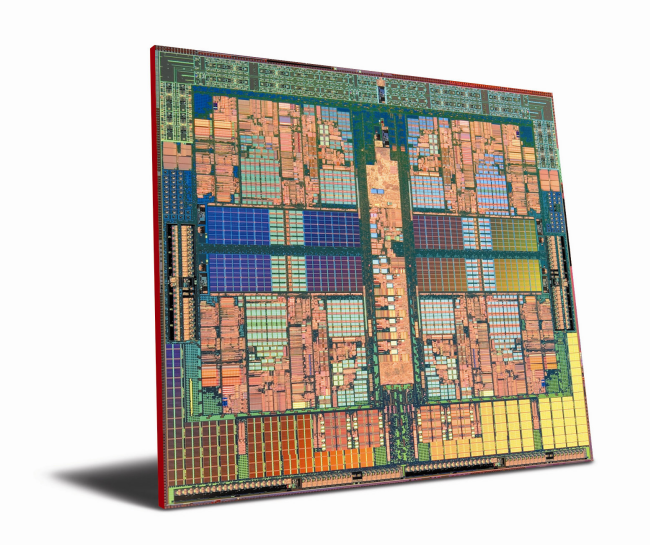
Multi-Processor Programming in the Embedded System Curriculum



Andreas Hansson[1]
Benny Åkesson[1]
Jef van Meerbergen[1,2]

[1] Eindhoven University of Technology
[2] Philips Research

Outline

- Introduction
- Assignment
- Structure
- Discussion

What are we preparing the students for?

Embedded systems

- …have non-functional requirements
  - hard or soft real-time constraints,
  - a limited power budget, and
  - limited resources, e.g. memory footprint
- …are constructed from highly programmable components
  - changes in applications and standards
  - algorithms (partially) implemented as embedded software
- …make use of a platform-based approach
  - many Intellectual Property (IP) blocks, with
  - processor cores, accelerators and communication infrastructure from different vendors
- …are using multiple processor cores
  - multi-processor with distributed memories for scalability and low power
  - requires parallelisation of algorithms with communication and synchronisation
- …are going from buses to networks on chip
  - programmable interconnect where the designer decides on the resource allocation
  - distributed, multi-hop communication with longer latencies

How is it done at Eindhoven University?

- Master program on embedded systems
  - joint program of EE and CS
- Set of four courses, bottom-up
  - lower levels of design, i.e. logic and RTL synthesis, developing ALUs, multipliers, memories etc, with focus on FPGAs
  - processor design, using the aforementioned blocks to build fully programmable microprocessors, DSPs, ASIPs etc
  - networks on chip, focusing on the issues related to the communication
  - Embedded Systems Laboratory, hands-on design exercise, integrating the previous courses and applying the lessons learnt in those courses

Course problem description

- Put an embedded JPEG decoder on the market within 12 weeks
  - a platform with multiple embedded VLIW cores is given
  - port application code to embedded VLIW cores
  - efficiently map application to platform
  - quantitative benchmarking
  - system optimisation

- Problem-driven assignment
  - design teams with four members
  - multi-disciplinary and multi-cultural cooperation

Course goals, low level

- Using the development and simulation environment
  - GNU make, command line compiler, linker, debugger
  - upload code and data to memories using the Hive Run-Time library
  - memory map variables and communicate using distributed memories
  - set up network connections using the Æthereal Run-Time library
  - simulate the system using development environment and run the code on FPGA
- Porting sequential C code to the target processor
  - identify which parts of the application that need modifications
  - handle file system and terminal I/O
  - statically allocate variables that are heap-allocated in the original code
  - use the frame buffer and peripherals on the FPGA board
- Parallelising the application
  - orchestrate parallel execution using the Hive Run-Time library
  - exploit data level and task level parallelism in a JPEG decoder
  - explore different ways of implementing inter-processor communication
  - benchmark the decoder in simulation and on the FPGA

Course goals, high level

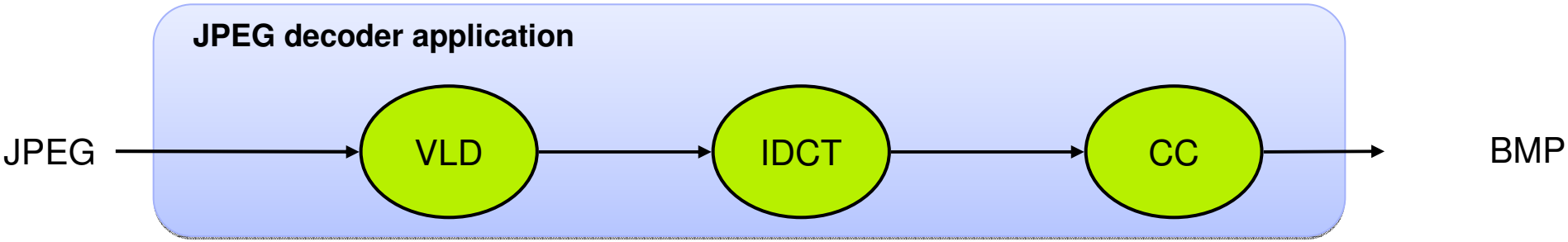
- Learn how embedded and desktop programming differs
- Learn how multi- and uni-processor programming differs
- Learn how to evaluate the performance of an embedded application
- Learn how design decisions impact the quality of the solution

