Hardware-Software-Codesign

Felix Leitl

16. September 2025

Inhaltsverzeichnis

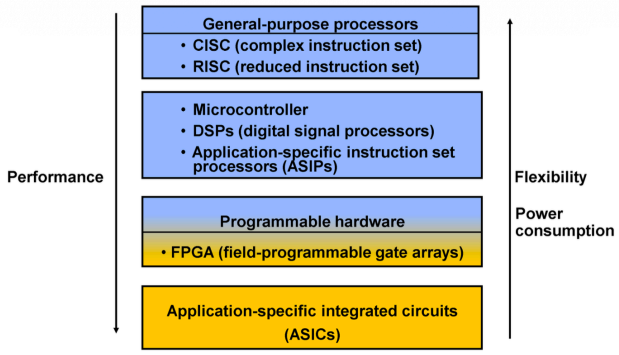


Abbildung 1: Implementation Styles

Architectures

See fig. 1

General-Purpose Processors

- Properties
  - high performance for a broad range of applications, not optimized for any particular application
  - high power consumption
- Design
  - highly optimized circuits
  - design times ¿ 100 person years
  - cost/device only cheap when fabricated in huge numbers
- Application areas
  - PCs
  - Smartphones

Reasons for High Performance

- Exploitation of parallelism
  - multiple scalar units (superscalar)
    - Functional units: integer units, floating-point unit, load/store-unit
    - Dynamic scheduling
    - instruction compatibility

        * complex control unit
    - deep instruction pipeline
        * fetch | decode | read | execute | write back
        * each scalar unit my have individual pipeline depth
        * branch prediction
- Multi-level memory hierarchy
    - $\uparrow$ Speed $\Rightarrow \downarrow$ Size $\Leftrightarrow \uparrow$ Size $\Rightarrow \downarrow$ Speed
    - Register $<$ Cache $<$ Main Memory

General-Purpose Processors and Real-Time

- Execution time of programs is not well predictable
    - Dynamic scheduling
    - Caching
    - Branch prediction
- Complex I/O and memory interface

Multimedia - Extensions and Sub-words

- Multimedia Applications
    - 8 / 16 Bit data types
    - many arithmetic operations
    - huge data sets $\Rightarrow$ Huge I/O-bandwidth
    - high data parallelism
- Sub-word execution
    - split 32/64-bit registers and ALUs into smaller sub-units
    - instructions execute in parallel on these sub-units
    - compromise between usage of parallelism and available data paths
    - currently most often based on hand-written assembly routines
    - Depends on language with defined bit width per data type and diverse overflow semantics
    - Compiler needs to automatically detect sub-word parallelism

Microcontrollers

- For control-dominant applications
  - control flow dominant programs (many branches and jumps)
  - few arithmetic operations
  - low throughput requirements
  - multitasking
- Microcontrollers are optimized for
  - Bit- and logic-level operations
  - Registers often realized in RAM: context switch through pointer operation, therefore shortest interrupt latencies
  - integrated peripheral units (e.g. A/D, D/A, CAN, Timer)
- Systems with 4/8 bit processors
- Microcontrollers with high performance
  - 16 - 64 bit processors
  - Application areas
    - Systems with high control-dominant parts and additional demand
    - High throughput (telecommunication, automotive)
    - Computational power (industrial control, signal processing)
    - As a part of an SOC

Digital Signal Processors

- Signal processing applications
  - data-flow dominant
  - many arithmetic operations, less branches and jumps
  - high degree of parallelism
  - high throughput requirements
- DSPs are optimized for
  - parallel instruction processing (MAC fig. 2)
  - Harvard Architecture, simultaneous access of multiple operands
  - zero-overhead loops
  - special addressing modes (circular, bit-revers)

