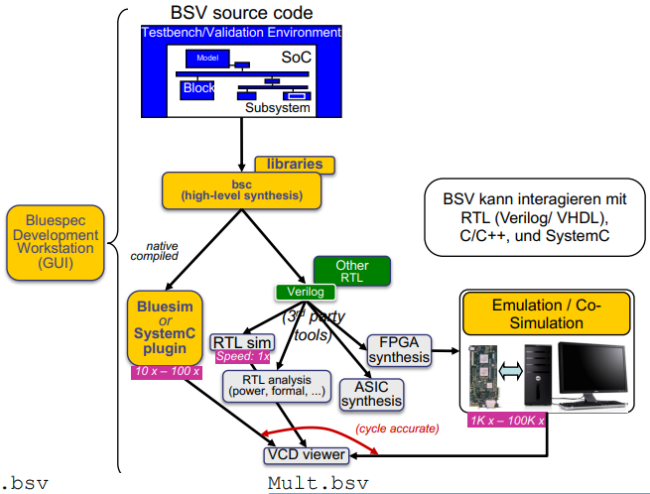
Zusammenfassung AER

Inhalt

Einführung in Hardware-Beschreibungssprache BlueSpec

- BlueSpec ist eine Weiterentwicklung der HDL Verilog (BSV = BlueSpec System Verilog)
- Verbindet Verhaltens- mit Strukturbeschreibung und bietet zudem Synthese



Verhaltensbeschreibung (Regeln für atomare Transaktionen)

- Zur Formulierung von parallelen Abläufen
- Basiert auf Theorie von Term-Ersetzungs-Systemen

Strukturbeschreibung (an Haskell angelehnt)

- Ausdrucksstarkes Typsystem
- Strenge Typprüfung
- Mächtige Parametrisierung

Testbench.bsv

```
package Testbench;

import Mult :: *;  // alles aus Package Mult importieren

module mkTestbench (Empty);
  Mult_ifc m <- mkMult;

  rule gen_x;
    m.put_x (9);
  endrule

  rule gen_y;
    m.put_y (5);
  endrule

  rule drain;
    let w <- m.get_w ();
    $display ("Produkt = %d", w);
    $finish ();
  endrule
endmodule: mkTestbench

endpackage: Testbench
```

Mult.bsv

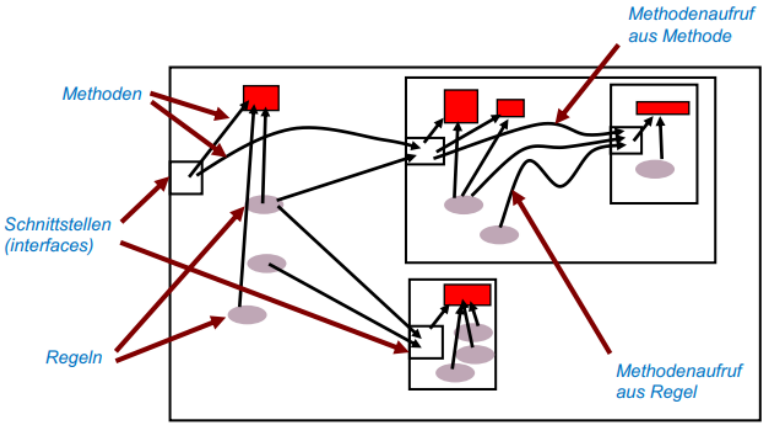
```
package Mult;

interface Mult_ifc;
  method Action put_x (int xx);
  method Action put_y (int yy);
  method ActionValue #(int) get_w ();   // w = xx * yy
endinterface: Mult_ifc

module mkMult (Mult_ifc);
  Reg #(int)  w    <- mkRegU;
  Reg #(int)  x    <- mkRegU;
  Reg #(int)  y    <- mkRegU;
  Reg #(Bool) got_x <- mkReg (False);
  Reg #(Bool) got_y <- mkReg (False);

  rule compute ((y != 0) && got_x && got_y) ;
    if (lsb(y) == 1) w <= w + x;
    x <= x << 1;
    y <= y >> 1;
  endrule

  method Action put_x (int xx) if (! got_x);
    x <= xx; w <= 0; got_x <= True;
  endmethod

  method Action put_y (int yy) if (! got_y);
    y <= yy; got_y <= True;
  endmethod

  method ActionValue #(int) get_w () if ((y == 0) && got_x && got_y);
    got_x <= False; got_y <= False;
    return w;
  endmethod
endmodule: mkMult

endpackage: Mult
```

Vorgehen

- Interface schreiben, um Eingaben und Ausgaben zu definieren
- Benutzung (z.B. Testbench)
- Verhalten (Definition des algorithmischen Verhaltens)

Entwurfshierarchie

- Zerlegungshierarchie von Modulen (Verilog, System-Verilog, SystemC)
- Blätter der Hierarchie sind primitive Zustandselemente (Register, Warteschlangen (FIFOs),…)
- Unterschied zu Verilog: Auch Register sind Module



Regeln und Schnittstellenmethoden

- Module stellen Schnittstellenmethoden bereit und enthalten Regeln, die Methoden anderer Module aufrufen
- Methoden können auch Methoden anderer Module aufrufen

Instanziierung:

- interface_type instance_name <- module_name (module_parameters);

Bezeichner:

- Variablen, Typvariablen und Methoden beginnen mit Kleinbuchstaben
- Konstanten und Typen beginnen mit Großbuchstaben
- Konvention für Modulnamen

- Traditioneller Präfik „mk“ – gelesen „make“ (Beispiel: mkMultiply, mkALU)

Methodendeklarationen

- Wert-Methoden (value methods)
  - Entsprechen mathematischen Funktionen
  - Können Zustand der Schaltung nicht ändern
  - Können lokale Zwischenwerte berechnen (=)
  - Haben Rückgabewert
- Aktions-Methoden (action method)
  - Können Zustand der Schaltung ändern (<=)
  - Haben keinen Rückgabewert
- Aktionswert-Methoden (action value method)
  - Können Zustand der Schaltung ändern (<=)
  - Haben Rückgabewert

```
method int foo (int x, int y, int z);
  let sum = x + y;
  return sum + z;
endmethod
```

```
Reg#(int) sum <- mkReg(0);
method Action inc(int x);
  sum <= sum + x;
endmethod
```

```
Reg#(int) sum <- mkReg(0);
method ActionValue(#int) inc2(int x);
  sum <= sum + x;
  return sum*2; // benutzt alten Wert
endmethod
```

Bedingungen

- Entscheiden über Bereitschaft von Regel oder Methoden zur Ausführung. Default: True
- Methoden werden (ggf. indirekt) aus Regeln aufgerufen. Regel ist nur bereit, wenn alle aufgerufenen Methoden ebenfalls bereit sind (zusätzlich zur Bedingung an Regel)
- Bereitschaft der Regel: CAN_FIRE (logisches Und (Konjunktion) von Regelbedingungen und Bedingungen an alle aufgerufenen Methoden)

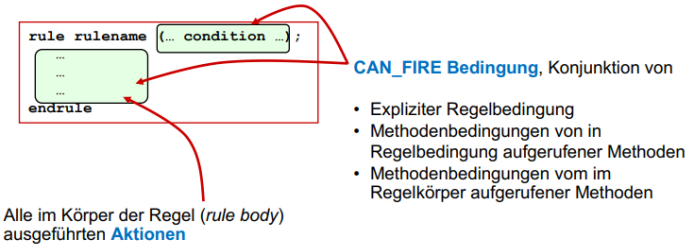
```
module mkMult (Mult_ifc);
   Reg #(int)  w     <- mkRegU;
   Reg #(int)  x     <- mkRegU;
   Reg #(int)  y     <- mkRegU;
   Reg #(Bool) got_x <- mkReg (False);
   Reg #(Bool) got_y <- mkReg (False);

   rule compute ((y != 0) && got_x && got_y);
      if (lsb(y) == 1) w <= w + x;
      x <= x << 1;
      y <= y >> 1;
   endrule

   method Action put_x (int xx) if (! got_x);
      x <= xx; w <= 0; got_x <= True;
   endmethod

   method Action put_y (int yy) if (! got_y);
      y <= yy; got_y <= True;
   endmethod

   method ActionValue #(int) get_w () if ((y == 0)
                                          && got_x
                                          && got_y);
      got_x <= False; got_y <= False;
      return w;
   endmethod
endmodule: mkMult
```

Ausführung von Regeln

- Wann immer das CAN_FIRE einer Regel wahr ist, führe die Aktionen im Regelkörper aus

Auffälligkeiten in Simulation von oben gegebenem Modul/Testbench

- Setzung von Steuersignalen, die in der Spezifikation nicht beschrieben waren
- Und eigentlich sollte eine Testbench keine Auswirkung auf die Verhaltens-/Architekturbeschreibung

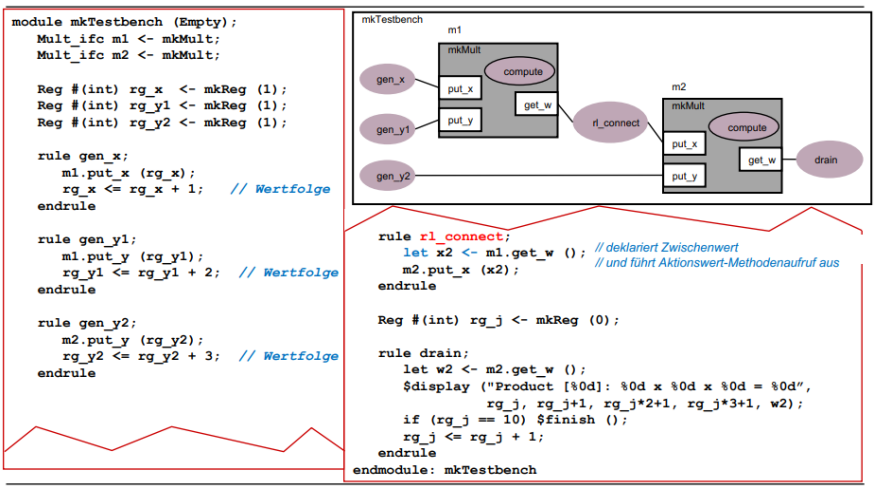


Generierung von Verilog aus BSV

- (* synthesize *) vor module mkFoo (); sorgt dafür, dass die Module in Bluespec erhalten bleiben und nicht inlined werden.
- Synthesize markiert die Grenzen von separaten Verilog-Modulen

