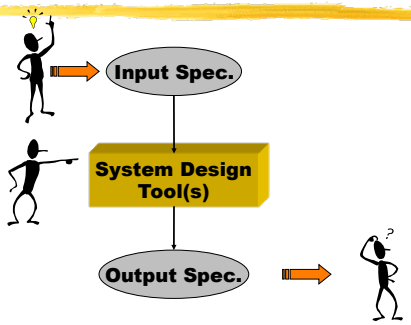
Design Methodologies

Kris Kuchcinski

kris@cs.lth.se

General Methodology



Input to System Design

- Executable specification (functional requirements):
  - usually provided as interacting processes/tasks,
  - very often multi-language specifications,
  - can be simulated and verified,
  - can be used to perform analysis, e.g, estimation.
- Specification languages: C, C++, VHDL, Verilog, SystemC, Esterel, SDL, CAL, etc.
- Set of (non-functional) design requirements (cost, speed, I/O rate, power consumption, etc.).

Good requirements

- Correct.
- Unambiguous.
- Complete.
- Verifiable: is each requirement satisfied in the final system?
- Consistent: requirements do not contradict each other.

Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Good requirements, cont'd.

- Modifiable: can update requirements easily.
- Traceable:
  - know why each requirement exists;
  - go from source documents to requirements;
  - go from requirement to implementation;
  - back from implementation to requirement.

Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Setting requirements

- Customer interviews.
- Comparison with competitors.
- Sales feedback.
- Mock-ups, prototypes.
- Next-bench syndrome (HP): design a product for someone like you.

Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Output from System Design

- A set of system modules assigned to system components (CPU's, DSP's, ASIC's, etc.).
- Communication modules.
- Each module can be further synthesized to hardware using *high-level synthesis* or *compiled to software*.

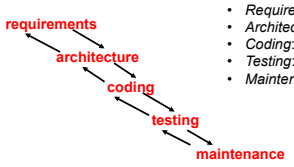
Design flow

- Design flow: sequence of steps in a design methodology.
- May be partially or fully automated.
  - Use tools to transform, verify design.
- Design flow is one component of methodology. Methodology also includes management organization, etc.

Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Waterfall model

- Early model for software development:



- *Requirements*: determine basic characteristics.
- *Architecture*: decompose into basic modules.
- *Coding*: implement and integrate.
- *Testing*: exercise and uncover bugs.
- *Maintenance*: deploy, fix bugs, upgrade.

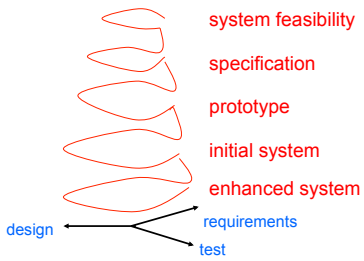
Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Waterfall model critique

- Only local feedback --- may need iterations between coding and requirements, for example.
- Doesn't integrate top-down and bottom-up design.
- Assumes hardware is given.

Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Spiral model



Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Spiral model critique

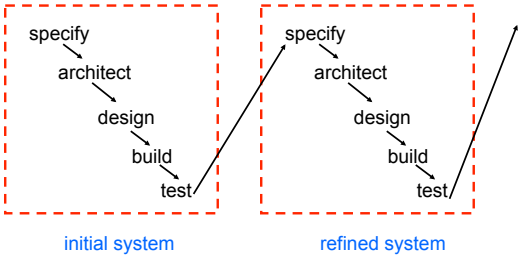
- Successive refinement of system.
  - Start with mock-ups, move through simple systems to full-scale systems.
- Provides bottom-up feedback from previous stages.
- Working through stages may take too much time.

Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

Stepwise refinement

- At each level of abstraction, we must:
  - *analyze* the design to determine characteristics of the current state of the design,
  - *refine* the design to add detail.
- Formal or informal refinement.
- System analysis and tool support.