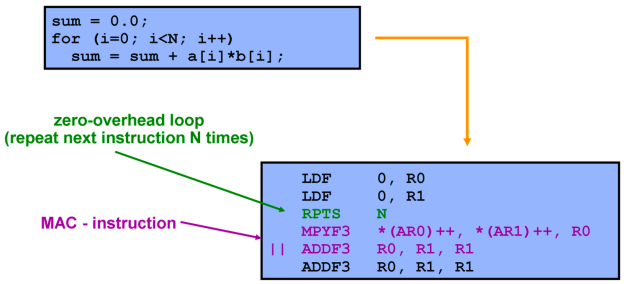


Abbildung 2: Multiply & accumulate

DSP - Arithmetics

- Number formats
  - Mantissa determines precision
  - Exponent determines dynamic range
- Fixed point
  - in case of equal mantissa length smaller (cheaper) and faster as floating-point
  - application design requires consideration of rounding and scaling problems
  - sufficient for many DSP applications
- Floating point
  - high dynamic range of numbers
  - easier application program development

DSP - Program Development

- Assembler, Complier
  - Many DSP features still only usable if code is written in assembly
  - Programming in C $\rightarrow$ profiling $\rightarrow$ time-critical parts in assembly code
- Libraries
  - Programming in C, call of hand-written optimized functions
- Code generation
  - Usage of design environments for modeling, simulation and code generation
  - e.g. Synopsis

DSP - Trends

- Multi-DSP Systems
  - for applications requiring highest performance
  - interfaces available to connect multiple processors
  - Multi-Processor SoC (MPSoC)
- VLIW (Very long instruction word)
  - multiple functional units
  - Compiler detects parallelism and creates a program that schedules multiple instructions jointly
  - Only useful for applications with high degree of instruction parallelism
  - suffers often from low code density
  - compilation difficult

Application-specific instruction set processors (ASIP)

- Specialization
  - instruction set (Operant chaining)
  - functional units (pixel operations, 1/sqrt(x))
  - memory architecture (parallel access onto multiple memory banks)
- Advantages
  - higher performance
  - lower cost
  - smaller code images
  - lower power consumption

Integrated Circuit

Phases of IC Design

1. Design
   (a) Modeling
   (b) Synthesis & Optimization
   (c) Verification
2. Fabrication
   (a) Masks
   (b) Wafer
3. Testing

4. Packaging
   (a) Slicing
   (b) Packaging

Design Styles for ICs

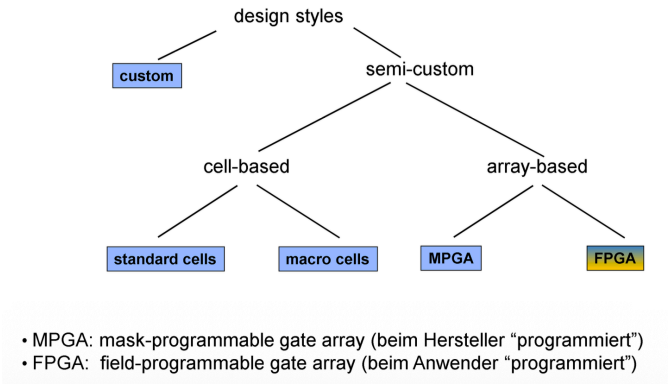


Abbildung 3: Design Styles for ICs

Comparison of Design Styles

	Custom	Cell-based	MPGA	FPGA
Density	very high	high	high	medium-low
Performance	very high	high	high	medium-low
Design time	very long	short	short	very short
Manufacturing time	medium	medium	short	very short
Cost- low volume	very high	high	high	low
Cost- high volume	low	low	low	high

Abbildung 4: Comparison of Design Styles

FPGA (field programmable gate arrays)

