ES2 1. Intro

Mobile Apps:

SoC: CPU, Media, RAM, Video Codec IP, I/O on 1 chip

Productivity Gap: 65 nm feature size >> 10nm for MPCPUs 300 million Transistors vs. >> 1 billion

Massive Data processing for Quad HD 3D Video streaming

ASIC can be used for that. Issues: Power, cooling, configurability, time-to-market

E-Textile: light-weight, power saving

Medical Diagnosis: ~~fault~~ error-free

Sensor Networks: Disaster Prevention, Traffic control, ~~energy~~ energy efficiency

Smart phones & Tablets

Characteristics for an embedded system:

- specialized for a task (domain of tasks): + probly more efficient because the specific requirements don't cause any overhead - configurability
- underlies tight constraints
- interacts with the physical surrounding via sensor/actor
- used in plenty of places → higher revenue than GPCPUs
- new application areas every day

Modern ES are more powerful and are designed for multiple applications

↳ power efficient, Energy ~, performance ~, area ~, cost ~

Dependable

↳ Reliability: propability of correct system functioning given it was functional at time $t=0$

↳ Availability: propability of system in function at a given time t

↳ Maintainability: effort to repair/operate the system

probability of correct system functioning after encountering an error

↳ Security: system retains confidently (no hackers)

↳ Safety: system doesn't harm people / prevents it

Constraints:

soft-real-time: ES becomes Quality of Service (QOS)

example: Video decoder FPS drop

hard-real-time: result becomes worthless or even causes harm

Behavior: transforming: stream apps → pace is imposed by the environment

reactive: continously reacts to events in physical world

interactive: for interaction with user

Composition: hybrid system: analog & digital components

Moore's Law: complexity (area) doubles (each) 18 months Memory challenge

Gaps: Logic Frequency vs Memory Frequency

: Moore's Law Logic vs Design of this complexity

HW - Design Gap

SW required for HW vs Hardware design productivity

HW including SW design gap

Shannon's law: growth of algorithmic complexity

Design Metrics: most popular: Performance

Power consumption (mobile devices, process variations)

Peak temperature (thermal issues, cooling, aging)

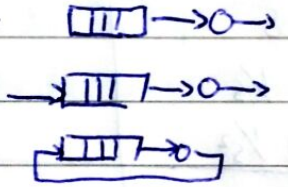
Reliability (under changing environmental aspects)

Size (package, code, chip)

Cost per Unit

NRE (non-recurring engineer cost)
Flexibility (effort for modify/update unit?)
Testability (effort for testing unit for functionality?)
Maintainability (effort for repairing unit/operating unit?)

Chip area tradeoff with time-to-market: getting chip produced quicker with sacrificing chip area, since it is nowadays possible to get really many components on 1 chip
Chip complexity measured in gate-equivalents: # gates (N4ND) nessecary to perform the same logical function
Issues: packaging cost, leakage power, power density, cooling cost
Kinds of systems:



Performance for fixed systems: time to finish workload
" open ": Latency/response time
" closed ": Throughput @ Pipeline
increasable by concurrency
" all: Speedup
Cycle-time: fastest clock
Ausbreitung → Propagation Delay: combinatoral logic input-output propagation delay

Power Consumption: ES are mostly battery driven
consumers: CPU, RAM, Buses & Communication ... (Hw) designtime
RTOS, application (SW) operating time

Heat & Temperature

heat density of CPU ~ nuclear reactor

Limitation / cooling expensive / better airflow / Layout enhancements

Hot spot migration

Failure of Dennard Scaling

- Transistor and power are no longer balanced

  scaling is limited by power

- Higher power density leads to thermal problems

  accelerate aging effect

Dennard Scaling (Failure)

Classic Dennard Scaling	Actual scaling
Device count S^2	S^2
Device Frequency S	S
Device power (cap) $1/S$	$1/S$
Device power (V_{DD}) $1/S^2$	~1
Power density: 1	$\sim S^2$

Due to leakage currents at shrinking gate sizes

More leakage → higher currents → higher temp → more leakage ~

↑ infinite loop !!

Randomized Doping Fluctuation and processing variants determine Threshold Voltage → some Gates may cause more leakage / be faster / lower than others

→ random aging effects / electron mitigation / dielectric breakdown

Soft errors may also occur by radiation / particle strikes

this scales up by shrinking technology / feature size / lower voltages / higher clocks

Soft Errors: Multiple bit upsets (Memory)
Single bit upset (Memory / Latch)
Single Event transient (logic gate @ function)
Single Event Functional interrupt

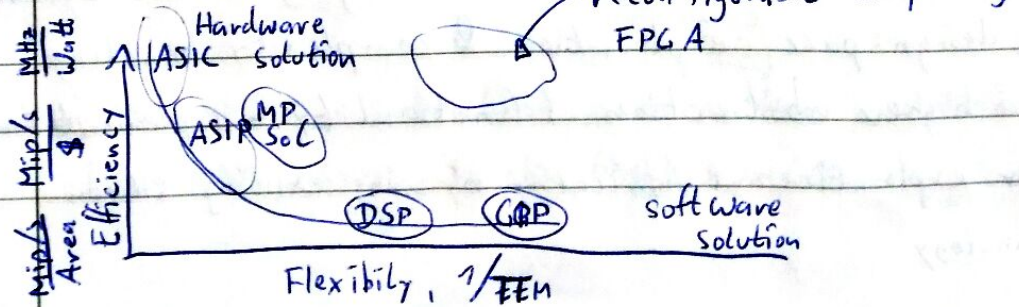
Product Lifecycle Generations overlap

Time To Market: Idea → Product
short TTM is key for revenue, being first @ Market with a product maximizes revenue
Too early also possible, market may not be ready 4 product

Cost
Cost_unit = (NRE + #units * cost_per_unit) / #units
NRE = one time cost for a new product (technology, designtime, tools)
Chips with new technology (22nm) are quite expensive
#units needs to be large (> 1mio) to amortize the costs
=> decreasing #ASIC starts
=> prices high (determined also by competitors)
Higher NRE may result in lower unit cost
$total depends on #units

