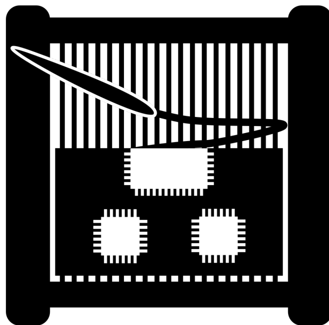
Architekturen und Entwurf von Rechnersystemen
Wintersemester 2016/2017

ThreadPoolComposer

TECHNISCHE UNIVERSITÄT DARMSTADT



Perspective: FPGAs - Why bother?

TECHNISCHE UNIVERSITÄT DARMSTADT

Every two years, FPGAs are 'finally going to arrive'.

Peter Lee, VP of Microsoft Research

FPGAs are for kids and academics. Everyone who needs to do real work uses ASICs.

Anonymous

Perspective: The Zoo of Computing Devices I

TECHNISCHE UNIVERSITÄT DARMSTADT

- **ASIC** - Application Specific Integrated Circuit
  Examples: Bitcoin Mining, Encryption, Audio Codec, Apple M7/8/9/M10, ...
- **μ-Controller** - Instruction Set Architecture (ISA), limited scope
  Examples: Arduino, ARM Cortex M-series, Atmel AVR series, ...
- **System-on-Chip** - small-scale computing architecture
  Examples: Raspberry Pi (BCM28xx), Altera D0-series, Xilinx Zynq-Series, ...
- **Low-Power CPU** - desktop-class CPUs w/focus on energy-efficiency
  Examples: Intel Celeron, Intel Core i3, ARM big.LITTLE, ...

Perspective:
The Zoo of Computing Devices II

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Multi-Core CPU / SoC** - desktop class and server class CPUs, SoCs
  Examples: Intel Core i5/i7-series, AMD Ryzen, Intel Xeon E5-series, . . .
- **GPGPU** - General Purpose Graphics Processing Unit
  Examples: NVIDIA GeForce-series, AMD R-series, NVIDIA Tesla P-series, . . .
- **Many-Core** - massively parallel CPUs (Intel Xeon Phi)
  Examples: Intel Xeon Phi & Knight's Landing, Sunway TaihuLight, Kalray MPPA, . . .
- **DSP** - Digital Signal Processors, massively parallel arithmetic units
  Examples: Texas Instruments Cx000-series

Perspective: The Zoo of Computing Devices III

TECHNISCHE UNIVERSITÄT DARMSTADT

. . . and last but not least:

- **FPGA** - Field Programmable Gate Array
  Examples: Xilinx Virtex-series, Altera Stratix-series, Lattice ICE-series, . . .

It's confusing! Why are there so many different technologies? Which ones should we use?

Perspective:
An attempt at a classification (incomplete)

TECHNISCHE UNIVERSITÄT DARMSTADT

Commodity ISAs *μ-Controller, LPCPUs, Multi-Core CPUs / SoCs*

- standardized instruction set
- extensive tool support (compilers, debuggers, . . .)
- programmable in mainstream languages (e.g., C)

Specialized ISAs *GPGPUs, DSPs, Many-Cores*

- non-standardized instruction set (e.g., NVIDIA PTX)
- limited tool support (vendor tools)
- programmable in specialized languages
  (e.g., OpenMP, CUDA, OpenCL, . . .)

Perspective:
An attempt at a classification (incomplete)

TECHNISCHE UNIVERSITÄT DARMSTADT

Reconfigurable Technology *PLDs, FPGAs*

- full-custom design, non-standard
- limited tool support (vendor tools)
- require hardware design languages (e.g., Verilog, VHDL, Bluespec, . . .)
- limited support for mainstream and specialized languages (e.g., C, C++, OpenCL)

ASICs *application specific hardware, e.g., Bitcoin mining, encryption*

- full-custom design, non-standard
- no compiler support (need to develop OS integration, API)
- require hardware design languages
- (user-)programmable only in low-level languages / via HW interface

Why are there so many different devices?
The End of Moore's Law

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Moore's Law (1965)

- number of transistors in ICs double approx. every 18 months
- named after *Gordon Moore*, founder of Fairchild Semiconductor and Intel
- has been accurate for decades
- informs International Technology Roadmap for Semiconductors (ITRS)

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Transistor count

Date of introduction

curve shows transistor count doubling every two years

Source: Wikipedia

Why are there so many different devices? The End of Moore's Law

Patterson's Walls

- recent nodes in ITRS roadmap have all missed their deadlines
- technological advances met *Patterson's Walls*
  - Power Wall + Memory Wall + ILP Wall = Brick Wall
- named after Prof. David Patterson
- summarizes reasons why Moore's Law could not be sustained

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Transistor count

curve shows transistor count doubling every two years

Date of introduction

Source: Wikipedia

Why are there so many different devices?
The End of Moore's Law

TECHNISCHE UNIVERSITÄT DARMSTADT

Beyond Moore's Law

- demise of Moore's Law was masked by lateral movements:
  - multi-core architectures
  - parallel computing
- . . . but also gave rise to new approaches:
  - GPGPUs
  - DSPs
  - FPGAs

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Source: Wikipedia

Why are there so many different devices?
Beyond Moore's Law

TECHNISCHE UNIVERSITÄT DARMSTADT

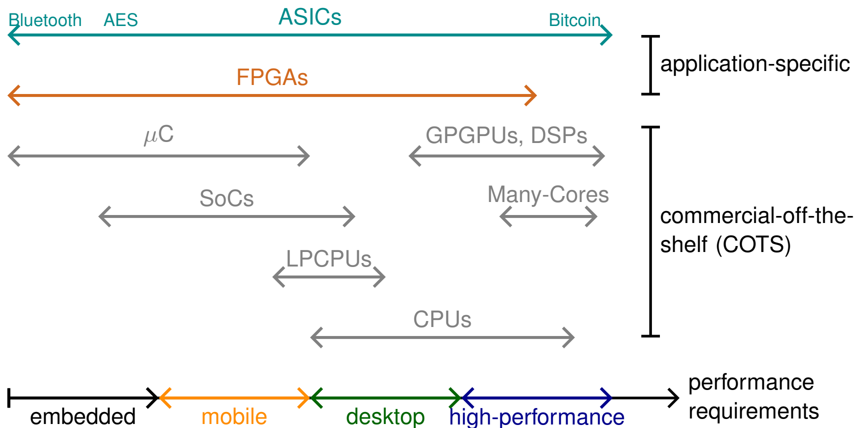
Beyond Moore's Law

- novel computing approaches to circumvent demise of Moore's Law
- *no silver bullet!* - each approach has its merits and demerits
- technology should be matched to the computational problem

Problem Dimensions

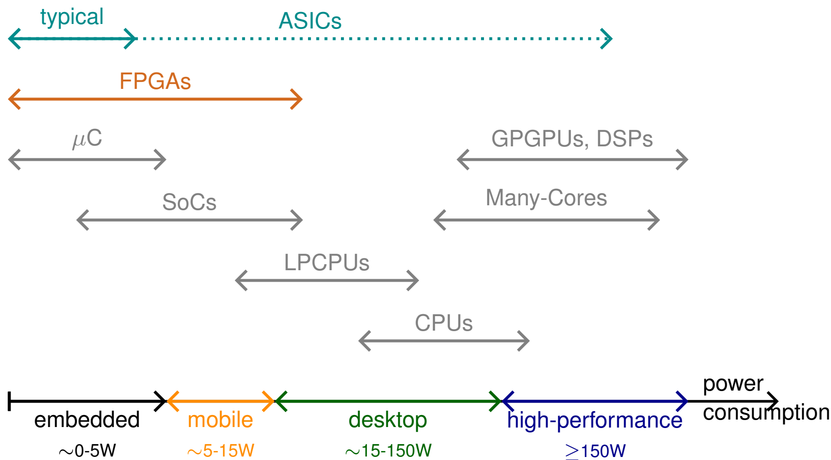
- **performance requirements** – is execution time critical?
- **energy sensitivity** – is power consumption critical?
- **development time/cost** – how quickly changes the problem/algorithm?

Which technology should we use?
Depends on the kind of computing



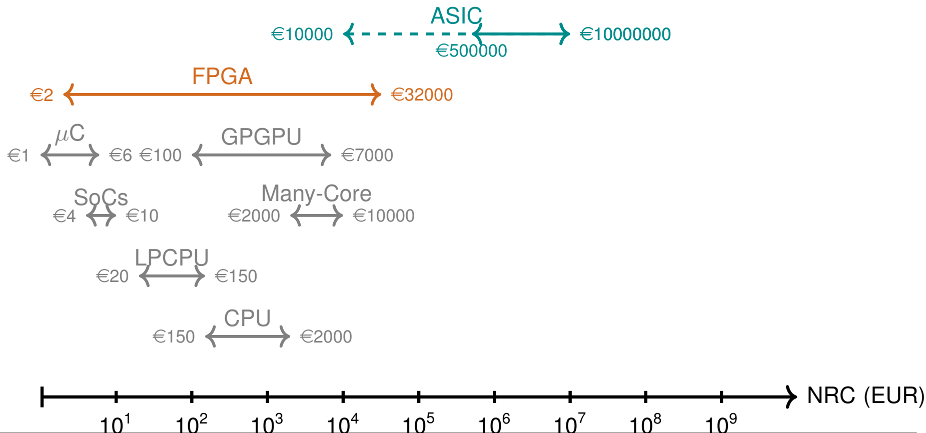
Which technology should we use?
Depends on the available energy

TECHNISCHE UNIVERSITÄT DARMSTADT



Which technology should we use?
Depends on the development cost

TECHNISCHE UNIVERSITÄT DARMSTADT



Which technology should we use?
Depends on the development cost

TECHNISCHE UNIVERSITÄT DARMSTADT

Non-Recurring Costs (Development)

- can be split into two components:
  1. cost of developers (€/h)
  2. time required for each development iteration ($\sim$ speed / flexibility)

1. cost of developers $\sim$ avail. number of developers trained in tech
2. development time $\sim$ level of customization
   - COTS solutions w/commodity ISAs have fastest turn-around (days)
   - COTS solutions w/specialized ISAs slightly slower (days, weeks)
   - FPGA solutions often slower by 10x or more (months)
   - ASIC solutions require manufacturing (months, years)

Focus of my work - methods, tools, frameworks, . . .

