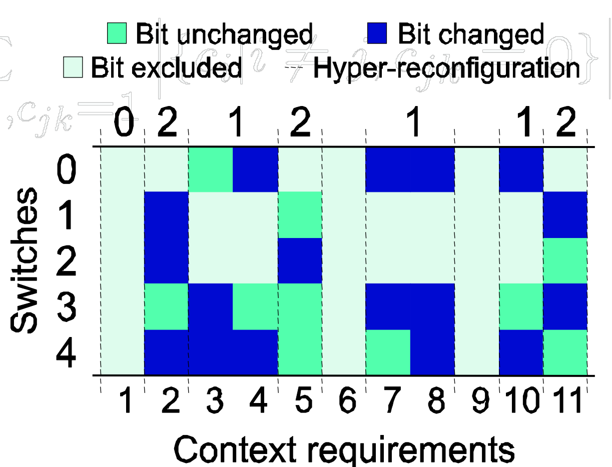




optimale PHC-Switch Lösung (ohne Cache)

optimale Perm-PHC-Switch Lösung

Daraus ergibt sich als optimale Lösung für eine Architektur mit Hypercache der Größe ≥3:



Heuristik für das Perm-PHC-Switch Problem:

Algorithm 1 Insertion sort inspired heuristic for the Perm-PHC-Switch problem

1: $S'_1 = \{c_1\}$
2: **for all** $i \in \{2 \ldots |S|\}$ **do**
3:     **for all** $pos \in \{1 \ldots i+1\}$ **do**
4:         Insert c_i in S'_{i-1} at position pos to obtain S'_{i_j}
5:         Solve PHC-Switch problem for S'_{i_j} and calculate $cost(S_{i_j})$
6:     **end for**
7:     $k = \arg\left(\min_{j=1}^{i+1}\left(cost(S_{i_j})\right)\right)$
8:     $S'_i = S'_{i_k}$
9: **end for**
10: $S_{result} = S'_{|S|}$

Beachte: Heuristik nimmt an, dass der Cache groß genug ist um alle Hyperkontexte aufnehmen zu können

→ ansonsten sind Anpassungen der Heuristik nötig

Heuristik

Heuristic for the Perm-PHC-Switch Problem:

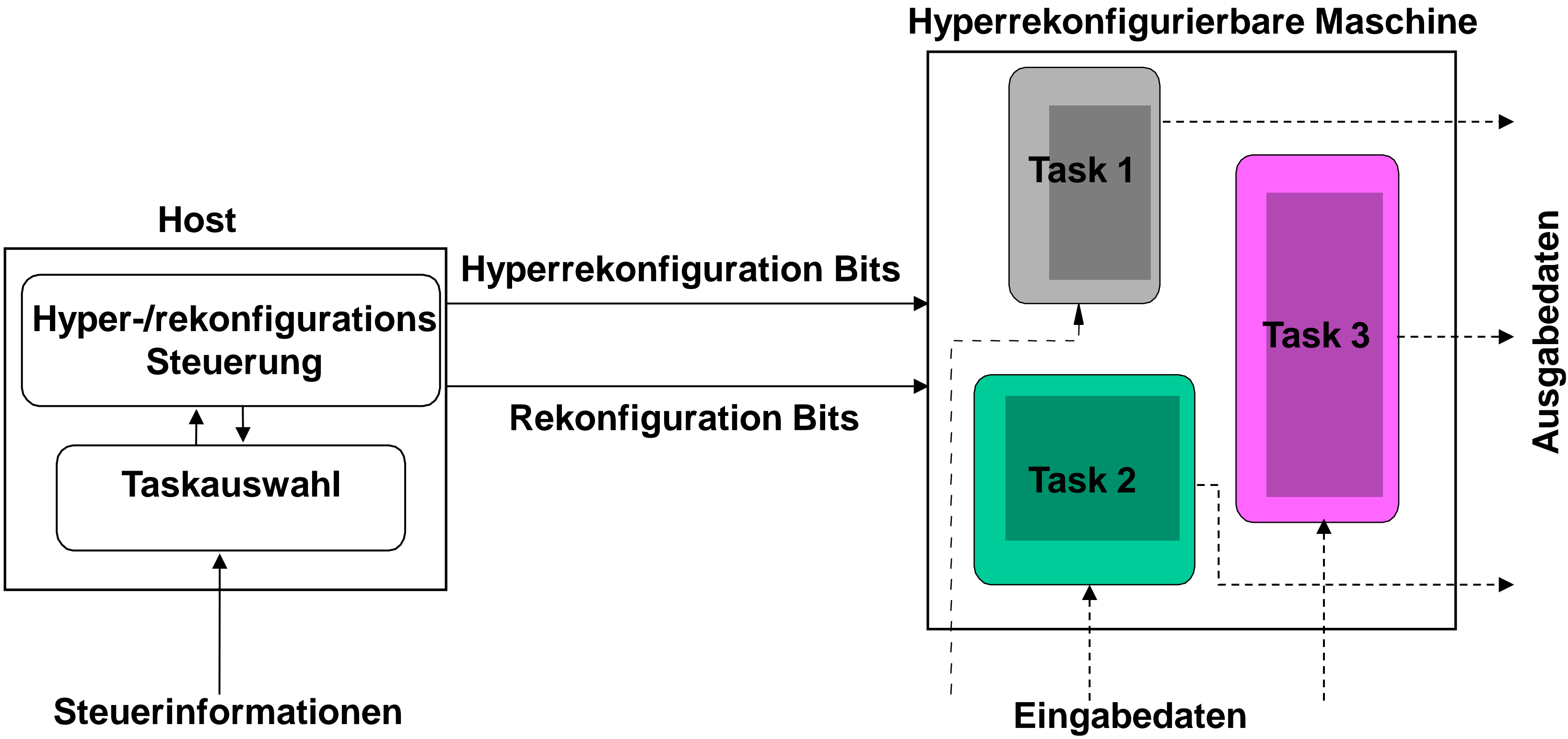
		1-Level	2-Level PHC-Switch	2-Level + Cache Heuristic
FIR	Cost	8000	5078 (63.5%)	3932 (49.2%)
	# H		64	8
Controller	Cost	20304	11024 (54.3%)	9206 (45.3%)
	# H		20	9
Counter	Cost	5280	3761 (71.2%)	2122 (40.2%)
	# H		30	4
LEDDec	Cost	34720	6925 (20.0%)	6820 (19.6%)
	# H		14	11