All these factors are interdependent => Tradeoff needs to be found since, for example, low temperature & performance are contradictory. => exploration of design space

Architectures for ES



GPP (General Purpose Processor)

Fully programmable, general purpose ALU, FPU, ...
For any Application, general " instruction set
Non-extensible, high SW flexibility, cheap/unit

ASIP (Application Specific Instruction-set Processor)

Fully programmable, general purpose OR specialized ALU, FPU, ...
embedded to an application environment (mediaplayer)
Instruction-set may be extensible, good SW flexibility
expensive

Reconfigurable Processors: 2D Array of configurable Register Files, Functional Units & ALUs, limited connections,
- adapted to application domain, requires complex tools
expensive
~ FPGA as pipeline

ASIC, Accelerator

hardwired "processor", no software, no instruction set,
if done right, low power consumption with high performance, very expensive

Multicore Processors

Levels of parallelization: Bit-Level (2 bit, ... 64 bit) cores
Instruction-Level
Data-Level (SIMD...)
Task/Threads

Electronic System Level Design (ESL)

Co-synthesis & co-simulation for designing a whole system
for designspace exploration & complexity handling
at a higher abstraction (functionality > implementation)
faster exploration and fabrication by using existing silicon
technology

HW & SW are no longer designed separately, their design tools have evolved and can now be used concurrently
⇒ System synthesis, IP are SW, HW, OS

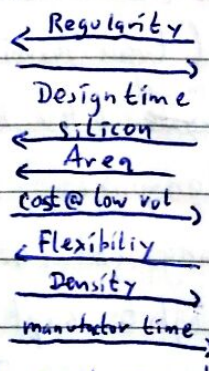
Embedded Software ≙ Software + interaction with real world
more than just a turing machine.
Needs to be able to read sensors (reactivity, interaction), underlies timing constraints (RTOS), ~~while~~ ~~out~~ controlling motors at the same time (concurrency) and has to handle the next action after finishing the current one (Liveness). Handles digital, as well as analog data (heterogenity)

ASICs Custom vs. Semi Custom vs structured

completely handcrafted	lower optimization	between FPGA
high optimazation	efficien tools	and standard cells
risk → tools	error freer	

- Standard cells taken from Library, ~~par~~ parametrizable
Macro-cells ⇒ building blocks, used when functional description is given (PLA, PAL..)
Interfaces mostly compatible with each other
Array based, configurable (by fuse, sea of gates...)
ASIC: FPGA, Gate Array, Standard Cells, Compiled, FCIC

Tradeoffs



ASICs may be pre-structured and can have predefined elements
Liken RAM, Array of structured elements

2. Spec & Mod

What are models for:
- performance modeling
- functional modeling and specification
- Design and synthesis
- validation & verification
- test vector generation
- test coverage analysis
- architecture evaluation and mapping
- technology mapping
- placement & routing

Why models:
- Describe ES behavior (radically growing complexity)
- natural languages often inprecise / ambigous
- easier to understand than 10^9 LOC

Using computation model
- → sequential program model (statements with semantic)
- → communication process model: (multiple seq. concurrently)
- → State machine model (for control dominated system)
- → Dataflow model (transforming, streaming systems)
- → Object-orientat~~e~~ model (for breaking complex software down)

Models capture all languages (programming ~)
One code in a programming language can be represented by different models, which might capture a specific aspect better

FSM: 6-tupel $(Q, q_0, \delta, \Omega, \Sigma, F)$
$Q \hat{=} \{q_0 \dots q_n\}$. $q_0 \hat{=}$ initial state $\delta \hat{=}$ transitions, $\Omega \hat{=}$ output,
$\Sigma \hat{=}$ input, $F \hat{=}$ accepting states

Moore type : outputs in state
Mealy type : output in transition
think of all possible states
how to: 1. List all states
2. List all variables
3. List all transitions
4. ∀ Q List actions
5. ensure exclusive & complete transitions ∀Q
There are tools for modelling FSMs (cost, Licence...)
mostly done by hand with switch-case-statement
Hierarchical / Concurrent FSM (HCFSM)
Program state Machine (PSM) can have programs in nodes (transition when @ end of code OR Signal triggered)
Concurrent Process Modell for describing programs that are inherently concurrent
Dataflow model: like petri-net & concurrency good for DSPs
MODEL ≙ a simplification of an entity which can be a physical thing or another model. A model contains exactly those characteristics and properties of the modeled entity which are relevant for a given task

2.

Properties of a model : inherent (finite state space)
static (memory for a FSM model)
dynamic (memory for a C programm)
Heterogeneous modelling : different models for the same entity
static system: memory-less, independant from inputs from the past
dynamic system: dependant from at least another input $u(t0)$ with
$t0 < t \Rightarrow y(t) = f(u(t), u(t0))$
time-invariant-system: offset in time doesn't change outputs

time-varying $=!$ time-invariant-system

state $\hat{=}$ that piece of information that is required to determine the output of a system at a point in time

state equations $\hat{=}$ transitions (δ)

state space $\hat{=}$ Q set of states reachable

can be continous ($\mathbb{R}$) or discrete (red, green, ...)