Which technology should we use?
Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

	NRC	Flexibility	Performance	Energy-Efficiency
commodity ISAs	+++	++	+	-
specialized ISAs	-	+	++	- - -
ASICs	- - -	- - -	+++	+++
FPGAs	- -	+++	++	++

working on it!

Recap quotes
True?

TECHNISCHE
UNIVERSITÄT
DARMSTADT

"FPGAs are for kids and academics. Everyone who needs to do real work uses ASICs."

Anonymous

Misunderstanding of the technology

- problems that qualify for an ASIC need to
  - change very little
  - stay relevant

  for the next **5-10 years!** (to amortize NRC)
- *no silver bullet* – different problems require different solutions

Recap quotes
True?

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Every two years, FPGAs are 'finally going to arrive'.

Peter Lee, VP of Microsoft Research

Project Catapult

- supervised by the same Peter Lee
- started in 2010 as datacenter tech to accelerate Bing w/FPGAs
- 2012: large scale pilot program, custom FPGA board (1623 servers)
- 2016: "Configurable Cloud" in nearly every production server (MICRO 2016)

Recap quotes
True?

TECHNISCHE UNIVERSITÄT DARMSTADT

Every two years, FPGAs are 'finally going to arrive'.

Peter Lee, VP of Microsoft Research

Amazon Elastic Compute Cloud (EC2)

- cloud computing datacenters now use FPGAs
- new "F1" instance provides user-configurable FPGAs (Xilinx UltraScale+)
- currently in developer preview, but considered a significant step

Recap quotes
True?

TECHNISCHE UNIVERSITÄT DARMSTADT

Every two years, FPGAs are 'finally going to arrive'.

Peter Lee, VP of Microsoft Research

Intel buys Altera (2015)

- one of two largest FPGA vendors, for approx. $16.7 Billion
- merging Intel Xeons with Altera Stratix FPGAs
- shrinking their traditional divisions by 11% at the same time (2016)

Recap quotes
True?

TECHNISCHE UNIVERSITÄT DARMSTADT

Every two years, FPGAs are 'finally going to arrive'.

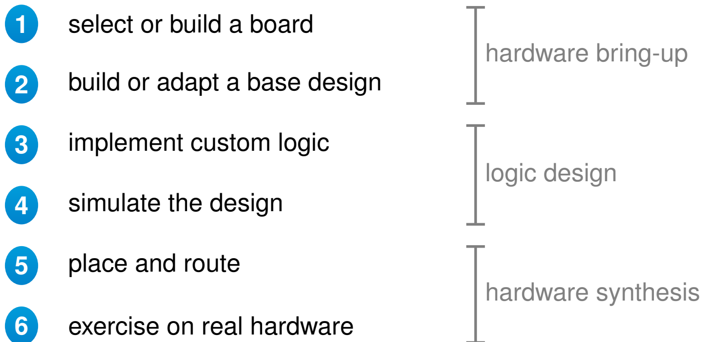
Peter Lee, VP of Microsoft Research

... not sure when they are "going to arrive", but:

- are not going away any time soon now
- FPGA devs and engineers are already in short supply
- will be a much sought-after skillset in years to come

Typical Workflow for an FPGA-based solution

TECHNISCHE UNIVERSITÄT DARMSTADT



Typical Workflow for an FPGA-based solution

1 select or build a board

TECHNISCHE UNIVERSITÄT DARMSTADT

High flexibility comes at a cost:

- COTS hardware differs in details, but is mostly homogeneous
- FPGA-based solutions are much more heterogeneous!
- even off-the-shelf FPGA boards vary wildly:
  1. **on-chip**, i.e., in the FPGA itself
  2. **off-chip**, i.e., in the peripheral components

Typical Workflow for an FPGA-based solution
Details: Differences between FPGAs I

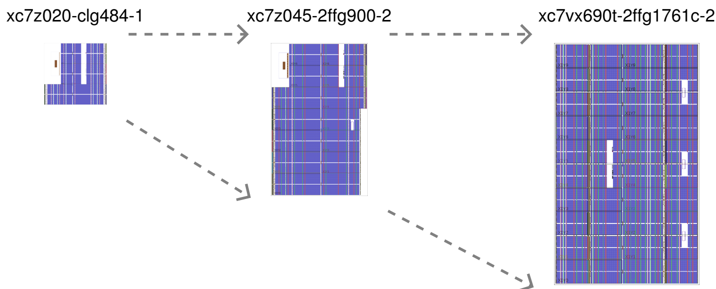
TECHNISCHE UNIVERSITÄT DARMSTADT

FPGA vary in their characteristics regarding

- architectural family (on-chip resources)
- speed grade ($\sim$ max. operating frequency)
- area and slice composition *(more on next slide . . .)*
- miscellaneous other properties:
  - energy consumption
  - space hardening
  - industrial
  - military
  - . . .

Typical Workflow for an FPGA-based solution
Details: Differences between FPGAs II

FPGAs differ in the size of the reconfigurable area:

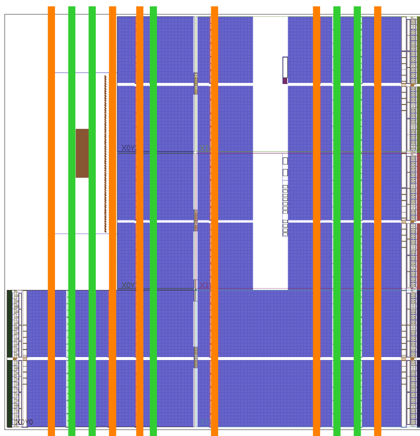


Typical Workflow for an FPGA-based solution
Details: Differences between FPGAs III

TECHNISCHE UNIVERSITÄT DARMSTADT

FPGAs differ in the composition of the reconfigurable area:

- Common Logic Blocks (CLBs) are interspersed with *hard blocks*:
  - block RAM (BRAM)
    on-chip memory (kBits)
  - digital signal processors (DSPs)
    dedicated arithmetic units
  - shift registers, Muxes, ...
- proportion, arrangement vary between *FPGA families*
- families target classes of applications
- variants address specific application requirements:
  - space hardened, automotive, military, ...



Typical Workflow for an FPGA-based solution
Details: Differences in the periphery

TECHNISCHE UNIVERSITÄT DARMSTADT

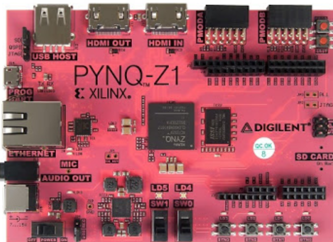
Off-the-shelf FPGA boards provide different peripheral components:

- connectivity to host / outside world, e.g.:
  - PCIe Gen2/3
  - Interlaken
  - RapidIO, Hypertransport
  - Gen-Z
  - Network
  - Shared Memory
  - ...
- accessible memories, caches, e.g.:
  - DDRx, LPDDRx, ...
  - RLDRAM, NVRAM, ...
  - HMC/3D-Memory, ...
- and secondary peripherals, e.g.:
  LEDs, Displays, Video In/Out, Sensors, ...

Typical Workflow for an FPGA-based solution
Details: Differences between FPGA boards

TECHNISCHE UNIVERSITÄT DARMSTADT

Xilinx PyNQ Z1 – Zynq-series FPGA-SoC (Artix-7)

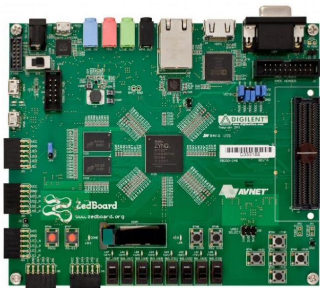


Host connectivity:
AXI3 hard IP (SoC)
Memory:
512MB DDR3, SD card
Other Periphery:
1G Network, HDMI In+Out, Analog Audio, Microphone, LEDs, Switches, Buttons, . . .
Size:
∼85.000 Logic Cells
Price:
∼229 USD — **65 USD (academic)**

Typical Workflow for an FPGA-based solution
Details: Differences between FPGA boards

TECHNISCHE UNIVERSITÄT DARMSTADT

Digilent ZedBoard – Zynq-series FPGA-SoC (Artix-7)

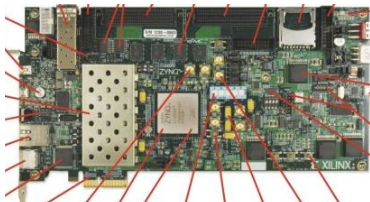


Host connectivity:
AXI3 hard IP (SoC)
Memory:
512MB DDR3, 256 Mbit Quad-SPI Flash, SD card
Other Periphery:
1G Network, FMC, VGA Out, Analog Audio/Video, PMOD, OLED display (128x32), LEDs, Switches, Buttons, . . .
Size:
∼85.000 Logic Cells
Price:
∼495 USD — **319 USD (academic)**

Typical Workflow for an FPGA-based solution
Details: Differences between FPGA boards

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Xilinx ZC706 – Zynq-series FPGA-SoC (Kintex-7) w/PCIe Gen2.0 x4

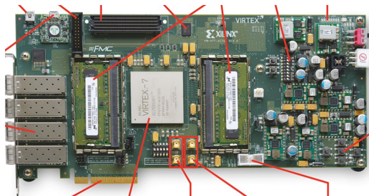


Host connectivity:
AXI3 hard IP (SoC), PCIe Gen2 x4, 1x SFP+ transceiver
Memory:
1GB DDR3 (PL), 1GB DDR3 (PS), 2x128Mbit (Dual) Quad-SPI, SD card
Other Periphery:
2x FMC, Differential InOuts, . . .
Size:
∼350.000 Logic Cells
Price:
∼2.850 USD