Outline

- Introduction
- Assignment
- Structure
- Discussion

Application

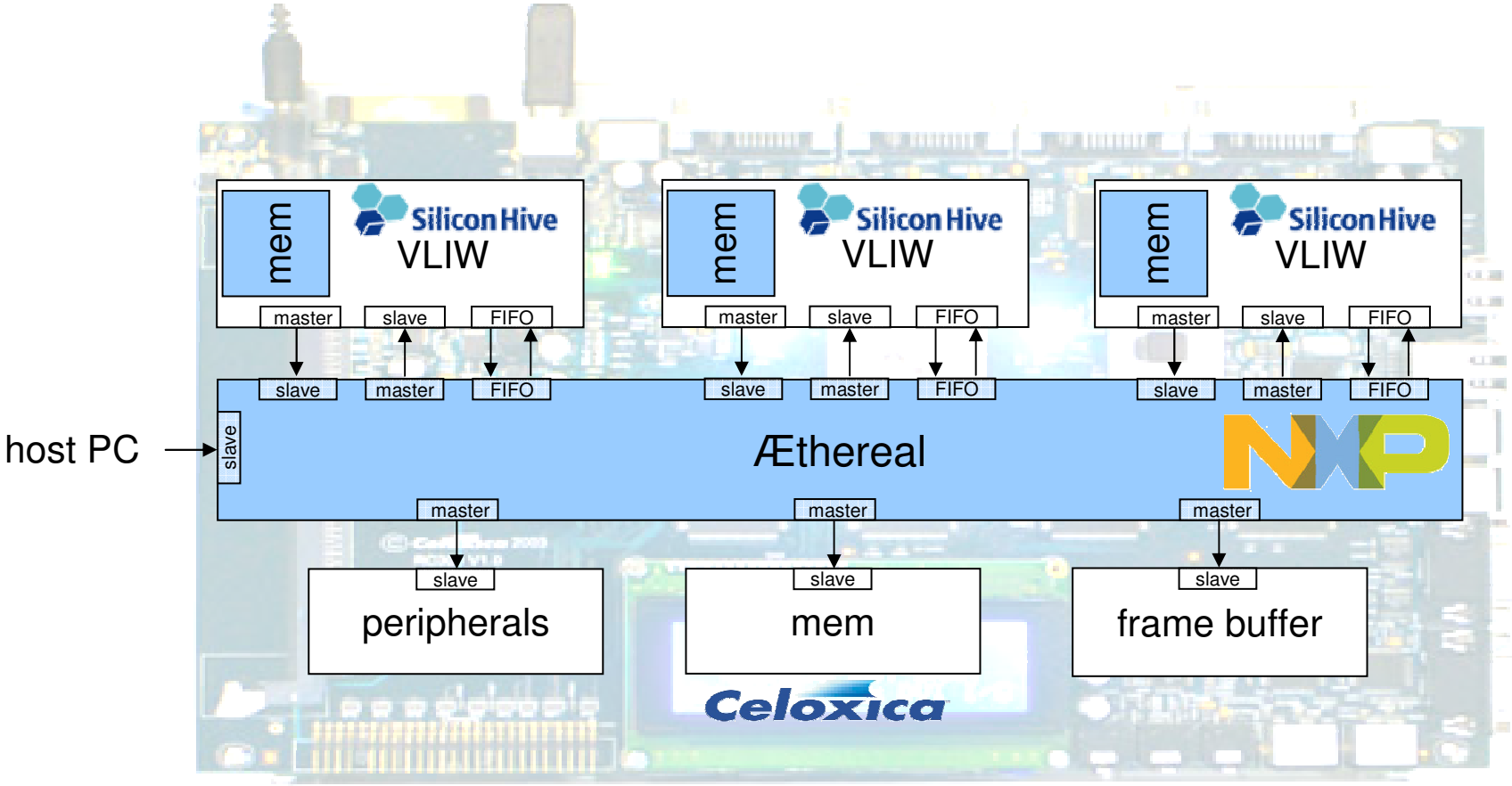
- Fully functional JPEG decoder written in sequential C
- Like many other audio/video decoders, the algorithm consists of
  - Variable-length decoding (VLD),
  - Inverse-discrete cosine transform (IDCT) and
  - Color conversion (CC)



What is important for the course?