Beispiele: FIR-Filter, Controller, Counter, LED-Decoder
jeweils auf der SHyRA

1-Level = nur „normal“ rekonfigurierbar, 2-Level = hyperrekonfigurierbar
2-Level + Cache = hyperrekonfigurierbar mit Cache genügender Größe

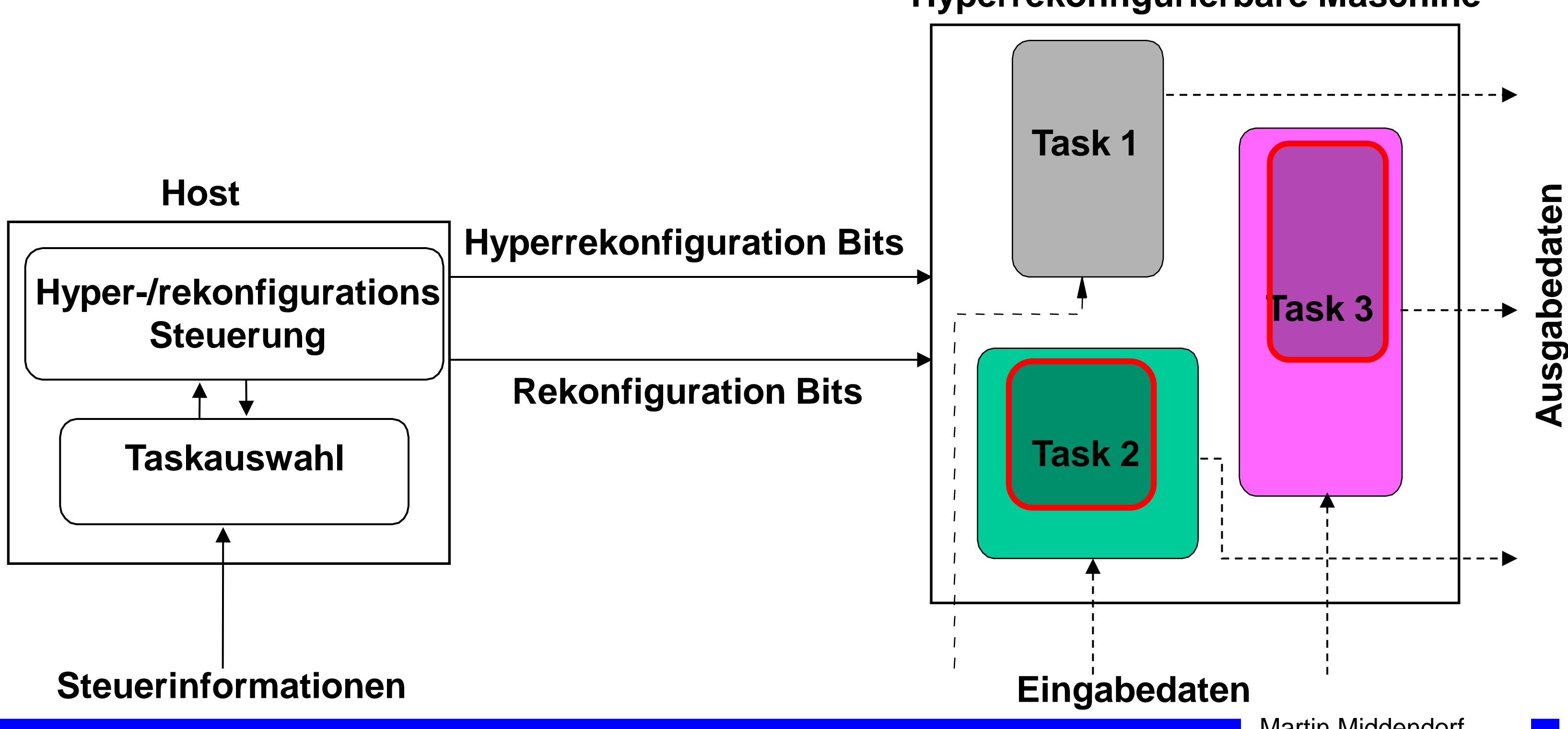
Control Systems (1/2)

- Hostrechner steuert (Hyper)rekonfiguration
- Control Tasks ändern sich um auf Eingaben zu reagieren