Typical Workflow for an FPGA-based solution
Details: Differences between FPGA boards

TECHNISCHE UNIVERSITÄT DARMSTADT

Xilinx VC709 – Virtex-7 FPGA w/PCIe Gen3.0 x8



Host connectivity:
PCIe Gen3 x8, 4x SFP+ transceiver
Memory:
2x 4GB DDR3, 128 MB BPI flash
Other Periphery:
1x FMC, Differential InOuts, 8pole DIP switch, . . .
Size:
$\sim$690.000 Logic Cells
Price:
$\sim$5.650 USD

Typical Workflow for an FPGA-based solution

2 build or adapt a base design

TECHNISCHE UNIVERSITÄT DARMSTADT

What is 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*
  - which physical pin is connected to which input/output in my design?
  - this is called a *pin constraint*
  - *Example: drive an LED from a logical output*
- great – almost complete wiring freedom, but:
  - you actually need to connect memories (e.g., DDR) on this level!
  - same is true for the connection to the host
  - completely different for each FPGA board
- a *base design* is an empty design with default connections (DDR, . . .)
  - done by vendor for off-the-shelf boards
  - must be done manually in case of custom PCB

Typical Workflow for an FPGA-based solution

3-4 Logic Design

TECHNISCHE UNIVERSITÄT DARMSTADT

Develop the custom logic

- develop a hardware module for the algorithm
- usually done in HDLs, e.g., Verilog/SystemVerilog, VHDL
- more recently: Bluespec, Chisel, . . . (alternative/high-level HDLs)

Test functional correctness by simulation

- behavioral hardware descriptions are simulated on RTL level
- similar to unit testing in software
- aim for coverage to gain confidence in functional correctness

Independent of target hardware (in theory)

- RTL is abstract; can be mapped to any FPGA, or ASIC
- in practice: scalability and portability issues

5-6 Hardware Synthesis

Place and route

- two step process:
  1. *place* step maps operations to hardware resources
  2. *route* step performs the wiring and checks timing constraints
- NP-complete problem!
- number of possible mappings grows exponentially with area
- many different algorithmic approaches:
  - heuristic approach with iterative backtracking
  - simulated annealing
  - analytical methods
- most time-consuming step (anywhere from 15min to days)

Exercise the design on real hardware

- necessary! several uncertainties in the process
- real-world perfomance is determined by many outside factors, e.g., OS interactions

Typical Workflow for an FPGA-based solution
Place and Route Example

TECHNISCHE UNIVERSITÄT DARMSTADT

A simple module: 2-bit counter

```
'timescale 1ns / 1ps
module blinkenlights(
    input wire rst,
    input wire clk,
    output reg[1:0] cnt
    );
  initial     cnt <= 'd0;
  always @(posedge clk) begin
    if (rst) cnt <= 'd0;
    else     cnt <= cnt + 1;
  end
endmodule
```

rst	cnt[1]	cnt[0]	cnt'[1]	cnt'[0]
0	0	0	**0**	**1**
0	0	1	**1**	**0**
0	1	0	**1**	**1**
0	1	1	**0**	**0**
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

rst=1

Typical Workflow for an FPGA-based solution
Place and Route Example

TECHNISCHE UNIVERSITÄT DARMSTADT

A simple module: 2-bit counter

LUT2

rst	cnt[0]	cnt'[0]
0	0	**1**
0	1	**0**
0	0	**1**
0	1	**0**
1	0	0
1	1	0
1	0	0
1	1	0

rst=1

LUT3

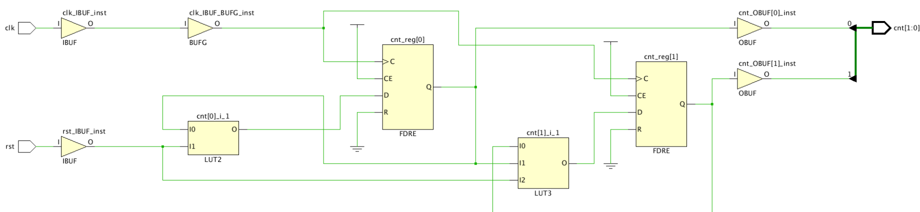
rst	cnt[1]	cnt[0]	cnt'[1]
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

rst=1

Typical Workflow for an FPGA-based solution
Place and Route Example

TECHNISCHE UNIVERSITÄT DARMSTADT

Synthesis result for module



- result of synthesis is called **netlist**
- consists of primitive **instances** and **nets**
- instances here as expected: 1x LUT2, 1x LUT3 + 2x FlipFlops
- **placing** is the process of mapping instances to hardware resources

Typical Workflow for an FPGA-based solution
Place and Route Example

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Recall: Common Logic Blocks (CLB)

- basic elements of configurable logic
- on Xilinx devices, CLBs usually contain
  1. 4x LUT6
  2. 8x Storage elements
  3. 3x Wide-function Multiplexer (MUX)
  4. Carry logic
- LUT6 can be configured as
  1. 1x 6-input-1-output, or
  2. 2x $\leq$5-input-2-output
- multiple LUTs can be combined with
  1. F7-MUXes to generate up to 2x 7-input-1-output LUTs
  2. F8-MUX to generate 1x 8-input-1-output LUT