linear system $\hat{=}$ $f(a_1 x_1 + a_2 x_2) = f(a_1 x_1) + f(a_2 x_2)$

$\wedge\ f(a_1 x_1 + a_2 x_2) = a_1 f(x_1) + a_2 f(x_2)$

deterministic system $\hat{=}$ given f and x_i, output always the same

~~stochastic~~ $\sim\ \hat{=}$ at least one x_i is random

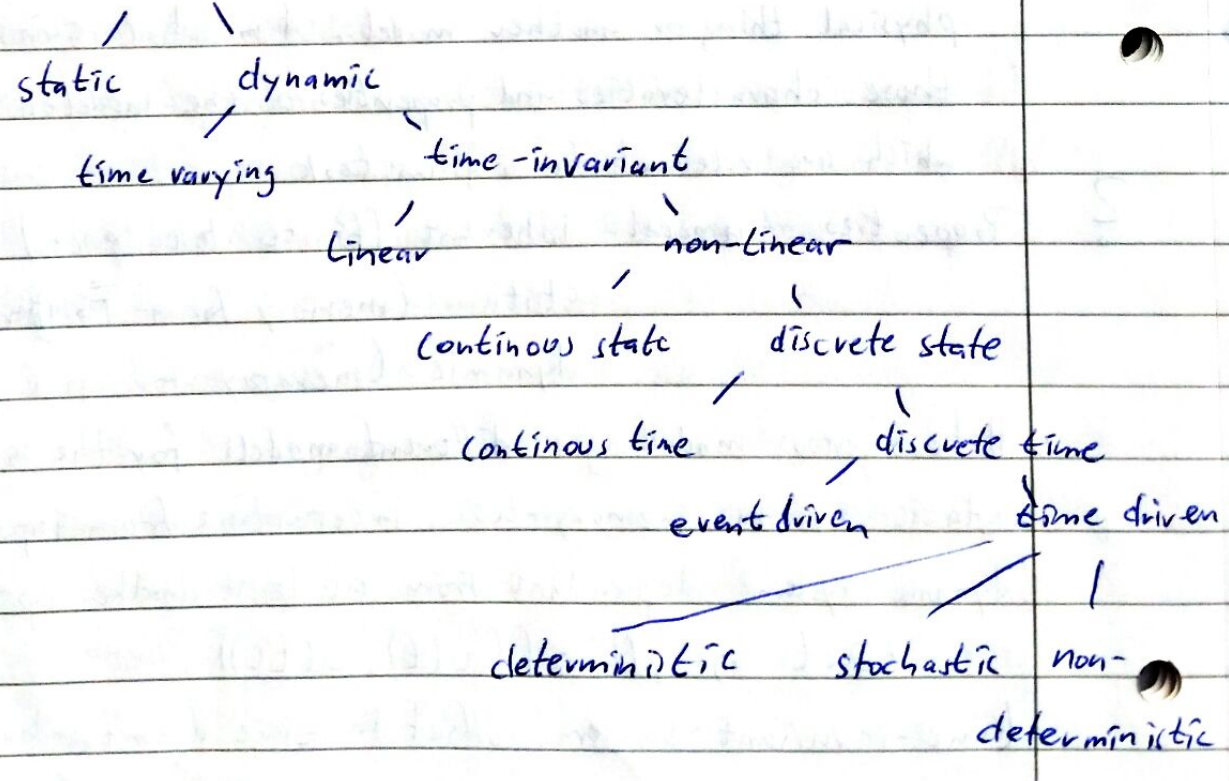
non-deterministic $\sim\ \hat{=}$ given f and x_i produce random output with unknown probability

Events come in a moment t and durate 0 sec.

Event-driven systems become active by event

Time- " " " " " time.

Systems classification:



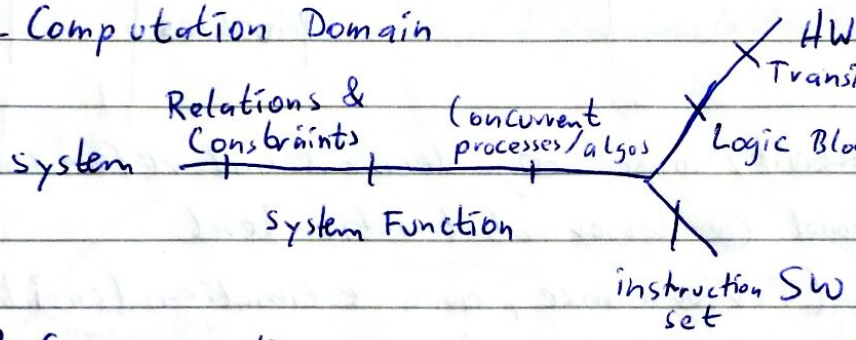
Hierarchy: partitioning of a model in such way, that ↑ hide
Information is hidden in lower hierarchical entities ↓ display

Abstraction: give the "what the system does" not "how it does it", remove irrelevant information for understanding the functionality ↑ remove ↓ add

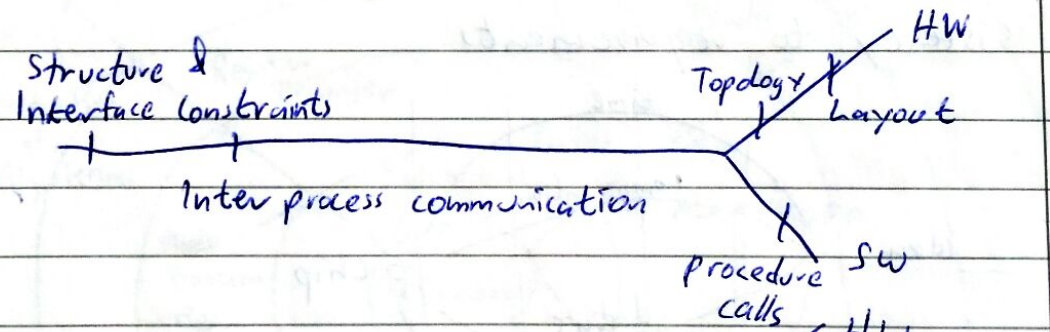
Domain: aspect of a model which can logically be analyzed independently from other aspects