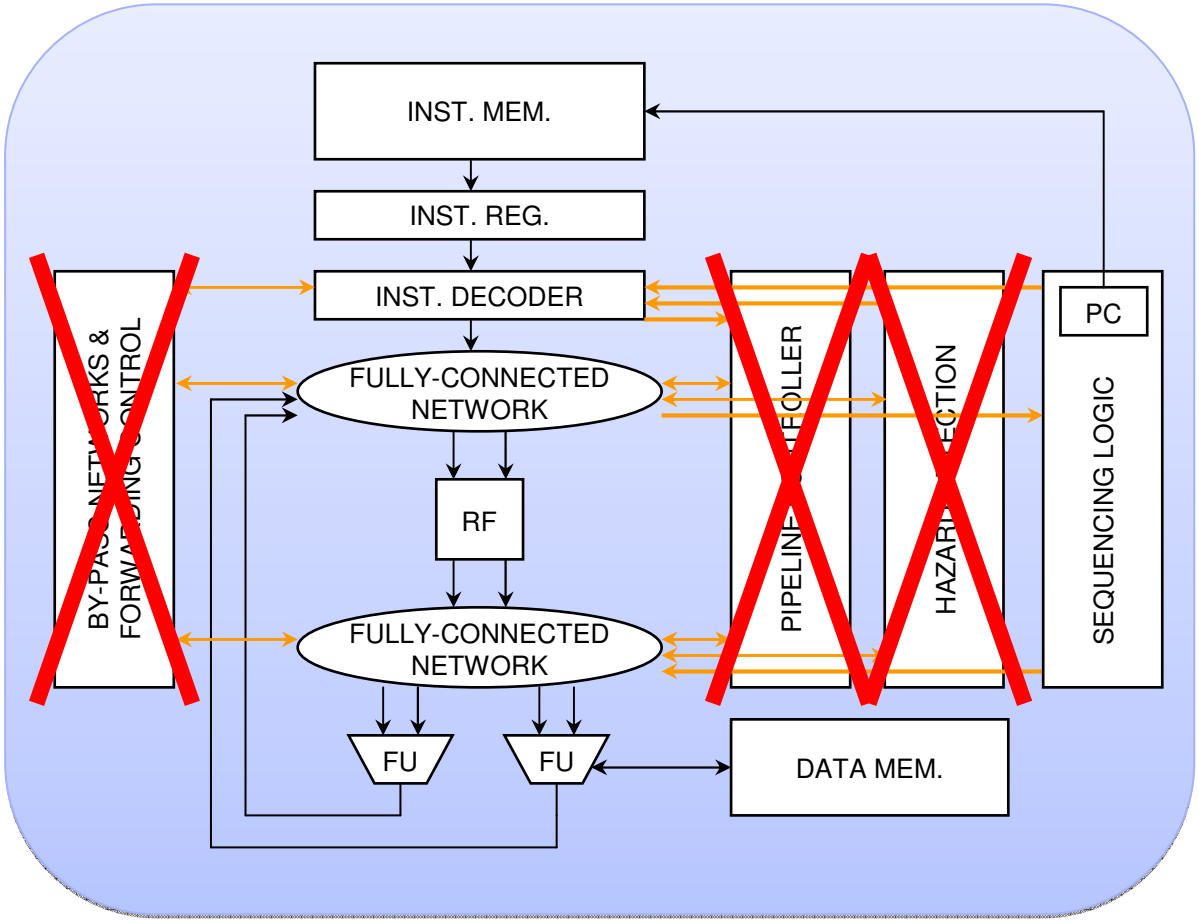
- Reasonable amount of code
- Makes use of dynamic memory and file system I/O
- Retains the technical difficulties of other audio/video codecs
- The algorithm is data dependent
- Not trivially parallel, i.e. the VLD is inherently sequential
- The code is small enough to fit in the local memory of a VLIW core (32 kb)
- Results can be presented on screen
- A JPEG decoder can be turned into M-JPEG, emphasising real-time

Multi-processor network-based archicecture



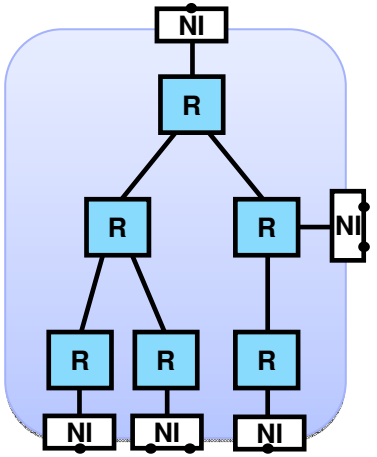
Silicon Hive VLIW template

- Dramatically reduce control overhead
  - expose all pipeline management to the instruction set
  - move complexity to the compiler
  - compiler explicitly schedules all pipeline stages



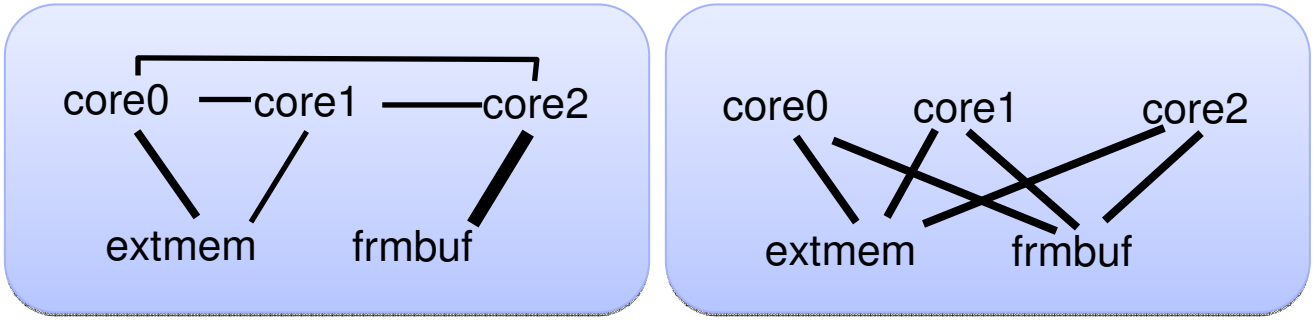
Æthereal Network on Chip

- modular and uniform with routers and network interfaces (NI)
- scalable on the physical and architectural level
  - mesochronouns, GALS, etc
  - more routers and/or NIs
- automation
  - NoC generation, simulation, programming, test bench and traffic generator (IP stub) generation, performance verification
- guaranteed service per connection
  - bounds on latency and throughput
- run-time programmable
  - allow late requirement / application changes



Network configuration

- Network connections must be setup before any communication takes place
  - Unlike a bus connections must be configured between ports
  - Æthereal Run-Time (ART) API is used to configure NoC
- Configuration determines the logical topology
  - What ports that are interconnected
    - Who can write to the frame buffer or read from a core's local memory?
  - What throughput and latency is given the different connections
    - How long time does it take to read and write to background memory?



functional parallelisation data parallelisation

What is important for the course?