- Logic blocks
- I/O-blocks
- Interconnect

FPGA Design Steps

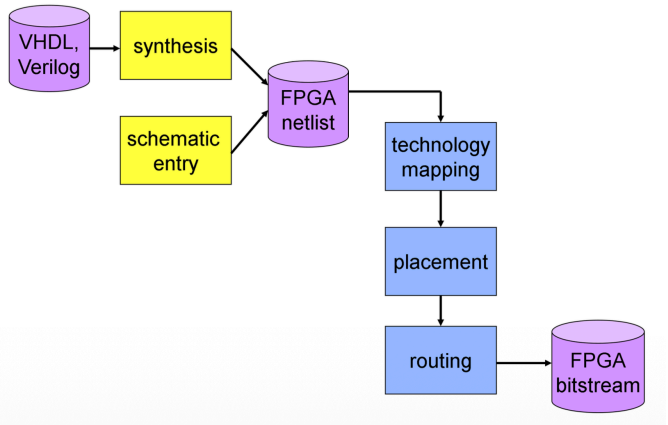


Abbildung 5: FPGA design steps

FPGA - Application Areas

- Glue Logic
- Rapid Prototyping, Emulation
- Embedded Systems
  - when when processors are too slow or too power-inefficient and
    - flexibility is a must
    - the sale volumes are too low to afford ASIC
- Custom Computing
  - Goal: Combine flexibility of processors with efficient advantages of ASICs
  - e.g. Outsourcing a complex task from SW to HW

System models

Dataflow Graph (DFG)

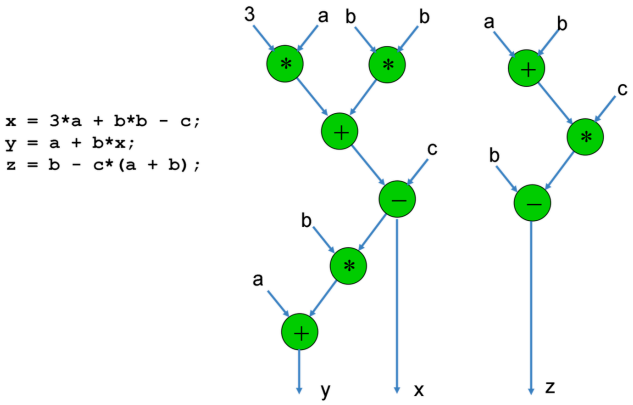


Abbildung 6: Data Flow Graph

Control flow graph (CFG)

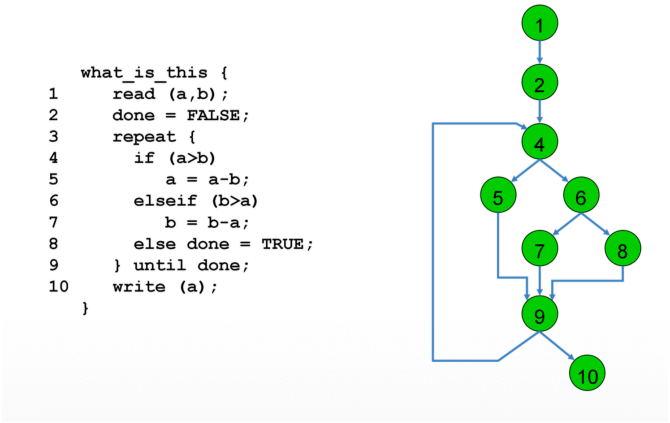
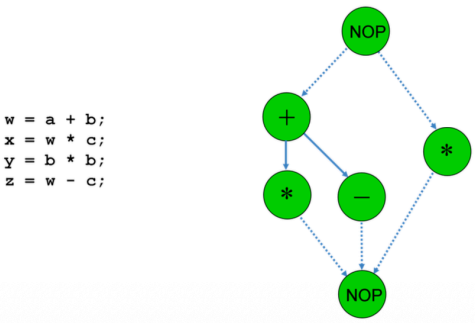