Successive refinement model

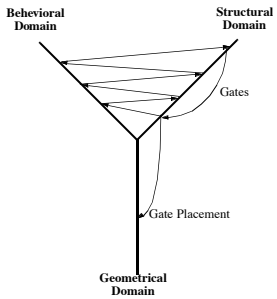


Source: W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*

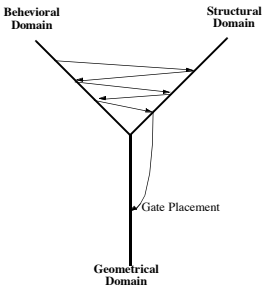
Top-down vs. bottom-up

- Top-down design:
  - start from most abstract description,
  - work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.

Bottom-Up Design



Top-Down Design



Specifications

"Instead of promoting the one language that will solve all design problems (which does not and will not ever exist), this book takes the view that different problems demand different languages, and a designer who knows the spectrum of available languages has the advantage over one who is trapped using the wrong language."

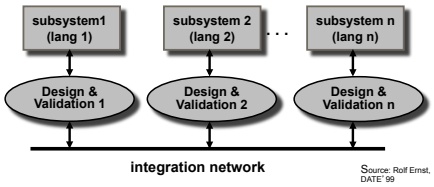
Stephen A. Edwards, "Languages for Digital Embedded Systems"

Specifications

- No single design language.
  - languages/semantics established in application domain: data-flow graphs, FSM, ...
  - domain specific optimizations required signal flow graph transformations, FSM transformations, ...
  - investment in language environments.

Multi-language System Design — Status

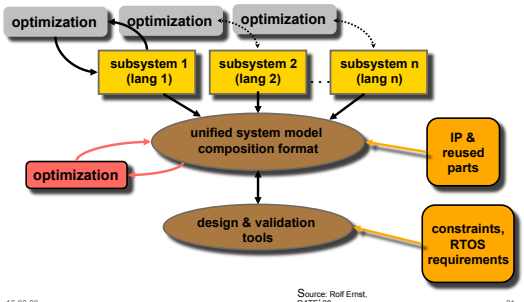
- co-modeling
- co-simulation
- individual implementation with low-level communication synthesis



Source: Rolf Ernst, DATE'99

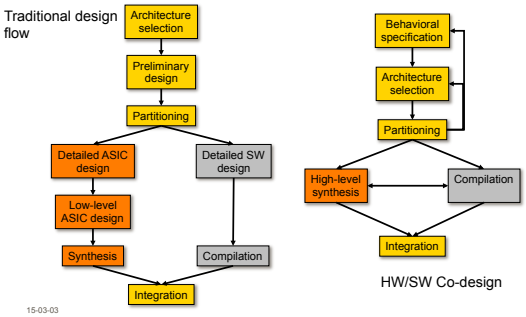
Global Optimization and Design Space Exploration

General View



Source: Rolf Ernst, DATE'99

HW/SW Co-design



Basic Characteristics of the Methodology

- *Behavioral specification* is given for the complete heterogeneous system, regardless of how different parts will be later implemented.
- *Analysis techniques* are provided; specially different estimation techniques.
- *Synthesis tools* are used to automatically explore a design space.
  - high-level synthesis, RTL synthesis,
  - compilers, cross-compilers,
  - interface generators,
  - etc.

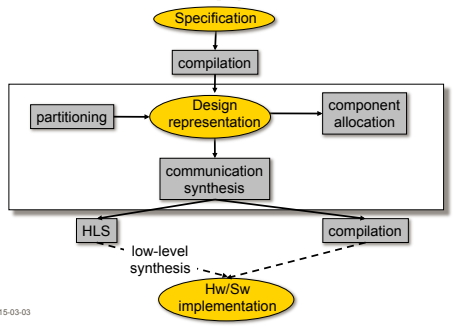
Estimation of Design Parameters

- Estimation of parameters such as size, cost, power consumption.
- Does not need to be very precise but has to be "consistent" — follows real design parameters.
- Usually 15%-20% inaccurate.
- Trade-off between accuracy and estimation time.