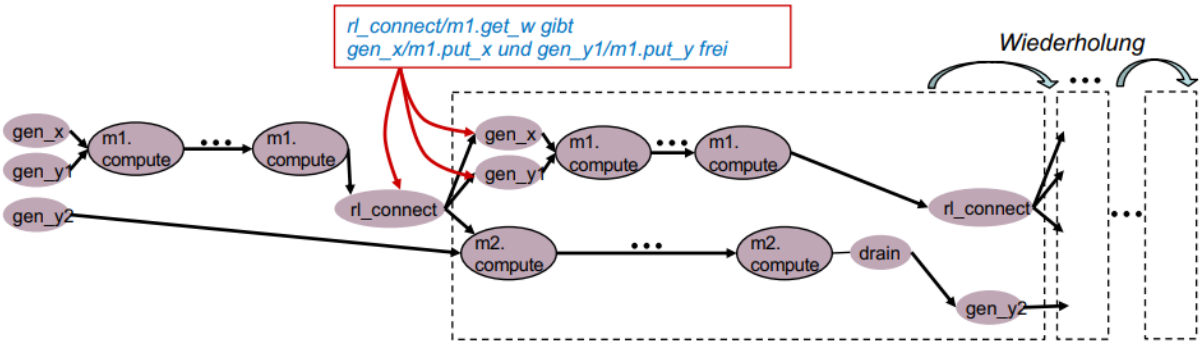
- Synthesize darf nur vor bestimmten Modulen auftauchen, da BSV mächtiger ist als Verilog und die Schnittstelle nach außen nur aus Bits, Skalaren und Bit-Vektoren bestehen darf
- Zwischen den inlined BSV-Modulen dürfen beliebige Schnittstellen gewählt werden
- Dabei verhält sich Verilog zu BSV wie Assembler zu C/C++
- Module in Verilog und BSV – Gemeinsamkeiten und Unterschiede
- Interface-Methoden werden auf Verilog Ports abgebildet
- Formale Methodenparameter -> input Ports
- Methodenergebnisse -> output Ports
- Ausführungsbereitschaft einer Methode -> output Port (RDY_xx == TRUE -> Methode ist bereit)
- Ausführen von Action und ActionValue-Methoden -> input Port (EN_xx == TRUE -> Methode ausführen)
- Optimierungen möglich – eliminiere RDY, wenn Methode immer bereits, eliminiere EN, wenn Methode jeden Takt ablaufen soll
- Clock- und Reset-Signale
- BSV-Beschreibungen haben nur eine Takt und eine Reset, Clock und Reset tauchen nicht im BSV-Quellcode auf
    - Werden in Verilog automatisch erzeugt

Präzedenzrelation



- es existieren Präzedenzrelationen zwischen Regeln/Methoden (Quellknoten gibt (Teil-) Bedingungen an Zielknoten frei)
- Register werden in HW als D-Flip-Flop realisiert, daher haben sie die Präzedenzrelation read < write, Wires hingegen haben write < read und halten ihre Daten nur innerhalb eines Taktes
- Die Relationen implizieren Halbordnung
    - Zwei zueinander (auch transitiv) ungeordnete Regeln können in beliebiger Reihenfolge feuern
- Die vorliegende Kaskade erlaubt Pipelineausführung
    - Dynamische Ausführung
        - Latenz ist nur datenabhängig variabel (je nach Anzahl 1-Bits im Multiplikator)
        - Gegenbeispiel MIPS: immer Fetch-Decode-Execute-Mem-Writeback (statisch)
    - Elastische Ausführung
        - Daten bewegen sich mit unterschiedlichem Fortschritt durch Pipeline (hier ohne Balancing Register, funktioniert hat aber reduzierten Durchsatz)

- Gegenbeispiel MIPS: alle Daten im Gleichschritt (inelastisch/starr)



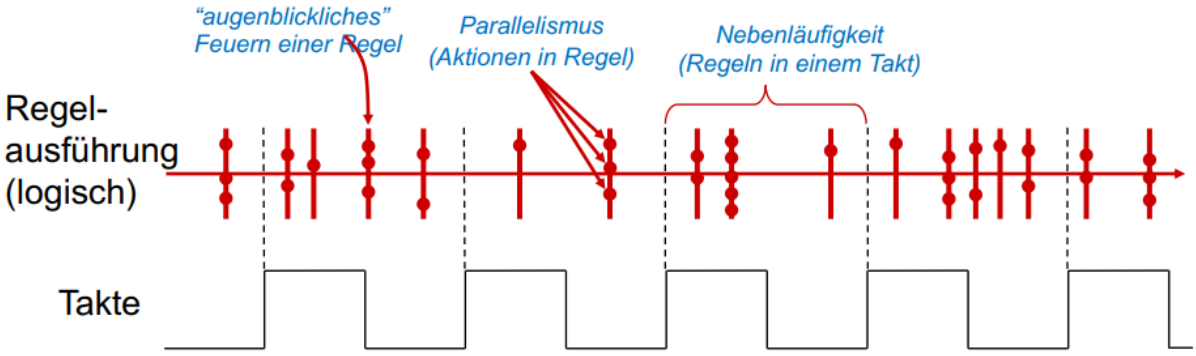
Ausführungssemantik

- Zweistufige Erklärung
  - Semantik einzelner Regeln
    - Parallele Ausführung von Aktionen, innerhalb einer Regel
  - Zusammenspiel mehrerer Regeln
    - Nebenläufige Ausführung mehrerer Regeln in einem Taktzyklus

Parallelität/ Ausführungssemantik für Aktionen innerhalb einer Regel

- Aktionen werden gleichzeitig und augenblicklich (Ablauf in „Nullzeit“) ausgeführt
- Reihenfolge von Aktionen im BSV Quelltext ist irrelevant (Aktionen feuern gleichzeitig)
- Betrachte jede Regel als Datenfluss von
  - Konstanten
  - Ergebnissen von reinen Funktionen (in BlueSpec Wertmethoden)
- Hin zu
  - Parametern für Aktionsmethoden (indirekte Veränderung von Zustandselementen)
- Ausführung von Aktionsmethoden (kurz: Aktionen), kurz: Feuern
  - Wird ausgelöst durch Enable
  - Bedingungen für Feuern werden berechnet als WILL_FIRE
  - Wichtig: Unterschied zwischen CAN_FIRE (Bereitschaft) und WILL_FIRE
- Selbst bereite Aktionen (CAN_FIRE = 1) können durch Bedingungen innerhalb der Regel am Feuern hindern (WILL_FIRE = 0)
- Gelesen wird vor dem Feuern aller Aktionen
- Geschrieben wird nach dem Feuern aller Aktionen -> atomares Aktualisieren des Zustandes
- Probleme bei parallelen Aktionen werden durch bsc Compiler entdeckt
  - Register kann nicht gleichzeitig mit zwei Werten geschrieben werden
  - FIFO kann nicht gleichzeitig zwei Werte einreihen
  - Registerfeld kann nicht gleichzeitig zwei Wert aus dem gleichen Port lesen

- Allerdings sind Register/FIFOs/Registerfelder mit mehr Ports möglich (diese ermöglichen dann auch innerhalb einer Regel mehrere Zugriffe



Nebenläufigkeit (Concurrency)/ Ausführungssemantik für Regeln innerhalb desselben Taktes

- Ermittle korrekte nebenläufige Ausführung aller Regeln
  - Untersuche jede Regel rj, ob sie konfliktfrei zu allen vorangegangen Regel r1 bis rj-1 ist.
  - Wenn dies zutrifft, führe rj als Einzelregel aus
  - Falls nicht, untersuche
    - mA < mB (Methode A müsste vor Methode B ausgeführt werden) -> Regeln können nebenläufig gefeuert werden, wenn die Regel, die mA aufruft logisch eher (CAN_FIRE ist logisch für rB erst erfüllt, wenn rA schon ausgeführt wurde) ausgeführt wird, als die Regel, die mB ausruft
    - mB < mA -> analog zu oben
    - mA konflikt mB -> nicht nebenläufig feuern

Nebenläufige Register

- Können neben der Datenhaltung von Takt zu Takt auch eine Historie von Werten innerhalb eines Taktes führen (Kontrollierte nebenläufige Ausführung von Regeln)
- Ersetzt ältere Mechanismen wie RWire, weil diese potenziell fehleranfälliger sind
- CReg bietet einen Array aus Reg-Schnittstellen, können untereinander nebenläufig betrieben werden
- Präzedenzrelation ports[0]._read < ports[0]._write < ports[1]._read < …
- CReg funktionieren mit beliebigen Ablaufplänen, bei einer Regel pro Takt: CReg == Reg

Konflikte bei Nebenläufigkeit

- ein Konflikt zwischen zwei Regeln r1 und r2 besteht, wenn in einer Instanz x zwischen irgendeinem Paar von Methoden x.m1 in r1 und x.m2 in r2 ein Konflikt besteht
- verschiedene Regeln laufen logisch sequenziell ab (nebenläufig nur, wenn konfliktfrei)
- Regeln laufen **instantan** (alle Aktionen einer Regel zum selben Zeitpunkt), **vollständig** (alle enthaltenen Aktionen einer Regel) und **geordnet** (eine Regel wird entweder vor oder nach, nie gleichzeitig mit einer anderen ausgeführt) -> atomare Ausführung

Allgemeine Regeln bei der Ausführung:

- Wähle eine freigegebene Regel aus und führe sie aus
- Jede Regel feuert maximal einmal während eines Taktes
- Konfliktbehaftete Regeln können nicht im gleichen Takt feuern

Häufige Ursachen für Konflikte:

- Zustandselemente können nur einmal je Takt umschalten (rule ordering conflict)
- Hardware-Ressourcen (z.B. Drähte) können nur einmal je Takt benutzt werden (rule ressource conflict)

Tatsächliches Vorgehen:

- Wähle willkürlich eine Regel zum Feuern aus und sperre alle – mit dieser Regel in Konflikt stehenden Regeln. Compiler gibt allerdings Warnung aus
- Bsc wählt Ablaufplan mit höchstem Grad an Nebenläufigkeit aus (möglichst viele Regeln/Takt,möglichst wenig Takte zur gesamten Ausführung)
- Falls
  - immer auftretende Konflikte erkannt werden: Entfernen aller konfliktbehafteten Regeln, alle übrigen werden in Hardware umgesetzt
  - Konflikte nicht immer auftreten: Führe konfliktfreie Regeln aus und erzeuge Hardware, um Ausführung noch konfliktbehafteter Regeln in diesem Fall zu unterbinden
- Benutzereingriff zur Auswahl der feuernden Regel möglich, wird hier aber nicht behandelt. Stattdessen Konflikte schon beim Schreiben der Regeln vermeiden

Warteschlangen (FIFOs)

- Einfache FIFOs/ mkFIFO
  - Wenn FIFO leer, kein deq möglich (selbst wenn gleichzeitig ein enq stattfindet)
  - Wenn FIFO voll, kein enq möglich (selbst wenn gleichzeitig ein deq stattfindet)
- Pipeline FIFOs/ mkPipelineFIFO
  - Auch wenn FIFO voll ist, enq möglich (wenn gleichzeitig ein deq stattfindet, first liefert noch den alten Wert – vor enq)
- Bypass FIFOs/ mkBypassFIFO
  - Auch wenn FIFO leer ist, deq möglich (wenn gleichzeitig ein enq stattfindet, first liefert schon den neuen Wert – nach enq)

Einfache Pipelines

Unterschnittstellen

- Komplizierte Interfaces, die nur einmal definiert werden müssen und häufig wiederverwendet werden können
- Die Bluespec Bibliothek hat eine Sammlung von wiederverwendbaren Interfaces (z.B. PutGet)
- Eine Konvertierung zwischen Standardschnittstellen ist möglich (z.B. interface put_x = toPut (fifo_in_x);)

Datentyp Maybe

- Basiert auf varianten Verbundtypen (tagged unions)
- Zu jedem Zeitpunkt existiert genau eine der beiden Komponenten: Festgelegt über type tag
- Werte des Typs können auf Gleichheit geprüft und als Bits dargestellt werden

```
typedef union tagged {
    void  Invalid;
    t     Valid;
} Maybe #(type t)
  deriving (Eq, Bits);
```