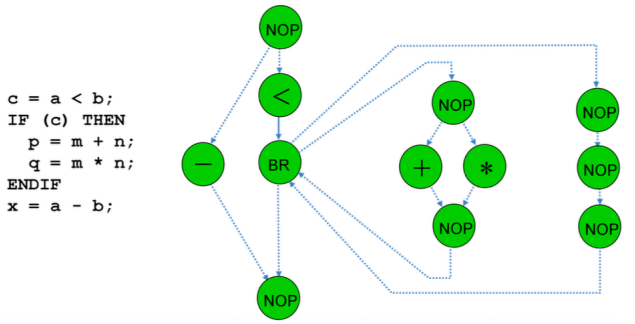
Abbildung 7: Control Flow Graph

Sequencing Graphs

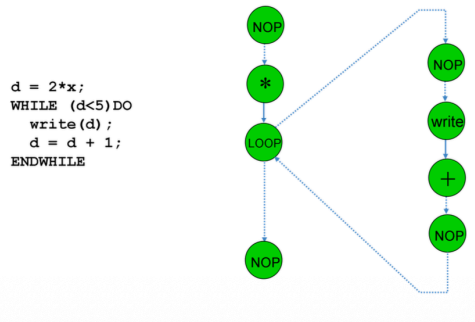
- Hierarchy of entities
  - entities modeling the data flow
  - hierarchy modeling the control flow
- Special nodes
  - Start/End nodes: NOP
  - Branch nodes: BRANCH
  - Iteration nodes: LOOP
  - Module call nodes: CALL
- Attributes
  - Nodes: computation times, costs
  - Edges: conditions for branches, iterations



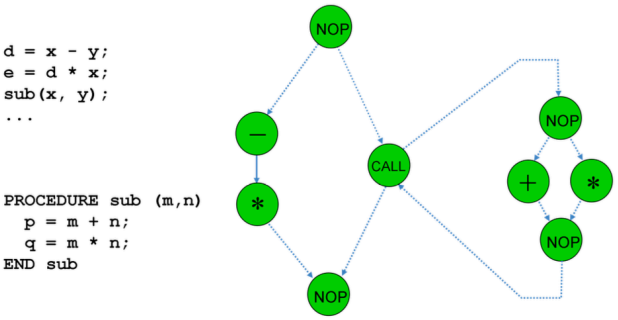
(a) Entity



(b) Branch



(c) Loop



(d) Call

Abbildung 8: Sequencing Graph

Scheduling

- static vs. dynamic
- preemptive vs. non-preemptive
- with vs. without resource constrains
- aperiodic vs. periodic (iterative)

Scheduling without resource constrains

- ASAP (as soon as possible)
  - determines the earliest start times of tasks
  - computes the minimal latency
- ALAP (as late as possible)
  - determines the latest start times of tasks (for a given latency bound)
- Mobility of a tsk is given by the difference between ALAO and ASAP start times
  - Mobility $0 \rightarrow$ task on critical path

Definition

Definition: Schedule

A schedule of a sequencing graph $G_S = (V_S, E_S)$ is a function $\tau : V_S \rightarrow \mathbb{N}$ that satisfies:

$$\tau(v_j) - \tau(v_i) \geq d_i \quad \forall (v_i, v_j) \in E_S$$

Definition: Latency

The latency L of a schedule τ of a sequencing graph $G_S = (V_S, E_S)$ is defined as:

$$L = \max_{v_i \in V_S} \{\tau(v_i) + d_i\} - \min_{v_j \in V_S} \{\tau(v_j)\}$$

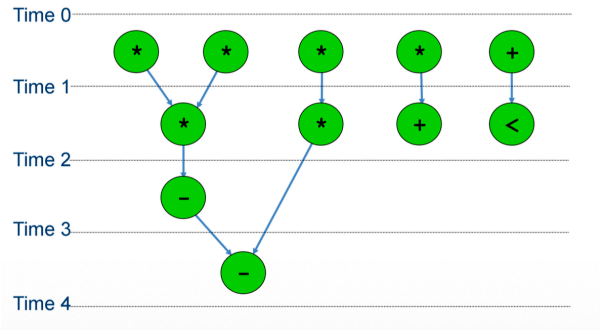
And therefore denotes the number of tine steps of the smallest time interval that includes the execution of all nodes $v_i \in V_S$

ASAP

ALAP

List scheduling

- Task are sorted into a list according to some priority function
- In each step, each free resource will be assigned a schedulable task of highest priority
- Priority functions: number of successor nodes, mobility, ...