Rugby divides idea → physical system progress in 4 parts

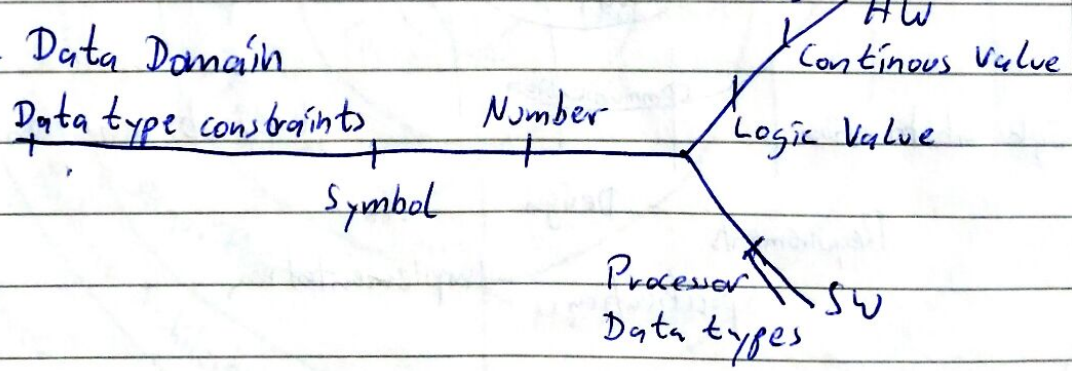
1. Computation Domain



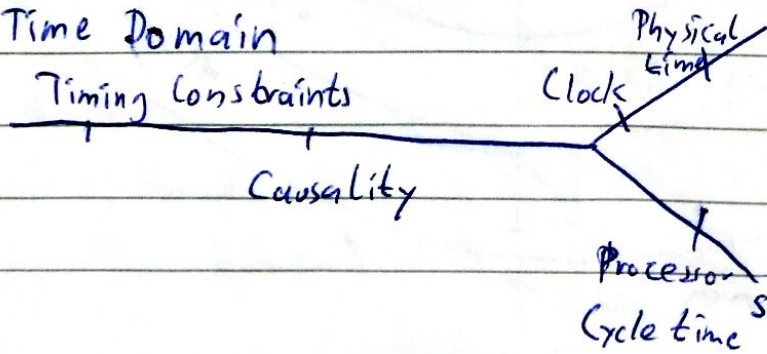
2. Communication Domain



3. Data Domain



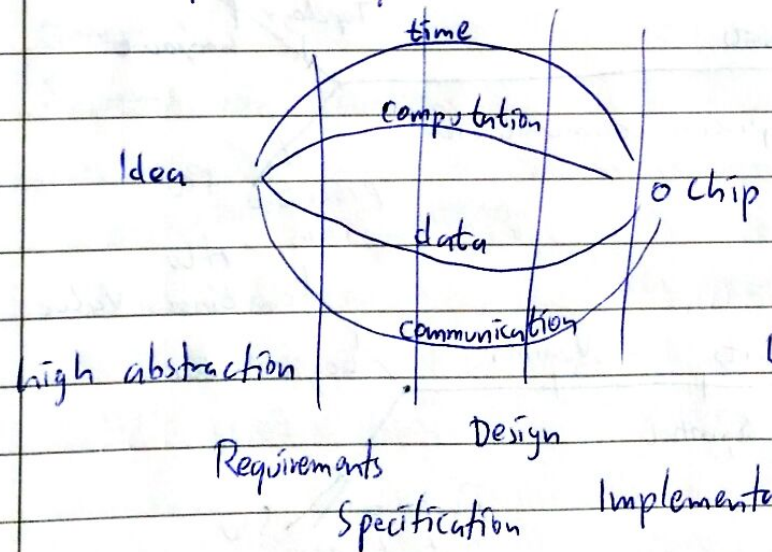
4. Time Domain



	Communication	Data	Time	Computation
Requirements Definition	Structural and Interface Const.	Data type Constraints	timing const	Relation Constraints
Specification	Inter-process communication	Symbol	Causality	System Functions
Design	~~Packet time~~	Numbers	Clocked	Algorithms
Implementation	Layout Topology	Logic / physical values	Physical time	Logic Blocks Transistors

Design activity make design decisions and refine the design into a model @ lower abstraction level

Analysis (Performance, area estimation, feasibility check, Logic simulation & timing check) ensure consistency to requirements