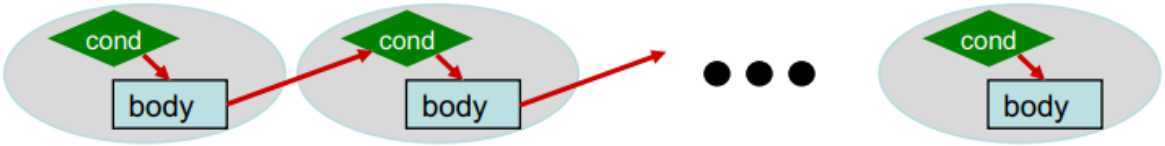
```
if (value matches tagged Valid .x)
    … hier ist x definiert, enthält den gültigen Wert…
else
    … hier den Fall "ungültiger" Wert behandeln…
```

Tupel-Typen

- Zusammenfügen von mehreren Einzelwerten zu einem zusammengesetzten Wert, Bluespec definiert bereits Tupel von 2 bis 8 Elementen
- Definition: Tuple2 #(typ1, typ2)
- Erzeugen von Werten: tuple2 (ausdruck1, ausdruck2)
- Lesen:
  - Mit Funktion tpl_j: tpl_1 (ausdruck), tpl_2 (ausdruck), …
  - Über „Mustervorlage“: match { .x, .y} = ausdruck; deklariert zwei temporäre Variablen mit den Werten der beiden Komponenten

Beeinflussen der Ablaufplanung

- Grundlage der Ausführungsreihenfolge ist Ablaufplan
- Einmal festgelegte Reihenfolge von Regeln, falls Regeln ausgeführt werden, werden sie immer in dieser Reihenfolge ausgeführt
- Regeln müssen aber nicht immer ausgeführt werden. Regeln werden unterdrückt, um Konflikte zu vermeiden
- Statisch: Schon zur Compile-Zeit, Regeln werden dauerhaft an Ausführung gehindert
- Dynamisch: Prüfung zur Laufzeit, Regeln werden nur unter bestimmten Umständen an Ausführung gehindert
- Bei N Regeln N! verschiedene Ablaufpläne, bsc wählt den mit der maximalen Nebenläufigkeit
- Regelbedingung rN.cond sind boole’sche Ausdrücke ohne Seiteneffekte

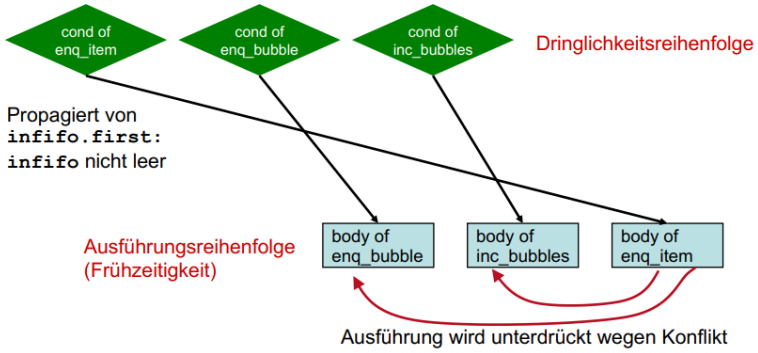


Dringlichkeit

- Reihenfolge der .cond Auswertungen
- Reihenfolge/Priorität der WILL_FIRE Bedingungen
- Attribut descending_urgency bestimmt Reihenfolge der .cond Prüfungen
- (* descending_urgency = “r1, r2” *)

Frühzeitlichkeit

- Reihenfolge der .body Auswertungen
- (* execution_order = “r1, r2“ *)
- Legt logische Ausführungsreihenfolge der Regelkörper fest



Preemption

- (* preempts = “r1, r2“ *)
- Erlaubt einer gefeuerten Regel das Feuern einer anderen Regel zu unterdrücken, auch wenn kein Konflikt vorliegt

Mutual exclusion

- (* mutually_exclusive = “r1, r2“ *)

- Zusicherung an Compiler, dass zwei Regelbedingungen niemals gleichzeitig wahr sind
- Zusicherung wird genutzt um effizientere Hardware zu erzeugen

Trends, Techniken und Werkzeuge des Hardware-Entwurfs

Entwicklung der Mikroelektronik

- 1970 erster Mikroprozessor
- Vorteil von ICs: Transistoren, Dioden und Widerstände sowie die Verbindung der Bausteine untereinander werden in einem gemeinsamen Herstellungsprozess in einem Silizium-Einkristall integriert und höhere Zuverlässigkeit, Schaltgeschwindigkeit und Packungsdichte
- ICs, oft auch als Chips bezeichnet sind nahezu überall verbaut
- Mikroelektronik in der Informatik: Einfluss auf Architekturen von Rechnersystemen und zur Entwicklung von Mikroelektronik wird Software verwendet, die Anforderungen werden per Algorithmen gelöst (Logikminimierung, Platzierung, …)

Entwicklung der Technik

- Moore: Alle 18-24 Monate verdoppelt sich die Anzahl der wirtschaftlichsten Transistoren auf einem Chip (nicht mehr aktuell)
- +53% pro Jahr Chip-Komplexität (Zahl der Transistoren, Speichergröße, etc.)
- +33% pro Jahr Packungsdichte (Elemente/Fläche)
- +33% pro Jahr Taktfrequenz
- +12% pro Jahr Chipfläche
- +12% pro Jahr mehr Pins (Flaschenhals)
- -6% pro Jahr Speicherzugriffzeit (Flaschenhals)
- Taktfrequenz:
    - Leistungssteigerung lange Zeit durch erhöhen der Taktfrequenz (aktuell 4.x GHz)
    - Bedingt durch die CMOS-Technologie steigt der Leistungsumsatz der Prozessoren mit dem Takt (U = Spannung, f = Transitionshäufigkeit, CL = Kapazitätenlast) ($P \approx U2 * f * CL$)
    - Problem: Abtransport der entstehenden Abwärme ist mit hohem Aufwand verbunden
    - Lösung: paralleles Rechnen
        - Integration mehrerer CPUs auf einem Chip
        - Massiv parallele Systeme mit mehreren tausend Prozessoren, Spezialarchitekturen (Vektorrechner, Cluster)
- Kostenentwicklung pro Fabrik +24% pro Jahr
- Reine Fertigungskosten ca x2 pro jahr
- Kosteneinsparung: ältere Technologie, Multi-Projekt-Chips, Programmierbare/konfigurierbare Schaltungen
- Kostensenkung durch Massenanfertigung

Hardware-Entwurfstechniken

- Probleme:
    - Sehr unübersichtlich („Puzzle mit 109 Teile“)
    - Hoher Zeitdruck (Time-to-Market, TTM)
    - Ein Fehler kann Unsummen kosten (erneute Chip-Fabrikation/ “respin“)
- Lösungsansätze
    - Abstraktere Vorgehensweisen
        - Beschreibe nicht mehr einzelne Transistoren, sondern komplette Systeme

    - Vergleichbar Software-Entwicklung: Statt Assembler, Systeme als interagierende Komponenten (service-oriented architectures)

Mittel der Wahl:

(Hardware-)Beschreibungssprachen („Hardware Description Languages“)

- Sehr abstrakt MATLAB/Simulink (Signalverarbeitung)
- Abstrakt SystemC
- Recht Hardware-Nah Verilog HDL, VHDL

Ebenen

- Verhaltensebene
    - Realisierung bleibt offen, (Verilog)
- Systemebene
    - Grobe Aufteilung von Struktur, Zeit, Daten und Kommunikation
- Registertransferebene
    - Verilog, etc.
    - Sehr wichtige Entwurfsebene
    - Effiziente Umsetzung in Hardware automatisch möglich
- Logik- und Gatterebene
    - Netze aus Gattern, Flip-Flops, etc.
    - Zeitverhalten grob abschätzbar
    - Keine endgültige Hardware-Beschreibung
- Transistorebene
    - Elektrischer Schaltplan
    - Beim Digitalschaltungsentwurf verborgen
- Layoutebene
    - Maßstabgetreue geometrische Anordnung des Chips mit verschiedenen Schichten
    - Ergebnis von Platzieren und Verdrahten x Genaues Zeitverhalten bekannt

Fundamentale Entwurfsmethoden

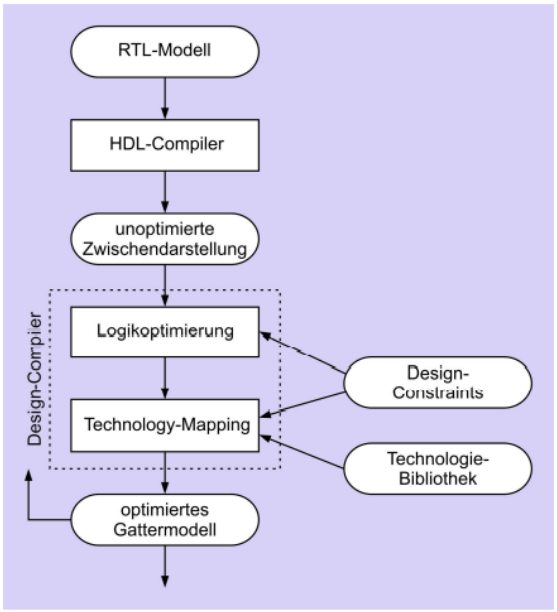
- Weglassen für die aktuelle Beschreibung unwichtiger Details
- Arbeiten auf unterschiedlichen Ebenen
    - Von ungenau bis sehr genau
    - Verhaltensebene, …, Layoutebene x Divide and Conquer
- Top-Down-Entwurf (Detaillierungsprozess)
    - Die zu realisierende Gesamtfunktion wird in kleinere Teilfunktionen (Teileinheiten, Komponenten) und ein Verbindungsnetz (Verbindung durch Signale) zerlegt.
    - Zerlegung kann hinsichtlich verschiedener Kriterien (Kosten, Geschwindigkeit, Chipfläche, etc…) erfolgen
    - Setzt voraus, dass sich Teilsysteme abstrakt als „black boxes“ beschreiben lassen, ohne dabei auf Details ihrer internen Realisierung einzugehen.
- Bottom-Up-Entwurf (Kompositionsprozess)
    - Aus den verfügbaren Primitiven, welche in einer Bibliothek angelegt sind werden komplexere Komponenten gebildet, die dann auf höheren Ebenen als (elementare) Bausteine verwendet werden können
- Meet-in-the-Middle-Entwurf
    - Vereinigt Top- und Bottom-Up-Entwurf, indem beim Vorgehen von der einen Seite die Auswirkungen auf der anderen berücksichtigt werden.