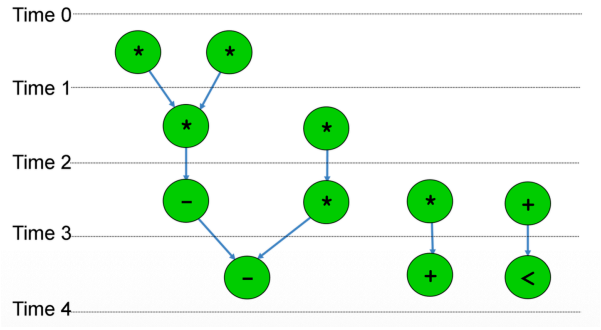


(a) latency-optimal schedule: $L = 4$

1: **function** ASAP($G_S(V_S, E_S), d$)
2: **for each** v_i without predecessor **do**
3: $\tau(v_i)^S \leftarrow 0$
4: **end for**
5: **repeat**
6: Select a node v_i whose predecessors have all been scheduled
7: $\tau(v_i)^S \leftarrow \max_{j:(v_j,v_i)\in E_S}\{\tau(v_j)^S + d_j\}$
8: **until** all nodes v_i scheduled
9: **return** τ^S
10: **end function**

(b) ASAP algorithm

Abbildung 9: ASAP



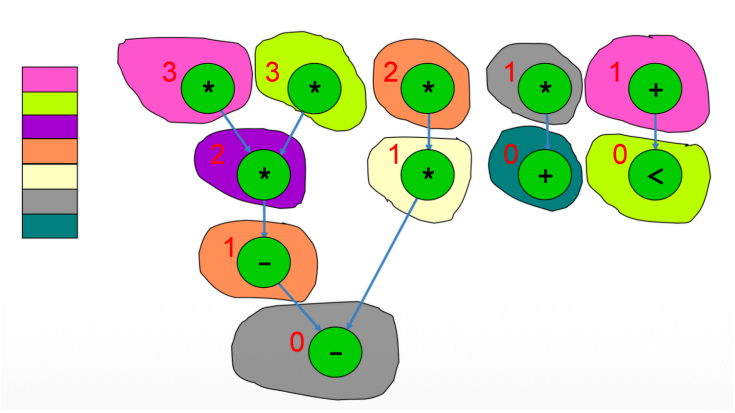
(a) latency bound: $\overline{L} = 4$

1: **function** ALAP($G_S(V_S, E_S), d, \overline{L}$)
2: **for each** v_i without successor **do**
3: $\tau(v_i)^L \leftarrow \overline{L} - d_i$
4: **end for**
5: **repeat**
6: Select a node v_i whose successors have all been scheduled
7: $\tau(v_i)^L \leftarrow \min_{j:(v_i,v_j)\in E_S}\{\tau(v_j)^L\} - d_i$
8: **until** all nodes v_i scheduled
9: **return** τ^L
10: **end function**

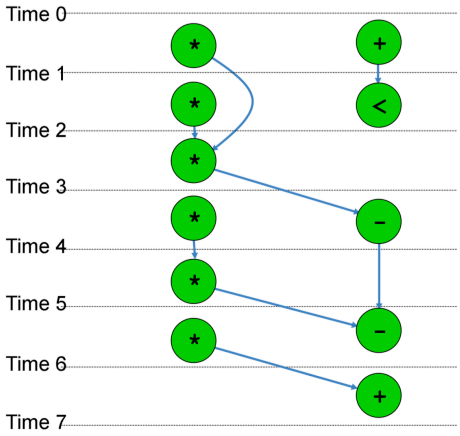
(b) ALAP algorithm

Abbildung 10: ALAP

Periodic scheduling



(a) Priority: Number of successor nodes
Resources: 1 multiplier, 1 ALU (+, -, <)