A6 O6
A5
A4
A3 O5
A2 O5
A1

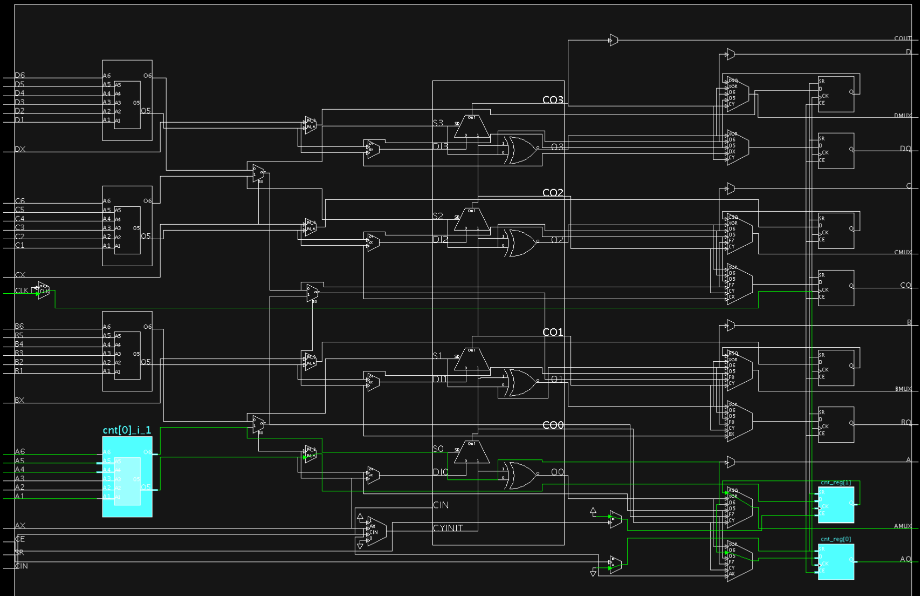
A6 O6
A5
A4
A3 O5
A2 O5
A1

A6 O6
A5
A4
A3 O5
A2 O5
A1

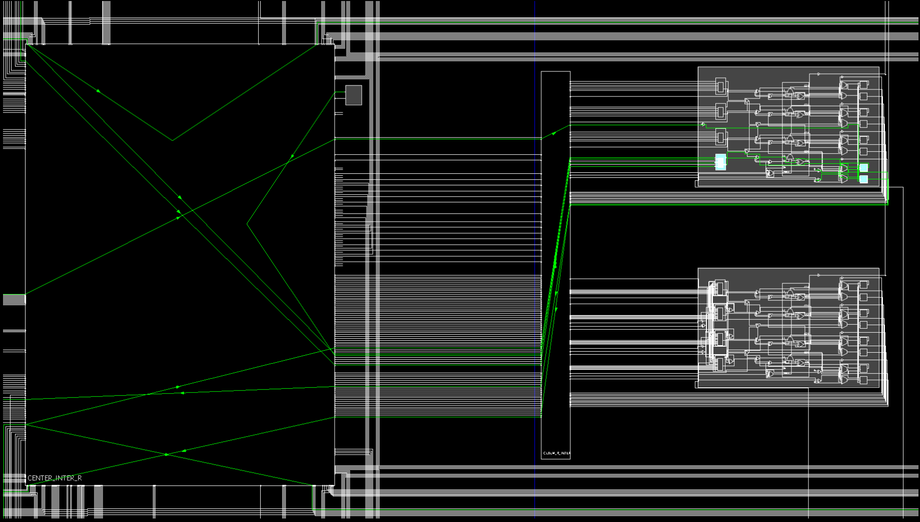
cnt[0]_i_1
A6 O6
A5
A4
A3 O5
A2 O5
A1

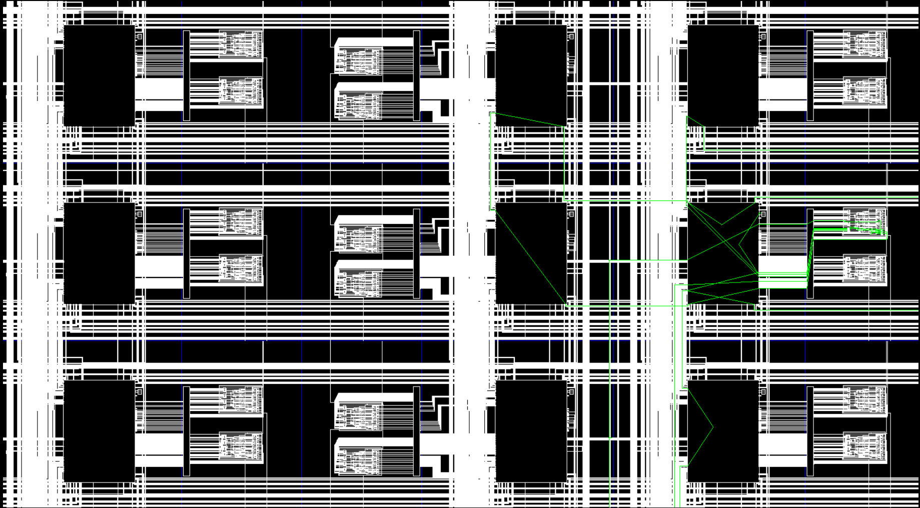
S3 CO3
DI3 O3
S2 CO2
DI2 O2
S1 CO1
DI1 O1
S0 CO0
DI0
CIN O0
CYINIT

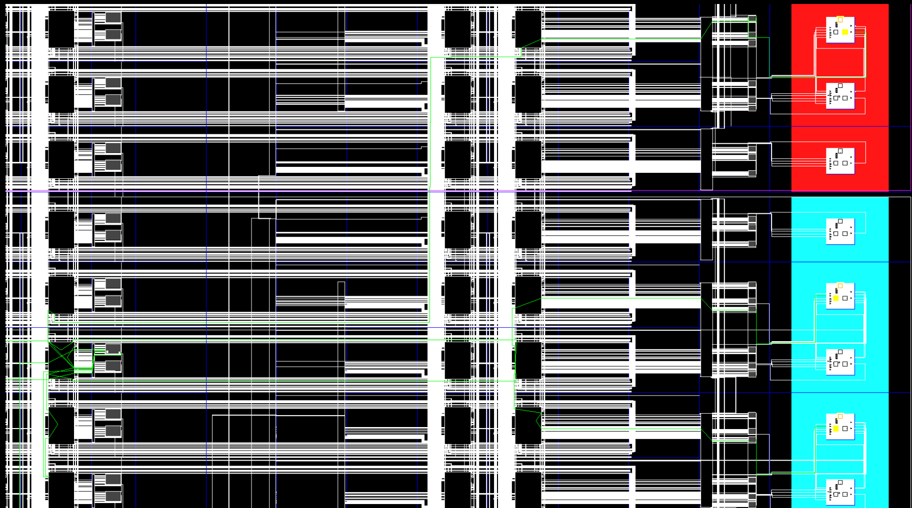
SR D CK CE Q
SR D CK CE Q
SR D CK CE Q
SR D CK CE Q
SR D CK CE Q
SR D CK CE Q
cnt_reg[1]
SR D CK CE Q
cnt_reg[0]
SR D CK CE Q

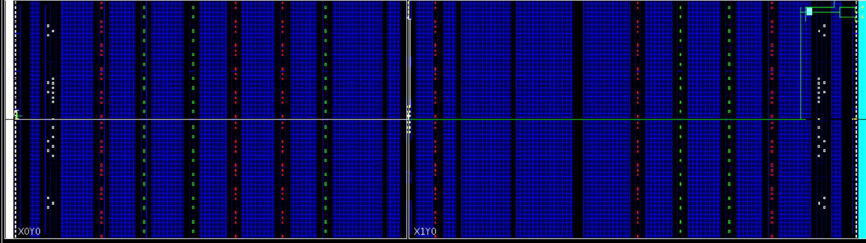


COUT
D
CO3
DMUX
S3
DI3
O3
DQ
C
CO2
S2
DI2
O2
CMUX
CQ
CLK
CO1
B
S1
DI1
O1
BMUX
BQ
CO0
cnt[0]_i_1
S0
DI0
O0
A
CIN
cnt_reg[1]
CYINIT
AMUX
cnt_reg[0]
AQ
D6
D5
D4
D3
D2
D1
DX
C6
C5
C4
C3
C2
C1
CX
B6
B5
B4
B3
B2
B1
BX
A6
A5
A4
A3
A2
A1
AX
CE
SR
CIN

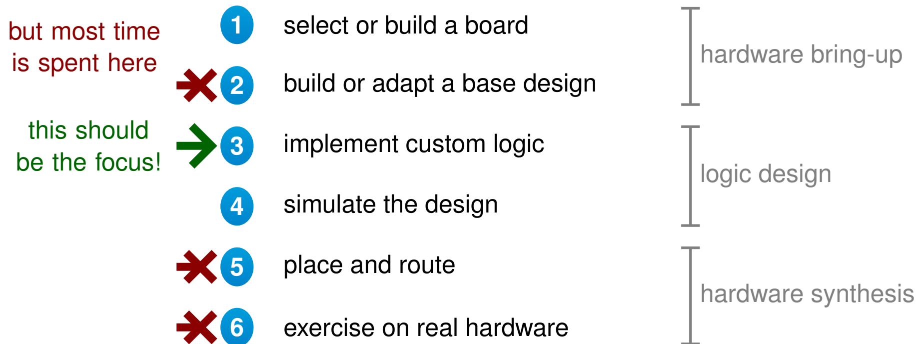

CENTER_INTER_R






X0Y2
X1Y2
X0Y1
X1Y1
X0Y0
X1Y0

Typical Workflow for an FPGA-based solution
Recap: Current Problems



Typical Workflow for an FPGA-based solution
Recap: Current Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

Two major issues driving up the NRC in FPGA-based solutions

Portability

- hardware choices made at start influence the entire workflow, result
- change of hw leads to repetition of most time-consuming steps
- makes choice of hardware **both critical and difficult**

addressed by **ThreadPoolComposer**
(this lecture)

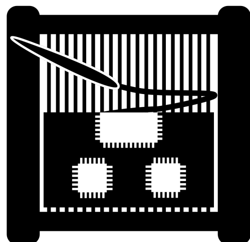
Scalability

- resources vary significantly; hardware designs often cannot scale the full range
- classic HDLs (Verilog, VHDL) are too verbose and not expressive enough
- good test coverage is hard to obtain; repetitive development efforts

addressed by Bluespec, **Chisel**, . . .
but also **ThreadPoolComposer** (coarse-grain)

ThreadPoolComposer

TECHNISCHE UNIVERSITÄT DARMSTADT



- automated tool-flow generating bitstreams for four *Platforms*:
  - Xilinx PyNQ
  - Digilent ZedBoard
  - Xilinx ZC706
  - Xilinx VC709
- automated *high-level synthesis (HLS)* pass to generate hardware from C/C++
- uniform *application programming interface (API)* - write software once!
- based on Scala and Vivado IP Integrator Tcl
- **free software, LGPLv2 license**

ThreadPoolComposer
Underlying model and assumptions

Task Parallelism

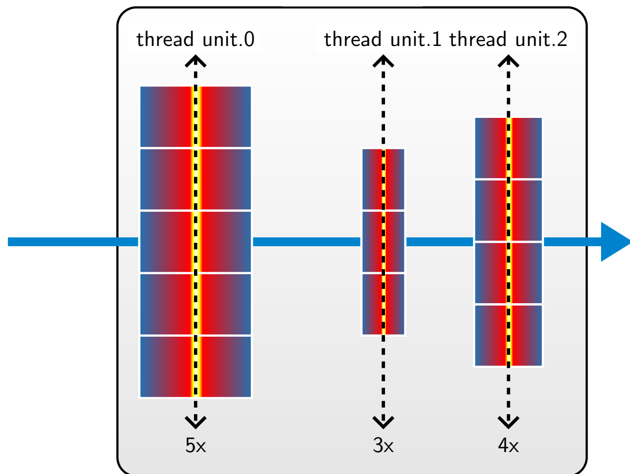
- TPC implements a **task parallel model of computation**
- computation can be split into independent **tasks**
- parallelization across multiple **processing elements (PEs)**
- data for each task can be partitioned into **jobs**
- jobs are scheduled on the PEs asynchronously

Assumptions / Requirements

- problem can be decomposed into independent jobs
- jobs can be executed in parallel, communication is limited:
  - inter-task communication is not performance critical
  - no inter-job communication, jobs can be completed independently

ThreadPoolComposer
Parallelising a sequential program

TECHNISCHE UNIVERSITÄT DARMSTADT



- profile sequential program
- identify **computational hotspots (kernels)**
- isolate kernel code, execute non-critical code on main thread
- replicate hardware for each kernel spatially: **parallel processing elements (PE)**
- PEs for each kernel are called **thread unit**
- **Composition**: size of thread units for each kernel *here:* $(5, 3, 4)$

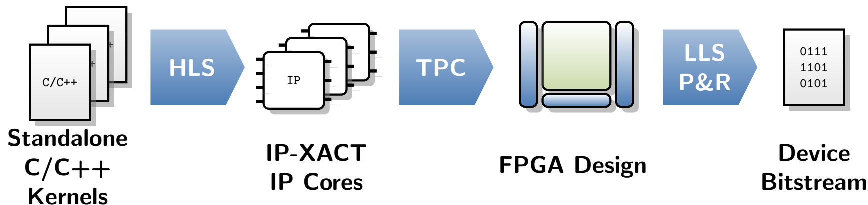
ThreadPoolComposer
Parallelism in TPC

TECHNISCHE
UNIVERSITÄT
DARMSTADT

- many terms for this type of parallelism:
  - task parallelism
  - coarse-grained parallelism
  - fork-join parallelism
  - work group parallelism
  - ...
- we use the term **thread pool**
  - conceptual abstraction over a *Composition*:
    - a pool consists of a number of thread units for kernels
    - each thread unit consists of a number of PEs (instantiations of kernels)
  - operational abstraction:
    - jobs for each kernel can be submitted to the pool
    - are executed on first available PE
    - results can be collected out-of-order
  - no inter-PE communication (handled on main-thread)