Control Systems (2/2)

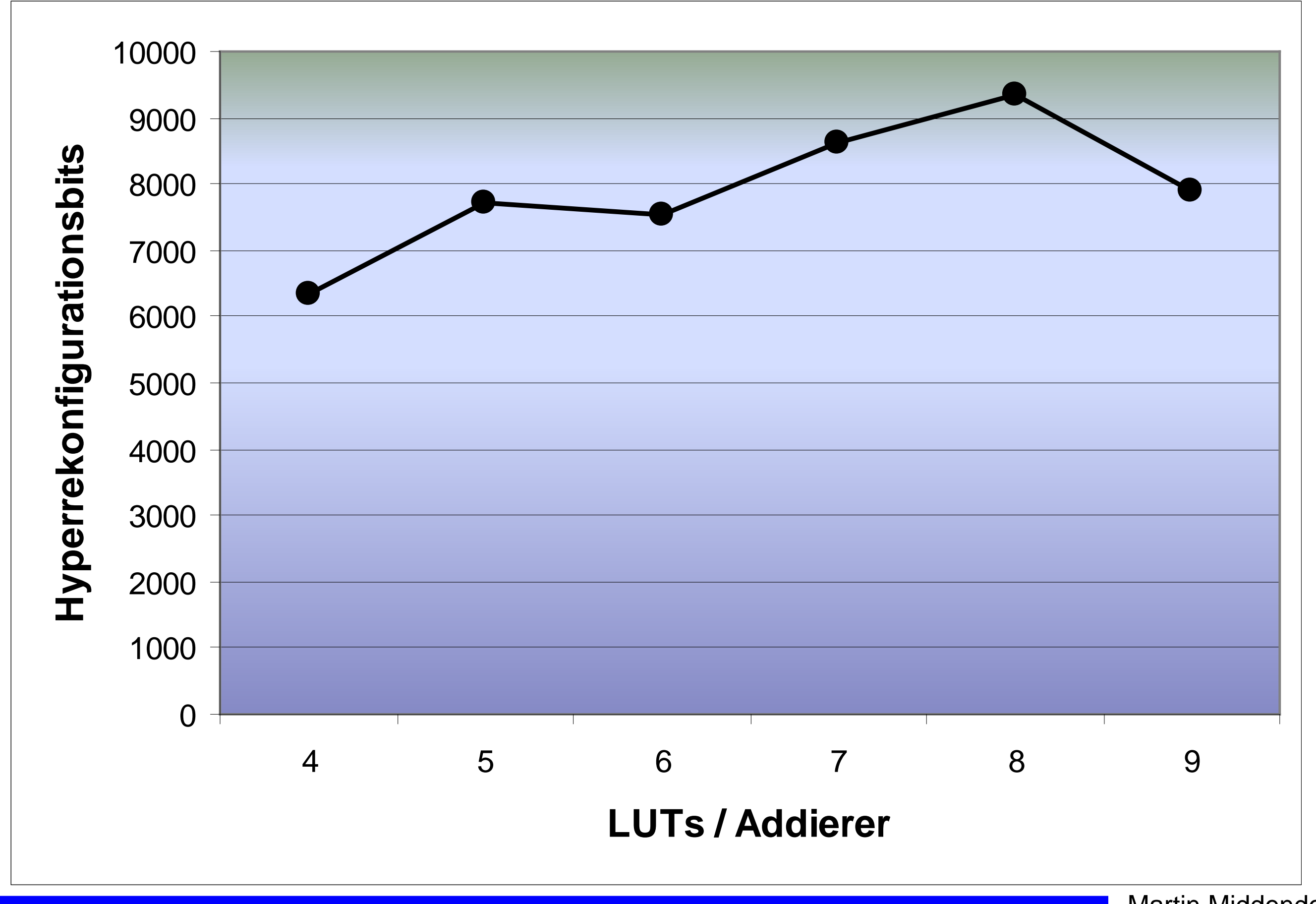
- Geschwindigkeitsänderung eines Tasks (hier Task 1) allein durch Anpassung der Hyperkontexte anderer Tasks (hier Tasks 2 + 3) möglich
- Verwendung von Designalternativen
- Task 1 beschleunigt, da Hyperkontexte von Task 2 und 3 durch neu geladene Designalternative kleiner geworden sind → dadurch sind die Rekonfigurationen schneller



Ergebnisse (4/4)

Zähler läuft parallel zu jeweils verschiedenen Designalternativen eines 4-Bit Addierers

- gesamte (Hyper)rekonfigurationskosten über 4 Zählerzyklen (in Abhängigkeit der Anzahl der LUTs, die der Addierer verwendet)



Multi-level Rekonfiguration

Multi-level Reconfiguration: Extension of hyperreconfiguration to $r \geq 2$ levels

Assumptions:

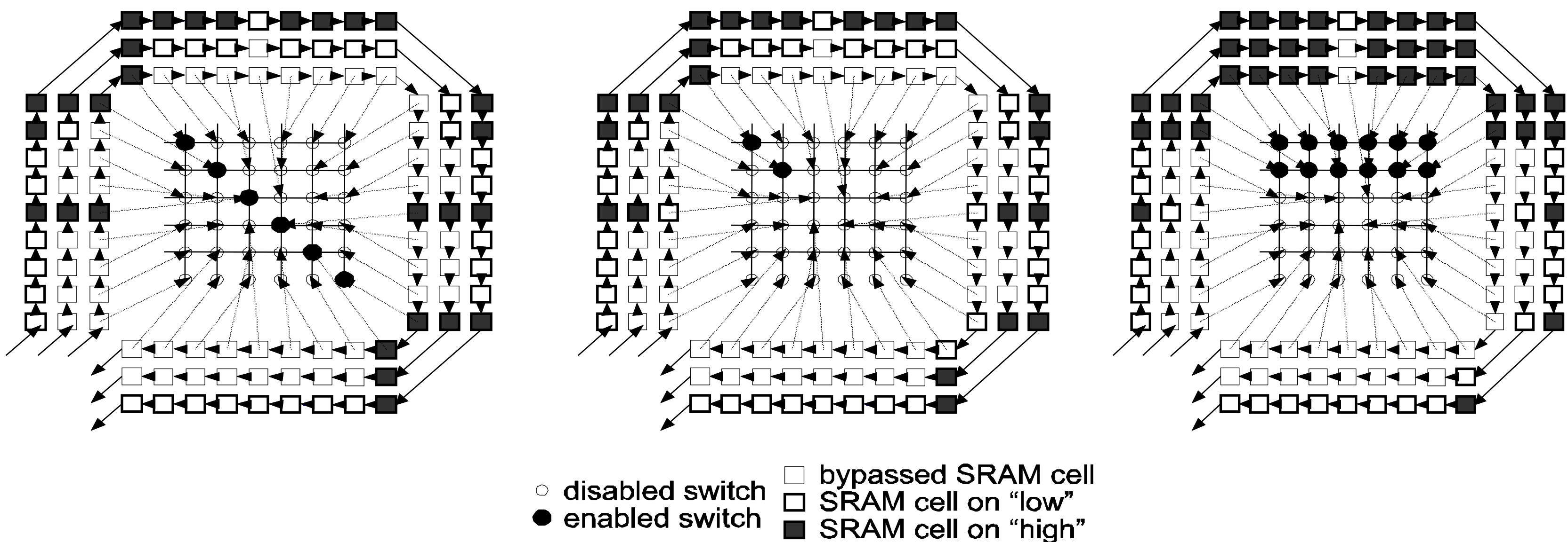
- If a k-level reconfiguration is done then a (k-1)-level reconfiguration is done directly afterwards, $2 \leq k \leq r$
- An r-level reconfiguration is always done at the start of a task

Switch model:

- r configuration chains each with n SRAM-cells
  → kth-configuration chain defines k-level context, $k \in [1:r]$
- content of s-th SRAM cell on level k defines whether the s-th SRAM cell on level k-1 is included in the (k-1)-level reconfiguration chain or not (i.e. it is shortcut), $2 \leq k \leq r$
- If s-th SRAM cell is included in the k-level reconfiguration chain then the s-th SARM cell is included on all levels $k+1,\ldots,r$ for $1 \leq k < r$

Multi-level Rekonfiguration

Example: 3-level reconfigurable switchbox



Given: An algorithm as a sequence of context requirements $c_1,\ldots,c_6$

$c_1=c_2=\{s_{11},s_{22},s_{33},s_{44},s_{55},s_{66}\}$ $c_3=c_4=\{s_{11},s_{22}\}$ $c_5=c_6=\{s_{11},\ldots,s_{16},s_{21},\ldots,s_{26}\}$

3rd-level rekonfiguration is done before c_1

2nd-level reconfiguration is done before c_1 and c_5

Multi-level Rekonfiguration

Theorem: The PHC-Switch problem can be solved exactly on multi-level reconfigurable architectures with $r \geq 2$ reconfiguration levels in polynomial time $O(nm^2 + m^2(m+n)(r-2) + m^3)$.

Reconfiguration Level Problem (RLP): For a given sequence of context requirements find the optimal number of reconfiguration levels so that the corresponding optimal solution of the PHC problem is minimal.

Observation: Level k cannot be useful if for the solution of the PHC-Switch problem a reconfiguration on level k is always done when a reconfiguration on level $k-1$ is done.

By assumption there is at least one r-level reconfiguration at the beginning.

→ $m+1$ is a simple upper bound for the best number of reconfiguration levels (where m is the number of given context requirements of the algorithm)

Corollary: RLP is solvable for multi-level reconfigurable architectures in polynomial time $O(nm^4 + m^5)$.

Multi-level Rekonfiguration

Heterogeneous Multi-level Reconfiguration: similar to multi-level reconfiguration but the number of reconfiguration levels can be different for each SRAM cell.