Verfeinerter Ablauf der Synthese

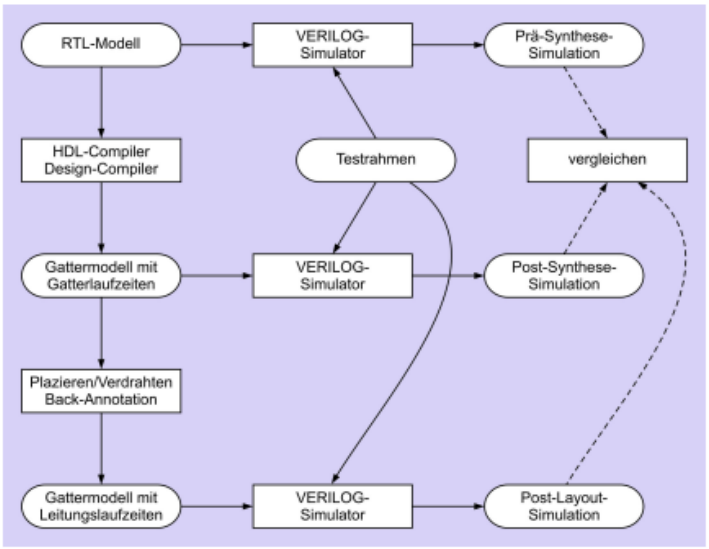
Technology-Mapping: Optimieren der Gatter und anpassen auf Ziel-Architektur

Design-Constraints

- Zeit
  - Timing-Analyse
  - Geschätzt nach Synthese
  - Exakt nach Platzierung und Verdrahten
  - Layout-Ebene
- Fläche
  - Geschätzt nach Synthese
  - Exakt nach Platzieren und Verdrahten
- Elektrische Leistungsaufnahme
  - Simulation auf Layout-Ebene
  - Bestimmung von Umschaltfrequenzen von Signalen (toggle frequencies)



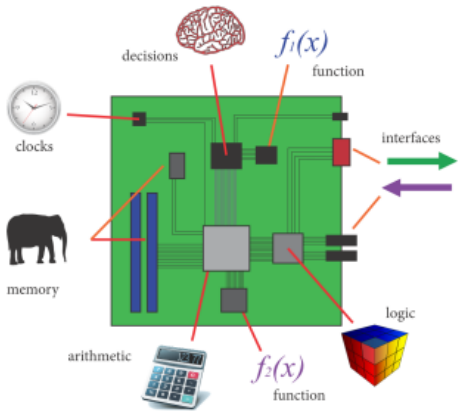
Verifikation



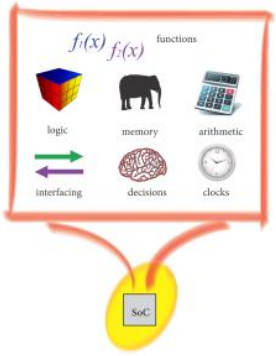
- Prä-Synthese-/ Post-Synthese-Simulation
- Gleiche Testdaten, bei Post-Synthese aber genauere Tests möglich, da hier nun genaueres Zeitverhalten
- Differenzen in Ergebnissen durch anderes Zeitverhalten
- Vergleich etwas aufwendiger, Interpretation nötig „Wenn man lange genug wartet ist das Ergebnis richtig“ – lang genug ist hier der kritische Pfad, damit passender Takt für RTL wählbar zwischen Eingangsregistern und Ausgangsregistern
- Post-Layout-Simulation schießt Gatterverzögerung und Leitungsverzögerung ein und kann noch Widerstände, Kapazitäten und Induktivitäten umfassen

Rekonfigurierbare System-on-Chips am Beispiel Xilinx Zynq 7000

- System-on-Board



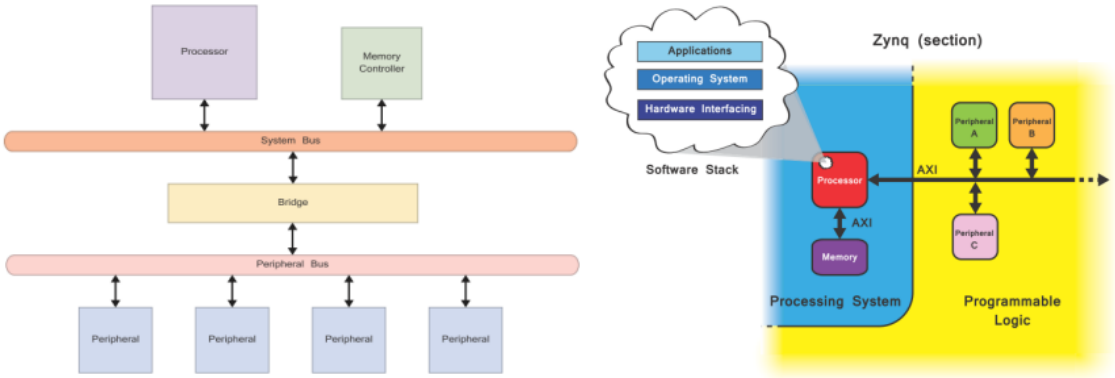
- System-on-Chip



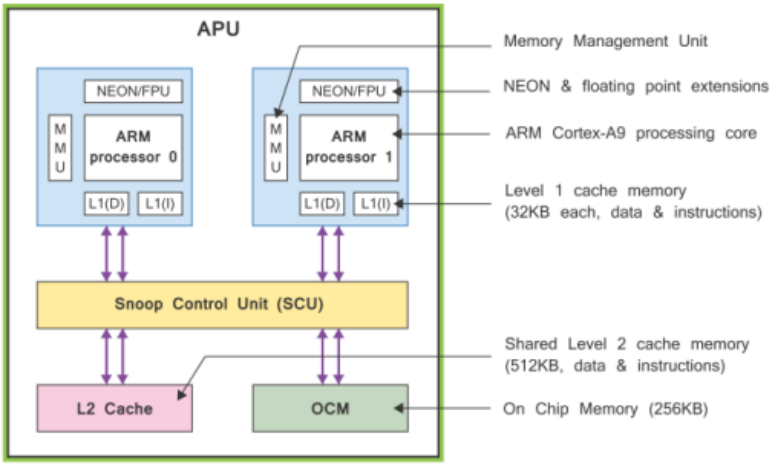
- Komponente eines modernen SoCs: ARM-Cores, Audio, DSP, Video, DISP, Face, Imaging, GPU, USB, SD, System-Bus und weitere

Xilinx Zynq 7000 reconfigurable System-on-Chip

- Basisarchitektur:

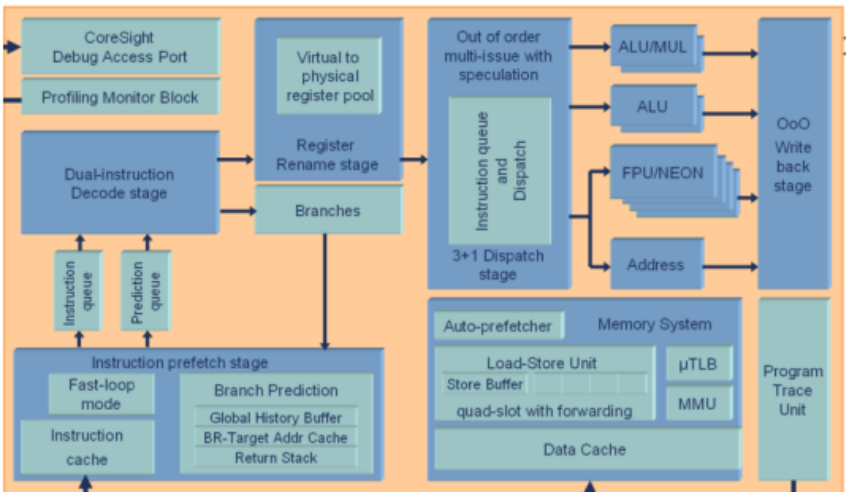


- Software-Programmierbare Prozessoren (Application Processing Unit)



ARM Cortex-A9 Prozessorkern

- Superskalar out-of-order
- Holt zwei Instruktionen je Takt
- Kamm je Takt bis zu vier Instruktionen ausführen: ALU/MUL; ALU; FPU/SIMD; Load-Store
- Mehr ILP durch umbenennen von Registern und dynamischen Vorziehen von unabhängigen Instruktionen
- Interner Aufbau mit 8 bis 11 Pipeline Stufen in 7 Phasen: FDRIEMW

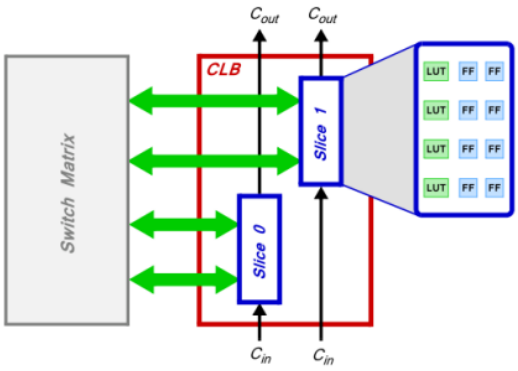


- Instruction Fetch (**F**etch)
  - Sprungvorhersage
  - Fast-Loop Mode
- Instruktionsdekodierung (**D**ecode)
  - Kann bis zu zwei Instruktionen je Takt dekodieren
- Umbenennen von Registern (**R**ename)
  - Zum Auflösen von Abhängigkeiten
  - Übersetzt virtuelle Registernamen in physikalische Register
- Instruktionen ausgeben (**I**ssue)
  - Akzeptiert zwei dekodierte Instruktionen je Takt
  - Kann bis vier Instruktionen an Ausführungseinheiten ausgeben
  - Dabei umsortieren von Instruktionen
- Ausführungseinheiten (**E**xecute)
  - Für unterschiedliche Arten von Instruktionen, zwei Additionen und eine Multiplikation pro Takt
  - ALUs haben eine Latenz von 1-3 Takten
  - ARM NEON hat eine Latenz von 1-32 Takten
  - ARM NEON ist eine advanced single instruction multiple data (SIMD) architecture extension für Arm Cortex Prozessoren
  - SIMD mit NEON
    - Single Instruction Multiple Data
    - Skalare Datentypen, Vektoroperationen
    - Wird aber auch als normale FPU genutzt
- Speichersystem (**M**emory)
  - Adressübersetzung zweistufig: Micro TLB (Table lookaside buffer) 1 Takt Latenz, TLB in MMU (Memory Management Unit) Variable Latenz
  - Prefetching: Holt vorweg Daten, bis zu 8 Datenströme gleichzeitig
  - Vier aktive Speicheranfragen: Store-to-load Forewarding
- **W**riteback

Cache-Kohärenz

- Prozessorkerne haben eigene L1-Caches, greifen aber auf gemeinsamen Hauptspeicher zu
- Prozessorkerne müssen sich austauschen darüber, wo aktueller Wert liegt
  - Noch im Hauptspeicher
  - Schon im L1- oder L2-Cache
  - Bereits in aktualisierter Form in einem Cache
- Snoop-Control-Unit (SCU)
  - Überwacht alle Speicherzugriffe
  - Gibt aktuellen Wert Weiter
  - Führt Protokoll: Modified; Exclusive; Shared; Invalid (MESI)
- APU On-Chip-Memory (OCM)
  - 256KB SRAM direkt auf dem Chip
  - Nicht sonderlich schnell ca. 1400 MB/s

- OCM hat aber geringere oder weniger schwankende Zugriffslatenz, i.d.R. 32-34 Taktzyklen



Programmierbare Logik

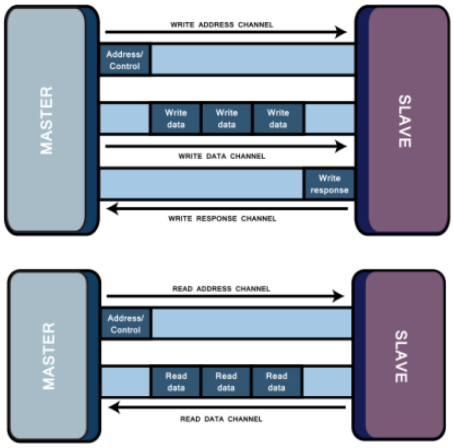
- Integriertes FPGA besteht aus programmierbaren Interconnects, konfigurierbaren Logikblöcken, Input/Output Blöcken (IOBs) und sind aufgeteilt in Logik Fabriken
- Konfigurierbarer Logikblock:
- Integrierte Speicher und Multiplizierer: BlockRAMs:
  Je 36kb oder 2x 18kb, Breite konfigurierbar, alle parallel zugreifbar
- Soft Core Processor, Prozessor in programmierbarer Logik, brauchen deutlich mehr Chip-Fläche als Hard Cores und laufen wesentlich langsamer
- Einsatz von Soft-Cores i.d.R. nicht mehr effizient
- Partitionierung von Algorithmen, Teile Ausführung auf, zwischen Ausführung
  - … als Software auf Prozessor (PS)
  - … als spezialisierte Hardware-Einheiten in programmierbarer Logik (PL)
- Partielle dynamische Rekonfiguration
  - Dynamisch. Tausche Funktionen in der PL zur Laufzeit aus
  - Partiell: rekonfiguriere nur die zu ändernden Teile der PL, der Rest läuft unverändert weiter
  - Erlaubt Änderung von Hardware-Funktionen während des laufenden Betriebs und effiziente Ausnutzung der Chip-Fläche