ThreadPoolComposer
Compilation Flow

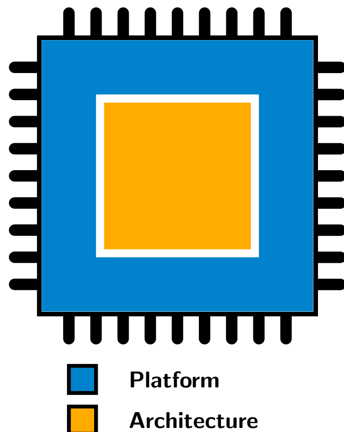
TECHNISCHE UNIVERSITÄT DARMSTADT



ThreadPoolComposer
Design Abstractions

TECHNISCHE
UNIVERSITÄT
DARMSTADT

- TPC designs are split into two distinct parts:
  1. **Architecture**
     - organization of thread units and PEs
     - independent of target platform / board
  2. **Platform**
     - connection to host and memory
     - hardware-dependent
- advantage: hardware-dependent parts are isolated in **Platform**
- represented in TPC as plug-in scripts
  - easy to modify / re-use existing scripts
  - easy to add new **Platforms** and **Architectures**



ThreadPoolComposer
Architecture Scripts

TECHNISCHE UNIVERSITÄT DARMSTADT

Tasks

1. define basic interface of PEs (HLS directives)
2. instantiate PEs according to composition
3. combine PEs into thread units + perform wiring
4. combine thread units into thread pool + perform wiring

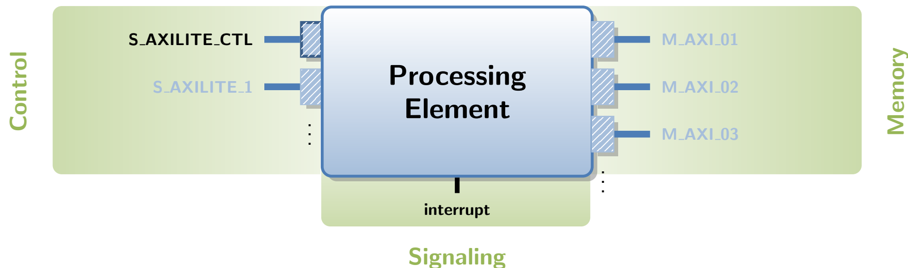
Example Architecture: baseline (AXI4)

- uses AXI4 interfaces for communication
- uses AXI4 interconnects to facilitate communication between host and PEs

ThreadPoolComposer
Example Architecture: baseline

TECHNISCHE UNIVERSITÄT DARMSTADT

PE Interface: AXI4-based

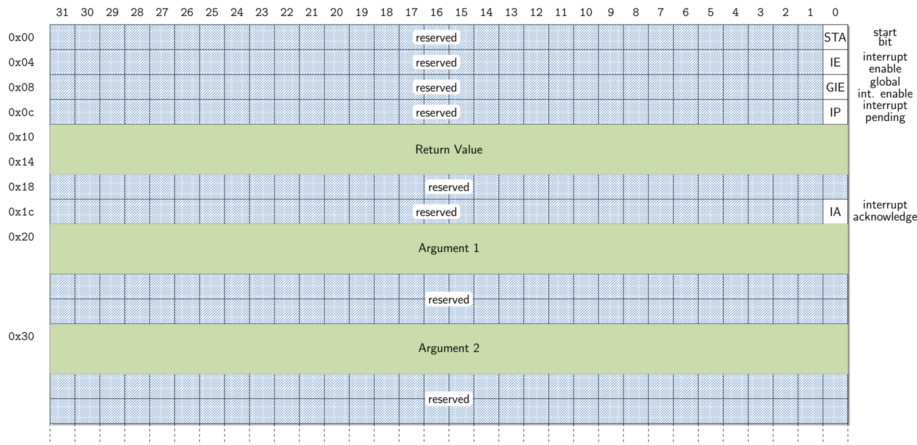


- AXI4Lite slave interface (memory-mapped)
- standardized control register file

- level-sensitive interrupt line
- logic 1 ⇔ completed execution awaits acknowledgement

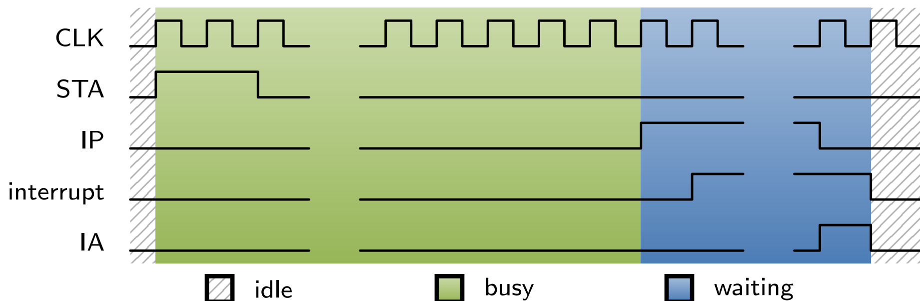
- *optional:* AXI4/AXI4Lite master interfaces

ThreadPoolComposer
Memory-Mapped AXI4Lite Control Register File



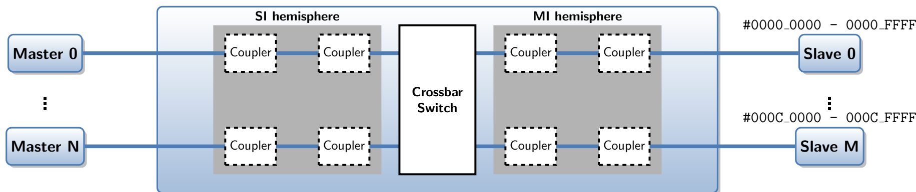
ThreadPoolComposer
PE timing diagram

TECHNISCHE UNIVERSITÄT DARMSTADT



ThreadPoolComposer
AXI4 Interconnect IP (Xilinx)

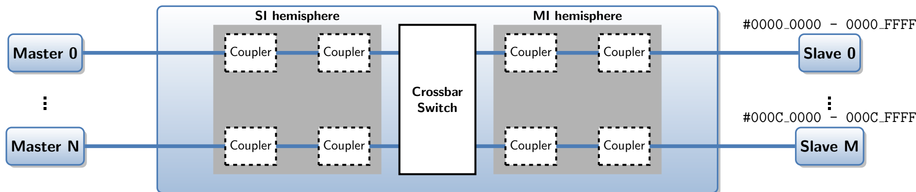
TECHNISCHE UNIVERSITÄT DARMSTADT



- connects up to 16 master interfaces with up to 16 slave interfaces
- slaves are mapped into master adress spaces to distinguish accesses
- **Shared-Address-Multiple-Data (SAMD)** topology
- recall: *AXI is not a bus!* point-to-point

- crossbar implements *time-multiplexed arbitration scheme*
- e.g., round-robin, priority-based, ...
- switches occur at end of bursts
- dual channel, read/write separate

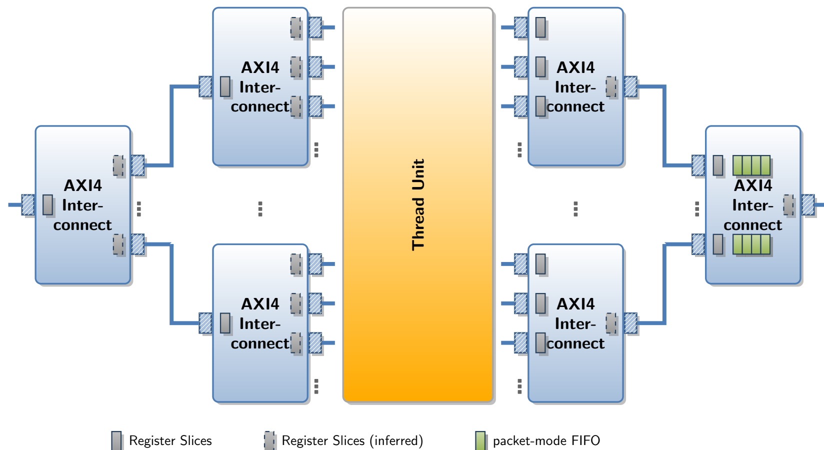
ThreadPoolComposer
AXI4 Interconnect IP (Xilinx)

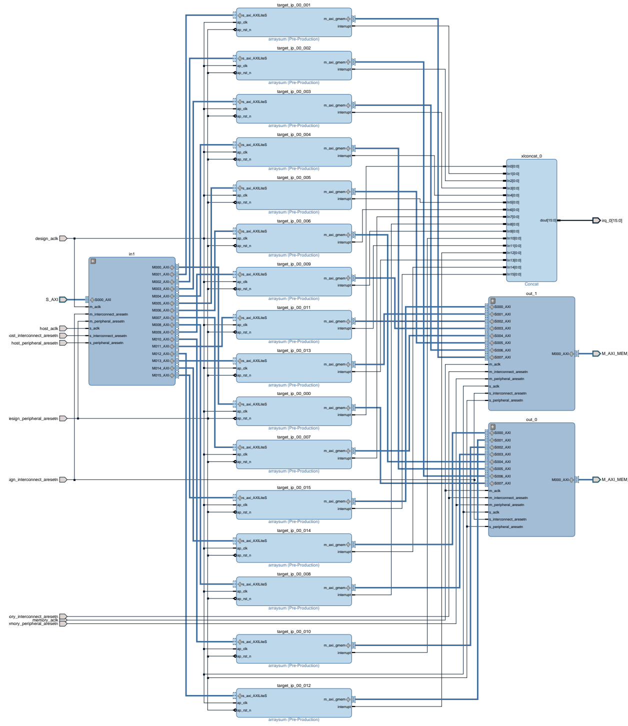


- more complicated: provides optional **couplers**
  - data width converter *(e.g., 32b → 8b)*
  - clock converter *(e.g., 250 MHz → 100 Mhz)*
  - protocol converter *(e.g., AXI4 → AXI4Lite)*
  - register slice *(relax critical paths)*
  - shallow/deep data FIFO *(latency-insensitive decoupling)*
  - AXI MMU
- AXI workhorse — highly versatile, high performance module
- simplifies AXI designs significantly