Improvements of the Design Process

- High-level specification is made before architecture selection and implementation *decisions can be made more accurate* (better exploration of architectures).
- A uniform description of HW and SW makes it possible to *move parts of the systems between HW and SW*.
- HW and SW development is moved closer and the *integration cost is reduced*.
- An *early evaluation of system characteristics* is possible.

An Example of a Design Flow

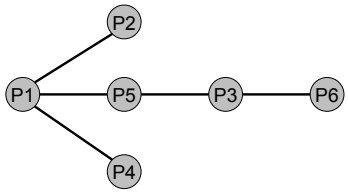


Specification Example

"Extended" VHDL

```
port(IP1,IP2:in INTEGER; OP1,OP2:out INTEGER);
• • •
signal S1,S2,S3,S4,S5,S6:INTEGER;
P1 : process
  • • •
  receive(IP1);
  • • •
  send(S1,...);
  • • •
  send(S3,...);
  • • •
  receive(S5);
  • • •
end process P1;

P2 : process
  • • •
  receive(IP2);
  • • •
  receive(S1);
  • • •
  send(OP2,...);
  • • •
end process P2

P3 : process
  • • •
  receive(S4);
  • • •
  send(S2,...);
  • • •
end process P3;

P4 : process
  • • •
  receive(S3);
  • • •
  send(S5,...,S6,...);
  • • •
end process P4;

P5 : process
  • • •
  receive(S1,S5);
  • • •
  send(S4,...);
  • • •
end process P5;

P6 : process
  • • •
  receive(S2);
  • • •
  send(OP1,...);
  • • •
end process P6;
```


Representation Example



Process communication graph

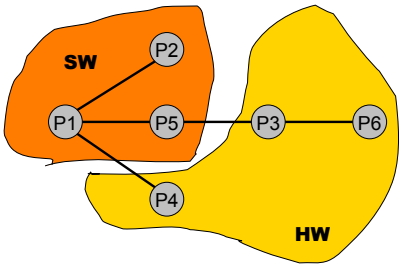
Allocation of System Components

- Decides about the kind and number of components for implementation of the system
  - processing elements: µprocesosrs, micro-controllers, DSP's, ASIP's, ASIC's, FPGA's, etc.
  - storage elements: memories, register files, registers, etc.
  - communication devices: buses, point-to-point links, networks, etc.
  - specialized I/O devices: A/D, D/A, frame grabbers, etc.

Partitioning

- *Functional* partitioning vs. *structural* partitioning.
- Abstraction level.
- Partitioning granularity (*fine* or *course*):
  - modules,
  - processes and procedures,
  - instructions.
- Partitioning objective:
  - performance,
  - minimal communication,
  - low power,
  - combination of several criteria.

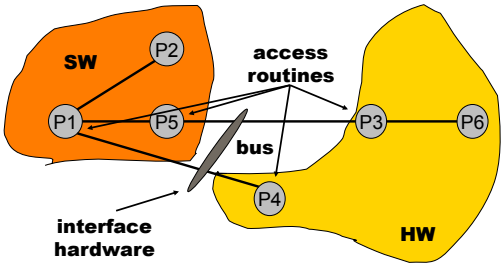
Partitioning Example



Communication Synthesis

- Creation of abstract communication channels by communication clustering.
- Communication refinement
  - selection of communication lines width,
  - protocol selection,
  - etc.
- Interface generation:
  - device drivers,
  - communication hardware,
  - etc.

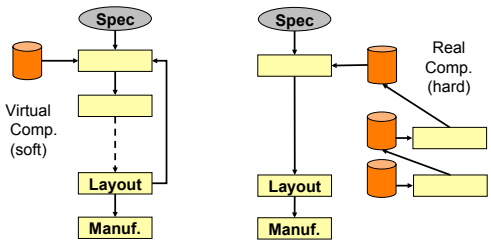
Communication Synthesis Example



IP components

- Hard IP — rigid design in silicon.
- Soft IP — written in HDL languages and synthesized on demand.