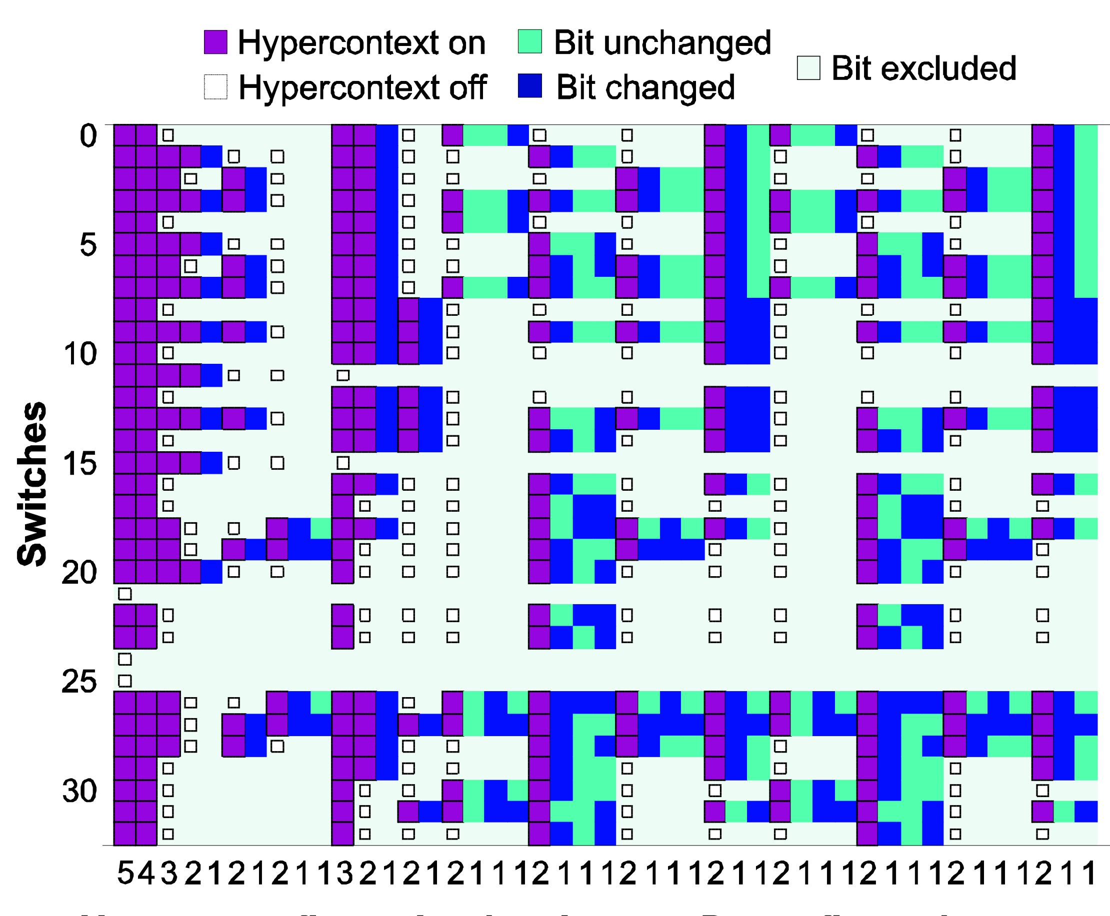
Heterogeneous Reconfiguration Level Problem (H-RLP): for a given sequence of context requirements find for each SRAM cell the optimal number of reconfiguration levels so that the corresponding optimal solution of the PHC problem becomes minimal.

Theorem: H-RLP is NP-hard even when the maximal number of reconfiguration levels is 2.

Multi-level Rekonfiguration

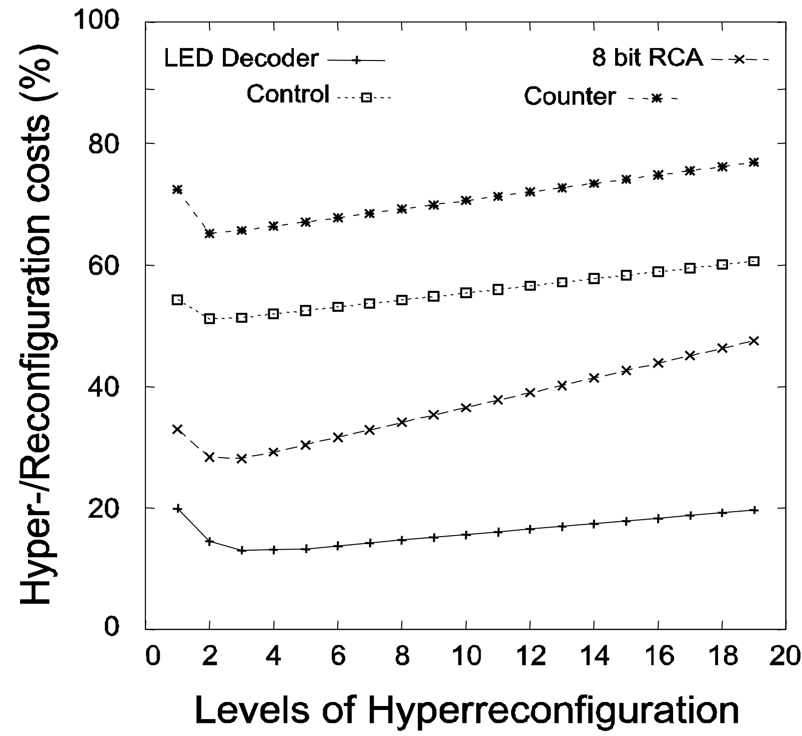
Example: **4 bit counter circuit on a 5-level reconfigurable SHyRA with 2 LUTs, 8 registers, 48 reconfiguration bits**

Counter was executed 10 times which requires a total number of 110 1-level reconfiguration operations



Multi-level Rekonfiguration

Examples:	Optimal number of reconfiguration levels	Reconfiguration cost	Cost reduction compared to 1-level reconfiguration
Adder	3	1245	71.86%
Counter	2	3443	34.79%
LED Decoder	3	4527	86.96%
Control	2	10385	48.85%