3. Specification

EDA ≙ Electronic Design Automation

for simulation, estimation, synthesis and optimization

increases design productivity & speed

manges high complexity

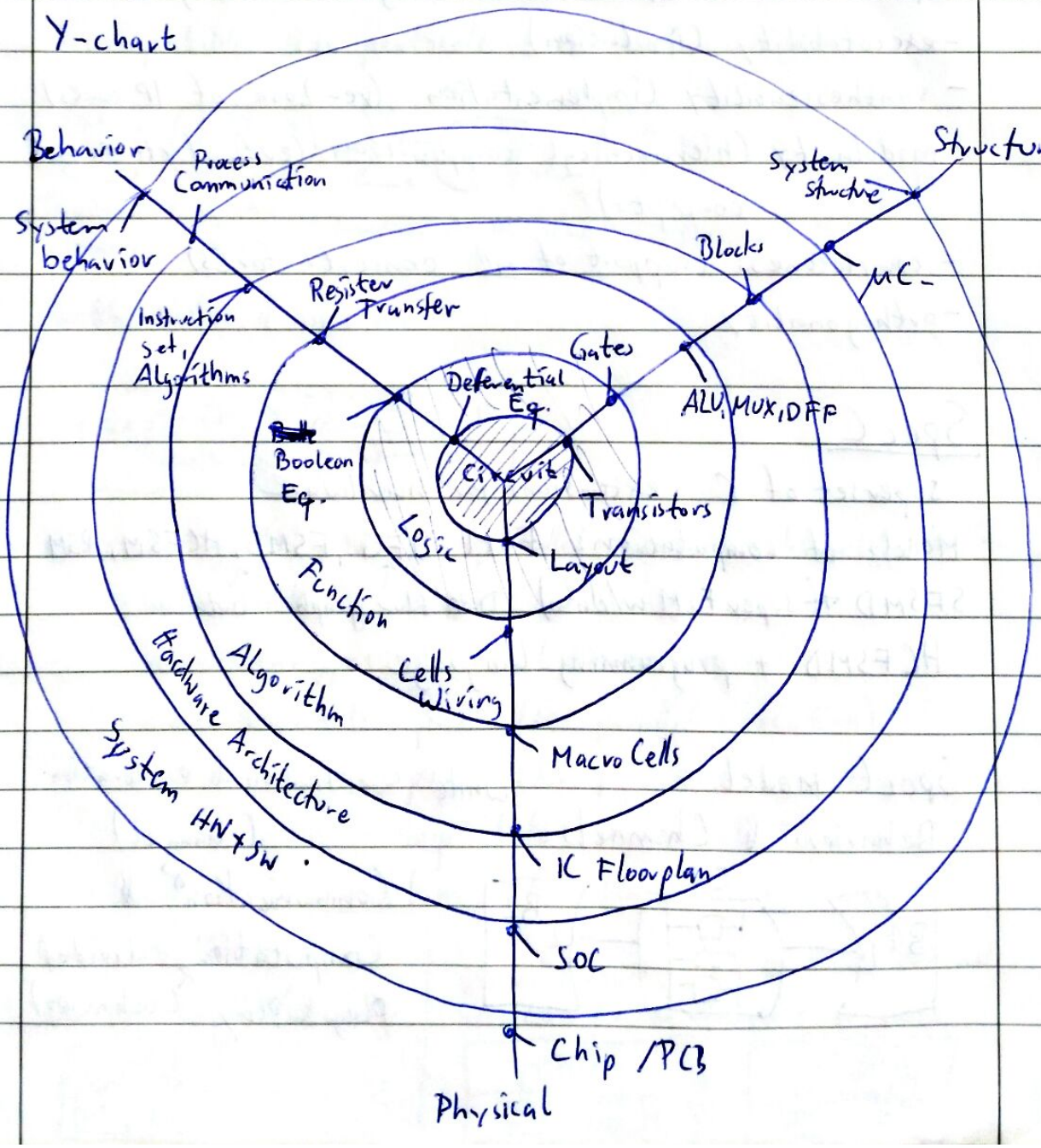
fore casts behavior

enables re-use of designs

improves design quality by avoiding errors

increases maintenance and documentation

Y-chart



EDA latest generation: @System-level designing w/o HW-SW difference

Design for re-use with new languages (SpecC)

New system architectures (MPSoC / Many Processors / NoC) with $> 10^9$ Transistors

Problems: complexity, power, test, signal integrity

Simulation @ high abstraction level with less details is fast but inaccurate (tradeoff)

Requirements for System-level design & modeling

- executability (simulation)
- synthesizability (implementation, (re-)use of IP cores)
- modularity (hierarchical composition (even of different concepts)
- completeness (support of all concepts for ES)
- orthogonality

SpecC

Superset of C, extensions for hardware

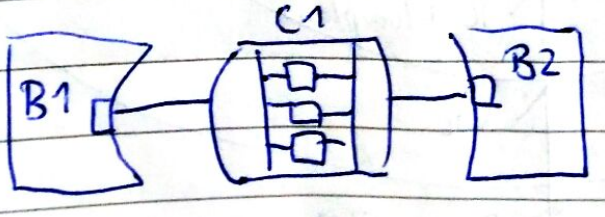
Models of computation: FSM, DFG, FSMD, HCFSM, PSM

SFSMD (= super FSM w/data) DFG: Data flow graph FSMD: Data

HCFSMD + programming language

SpecC Model while Specification & exploration

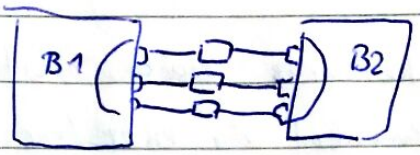
Behaviors & Channels



Communication (channel) & Computation (behavior) divided

Plug & Play

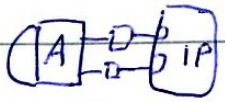
when implementing channel disappears



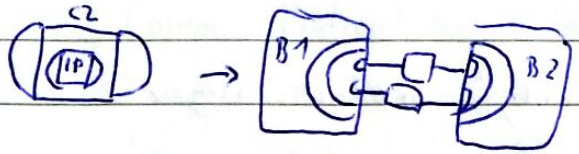
Communication inlined in behaviors (protocol inlining)

IPs come in wrappers with channel behavior same as usual

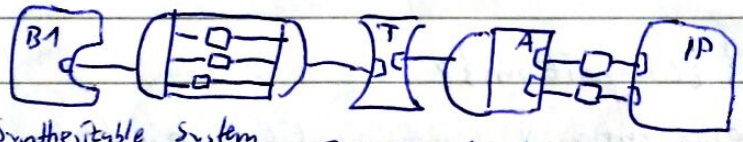
If no channel is available, use adapter



Even if the IP is for communication:

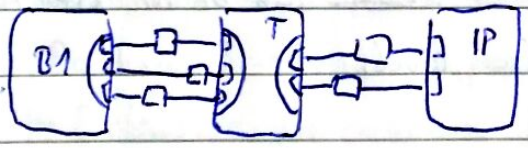


Incompatible bus protocols need a transducer



Synthesizable Behavior | System bus | Transducer | Adapter | IP bus | Synthesizable IP

With protocoll inlining:



Code: „Objects" like things called behavior/channel/interface

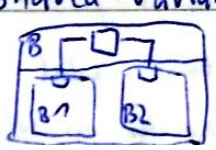
main-method as usual

more datatypes: bool, bit (vector), event

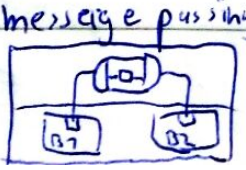
more keywords: par (for parallel execution)
pipe (for pipelined execution)
seq (for sequential ")
fsm (for finite state machines)

Communication:

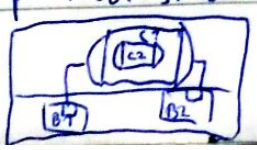
shared variable/memory



virtual channel message passing



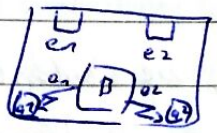
hierchical channel protocol stack

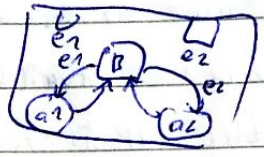


event ~~types~~ keywords : wait, notify, notifyone

usually implemented via interface : 