(b) Result

Abbildung 11: List schedule

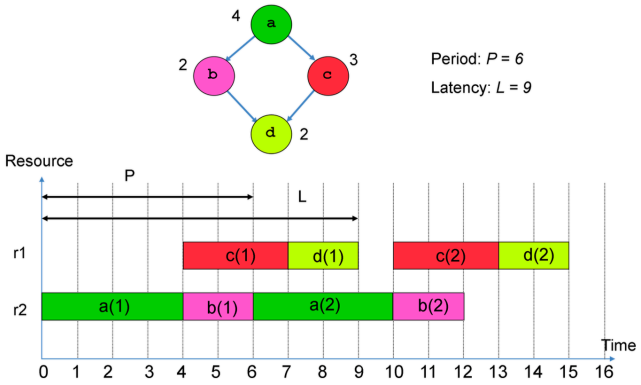


Abbildung 12: Periodic scheduling

Code generation

Double Roof

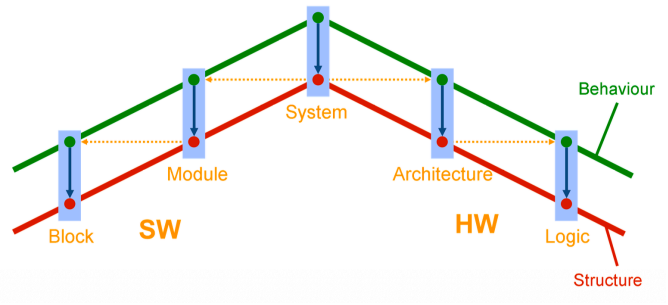
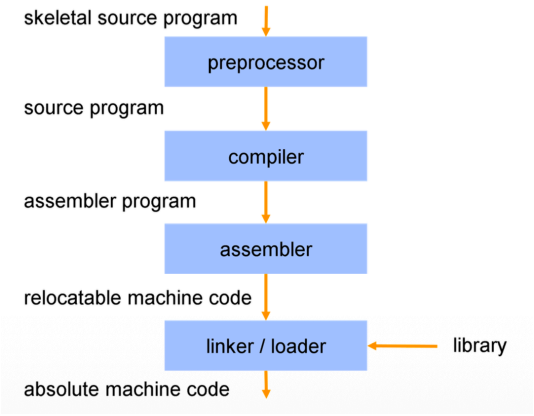
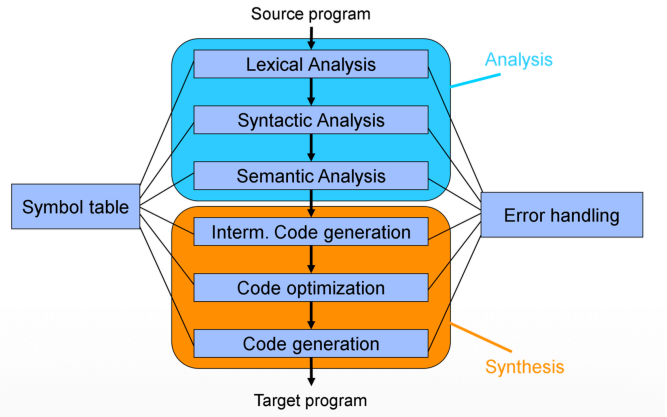


Abbildung 13: Double Roof

Compiler - Basics



(a) Compilation Process



(b) Compilation Process

Abbildung 14: Compiler

Analysis

- Lexical Analysis
  - scan source program and decompose it into symbols

  - regular expression: recognition through finite automata
- syntactic analysis
  - parse sequences of symbols and determine clauses
  - clauses are described by a context free grammar
    - Z → Identifier := A
    - A → A + A | A * A | Identifier | Number
- semantic analysis
  - assure that the pieces fit together semantically
  - Example: type conversion