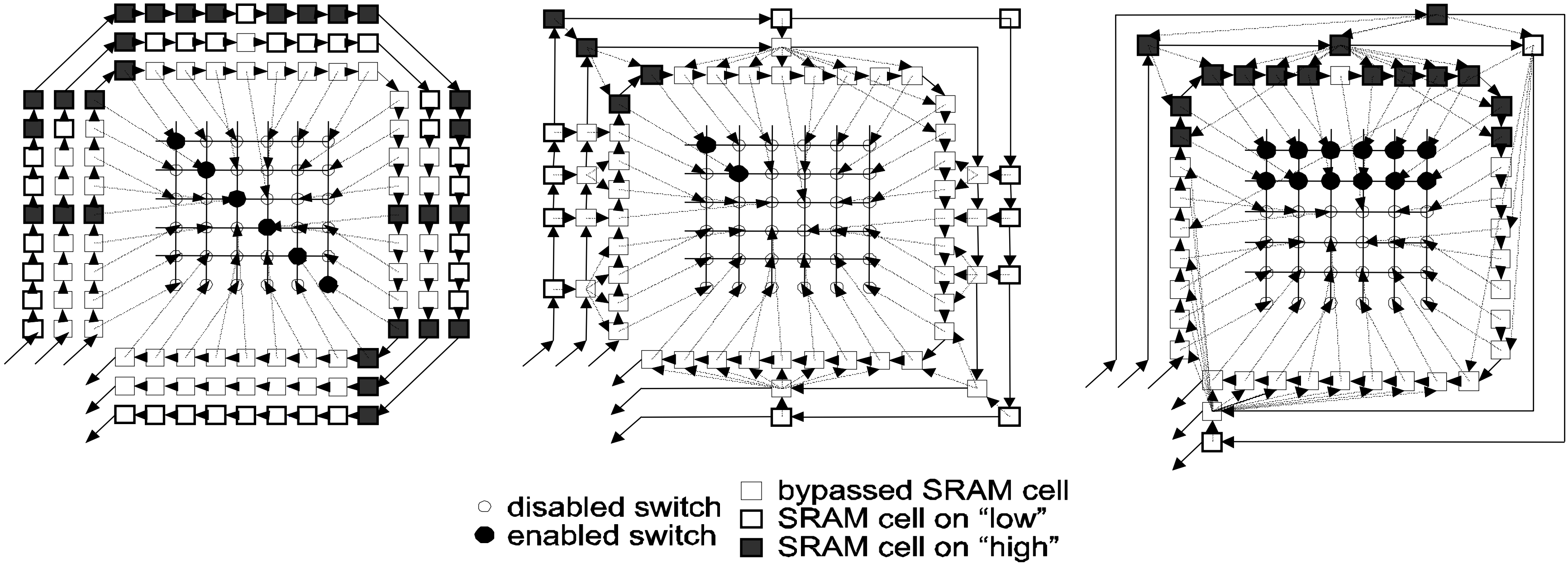


Granularität

Aim: Reduce the number of additional SRAM cells that are used for mutli-level reconfiguration

Idea: Let one SRAM cell on level *k* determine for several SRAM-cells on level *k-1* whether they are included in the (*k-1*)-level reconfiguration chain or shortcut

Example:



Granularität

Hypercontext Design problem for the multi-level reconfigurable Switch-model (Multi-HD-Switch):

Given: An algorithm as a sequence of context requirements $c_1,\ldots,c_m$ for a set of n switches $X^{(1)} = \{x_1^1, \ldots, x_n^1\}$

Find: A multi-level reconfigurable machine with r levels of reconfiguration in the Switch-model with sets of SRAM-cells

$$X^{(k)} = \{x_1^k, \ldots, x_{n_k}^k\}, k \in [1, \ldots, r],\ k \in [2 : r]$$

for chosen numbers $n \geq k_2 \geq \ldots \geq k_r \geq 0$, and define a partition π_k of $X^{(k)}$, $k \in [1,\ldots,r-1]$ into n_{k+1} sets $X_1^{(k)}, \ldots, X_{n_{k+1}}^{(k)}$ such that the optimal total reconfiguration cost for the given algorithm are minimized when executed on the defined architecture.

The **partition** of $X^{(k)}$ into n_{k+1} subsets defines then an **assignment** of the j-the SRAM cell on level $k+1$ to all SRAM cells in the j-th subset of the partition.

This means that either

- all cells in the j-th subset of the partition are included in a context (when the j-th SRAM cell on level $k+1$ is set to 1) or
- all are excluded (when the j-th SRAM cell on level $k+1$ is set to 0).

Granularität

Problem: It might be difficult to connect an SRAM cell on level *k* to a subset of the SRAM cells on level *k-1* because

- long wires might be necessary for the connections
- wires might be necessary through areas on the chip which are occupied

The following variant of **Multi-HD-Switch** might be easier to realize

Multi-Interval-HD-Switch (**Multi-I-HD-Switch**): same as Multi-HD-Switch but with the following additional restriction

- indices of the switches in each set of the partition $X_1^{(k)}, \ldots, X_{n_{k+1}}^{(k)}$ form a subinterval of $[1,\ldots,n_k]$.