Exception handling : abortion / interrupt





Timing "wait for"

"do" & "timing" syntactic sugar

Transaction-Level-Modeling (TLM)

TLM = <{objects}, {compositions}>

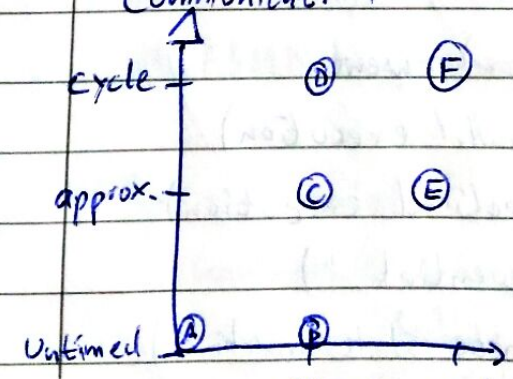
↑ computation & communication objects

↑ hey they are composed

Advantages: object independence, modularity

abstraction " , ~~modules~~ objects can be modeled @ different abstraction levels

Abstraction Models:



A ≙ specification model

B ≙ component-assembly model (Architecture model)

C ≙ Bus-arbitration model (Transaction model)

D ≙ Bus-functional model (Communication model)

E ≙ Cycle-accurate comp. model

F ≙ Implementation model (Resister-Transfer-model)

A objects: behavior (Comp)
Comm: Variables

B Comp: Proc / IP / Memories
Comm: variable channels

C Comp: Proc / arbiters / Memories
Comm: Abstract bus channels

D Comp: "
Comm: Protocol bus channel

E Comp: Proc / IP / Arbiters / Memories / Wrappers
Comm: Abstract bus channels

F Comp: Proc / IP / Memories
Comm: Wires

Foliensatz 5 ausgelassen!

6. Code Generation

Retargetable code generation for embedded processors
Processor families have different architectures
ASIPs have 3 classes of parameters:
- Extensible instructions (completely customized)
- Parameterizations (cache size / policy ...)
- In/exclusion of predefined blocks (SFR, test ...)

Writing the same application for every ASIP is expensive
→ use retargetable code

ES often have:
DSP: for arithmetic intensive calculation (FFT)
Hardware Multiplier
Adress generation unit (AGU)
Special purpose registers

Instruction level parallelism (ILP)
Muliply-And-Accumulate (MAC)
→ They make code generation quite complex
Instruction word: VLIW
controls level of parallelism by instructing FUs
- large code size (cc)
- variable length of words
ASIP vs DSP
↑ designed for 1 application, but can be configured better
↑ class of applications
DSPs often programmed in Assembly ⇒ efficient, but costly to develop
ES applications can have 1 mio LOC, impossible to write down in Assembly ⇒ code-generation may end up with inefficient code
※1 goal in code generation: performance
compilers often trade-off performance against compile-time
ES need to be efficient (battery life)
Code size needs to be reduced, memory is expensive
Compilation speed doesn't matter as much as with a GPP.
only 1 application runs on the ES, therefore this one shall be highly optimized.
⇒ unusual steps in code generation (simulated annealing, genetic algorithms)

Steps in code generation for ES
1. Code selection

Analize code and choose proper instructions (MAC...)
SIMD

2. Instruction scheduling
When will which instruction be executed
considers data dependencies (and control dep.), pipelining effects and degree of ILP
Goal => minimize # of cycles, sometimes minimize power usage, or maximize reliability
Simulated annealing used here (together with list-scheduler)

3. Register allocation
Data given in intermediate format (DFT)
Basic block ≙ Data path between jump-instructions
Simulated annealing also used here

4. Adress code generation
AGU supports: immediate load/modify, auto inc/dec
create access graph V: variables, E: access after another W: times accessed after.
calculate minimal spanning tree => order of variables in memory. Use auto inc/dec for accesses

Before all of that (4 steps):
IR optimization (Intermediate Representation)
Architecture independent

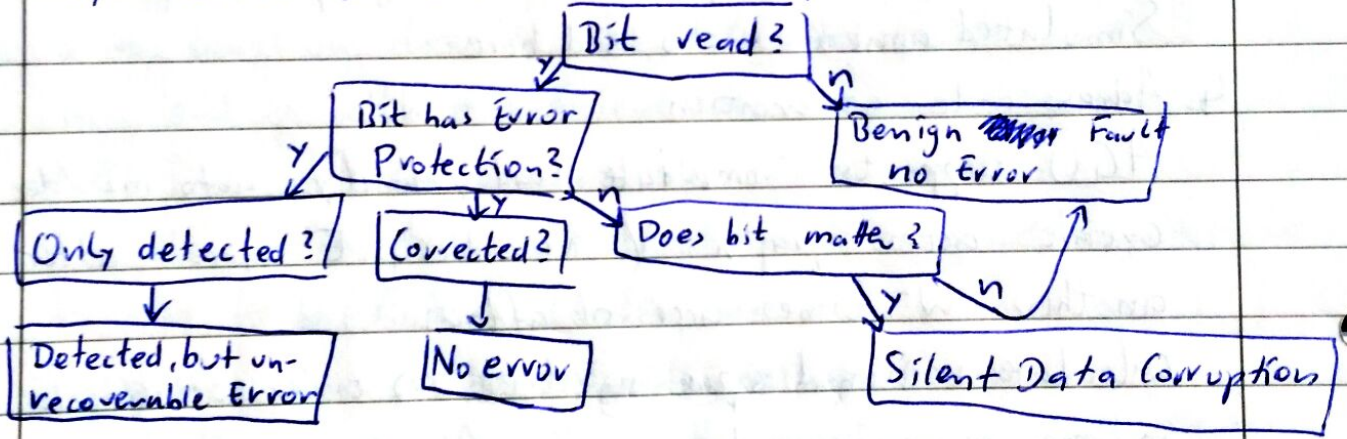
7. Reliability

Moore's Law was a win-win situation...
...but now we have new problems:

Complexity (100 billion transistors/chip)
⇒ productivity gap
Thermal issues
⇒ aging effects
Manufactoring defects
⇒ process variations (RDF)
Physical limits reached
⇒ stochastic effects
⟹ decreasing yield

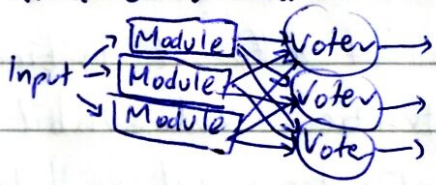
} Reliability

Effects for system failure: Manufactoring faults, Temperature, Aging
Particle strike (^4_2He) become more staving since we lower voltages and logical values are represented by shorter (⇒ weaker) potentials



Temperature may lead to errors due to electromigration and process variations. Holes and hillocks caused by thermomigration ⇒ decrease peak temperature is a goal
Negative Bias Temperature Instability (NBTI) ≙ Breakdown of Si-H bonds @ siliconoxide interface
results in a V_{th} shift ⇒ transistor delay
Smaller feature sized components are more affected by NBTI

higher temperatures affect key metrics of any chip (cost, power, performance, ...)
Packaging & cooling - cost increases, life-time and reliability decreases
What can be done? Hardware - improvements:
Fin FET - Transistor / Spin - Transistor / Single - Electron Transistor / Graphene - Transistor!...
Architectural changes: DMR / TMR
N-module - redundancy
massive power and area overhead
Triplicated TMR



Error correction codes:
SECDED: single error correction, double error detection
needs $\log_2(n)$ bits, n = data size
much overhead, high power consumption
Parity protection...
most of all memories are protected
Pipeline protection: Razor Flip-Flop
Simultaneous Multithreading (SMT) + Fault detection = SRT
Sphere of replication: 2 Threads calculate the same & check for equal results. If not, signalize error.
Software: duplicate commands and branch-on-not-equal
massive memory and calculation overhead (EDDI)
Control flow checking using Software Signatures (CFCSS)
Enhanced version ↑ (ECFC) even transfers signatures
Software implemented Fault tolerance (SWIFT)

is basically EDDI + ECFS but only checks signatures @ store commands and removes branch checking
SWIFTR: SWIFT + Recovery: has majority instruction
Like TMR in software
Both HW & SW reliability solutions cost power and time.
Compiler-directed reliability methods with accurate reliability estimation is the key for reliable software

Abstraction of all Errors to bit-flip-level.
can be of different criticality: bad op-code / bad pixel
IVI ≙ Instruction vulnerability index
AVI ≙ Application vulnerability index
FVI ≙ Function " "
Errors can be masked
Loop-unrolling

8. Multicore Processors

old technique of gaining performance: increase frequency
BUT: doubled frequency → fourfold of ~~energy~~ power consumption: power = capacitance · frequency · (voltage)²
higher frequency requires higher voltage
" " → more leakage → generates heat →
better cooling required → decreases reliability → costs money
Single core vs. Dual core
↳ increase frequency by 50% → 2x power consumption
↳ double calculation power, only 30% more energy drawn

Single Core	vs	Dual core
Voltage = 1		Voltage = 0,85
Frequency = 1		Frequency = 0,85
Power = 1		Power = 1
Performance = 1		Performance = ~1,8

Multicore breaks the 3 Walls:

Power-Wall, see above.

ILP-Wall difficult to find enough parallelism in a single instruction stream to keep a single core busy. Doubling the frequency doesn't double the performance due to timing issues

Memory-Wall fast single-cores need to wait long for memory access, since the bus frequency is much slower

Homogeneous: all cores are equally built and have the same performance

Heterogeneous: different cores, perform different tasks @ different speeds

Memory Designs for multicore CPUs

Distributed: each core has its own exclusive memory

Shared: global memory for all cores

Hybrid: both

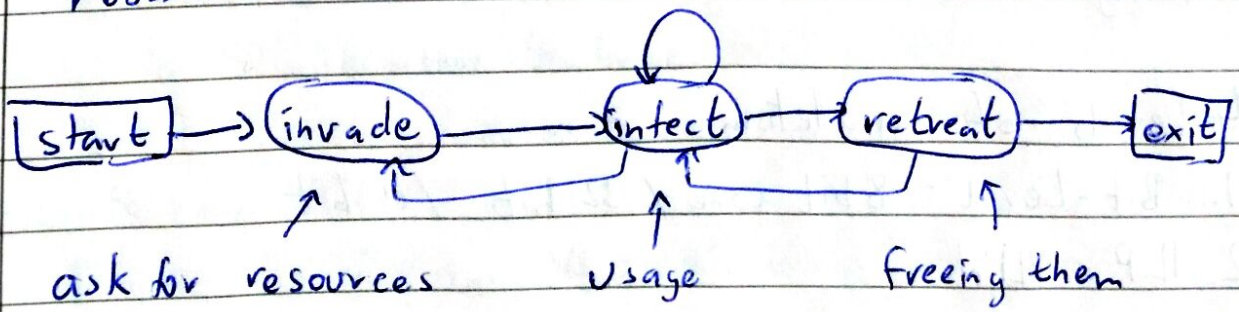
4 Levels of parallelism

1. Bit-Level: 8 bit core / 32 bit / 64 bit ...
2. ILP: VLIW
3. Data Parallelism: SIMD
4. Task / Thread parallelism

Problems with multicore:
- all programs need to be rewritten
- asymmetric multiprocessing or symmetric multiprocessing? synchronization

Challenges:
- How to dynamically map applications to 1000+ processors while considering memory, communication & computation resourse constraints? (Complexity)
- How and to what degree shall algorithms adapt?
- How to generate programs, that use n coves? (Scalability)
- Management of overhead? (Constraints)
- How to hanelle process variation, transient and permanent defects? (Reliability)
- How to exploit and adres heterogenous coves?