- Multi-processor architecture with network on chip and multiple memories
  - communication infrastructure where resource dimensioning is done, but resource assignment is left for the students
  - cores with fixed-point arithmetic, no operating system, no caches and explicit memory management
- Complete system simulation environment
  - one environment for functional verification, performance evaluation and debugging, continuous refinement, etc
- An actual hardware implementation on FPGA
  - face all the real problems, but gives tangible results
- Industrially relevant IP components and tools
  - good for headhunting students for internships, hiring, etc

Outline

- Introduction
- Assignment
- Structure
- Discussion

Design-team roles

- A design team has four members
  - Application expert: understands the JPEG standard and algorithm
  - Hardware expert: has detailed knowledge about the HW building blocks
  - Embedded programming expert: knows about porting and communication
  - Group leader: overviews project, distributes work and reports progress

- Roles are determined within the first week

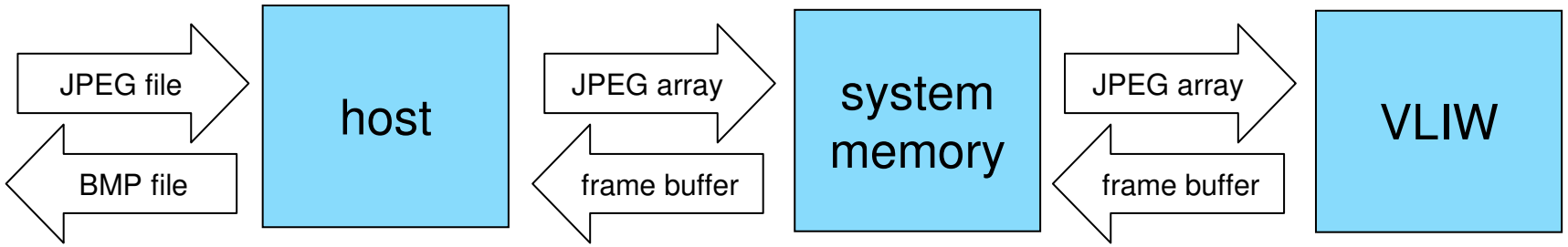
Step 1 – Install and familiarise

- Getting of the ground
  - Students download, unpack and test source code distribution (JPEG decoder)
  - Acquiring documentation for the development environment, HW blocks and APIs
- Familiarise with a simple ‘add’ example
  - deciphering Makefile and source code
- Run the application on FPGA

Done during the first lab session

Step 2 – Single core solution

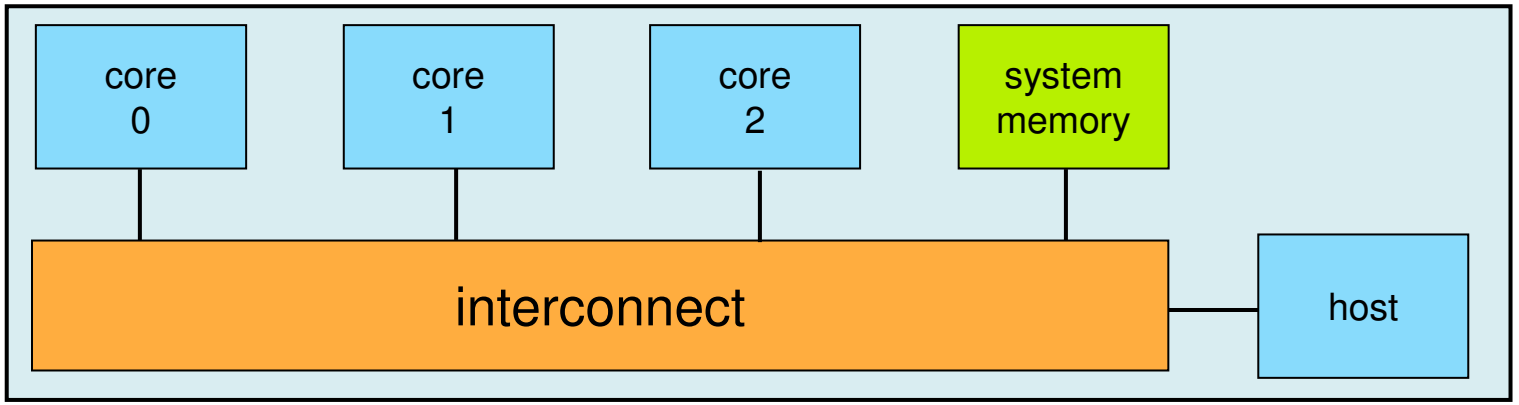
- Run the decoder on a single VLIW core
  - the host reads a JPEG file and stores it as a byte array in system memory
  - the host uploads program code to the VLIW and starts its execution
  - the VLIW decodes image to frame buffer in system memory
  - The host downloads the contents of the system memory and writes it to a bitmap



Typically takes three to four weeks

Step 3 – Distribute code over multiple cores

- Code can be parallelised in many ways
  - functional partitioning (e.g. VLD, IDCT and CC), or data partitioning (tiling)
  - get a balanced load on the cores for high performance, measure stall cycles



Everything from one day for naïve tiling to six weeks for pipelining

Requirements

- Group approved when
  - working single core solution
  - two working parallelisations of the code
  - benchmarks comparing solutions
  - presentation and report explaining approach and results
  - (compare with the low-level goals)
- Grades determined by how well approach and results are explained (compare with the high-level goals)
- Expected load: 9 hours per week
  - 3 + 3 = 6 hours in lab.
  - 3 hours outside lab.