ThreadPoolComposer
Concrete Architecture: baseline

TECHNISCHE UNIVERSITÄT DARMSTADT





target_ip_00_001
target_ip_00_002
target_ip_00_003
target_ip_00_004
target_ip_00_005
target_ip_00_006
target_ip_00_009
target_ip_00_011
target_ip_00_013
target_ip_00_000
target_ip_00_007
target_ip_00_015
target_ip_00_014
target_ip_00_008
target_ip_00_010
target_ip_00_012
arraysum (Pre-Production)
s_axi_AXILiteS
ap_clk
ap_rst_n
m_axi_gmem
interrupt
xlconcat_0
Concat
dout[15:0]
irq_0[15:0]
in1
S000_AXI
M000_AXI
M015_AXI
m_aclk
m_interconnect_aresetn
m_peripheral_aresetn
s_aclk
s_interconnect_aresetn
s_peripheral_aresetn
out_1
out_0
M_AXI_MEM
design_aclk
S_AXI
host_aclk
host_peripheral_aresetn
memory_aclk

ThreadPoolComposer
Address Map: PE Slaves and Masters

TECHNISCHE UNIVERSITÄT DARMSTADT

Host/ps7
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
InterruptControl/axi_intc_00	s_axi	Reg	0x8180_0000	64K	0x8180_FFFF
Threadpool/target_ip_00_000	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF
Threadpool/target_ip_00_001	s_axi_AXILiteS	Reg	0x43C1_0000	64K	0x43C1_FFFF
Threadpool/target_ip_00_002	s_axi_AXILiteS	Reg	0x43C2_0000	64K	0x43C2_FFFF
Threadpool/target_ip_00_003	s_axi_AXILiteS	Reg	0x43C3_0000	64K	0x43C3_FFFF
Threadpool/target_ip_00_004	s_axi_AXILiteS	Reg	0x43C4_0000	64K	0x43C4_FFFF
Threadpool/target_ip_00_005	s_axi_AXILiteS	Reg	0x43C5_0000	64K	0x43C5_FFFF
Threadpool/target_ip_00_006	s_axi_AXILiteS	Reg	0x43C6_0000	64K	0x43C6_FFFF
Threadpool/target_ip_00_007	s_axi_AXILiteS	Reg	0x43C7_0000	64K	0x43C7_FFFF
Threadpool/target_ip_00_008	s_axi_AXILiteS	Reg	0x43C8_0000	64K	0x43C8_FFFF
Threadpool/target_ip_00_009	s_axi_AXILiteS	Reg	0x43C9_0000	64K	0x43C9_FFFF
Threadpool/target_ip_00_010	s_axi_AXILiteS	Reg	0x43CA_0000	64K	0x43CA_FFFF
Threadpool/target_ip_00_011	s_axi_AXILiteS	Reg	0x43CB_0000	64K	0x43CB_FFFF
Threadpool/target_ip_00_012	s_axi_AXILiteS	Reg	0x43CC_0000	64K	0x43CC_FFFF
Threadpool/target_ip_00_013	s_axi_AXILiteS	Reg	0x43CD_0000	64K	0x43CD_FFFF
Threadpool/target_ip_00_014	s_axi_AXILiteS	Reg	0x43CE_0000	64K	0x43CE_FFFF
Threadpool/target_ip_00_015	s_axi_AXILiteS	Reg	0x43CF_0000	64K	0x43CF_FFFF
tpc_status	S00_AXI	S00_AXI_reg	0x7777_0000	64K	0x7777_FFFF
Unconnected Slaves					
Host/ps7	S_AXI_ACP	ACP_DDR_LO...			
Host/ps7	S_AXI_ACP	ACP_QSPI_LIN...			
Host/ps7	S_AXI_ACP	ACP_IOP			
Host/ps7	S_AXI_ACP	ACP_M_AXI_G...			
Host/ps7	S_AXI_ACP	ACP_M_AXI_G...			

Threadpool/target_ip_00_000					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_001					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_002					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_003					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_004					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_005					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_006					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_007					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_008					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_009					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_010					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_011					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_012					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_013					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_014					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP0	HP0_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF
Threadpool/target_ip_00_015					
Data_m_axi_gmem (32 address bits : 4G)					
Host/ps7	S_AXI_HP2	HP2_DDR_LO...	0x0000_0000	512M	0x1FFF_FFFF

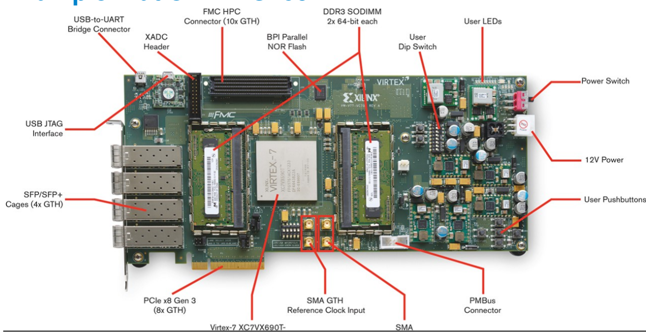
ThreadPoolComposer
Platform Scripts

TECHNISCHE UNIVERSITÄT DARMSTADT

Tasks

1. provide control connection from host to PEs
2. provide PEs with access to memory
3. provide signaling interface from PEs to host
4. *optional: instantiate additional hardware infrastructure*

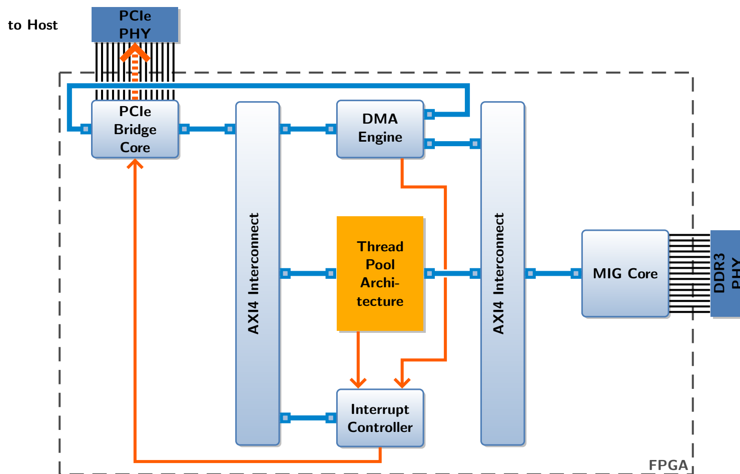
Example Platform: VC709



- *host connection:* PCIe Gen3 x8
- *memory:* 2×4 GiB DDR3
- *signaling:* MSI-X packetized in-band interrupts
- *other:* User LEDs (GPIO)

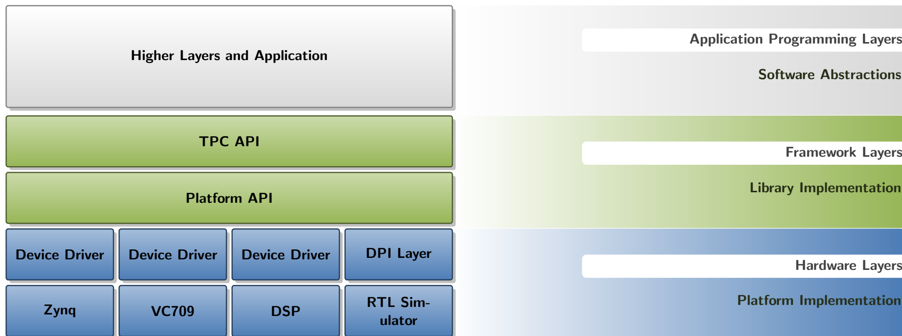
ThreadPoolComposer
Concrete Platform: VC709

TECHNISCHE UNIVERSITÄT DARMSTADT



ThreadPoolComposer
Software Stack

TECHNISCHE UNIVERSITÄT DARMSTADT



ThreadPoolComposer
OS-level Integration

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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 (exactly like "regular" hardware)
- in case of monolithic kernels (Linux, Windows, Mac OS):
  - driver code becomes part of the 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*
- *. . . no details here, will be discussed in OS lecture*

Platform API

- software counterpart to Platform on-chip
- minimal userspace abstraction over the device driver layer
- low-level integration tasks:
  - read/write control registers
    1. `platform_read_ctl`
    2. `platform_write_ctl`
  - manage + read/write device memory
    1. `platform_alloc`
    2. `platform_dealloc`
    3. `platform_read_mem`
    4. `platform_write_mem`
  - wait for signals
    1. `platform_wait_for_irq`
    2. `platform_write_ctl_and_wait`
  - query device address space
    1. `platform_address_get_pe_base`
    2. `platform_address_get_infrastructure_base`
  - ... and initialization/administrative functions

TPC API

- user-facing API for TPC applications
- device memory management
  1. `tpc_device_alloc` — wrapper for `platform_alloc`
  2. `tpc_device_dealloc` — wrapper for `platform_dealloc`
  3. `tpc_device_copy_to` — wrapper for `platform_write_mem`
  4. `tpc_device_copy_from` — wrapper for `platform_read_mem`
- high-level task management
  1. `tpc_device_acquire_job_id`
  2. `tpc_device_release_job_id`
  3. `tpc_device_job_set_arg`
  4. `tpc_device_job_get_arg`
  5. `tpc_device_job_get_return`
  6. `tpc_device_job_launch`
- ... and initialization/administrative functions

TECHNISCHE UNIVERSITÄT DARMSTADT

ThreadPoolComposer Example Application

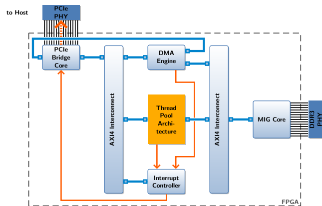
int some_kernel(uint8_t data[10

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data          , TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

- allocate 1 KiB of device-accessible data
- copy user data block to device
- acquire job
- set pass-by-reference arg
- set pass-by-value arg
- launch blocking returns after completion
- retrieve by-value return value
- retrieve by-reference data block
- release job
- free device-accessible data