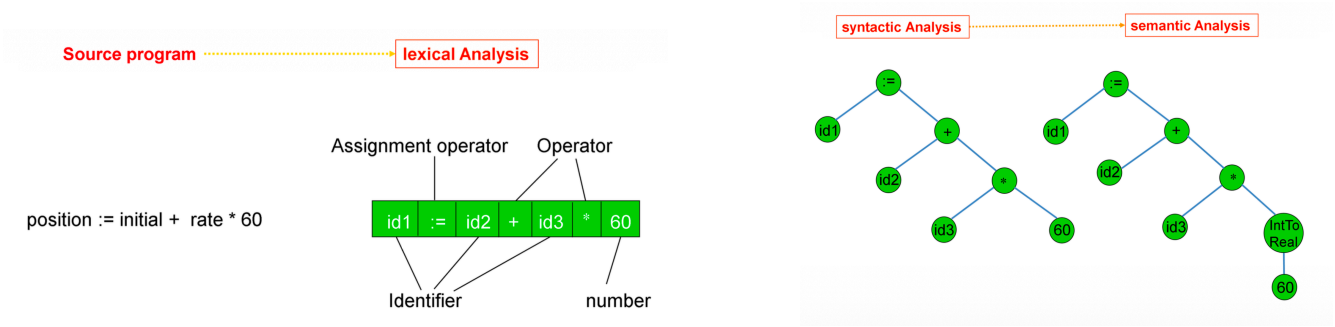


Abbildung 15: Analysis Example

Synthesis

- Generation of intermediate code
  - machine independent → retargeting easier
  - easy to generate
  - easy to be translated
- Optimization
  - General-purpose processors: fast code, fast translation
  - Special processors: fast code, compact code, small memory image
  - of intermediate code or of target code
- Code generation

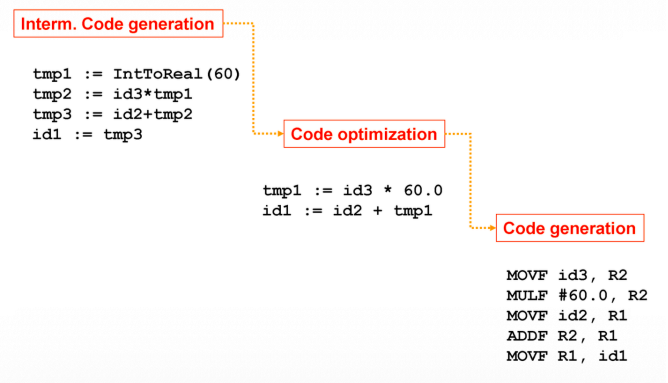


Abbildung 16: Synthesis Example

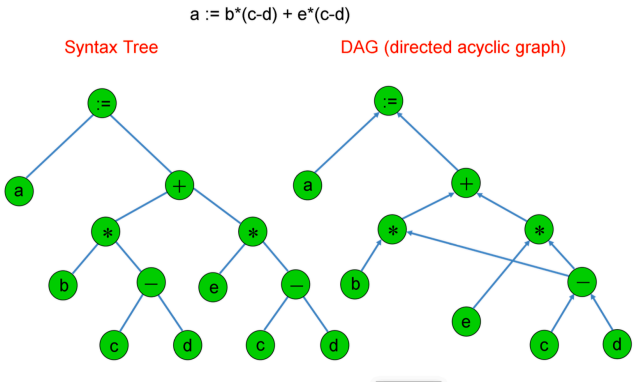


Abbildung 17: Syntax Tree and DAG

Syntax Tree abd DAG

3-Address Code

- Instructions
  - maximum 3 addresses (2 operands, 1 result)
  - maximum 2 operants
- Assignments
  - `x := y op z`
  - `x := op y`
  - `x := y`
  - `x := y[i]`

  - `x[i] := y`
  - `x := &y`
  - `y := *x`
  - `*x := y`
- Control flow
  - `goto L`
  - `if x relop y goto L`
- Sub programs
  - `param x`
  - `call p,n`
  - `return y`
- Advantages
  - resolution of lengthy expressions and nested loops
  - temporary names allow for easy reordering
  - represents already a valid schedule
- Definition
  - `x := y op z`
  - defines x and uses y and z