Schnittstellen zwischen Prozessor und FPGA

- Prozessor und programmierbare Logik müssen Daten austauschen, Eingabeparameter und Ergebnisse von Hardware-Funktionen
- Kommunikation benötigt Schnittstellen zwischen PL und PS -> Varianten des ARM AMBA AXI4 Protokolls
  - **A**dvanced **M**icrocontroller **B**us **A**rchitecture **A**dvanced e**X**tensible **I**nterface
  - Getrennte Signale für Lese- und Schreiberichtungen, es müssen aber nicht immer Beide vorhanden sein
- Interfacetypen: general purpose interface (GP), ACP interface, high performance interface (HP)
- Zugriffe von Prozessoren, PL und Master-fähigen Peripheriegeräten teilen sich den Übertragungsdurchsatz von Interconnects, L2-Cahce und dem Speicher-Controller
- High-Performance Interface
  - HP greift auf den Speicher zu und ist cache-inkohärent zu CPU
  - Greift immer auf DDR3-SDRAM zu (PL-interne Caches wären aber möglich)
  - Belastet nicht CPU L2-Cache
- ACP greift auf den Speicher zu und erlaubt cache-kohärente Ausführung zusammen mit CPU (shared memory model)
  - Greift via SCU auf L2 Cache zu
  - Liefert Daten schnell, solange sie im Cache liegen
  - Teilt sich aber L2 Cache mit CPU
  - Nach erfolglosem Zugriff auf Cache (miss), Zugriff auf externen DDR3-SDRAM

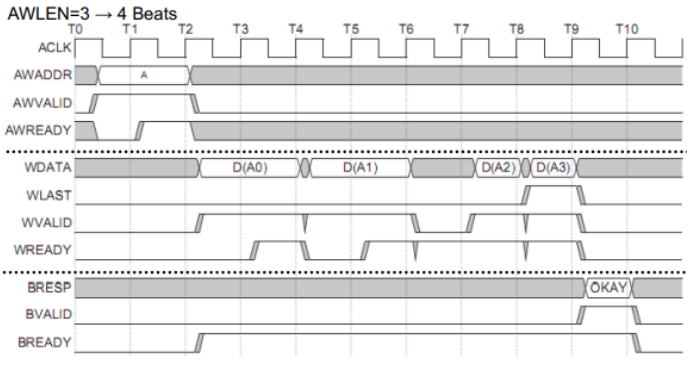
ARM AMBA AXI4

- AXI4: Mächtigste Implementierung, benötigt am meisten Chip-Fläche
  - Unterstützt memory-mapped I/O (Adressen und Daten)
  - Erlaubt effiziente Übertragung von ganzen Datengruppen (burst transfer)
- AXI4-lite: Einfacher, braucht weniger Chip-Fläche
  - Unterstützt memory-mapped I/O (Adressen und Daten)
  - Aber nur noch Übertragung von Einzeldaten, keine Burst-Transfers mehr
- AXI4-Stream: Spezialisierte Realisierung
  - Überträgt nur reine Datenströme (keine Adressen mehr, damit ungeeignet für memory-mapped I/O)
  - Unbegrenzt lange Bursts
  - Unidirektionale Datenübertragung von Master zu Slave, bidirektionale Übertragung benötigt zwei separate AXI4-Streams
- Getrennte Übertragungskanäle für Adressen/Kommandos und daten (ausgenommen AXI4-Stream)
- Master löst Übertragung aus, Slave reagiert auf Übertragung
  - Liefert Daten an Master bei Lesezugriff
  - Nimmt Daten vom Master entgegen bei Schreibzugriff
  - Liefert Status an Master (über Lesekanal oder extra Rückkanal)
- Mehrere AXI Kommunikationspartner kommunizieren über ein Verbindungsnetz (interconnect) und ist nicht zwangsläufig ein Bus
- Master schreibt Daten zu Slave: Burst Übertragung: Master setzt nur Start-Adresse
  - Slave muss Folgeadressen selbst bestimmen
  - Maximale Burstlänge sind 256 Datensätze (beats)
- Master liest Daten von Slave: Burst-Übertragung
  - Ein einzelner Beat kann 1 bis 128 Bytes umfassen



Signale

- Handshake zwischen Quelle und Ziel von Daten, jeweils ausgewertet @posedge
  - Quelle setzt VALID, wenn gültige Daten anliegen
  - Ziel setzt READY, wenn Daten übernommen werden können



Transaction Channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

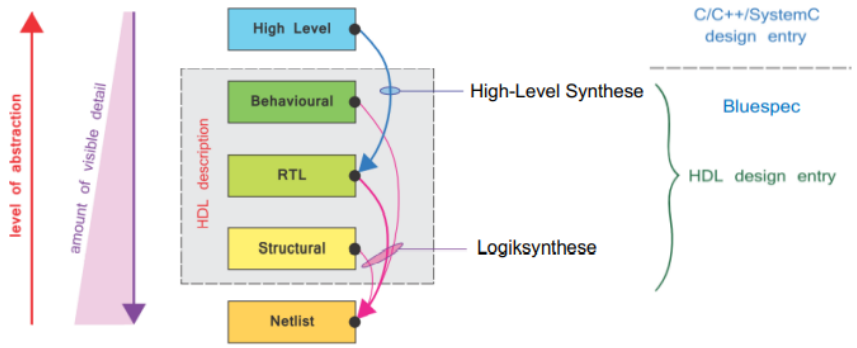
IP Blöcke

- IP-Blöcke sind vordefinierte Hardware-Funktionen (intellectual property blocks/cores) für Standardaufgaben
- Hardwareentwurf ist sehr aufwendig und teuer
- Einpacken der IP Blöcke: Standardisierte Beschreibung der Schnittstellen (IP XACT)

- Blöcke umfassen nicht nur Hardware sondern auch Dokumentation, Simulationsmodelle, Beispiele, etc.

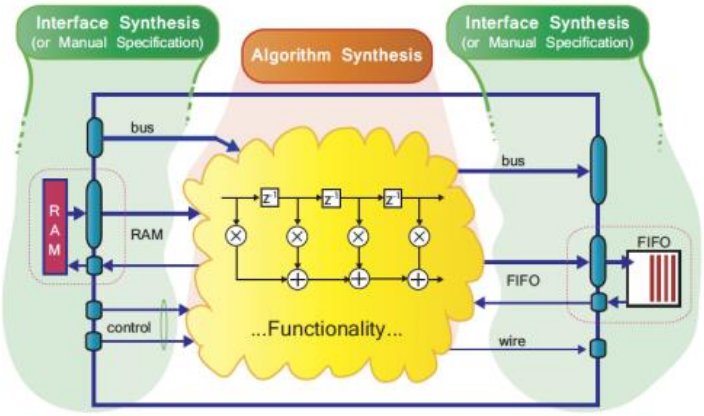
High-Level-Synthese

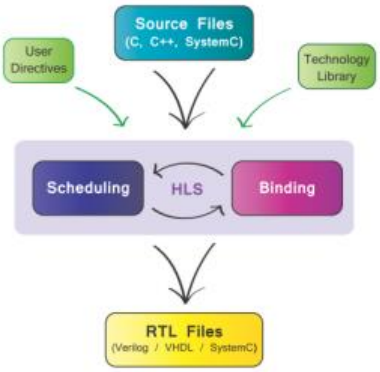
- HLS-Werkzeug probiert verschieden Implementierungs-alternativen aus und bietet eine an, die die Design Constraints erfüllt

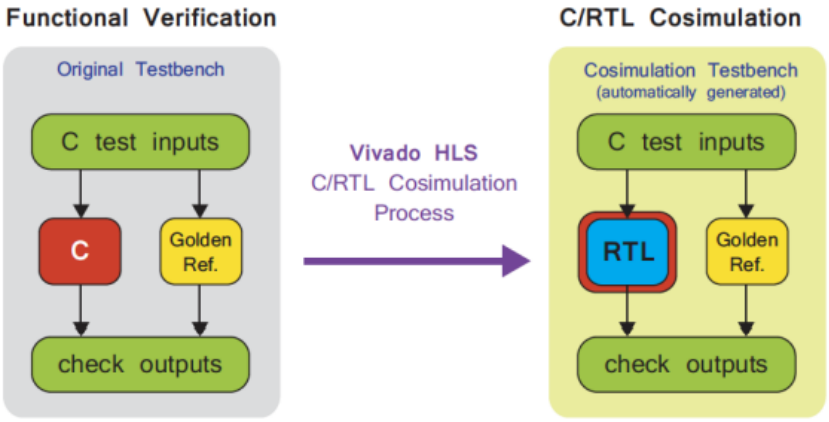


Teilaufgaben des HLS-Vorgehens

- Ablaufplanung (scheduling), Zuordnung von Operationen an Zeitschritte
- Bindung (binding), Zuordnung von ablaufgeplanten Operationen an echte Hardware-Ressourcen
- Einige Implementierungsvarianten
  - Verwende so wenig Ressourcen wie möglich und benutze diese wieder: **Resource Sharing**
  - Verwende mehr Hardware-Ressourcen und kaskadiere mehrere Operationen in einem Takt: **Operator Chaining** (hier kombiniert mit Resource Sharing), Bedingung: Summe der Propagation Delays der Ressourcen ist kleiner als Taktperiode
  - Verwende spezialisierte Hardware-Ressourcen (in der Regel schneller als allgemeine), aber davon deutlich weniger auf Chip vorhanden als allgemeine Logikblöcke







- Interface-Synthese in HLS
  - Software-Welt: C/C++ Konstrukte werden durch HLS in Hardware übersetzt
  - Restliche Hardware-Welt: Üblicherweise in RTL beschriebene Digital-Schaltungen
  - Verbindungen zwischen Hardware-Komponenten, Leitungen und Protokolle, die C/C++ nicht ausdrücken

  - Prinzip: Zunächst Ports für Eingabe/Ausgabe-Daten definieren und dann weitere Befehle zum Zurücksetzten und starten des Blocks. Außerdem Statusabfragen, idle, done, ready

Entwurfsverfahren und Werkzeuge für Hardware-Rechenbeschleuniger

Übersicht Rechenbeschleuniger

- ASIC: Application Specific Integrated Circuit
- Mikro-Controller: Instruction Set Architecture (ISA), limited scope
- System-on-Chip: small-scale computing architecture
- Low-Power CPU: desktop-class CPUs with focus on energy-efficiency
- Multi-Core CPU/SoC: desktop class and server class CPUs, SoCs
- GPGPU: General Purpose Graphics Processing Unit
- Many-Core: massively parallel CPUs
- DSP: Digital Signal Processors, massively parallel arithmetic units
- FPGA: Field Programmable Gate Array