Theorem: Multi-HD-Switch und Multi-I-HD-Switch are NP-hard even when the maximal number of reconfiguration levels is 2.

Theorem: With the additional requirement that for all levels the time steps when (hyper)reconfiguration operations are done are known and fixed then Multi-I-HD-Switch can be solved optimally in polynomial time.

Granularität

Beispiel: Vergleich der minimalen (Hyper)rekonfigurationskosten (cost) für Switch, I-HD-Switch und HD-Switch-Modell für 3 Testschaltungen auf der SHyRA

	Reconf.	Switch	I-HD-Switch	HD-Switch
Switchbox	Cost	644(89.4%)	487(67.6%)	404(56.1%)
	#HR	8	35	41
	#Sets	N/A	5	4
Adder	Cost	581(48.3%)	554(46.0%)	444(36.9%)
	#HR	5	5	7
	#Sets	N/A	36	12
Counter	Cost	2145(87.1%)	2025(82.2%)	1740(70.6%)
	#HR	2	32	49
	#Sets	N/A	11	6

Prozentzahlen bezogen auf Kosten bei “normaler” Rekonfiguration (d.h. keine Hyperrekonfiguration)

#HR = Anzahl der Hyperrrekonfigurationen (#HR)

#Sets = Anzahl der SRAM-Zellen auf Ebene 3 (#Sets)

Granularität

Problem: Multi-level reconfiguration can reduce the total reconfiguration costs when measured by the number of necessary reconfiguration bits but it also increases the necessary amount of hardware (SRAM cells, wires)

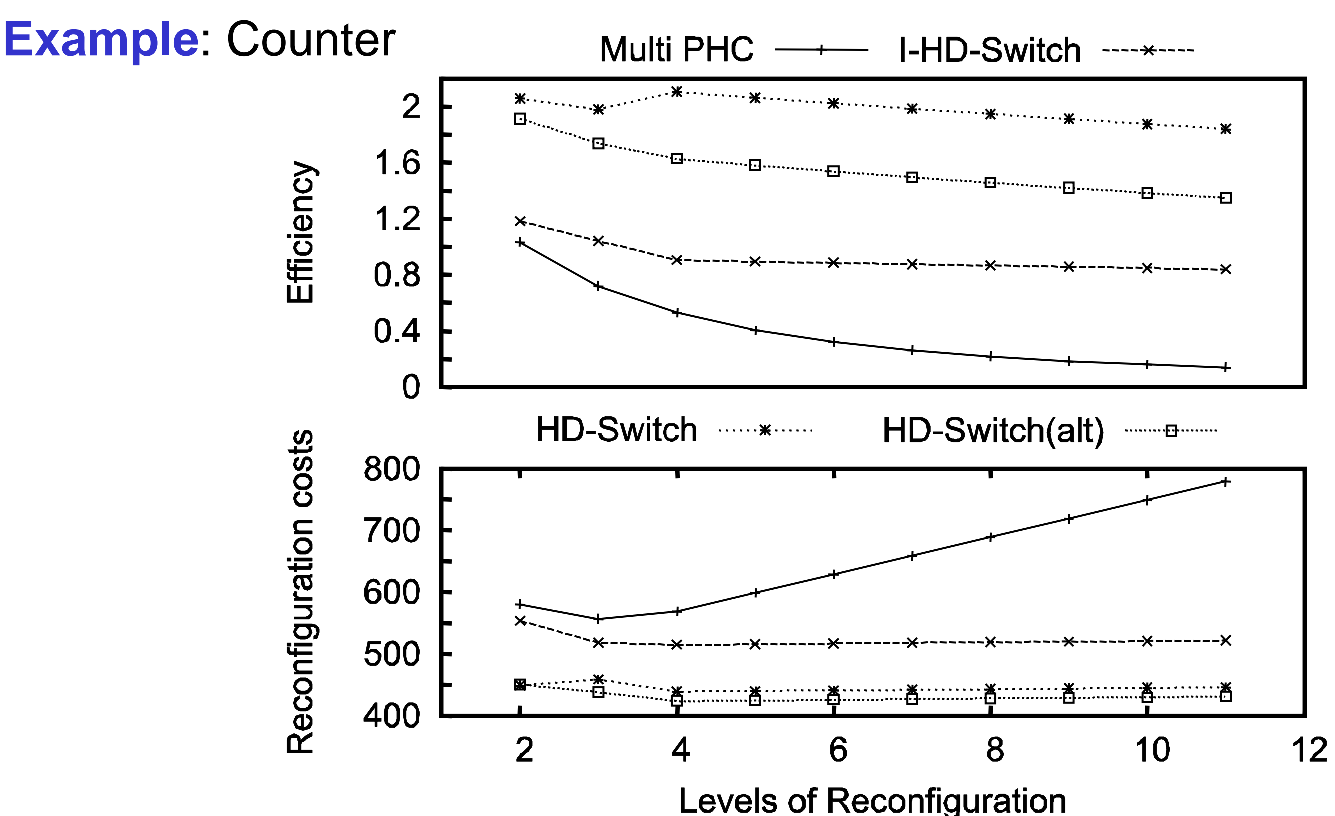
Efficiency of multi-level reconfiguration:

$$\varepsilon = \Delta p / \Delta r$$

- Δp is the reduction of reconfiguration costs compared to reconfiguration costs of one level reconfiguration
- Δr is the increase in the amount of necessary resources (measured in number of SRAM cells) compared to one level reconfiguration