Assessment

- Oral presentations
  - short presentation per group (15 minutes)
  - individual presentations (10 minutes)
- Written report per group (4 pages sig-alt template)
- Meetings (~1 per week)
  - Group meetings
  - Group leader meetings
  - Application group
  - Benchmark committee

Outline

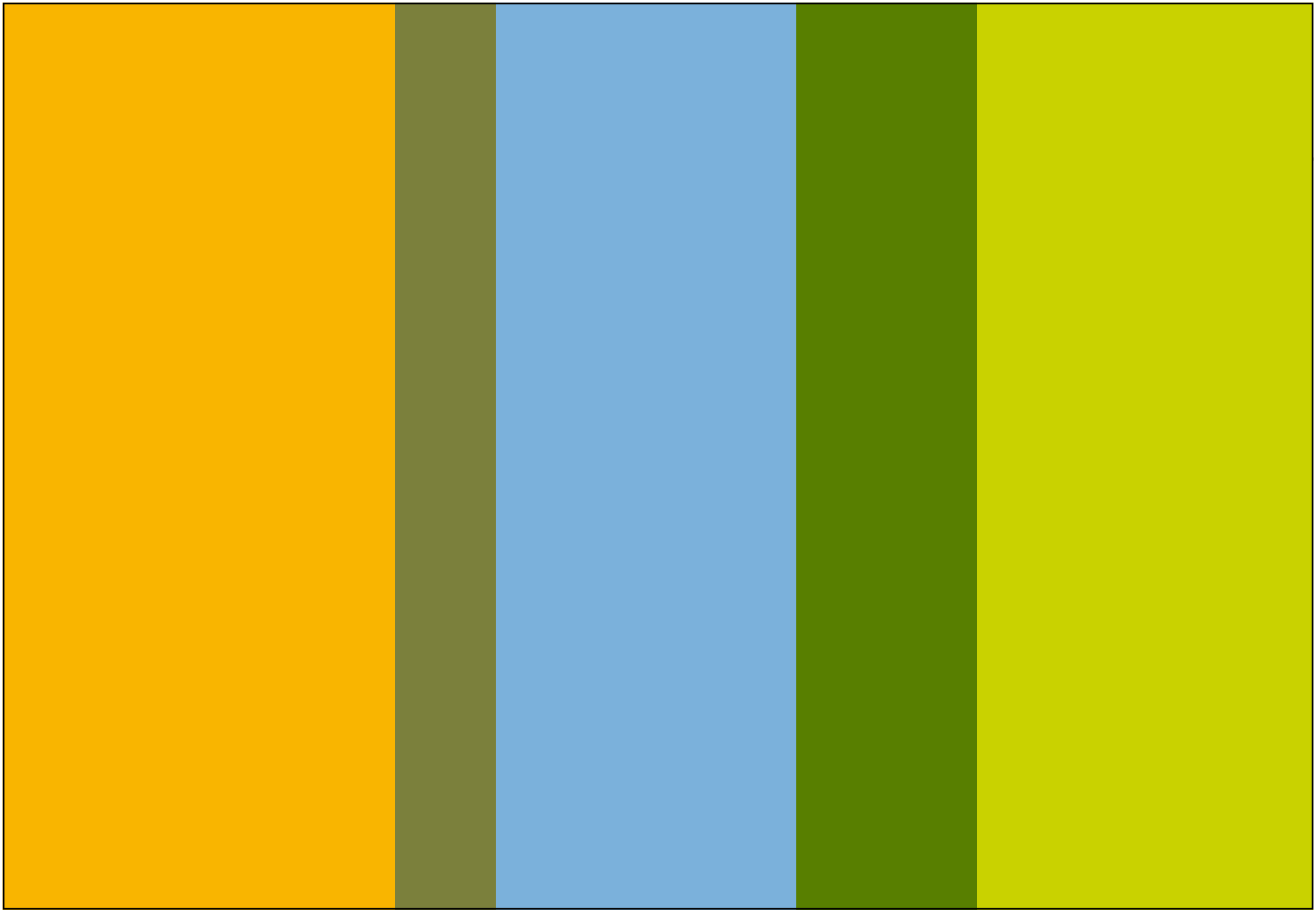
- Introduction
- Assignment
- Structure
- Discussion