Klassifikation von Rechenbeschleunigern

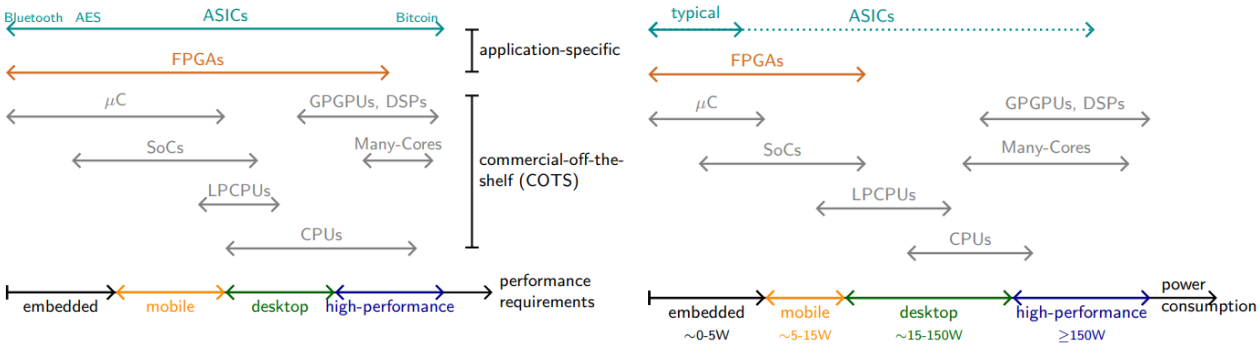
- Commodity ISAs (Mikro-Controller, LPCPUs, Multi-Core CPUs/SoCs
  - Standardized instruction set
  - Extensive tool support
  - Programmable in mainstream languages
- Specialized ISAs (GPGPUs, DSPs, Many-Cores)
  - Non-standardized instruction set
  - Limited tool support
  - Programmable in specialized languages
- Reconfigurable Technology (PLDs, FPGAs)
  - Full-custom design, non-standard
  - Limited tool support
  - Require hardware design languages
  - Limited support for mainstream and specialized languages
- ASICs (Application specific hardware)
  - Full-custom design, non-standard
  - No compiler support
  - Require hardware design languages
  - Programmable only in low-level languages / via HW interface

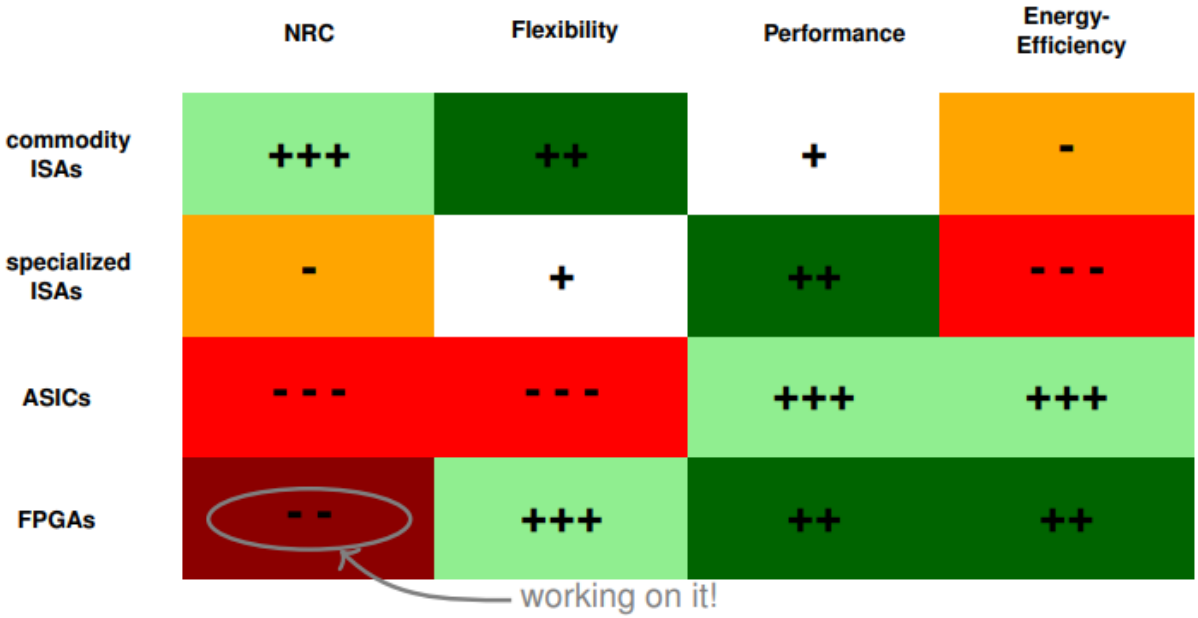
Moore's & Patterson's Law

- Moore: number of transistors/area doubles every two years
- Turned from observation (10 years) to imperative
- Patterson's Walls : Power Wall + Memory Wall + ILP Wall = Brick Wall
- Power Wall: capacitors need to charge/discharge faster, limited by power and heat
- Memory Wall: access latencies + transfer speed, memory is slow and therefor a bottleneck
- Instruction-Level Parallelism Wall: limited by number of non-contentious hardware resource requirements
- Patterson's Law: If you circumvent two walls, you'll hit the third

- Demise of Moore's Law was masked but gave rise to new approaches (GPGPUs, DSPS, FPGAs), consider heterogeneous architectures

Vergleich Rechenbeschleuniger





Typischer Workflow einer FPGA-basierten Lösung

1. Select or build a board
   - COTS hardware differs in detail, but mostly homogeneous
   - FPGA based solutions are much more heterogeneous and differ in architectural family, speed grade, area composition, periphery and other properties
2. Build or adapt a base design
   - The board PCB design controls the pin connections on the physical level
   - The FPGA bitstream controls the pin connections on the logical level
   - Almost complete wiring freedom, but memory needs to be connected on this level as well as the connection to the host
   - A base design is an empty design with default connections
3. Implement the design
   - Develop a hardware module for the algorithm, usually in HDLs
   - Independent of target hardware, RTL can be mapped to any FPGA or ASIC, in practice not quite realizable
4. Simulate the design
   - Test functional correctness by simulation on RTL level, aim for coverage
5. Place and route

- Two step process:
    - Place step: maps operations to hardware resources
    - Route step: performs the wiring and checks timing constraints
- Number of possible mappings grows exponentially with area
- Many different algorithmic approaches: heuristic approach, simulated annealing, analytical methods
- Most time-consuming step

6. Exercise on real hardware
    - Necessary due to several uncertainties in the process
    - Real-world performance is determined by many outside factors

Aktuelle Probleme

- Portability: hardware choices from the start influence the entire workflow, change of hw leads to repetition of most time-consuming steps
- Scalability: resources vary significantly, classic HDLs are not expressive enough and good test coverage is hard to obtain

TaPaSCo

Parallelismus in TaPaSCo

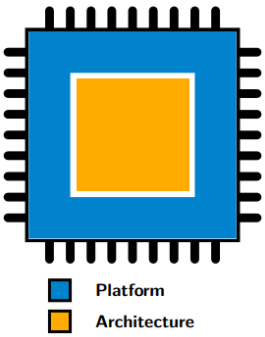
- TaPaSCo implementes a task parallel model of computation
- Computation is grouped into phases called tasks
- Tasks may depend on the outputs of their predecessors
- Data for each task can be partitioned into jobs
- Parallelization across multiple processing elements (PEs)
- Jobs are scheduled on the PEs asynchronously
- Requirements: problem can be decomposed into independent jobs and the jobs can be executed in parallel
- Parallelising a sequential program: profile, identify computational hotspots (kernels), isolate, replicate hardware
    - PEs for each kernel are called thread unit
    - Composition: size of thread units for each kernel
- Thread pool
    - Conceptual abstraction over a Composition: pool = number of thread units; thread unit = number of PEs
    - Operational abstraction: jobs for each kernel can be submitted to pool, are then executed on first available PE and results can be collected out-of-order
    - No inter-PE communication

Kompilierungsablauf

Design Abstractions

- Architecture: thread units and PEs, according to Composition, interconnects, independent of target platform/board

- Platform: connection to host and memory, hardware dependent
- Hardware-dependent parts are isolated in Platform
- Represented in TaPaSCo as plug-in scripts



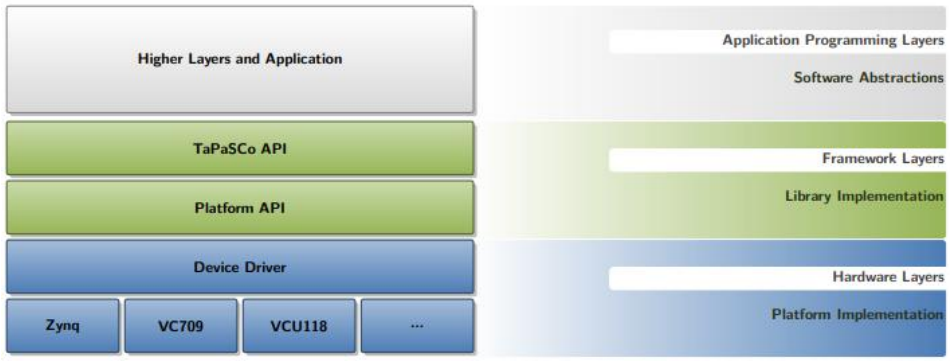
Architektur Skripte

- define basic interface of PEs
- instantiate PEs according to composition
- combine PEs into thread units + perform wiring
- combine thread units into thread pool + perform wiring

Platform Skripte

- provide control connection from host to PEs
- provide PEs with access to memory
- provide signalling interface from PEs to host
- optional: instantiate additional hardware infrastructure

Software Stack



OS-Level integration

- device driver provides OS-level integration
  - on-chip address space is mapped into global/bus address space
  - each slave interface has physical address range
  - hardware functions are accessible for all bus devices
- in case of monolithic kernels (Linux, Windows, MacOS)
  - driver code becomes part of kernel, executes with Ring-0 privilege
  - security-critical, can crash entire system
  - tedious to develop
- OS provides facilities to allow and control userspace access
  - Physical ranges can be mapped into virtual address spaces
  - Driver can permit access to special files

Platform Bibliothek

- Software counterpart to Platform-on-chip
- Minimal userspace abstraction over the device driver layer
- Low level integration tasks:
  - Platform_alloc
  - Platform_dealloc
  - Platform_read_mem
  - Platfrom_write_mem

    - Much more like read/write control registers, wait for signals, query device address space and initialization /administrative functions

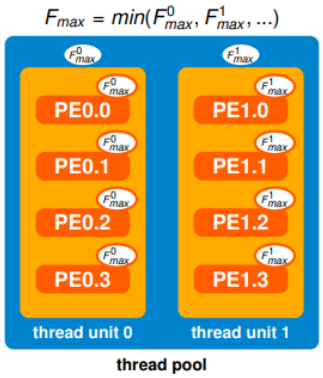
Architektur Bibliothek

- User-facing API for TaPaSCo applications
- Tapasco_device_alloc – wrapper for platform_alloc
    - TaPaSCo library forwards memory allocation request to platform library, which then allocates memory on device
- Tapasco_device_dealloc – wrapper for platform_dealloc
- Tapasco_device_copy_to – wrapper for platform_write_mem
    - Device driver copies data from virtual address in userspace to kernel space buffer, device driver translates kernel virtual address to physical/bus address
    - Device driver sets DMA (Direct memory access) engine registers: kernel buffer physical base address, kernel buffer length, device memory destination address, copy direction: to device
    - DMA engine copies data and raises interrupt signal to host
- Tapasco_device_copy_from – wrapper for platform_read_mem
    - Device driver reserves kernel buffer
    - Device driver sets DMA engine registers: device memory source address, kernel buffer length, kernel buffer physical base address, copy direction: from device
    - DMA engine copies data and raises interrupt to signal host
- Tapasco_device_acquire_job_id
    - TaPaSCo library acquires job object userspace struct, containing thread unit ID and buffers for all argument values
- Tapasco_device_release_job_id
    - TaPaSCo library releases job object
- Tapasco_device_job_set_arg
    - Application prepares job in memory: sets argument values in job object
- Tapasco_device_job_get_arg
- Tapasco_device_job_get_return
    - Application fetches return value from job object
- Tapasco_device_job_launch
    - Application submits job object to pool
    - TaPaSCo library acquires first idle PE in the thread unit
    - TaPaSCo library writes register values via Platform library
    - TaPaSCo library writes start bit and waits for interrupt via Platform library
    - TaPaSCo library copies back any register values and arguments to the job object
    - TaPaSCo library releases the PE
- Tapasco_device_free
    - TaPaSCo library forwards memory release to platform library
    - Platform library marks block as free in buddy allocator