Granularität

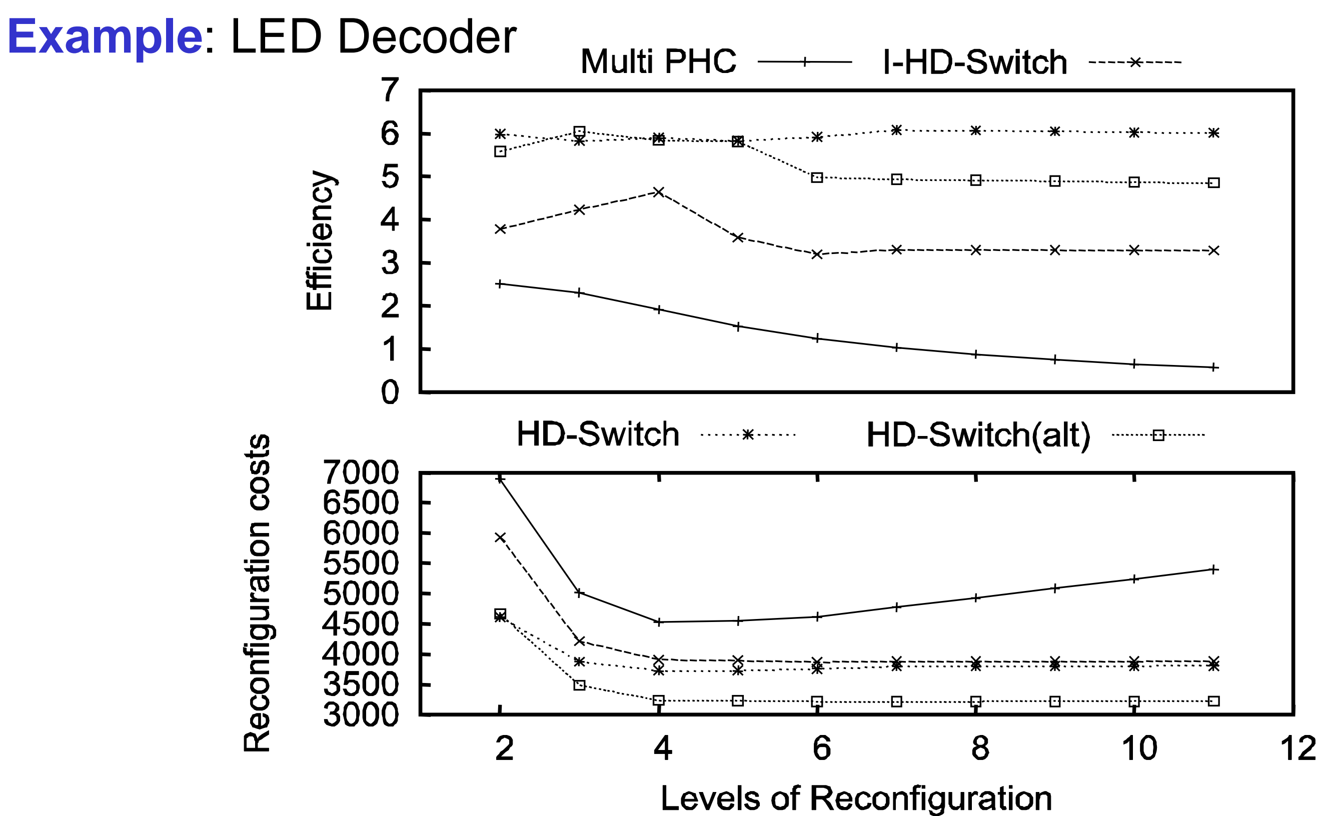
Example: Counter



Multi-PHC: Results are for the case of a same number of SRAM cells on each level
I-HD-Switch: Results correspond to the optimal solution (assuming the reconfiguration steps are the same as for the Multi-PHC solution)
HD-Switch and HD-Switch-(alt): Results are for two heuristics to solve the Multi-HD-Switch problem

Granularität

Example: LED Decoder



Multi-PHC: Results are for the case of a same number of SRAM cells on each level

I-HD-Switch: Results correspond to the optimal solution (assuming the reconfiguration steps are the same as for the Multi-PHC solution)

HD-Switch and HD-Switch-(alt): Results are for two heuristics to solve the Multi-HD-Switch problem

Zusammenfassung Hyperrekonfiguration

- **Ziel**: Verringerung des Aufwandes (in Anzahl der nötigen Rekonfigurationsbits) von dynamischer Rekonfiguration durch selektive Auswahl der zu beschreibenden Ressourcen

  - **Formales Modell** von Mehrebenen-Rekonfiguration
  - Zielarchitektur neutral (hierarchisch, linear, Arraystruktur)
  - Beispielmodell(e): **Switch-Modell** (und DAG-Modell)
  - Modell eines Algorithmus als **Folge von Ressourcenanforderungen**
  - **Kostenmodell** und zugehörige Optimierungsprobleme (PHC und Varianten)
  - Explizite Behandlung **mehrerer Tasks**
  - **Cache** für Hyperkontexte
  - Unterschiedliche **Granularität** auf den einzelnen **Rekonfigurationsebenen** möglich

Ende Teil 1 der Vorlesung