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)



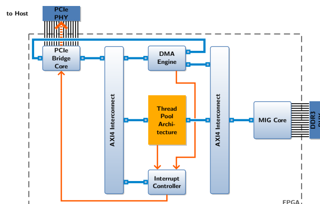
`tpc_device_alloc`

1. TPC library forwards memory allocation request to Platform library
2. Platform library allocates memory on device (via *three-tiered buddy allocation*)
   - `tpc_handle_t` contains the device address

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)



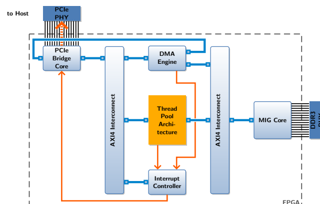

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

`tpc_device_copy_to`

1. device driver copies data from virtual address in user space to *kernel space buffer*
2. device driver translates *kernel virtual address* to *physical/bus address*
3. device driver sets DMA engine registers:
   3.1 kernel buffer physical base address
   3.2 kernel buffer length
   3.3 device memory destination address
   3.4 copy direction: to device
4. DMA engine copies data
5. DMA engine raises interrupt to signal host

ThreadPoolComposer Example Application

int some_kernel(uint8_t data[1024], const int x)



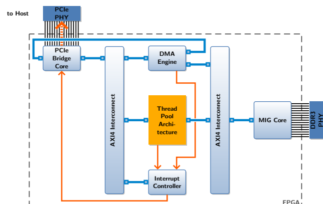
tpc_device_acquire_job_id

1. TPC library acquires *job object* userspace struct, contains
   - thread unit ID
   - buffers for all argument values

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)



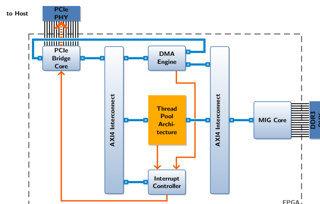
tpc_device_job_set_arg

1. application prepares job in memory
   - sets argument values in job object

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

ThreadPoolComposer Example Application

int some_kernel(uint8_t data[1024], const int x)

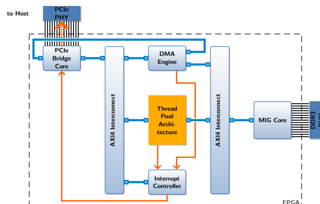



```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

`tpc_device_job_launch`

1. application submits job object to pool
2. TPC library acquires first idle PE in the thread unit
3. TPC library writes register values (AXI4Lite register file) via Platform library
4. TPC library writes start bit and waits for interrupt via Platform library
5. TPC library copies back any register values and arguments to the job object
6. TPC library releases the PE

ThreadPoolComposer
Example Application
int some_kernel(uint8_t data[1024], const int x)



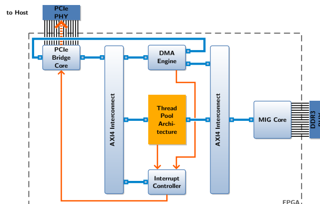
tpc_device_job_get_arg

1. application fetches return value from job object

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)



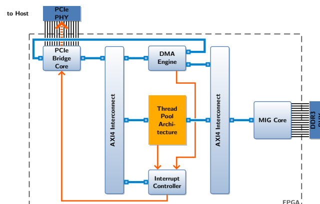

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

`tpc_device_copy_from`

1. device driver reserves kernel buffer
2. device driver sets DMA engine registers:
   2.1 device memory source address
   2.2 kernel buffer length
   2.3 kernel buffer physical base address
   2.4 copy direction: from device
3. DMA engine copies data
4. DMA engine raises interrupt to signal host

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)



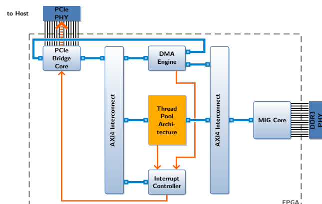
tpc_device_release_job_id

1. TPC library releases job object

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)



tpc_device_free

1. TPC library forwards memory release to Platform library
2. Platform library marks block as free in buddy allocator

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);

tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, "some_kernel");

tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
tpc_device_job_set_arg(dev, j_id, 1, sizeof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

ThreadPoolComposer
Example Application

int some_kernel(uint8_t data[1024], const int x)

```
tpc_device_t dev = ...;
const int x = 42;
tpc_handle_t h = tpc_device_alloc(dev, 1024, 0);

tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);
```

... Wait! The 80’s called, they want their C-code back! All this for a single kernel execution?!

```
ob_id(dev, "some_kernel");

eof(h), &h);
eof(x), &x);

tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);

int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
tpc_device_copy_from(dev, h, &data, 1024, TPC_BLOCKING_MODE);

printf("result of job: %d\n", r);

tpc_device_release_job_id(dev, j_id);

tpc_device_free(dev, h);
```

TPC++ API: Example

C++11 to the rescue!

TECHNISCHE UNIVERSITÄT DARMSTADT

hides initialization, device setup, . . .

variadic template method: handles job, manages copies, . . . automatically!

```
TPC tpc();

int r = tpc.launch("some_kernel", &data, x);