Design Frequency

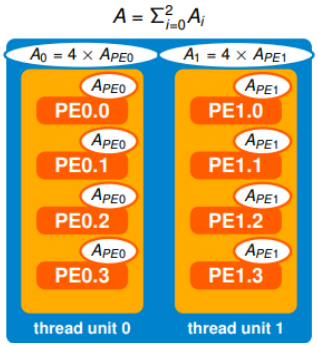
- Upper bound for each thread unit is given by upper bound of its PEs
- Upper bound for pool is given by lowest upper bound of thread units

- F_max for PE is obtained by performing synthesis + place and route in out-of-context mode
  - Synthesize netlist for PE, place and route in fixed location
  - Length of critical path -> estimate for F_max
  - Also yields estimation of area



Area Utilization

- Area estimation for each thread unit: area estimation for PEs x number of instances + Architecture overhead estimation
- Area estimation for each thread pool: sum of area estimation for each thread unit plus Architecture overhead estimation
- Compare area estimation for pool to available resources
  - Feasibility: A <= 1
  - Optimality: A_opt = 1 – A

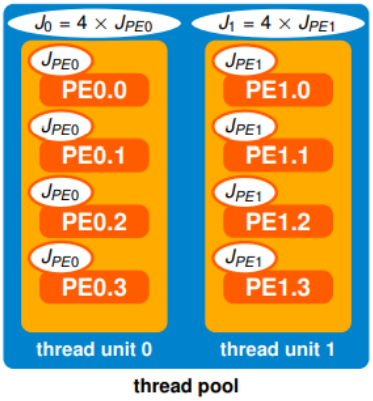


Optimierungsprobleme

- Maximization of area and design frequency
- Standard approach: heuristic function and optimize this function
- Optimize job throughput instead of area and design frequency

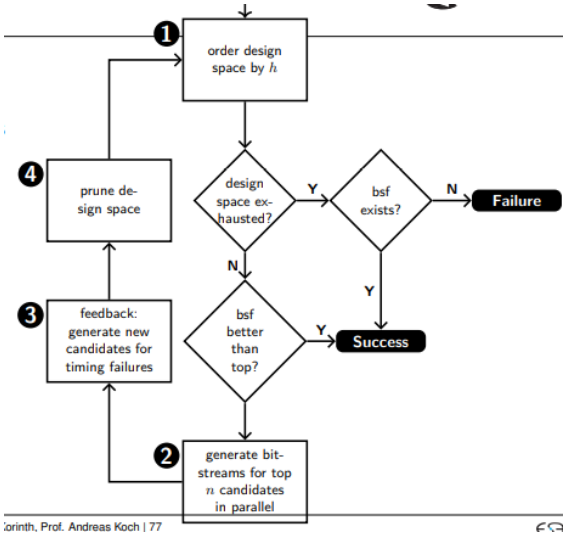
Job Throughput

- Job throughput estimation for each unit: job throughput PEs x number of instances – Architecture (Platform overhead estimate
- Job throughput estimation for thread pool: sum of job throughput estimations for each thread unit – Architecture/Platform overhead estimate
- Job throughput for single PE:
  - Job frequency = number of cycles per job * clock frequency
  - Number of cycles can be obtained by register transfer level (RTL) simulation of PE
  - Max frequency can be obtained by out-of-context approach
  - If number of cycles is input dependent: use average number of cycles in typical workload for approximation



Automatic Design Space Exploration

- Fully automatic process in TaPaSCo
- Heuristic h (configurable)
- Batch size n (configurable)
- Bsf = best-so-far; top = best-in-batch
- Very useful, automates tedious process
- Can be run on clusters and increases portability and performance
- TaPaSCo supports multiple variants of PEs, automated search across all variants of a composition

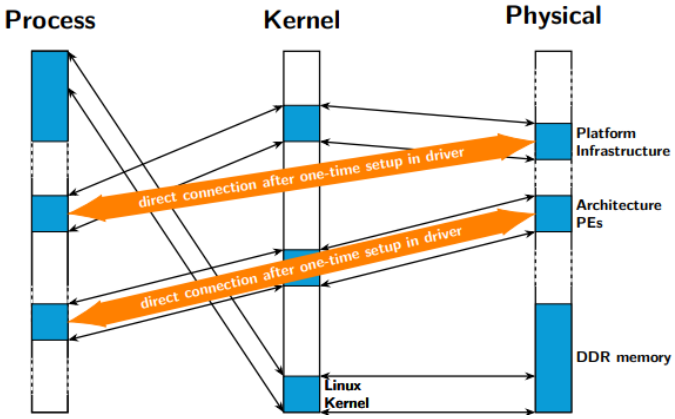


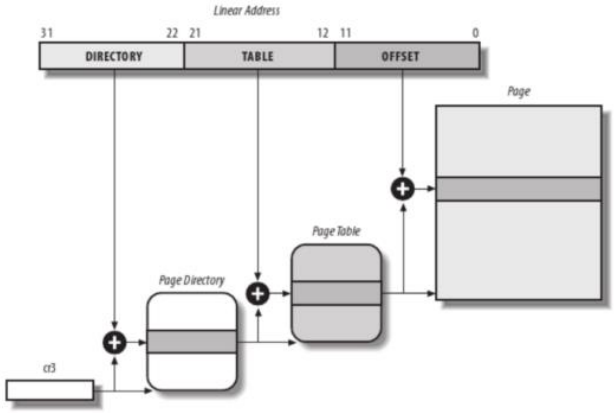
Address Spaces and Address Maps

- Devices are controlled via registers

- Registers are organized in a local address map
- Local address map is mapped into a controlling master at a base address
- Classic Memory-as-I/O abstraction: CPU controls memory master, which accesses memory addresses
- An interconnect controls the global address map

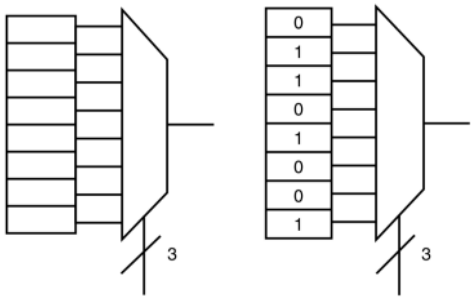
Page Table Mapping

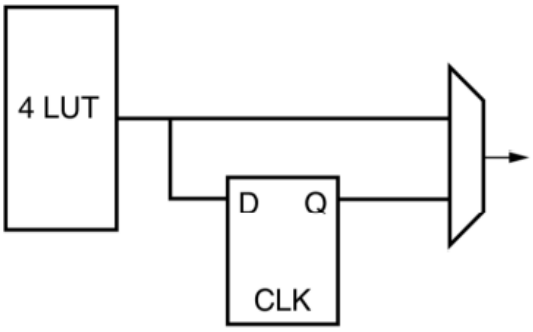




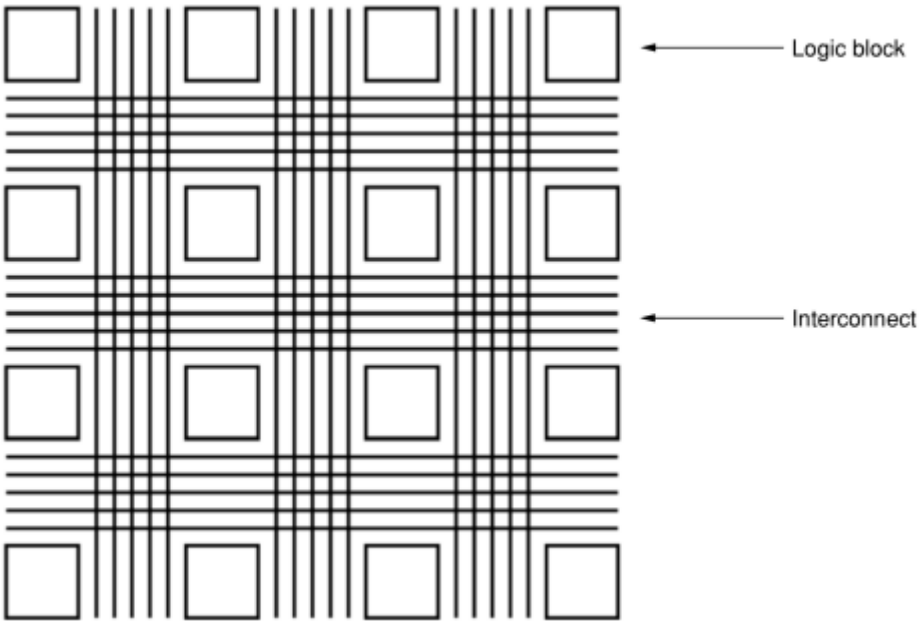
FPGA Structures

Look-Up-Tables (LUT)