Challenges

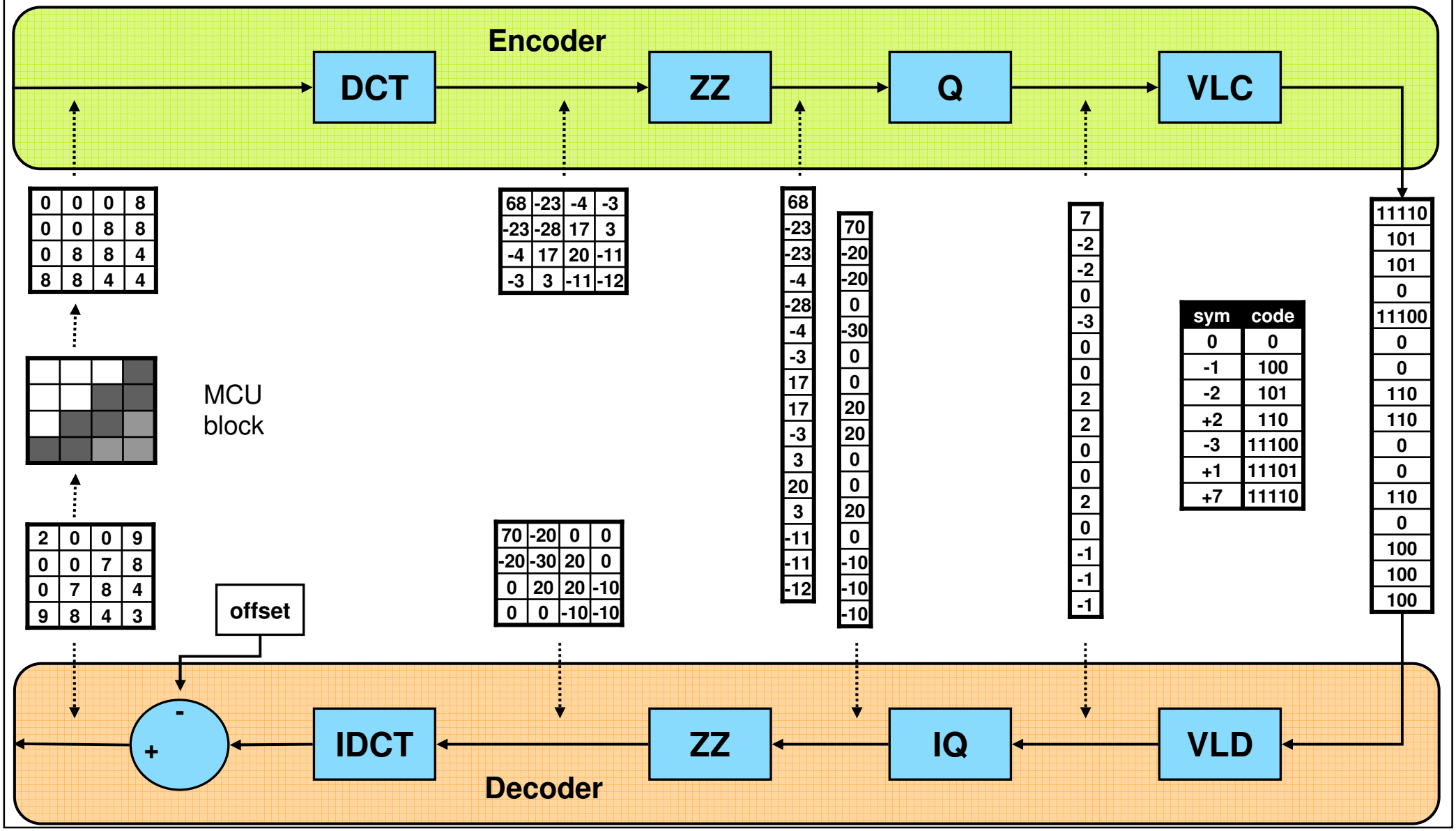
- How to help without solving the problem for them?
  - minimising the problems involving understanding interfaces and I/O devices
  - tutorial exercises and demonstrative examples
  - inter-group discussions where problems/solutions are shared
- How to debug the students non-working code?
  - show them how to do structured test and version control already from day one
  - …in real life there is no one out there to help you
- How to cope with research-quality (buggy) tools?
  - we work closely with the tool developers, good for them, good for us
- How to give even more freedom to the students?
  - not feasible with current 5 ECTS credits
- How to set a grade on such a “fuzzy” course?
  - lots of time spent with the students has proven to make it easier than we initially thought

Conclusions

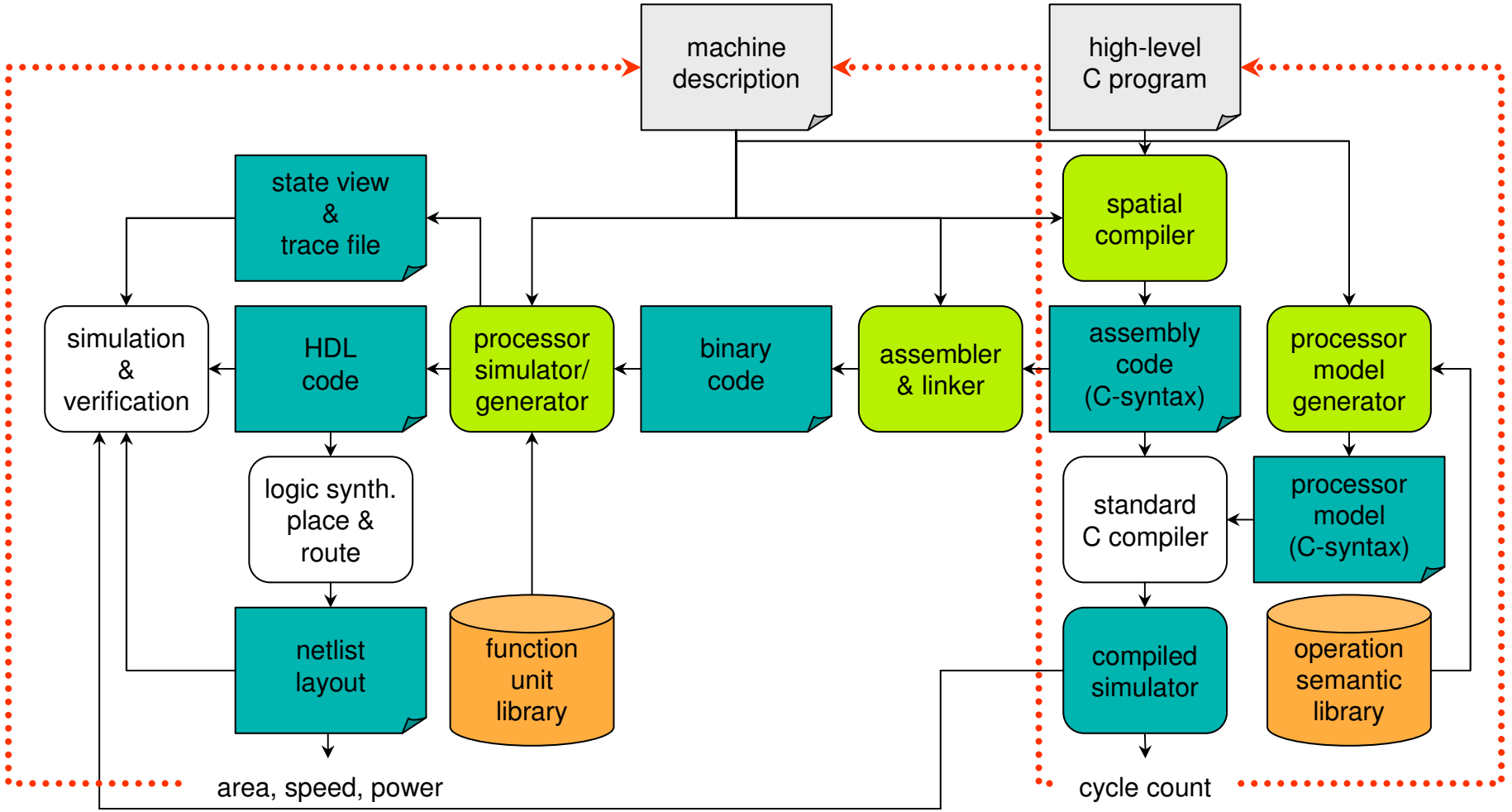
- Like other courses, we emphasize
  - the growing importance of software in embedded systems
  - resource-limited performance-oriented design
  - challenges in areas like personal time management and teamwork
- In contrast to other courses, we stress
  - the challenges involved in going from uni- to multi- processor systems, and
  - the importance of communication and synchronisation
- Based on student evaluations, we believe
  - that the Embedded System Laboratory delivers a level of realism that helps in motivating the students and reinforcing the experiences gained during the course