Invasive computing:
programming for heterogeneous processors.
Deals with Languages, system software, architecture (HW/SW)
runtime - scalability
efficient resource utilization
Fault-tolerance
robustness



Middleware for observing resource utilization and managing them (so called agent)
realizes concept of self-adaption
Sea of logic units, so called tiles
connected by NoC
i-Cores for runtime adaptive acceleration
TCPA ≙ reconfigurable processor in invasive network
Memory and I/O-Tiles

9. ASIP

Designing an embedded processor with a tool-suite
They help with architectural exploration, integration & verification, implementing the architecture and designing software & development tools

1. Instructionset: can be completely customized, from scratch or extended for a given core (& well tested)
2. Processor components: add specialized cores which can be instructed by the SI
3. System components: can be added / omitted / parametrized (cache size / policy, MMU, ...)
4. On-chip communication infrastructure: busses, topology, hierarchy (→ typically fixed)

3rd design Technology: reconfigurable / adaptive

Tensilica

provides IP-coves, well tested processor coves that are subject to receive an application specific instruction set.

Two levels of customization: 1. choose cove 2. add SI

tool chain: XTensa

XTensa Xplorer for HW:

Algorithims → Simulation → Analyze → configure/extend

SW:

Source Code → Compiler → Simulation/Debug → Profile

XPRES Flow:

C/C++ → XPRES Compiler → TIE instructions automatically generated + user defined TIE instructions + processor configuration input → Xensa Processor generator → Hardware RTL + System Models + Complete Software Tools

TIE ≙ Tensilica Instruction Extension

extend the processors instruction set and architecture

written in Verilog

are part of the processor's programming model

used for TIE compiler & processor generator

combines concepts of VLIW, FLIX, Fusion, SIMD

TIE Compiler generates HDL from TIEs

makes SI accessible in C code

Fusion ≙ combine dependent operations in one instruction

i.e. AVERAGE

Also possible with SIMD Vectors

FLIX ≙ VLIW ~ in fixed word lenghts

External co-verification tool links SW & HW simulation models
Instruction set simulator with assembly code is simulated on the processor interface, written in HDL

LisaTek
Combining architectural exploration & implementation
The whole Instruction Set Architecture (ISA) is customized
Flow: Description of ASIP → LisaTek Processor Designer
→ Application + C Compiler + Assembler + Linker + Simulator
→ Analyze + Refining / Adjustments → System Model
+ Tools + RTL description
Compiler generation: Processor model → CoSy system → C Compiler
With that compiler, the ASIP assembly is generated

Basic idea: closing gap between structural oriented Languages (VHDL, Verilog) and instruction set languages @ different abstraction Levels
retargeting various tools: compiler, assembler, simulator tools that work for various architectural scenarios
retargeting requires different types of architectural information

Memory model: registers, memories, with width ranges
Resource model: available hardware & resource requirement
Instruction set model: instruction word coding, spec. of valid operands & addressing modes, written in assembly, collects all instructions as combinations of HW operations that are permitted by the CPU controller comprises instruction semantics

Behavioral model: activities of HW are abstracted to operations, notion of state for simulation, change the state of system, abstraction level can vary widely between HW implementation & high-level-Language

Timing-model: specifying the (activation) sequence of hardware operations and units

Micro-architetural-model: grouping of HW operations to FUs, describes the details of μ-architectural implementation of RTL-components

Processor Architecture

- Mixed behavioral & structural model:
  based on C/C++, VLIW datatypes, strong memory modeling capabilities (ind $), include external IP (libs.)
- Enriched by timing information
  clocked register behavior
  operation scheduling
  extensive pipeline model with predefined functions (stall, flush, ...)

Instruction Set Description:

Instruction word coding: variable length, multiple words
Assembly syntax: mnemonic based or C-like
Instruction semantics / compiler semantics
Configurable instruction set information (power, ...)

Generating HDL from LISA

Models: Resource model, memory model, ISA, behavior model, timing

HDL: Structure, FUs, Memory Configuration, Decoder, Pipeline Controller

10. Special Instructions

Goals: High performance (speedup)
Low energy consumption
reduce used HW area and code size

ISA ≙ abstraction level between HW and application
each processor provides a core ISA
SI extend core ISA, they look like a common assembly instruction
... but: how to create and choose them?

2 ways to build extended ISA: from scratch: everything custom
or extend core ISA by adding a few

Manual effort for higher performance but complex to design
→ automated tools (XPRES Compiler)

ISA customization with templates for application domain
well known facts about " : Number of in/outputs,
area usage, power consumption, timing, memory accesses,

Generation of SI is complex in exploration:
Heuristics vs. optimal, additional constraints may
tighten the exploration space

~~Selecting~~ Selection: select a set of SI that fulfill the constraints
and maximize the performance
a lot of Heuristics: longest path as short as possible,
minimal number of distinct templates, maximum number
of instances of one template, similarity of clusters for
reuse, minimize memory accesses, minimize power/area,
basing on the regularity of the frequency of execution, ...

When are SIs created? ASIP ⇒ design-time
reconfigurable-processors ⇒ usually @ compile-time

but why not @ runtime ? => RAS
Online synthesis: convert binary to high-level-format
find computational hot-spot (usually 10% of code cause 90% calculation time)
HLS of that part
generate new binary and update configuration
huge runtime overhead. simplified FPGA structures
as a tradeoff: quality vs. overhead

SI integration
Loosely coupled: via I/O pins: comm over databus, high comm-overhead
processingunit: faster on-chip communication
may access external $, if using $-coherency protocoll
requires new IC + still no internal CPU into accessible
coprocessor: using dedicated co-processor interface and dedicated control signals
tightly coupled: reconfigurable fabric on CPU: very low comm overhead & high data bandwidth
-new IC necessary
soft-cove CPU on FPGA: only drawback:
Lower CPU frequency