Top-down vs. Bottom-up methodology with IP components

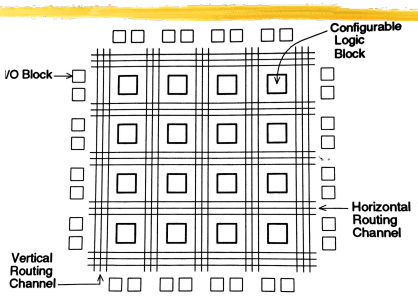


Source: Daniel D. Gajski, *IP-Based Design Methodology*, DAC'99.

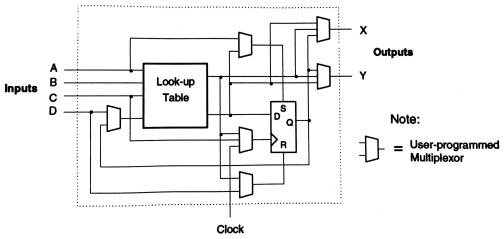
Verification Methods

- *Simulation* - making a model of the system, executing the model and observing relevant signals, etc.
- *Prototyping* - building the system of discrete components (FPGA's, prototyping environments, etc.).
- *Formal verification* - proving certain properties of the system
  - equivalence checking,
  - model checking,
  - theorem proving, etc.
- *Correct by construction*.

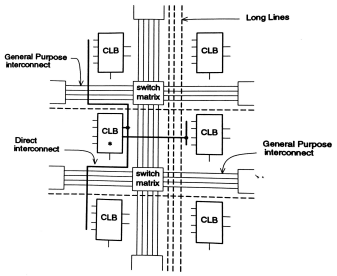
FPGAs Architecture



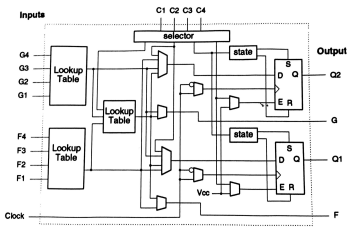
FPGAs Architecture (cont'd)



FPGAs Architecture (cont'd)



FPGAs Architecture (cont'd)



Design Automation

- Uses internal representations which are usually based on graphs.
- Graph algorithms (shortest path, Hamiltonian circuit, topological sort, depth-first-search, breadth-first-search, etc.).
- Optimization methods — (M)ILP, CP, heuristics, etc.
- *Tractable* and *intractable* problems.
- *Decidable* and *undecidable* problems.
- *Decision problems* and *combinatorial optimization problems*.

Complexity Classes (informal introduction)

- Two types of computational complexity are distinguished: *time complexity* and *space complexity*.
- If there exist an algorithm which can solve a problem in a polynomial time (e.g., $O(n^2)$) then we have a class *P*.
- If an algorithm has an exponential complexity then it is *NP-complete* problem.
- *NP-hard* refers often when the problem is at least as hard as NP-complete. It is also used for the optimization version of combinatorial problems for which the decision version is known to be NP-complete.

Consequences

- Most of the problems which need to be solved in design automation are NP-complete or NP-hard.
- Usually only small problems can be solved exactly.
- Need for algorithms which do not guarantee optimal solutions but "good enough" solutions
  - *approximation algorithms* — guarantee a solution with a cost that is within some margin of the optimum,
  - *heuristics* — algorithms that are constructed based on "rules-of-thumb"; nothing can be said in advance about the quality of the result.

Undecidable problems

- There are decision problems for which no algorithm at all exist, regardless of its complexity.
- *Halting problem* — algorithm that accepts a computer program and decide whether or not this program will stop after a finite computation time.
- Undecidable whether or not a given instruction can be executed — no static analysis is possible in a general case.