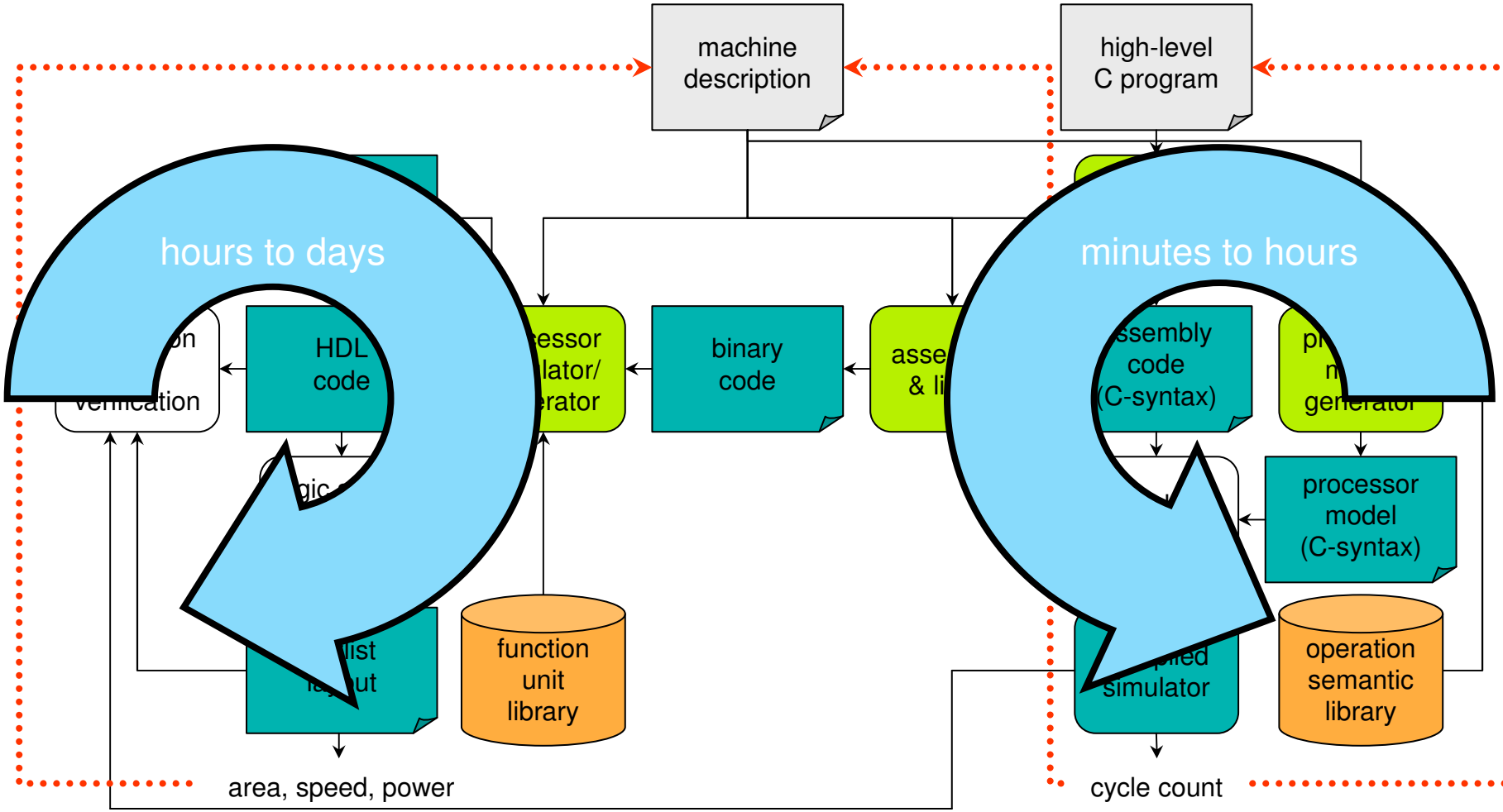
Overview: JPEG encoding/decoding



Internal design flow



Internal design flow

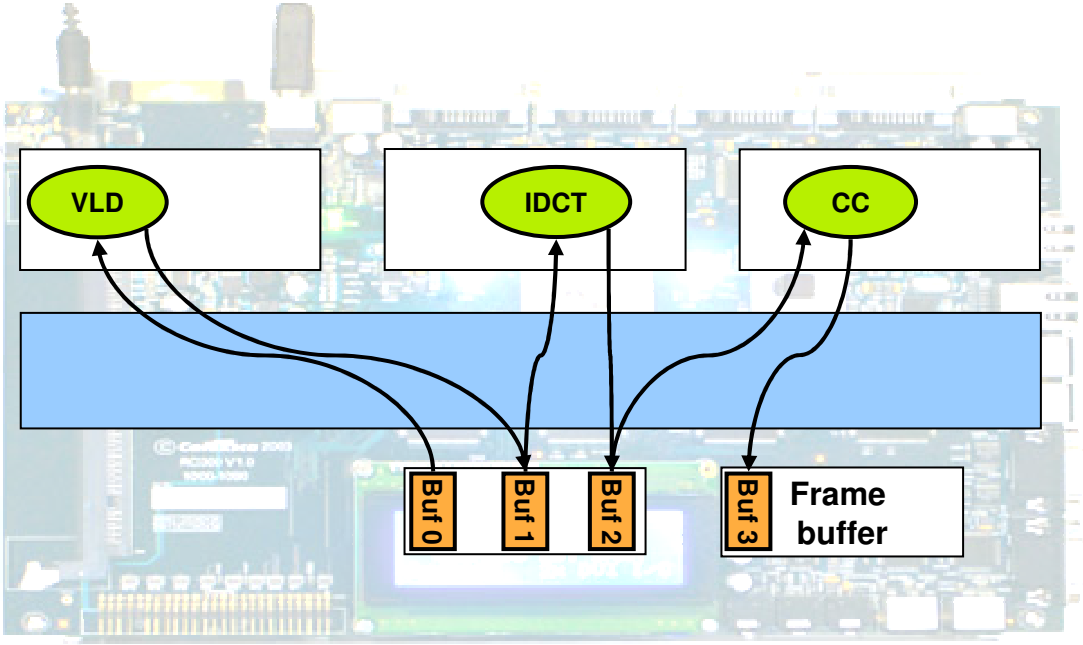


Four simulation levels

- Makefile target *crun*
  - All code is compiled with *gcc*
  - Check correctness of application code and testbench
- Makefile target *unsched*
  - Core code is compiled with *hivecc*, but still non-optimised
  - Generate code with instruction semantics of the specified core
- Makefile target *sched*
  - Schedule to maximize Instruction Level Parallelism (ILP)
  - Check if core has enough resources, i.e. register files and interconnect
  - Cycle count for *ideal case*, infinitely fast communication