std::cout << "result of job: " << r << std::endl;
```

- simple is beautiful - simplest API that "gets the job done"
- full access to lower levels for corner cases
- C API resembles *OpenCL* - no coincidence, reduce learning curve for developers
- C++ API - *facilitate experimentation* with minimal effort/code

ThreadPoolComposer
Challenges addressed by TPC

TECHNISCHE UNIVERSITÄT DARMSTADT

Portability

- isolated Platform from Architecture
- can reuse Architectures on all Platforms
- can reuse software code via API hierarchy

Scalability

- can control design frequency
- can control area utilization via composition
- can generate design automatically for frequency + composition pairs

. . . but what is the maximal pair for any given Platform?

ThreadPoolComposer
Estimating Upper Bounds: Design Frequency

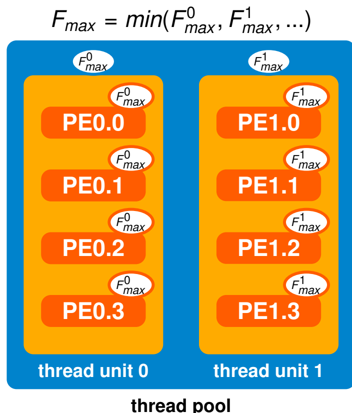
TECHNISCHE UNIVERSITÄT DARMSTADT

Hierarchical Approach

- 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

How to obtain an upper bound of F_{max} for a PE?

- perform synthesis + place and route in **out-of-context mode**
  - synthesize netlist for PE
  - place and route in fixed location
  - length of critical path $\rightarrow$ estimate for upper bound of frequency
  - also yields estimation of area!
- very few constraints for place and route, can be done quickly (minutes)
- ideal scenario $\rightarrow$ optimistic approximation



ThreadPoolComposer
Estimating Upper Bounds: Area Utilization

TECHNISCHE UNIVERSITÄT DARMSTADT

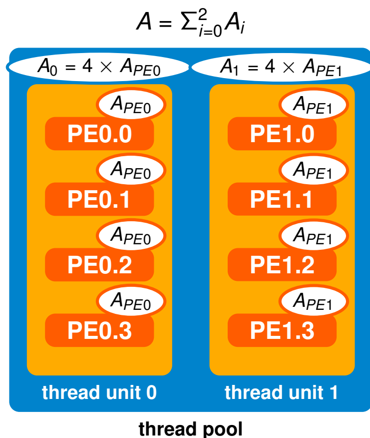
Area Utilization

- measures the utilization of limited FPGA hardware resources
- area utilization $\sim$ spatial parallelism $\sim$ performance
- more precisely: one utilization factor per resource type
  - LUTs
  - FlipFlops
  - MUXes
  - BRAM
  - DSP slices
  - ...
- good approximation for amount of logic: LUTs

ThreadPoolComposer
Estimating Upper Bounds: Area Utilization

Hierarchical Approach

- area estimation for each thread unit
  - area estimation for PEs $\times$ number of instances
  - plus Architecture overhead estimation
- area estimation for each thread pool
  - sum of area estimations for each thread unit
  - plus Architecture overhead estimation
- compare area estimation for pool to available resources
  - **feasibility:** $A \leq 1$
  - **optimality:** $A_{opt} = 1 - A$



ThreadPoolComposer
Optimization Problem

TECHNISCHE UNIVERSITÄT DARMSTADT

Optimize Performance

- optimization of multiple variables
  - maximization of area utilization
  - maximization of design frequency
- standard approach: **heuristic function**
  - define mathematical function of all variables that maps to single value ($\sim$ performance estimation)
  - optimize this function instead (one dimension)
  - example: use product of max. frequency and area optimality
    $h : (F_{max}, A_{opt}) \mapsto F_{max} \cdot A_{opt}$
- **but: variables here are not independent!**
  - critical path length increases with area utilization $\Rightarrow$ design frequency decreases
  - increasing design frequency limits length of critical path $\Rightarrow$ limits area utilization

ThreadPoolComposer
Optimization Problem

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Optimize Performance

- step back: recall that TPC implements the *task parallel model* of computation
- what is a measure of performance for task parallelism?
- **job throughput** (average jobs/s)
- optimize job throughput instead

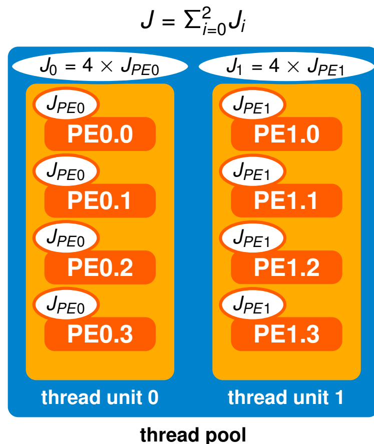
ThreadPoolComposer
Estimating Upper Bounds: Job Throughput

TECHNISCHE UNIVERSITÄT DARMSTADT

Hierarchical Approach

- job throughput estimation for each unit
  - job throughput PEs × number of instances
  - minus Architecture/Platform overhead est.
- job throughput estimation for thread pool
  - sum of job throughput estimations for each thread unit
  - minus Architecture/Platform overhead est.

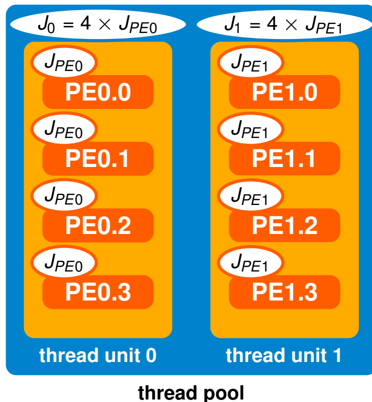
$$J = \sum_{i=0}^{2} J_i$$



ThreadPoolComposer
Estimating Upper Bounds: Job Throughput

Job Throughput for single PE

- job frequency = number of cycles per job * clock frequency
- number of cycles can be obtained by RTL simulation of PE
- max. frequency can be obtained by out-of-context approach
- what if number of cycles are input-dependent?
  - analyze typical workload for the application
  - use average number of cycles in typical workload for approximation



ThreadPoolComposer
Example: PE Analysis

TECHNISCHE UNIVERSITÄT DARMSTADT

Out-of-context results for a simple PE

Clock Period (max.)		5.125	ns
Clock Frequency (max.)		195	MHz
	used / available		
LUTs	443 / 53200	0.833	%
FlipFlops	753 / 106400	0.708	%
BRAM Tiles	0.5 / 140	0.357	%
DSPs	0 / 220	0.000	%
Feasible #	$\lfloor 53200 / 443 \rfloor =$	120	PEs
Clock Cycles/Job		556	cycles
Job Period	$556 * 5.125ns =$	2849.5	ns
Job Frequency		$\approx$ 351	kHz
Heuristic Value		42 112 651	jobs/s

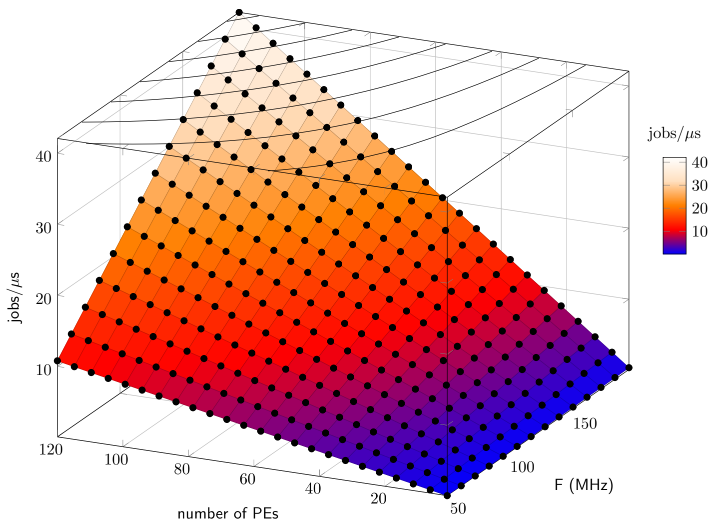
$\Rightarrow$ **120 PEs @ 195 MHz (best heuristic performance)**

Area utilization $\approx$ 99%, will not achieve timing closure

ThreadPoolComposer
Exploring the Design Space

TECHNISCHE
UNIVERSITÄT
DARMSTADT

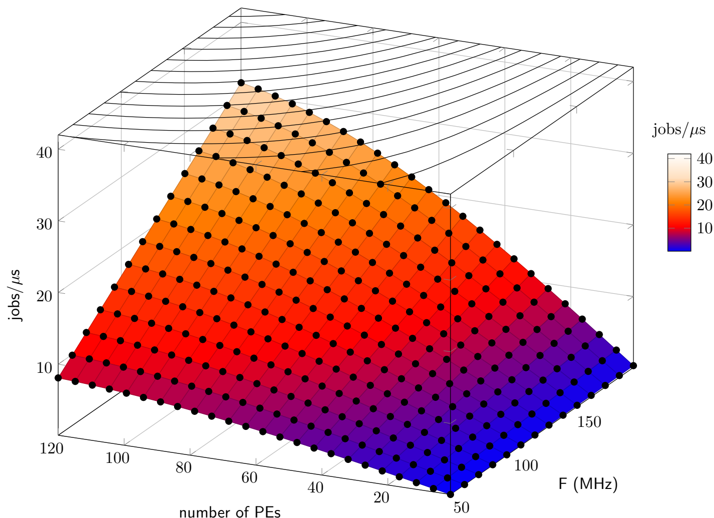
Assuming worst case performance penalty=0



ThreadPoolComposer
Exploring the Design Space

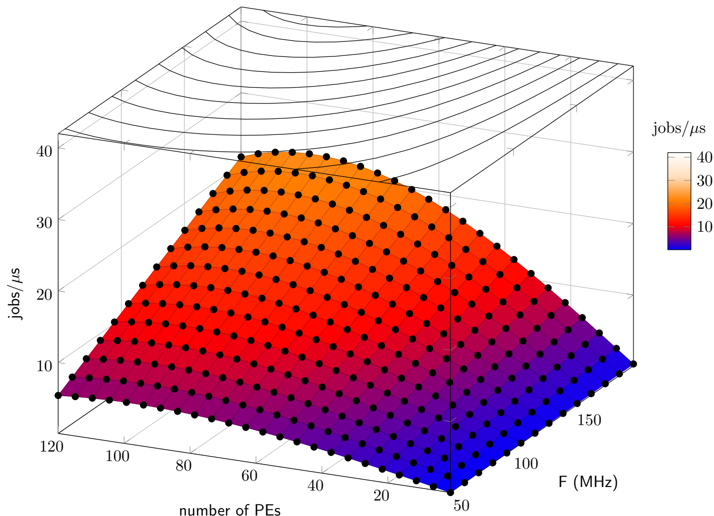
TECHNISCHE UNIVERSITÄT DARMSTADT

Assuming worst case performance penalty=25%



ThreadPoolComposer
Exploring the Design Space

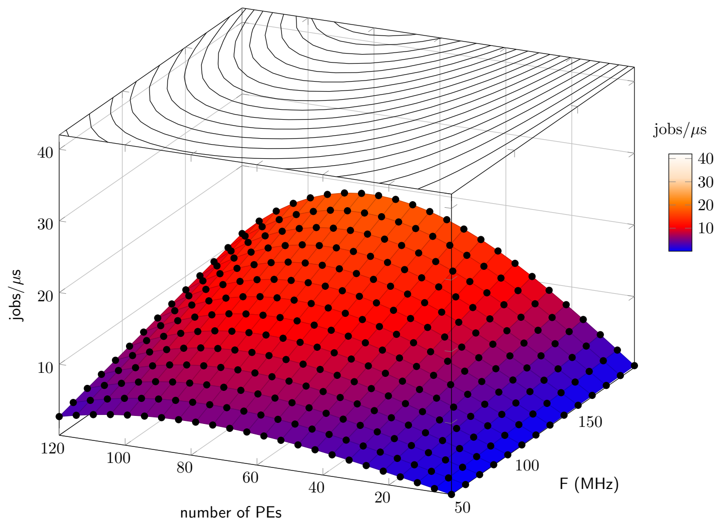
Assuming worst case performance penalty=50%



ThreadPoolComposer
Exploring the Design Space

TECHNISCHE UNIVERSITÄT DARMSTADT

Assuming worst case performance penalty=75%

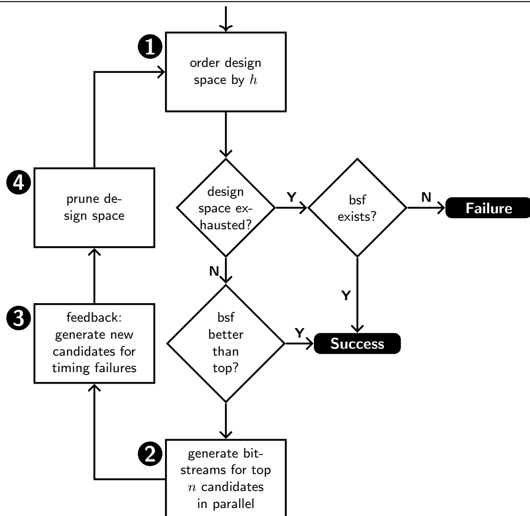


ThreadPoolComposer
Automatic Design Space Exploration

TECHNISCHE UNIVERSITÄT DARMSTADT

Automatic DSE

- fully automatic process in TPC
- heuristic h (configurable)
- batch size n (configurable)
- bsf = best-so-far
- top = best-in-batch



ThreadPoolComposer
More Design Space Exploration

TECHNISCHE UNIVERSITÄT DARMSTADT

Automatic DSE

- very useful, automates tedious process
- can be run on compute clusters (e.g., Lichtenberg Cluster)
- increases portablity *and* performance

More DSE: Variants

- variables so far: composition ($\sim$ area), target frequency
- TPC supports multiple **variants of PEs**
  - different implementations of the algorithm
  - different area requirements
  - different levels of parallelism (fine-grained)
- automated search across all variants of a composition
- extremely useful, often finds non-obvious solutions

ThreadPoolComposer

. . . is free software[1] !

- complete source on our public **GitLab**

  `https://git.esa.informatik.tu-darmstadt.de/REPARA/threadpoolcomposer`
- free software under LGPLv2 license

. . . is lacking extensive documentation

- but the code is well-documented inline
- some more information can be found in our papers:

  `https://www.esa.informatik.tu-darmstadt.de/twiki/bin/view/Staff/JensKorinthDe.html`
- mostly outdated, but some in-depth info can also be found here:
  REPARA Project, Work Package 5 Deliverables

  `http://repara-project.eu/?cat=7`

[1] requires Vivado Design Suite