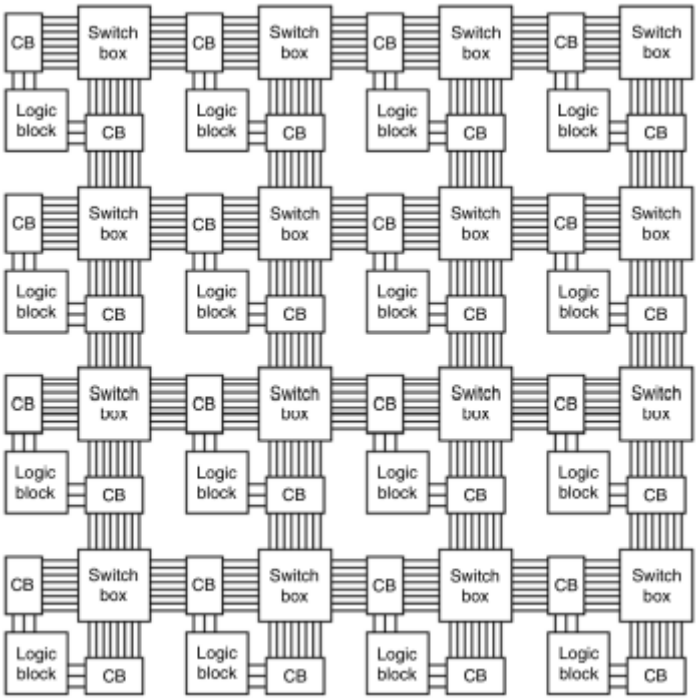




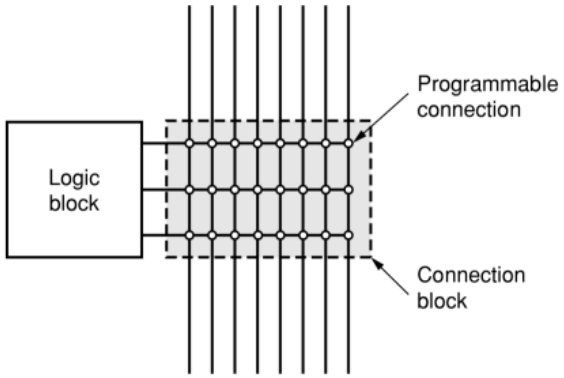
Interconnections



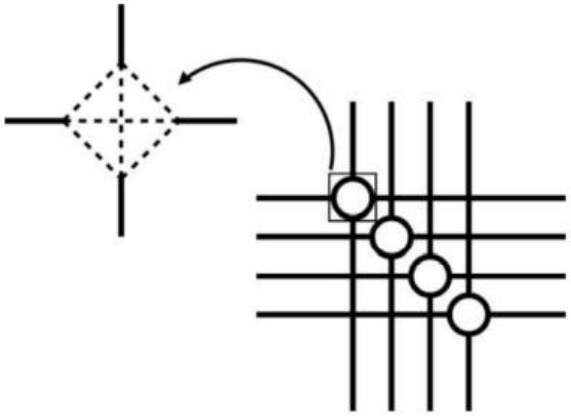
Island Style Architecture



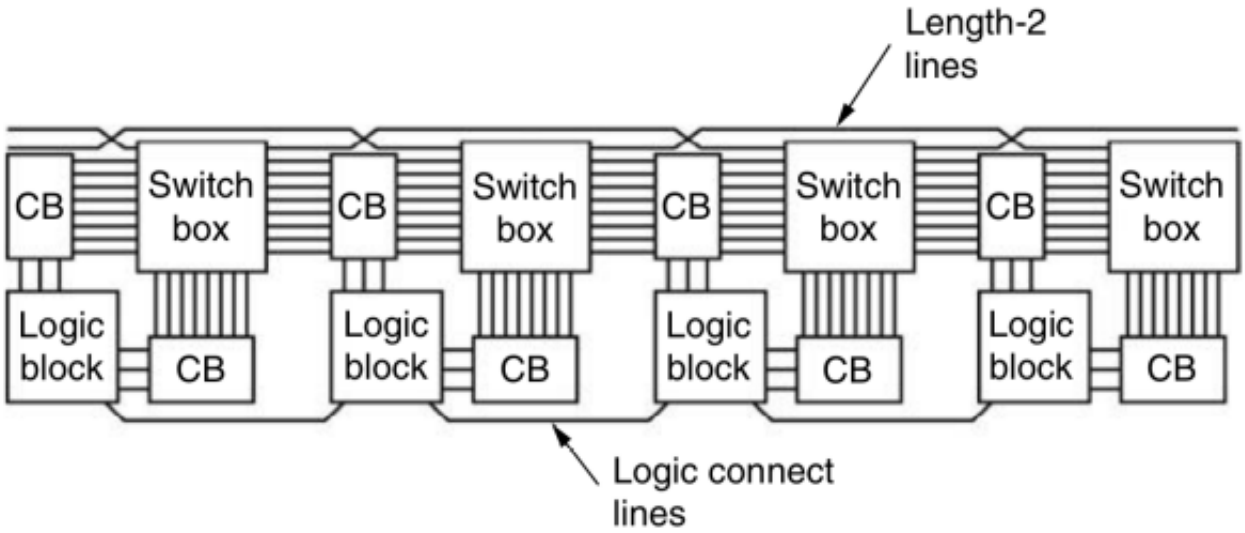
Connection Blocks



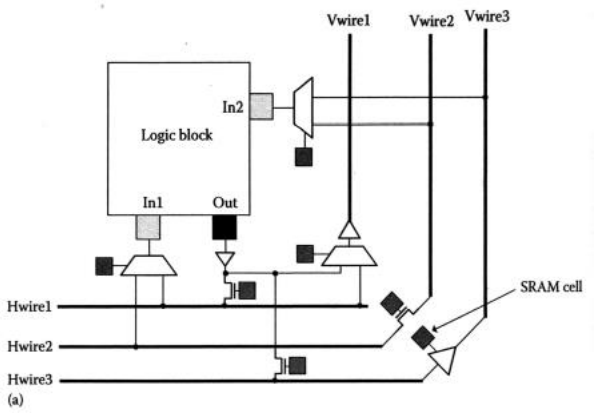
Switch Blocks



Interconnect Hierarchical Connections



Programmable Routing (electrical level)



Caches

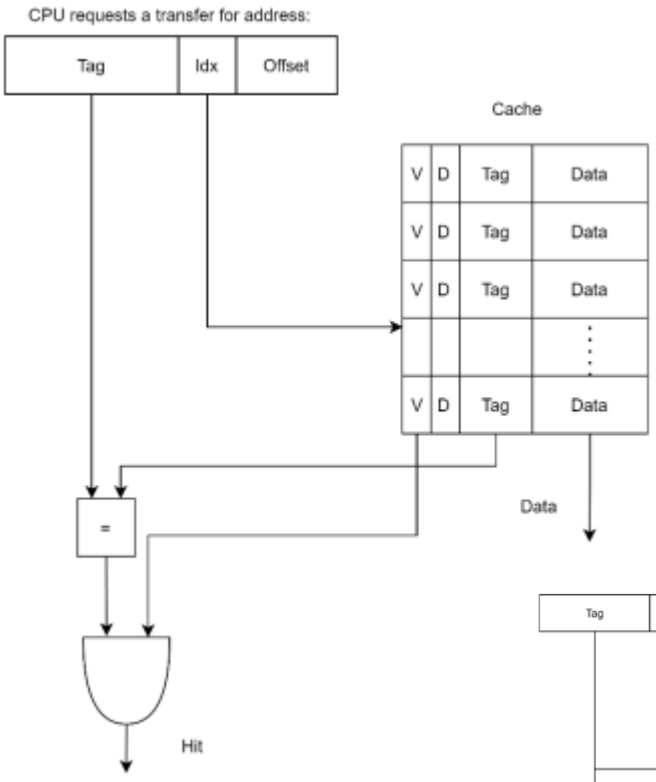
- CPU versucht zunächst die Daten aus dem Registersatz zu holen, sind diese nicht vorhanden, geht er zum Cache und versucht sie von dort zu holen
- Sind die Daten im Cache vorhanden: Cache-HIT
- Sind die Daten nicht im Cache, müssen sie aus dem Hauptspeicher geholt werden: Cache-MISS

Funktionalität

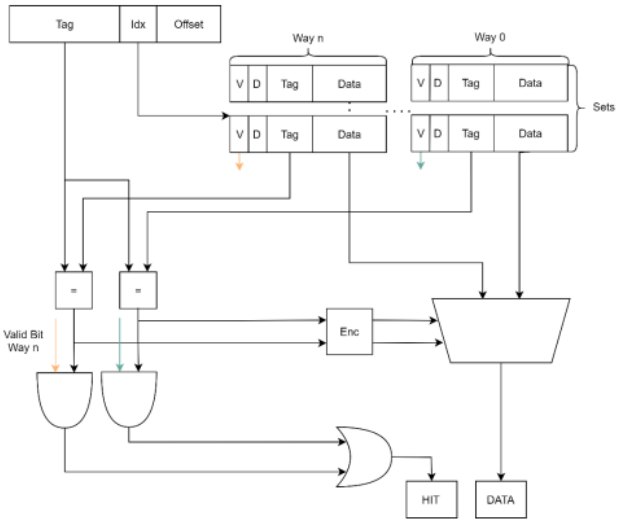
- CPU benötigt Daten, dazu ruft die CPU die Daten aus dem Cache ab, von einer bestimmten **Adresse**
- In der Adresse ist ein **Offset** vorhanden
- Das **Valid Bit** im Cache gibt an, ob die Daten gültig sind, also der Cache Block nicht leer ist
- Die Daten aus dem Cache sind eine Kopie der Daten aus dem Hauptspeicher, um zu wissen ob die Daten durch die CPU in den Cache geschrieben wurden, gibt es das **Dirty Bit**
- Da der Hauptspeicher größer als der Cache ist, sind die Adressen in einen **TAG** und einen **INDEX** aufgeteilt
- Mehrere Daten werden derselben Cache Zeile zugewiesen
- Um die Anzahl der Misses zu reduzieren, verwendet man einen Cache mit mehreren Wegen **(Multi-Way Cache/ n-Way Cache)**
- Da ein Set mehrere Konflikte haben kann, währen ein anderes unbenutzt ist, kann man einen Cache mit einem Set und mehreren Wegen wählen **(Fully-Associative-Cache)**

Cache-Arten

- Direct mapped Cache: Ein Weg, mehrere Sets



- Set-Associative Cache: Mehrere Wege, mehrere Sets



- Fully-Associative-Cache: Mehrere Wege, ein Set

Cache-Strategien

- Wenn der Cache voll ist:
  - Random replacement (RR)
  - Least Recently used (LRU)
  - Least-frequently used (LFU)
  - …
- Beim Schreiben (Write policies)
  - Wann schreiben der Daten in den Hauptspeicher?
    - Write through
    - Write back
  - Laden wir die Daten in den Cache bei einem Schreibzugriff?
    - Write allocate
    - No-write allocate
- Arten von Cache-Misses

  - Compulsory misses: Daten von Adresse X wurden nie in den Cache geladen
  - Conflict misses: Viele Daten werden demselben Set zugewiesen und verursachen das Ersetzen der Daten
  - Capacity misses: Der Cache ist nicht groß genug für unsere Anwendung und die Daten werden deswegen ersetzt

StmtFSM

- Extremely useful library for developing testbenches which allows you to quickly generate C style loops, sequences, parallel blocks etc
- Basic StmtFSM:
  - import StmtFSM::*;
  - Create sequence: Stmt test = seq … endseq;
  - Instantiate FSM: FSM testFSM <- mkFSM(test);
  - testFSM.start to start the FSM
  - testFSM.done to check if FSM is done
- AutoFSM
  - Instantiate FSM: mkAutoFSM(test);
  - Sequence can also directly be written as argument
  - AutoFSM automatically starts the sequence and ends the system with $finish
- A sequence is basically an execution schedule for actions that should be executed after each other
- In a sequence you can use branching,
  - If (cond) seq … endseq else seq … endseq
- loops
  - repeat(n) repeats n number of times
  - while(cond) seq … endseq
  - for (init; cond; comm) seq … endseq
- and parallel execution
  - par seq … endseq seq … endseq endpar
  - tries to fire both sequences in parallel
- await (cond) pauses the FSM until the condition becomes True

BlueCheck

- BlueCheck is a Unit-Test Tool based on QuickCheck, a Unit-Test Tool for Haskell
- Create specification Module:
  - Module [BlueCheck] mkSomeModuleSpec(Empty); … endmodule
  - Instantiate dut (Interface dut <- mkModuleName;) and Ensure (Ensure ensure <- getEnsure;)
  - Create functions that return a Stmt (do calc on dut and in the module itself, then ensure (value1 == value2);)
  - Tell BlueCheck the properties to check: prop(“someFunction”, someFunction);
- Create synthesizable module that automatically creates and starts the FSM
  - Module [Module] mkSomeModuleCheck(Empty); blueCheck(mkSomeModuleSpec); endmodule
- Instead of ensuring each value by hand we can use equiv to generate test inputs and test the equivalency of the implementation
- Define functions that return Actions/ActionValues (return action … endaction;)
- Equiv(“someFunction”, someFunctionref, dut.function);