- Makefile target *fpga*
  - Generate microcode for cores and binary for host PC
  - Full detail of communication and arbitration overhead

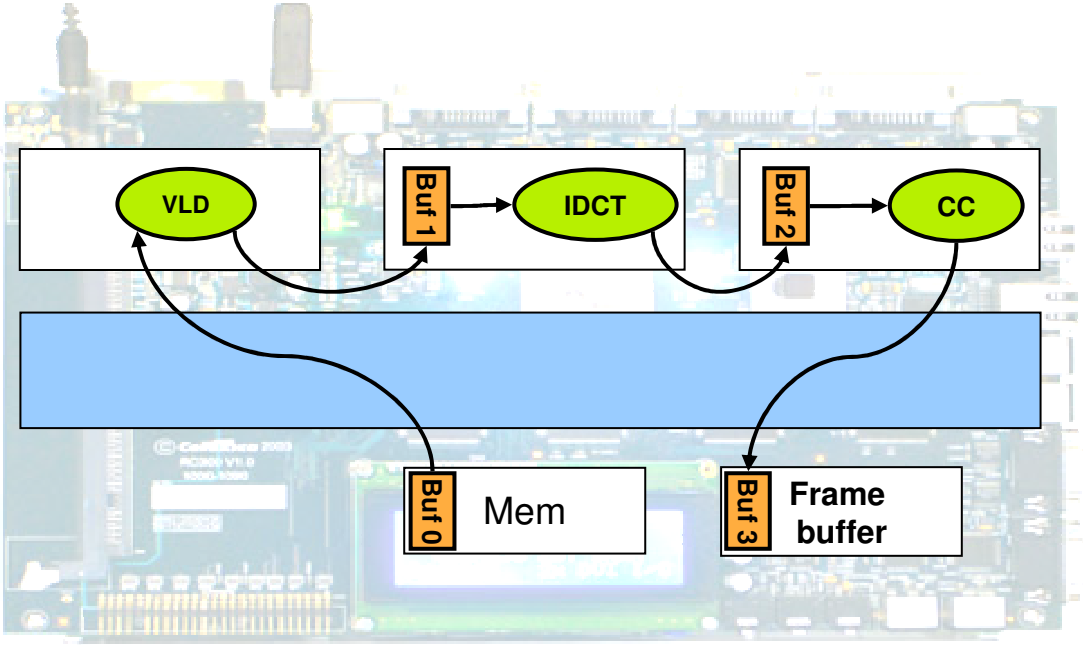
Functional partitioning (1)

- One task per core
  - No code duplication
  - Balanced load?
- Communication through system memory
  - Creates contention for memory



Functional partitioning (2)

- Some tasks read from local memory
  - Faster
  - Does communicated data fit?



Data partitioning

- All tasks on all cores
  - Code duplication
  - Each core decodes 1/3 image
  - Balanced load?
- All cores must read entire image
  - Severe contention for memory