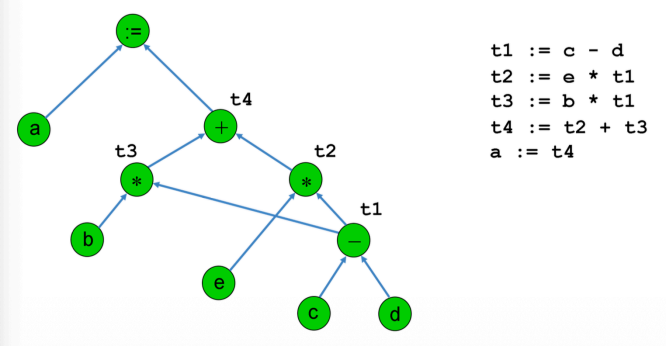


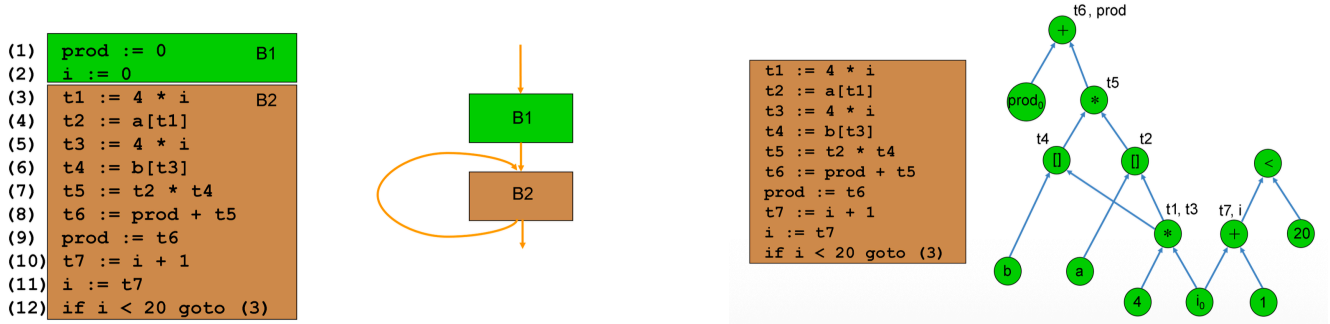
Abbildung 18: Generation of 3-address-code from DAG

Basic Block

Definition: A basic block is a sequence of consecutive instructions in which the control flow enters at the beginning and leaves at the end without branching except at the end.
Sequence of 3-address instructions → set of basic block:

1. Determine block start points:
   - the first instruction
   - targets of branch and jump instructions
   - instructions directly following a branch or jump instruction
2. Determination of basic blocks
   - each basic block includes its block start point
   - includes all instructions (but excluding) the next block start point or until the program ends

CFG and DAG for basic blocks



(a) Degenerated Control Flow Graph

(b) Directed acyclic graph

Abbildung 19: CFG and DAG for basic blocks

Code generation

- Requirements
  - correct code
  - efficient code
  - efficient code generation
- Code generation = Software synthesis
  - Allocation: often given

  - Binding:
    * Register allocation, register binding
    * Code (instruction) selection
  - Scheduling
    * Instruction sequencing

- Goal: efficient register usage
  - instructions on register operands typically shorter and faster as instructions with memory operands
- Register allocation, register binding
  - determine at each point of the program the set of variables hat shall be stored in a register
  - bind each variable to a physical register
  - optimal register binding in a NP-complete problem
  - additional constrains given by special registers of the CPU architecture, compiler and operating system

Code (Instruction) Selection

- Code pattern for implementing a 3-address instruction [fig. 20]
- Problems
  - often inefficient code → Code optimization
  - There may be many alternative instructions
  - some instructions are only executable on certain registers
  - exploitation of special processor properties

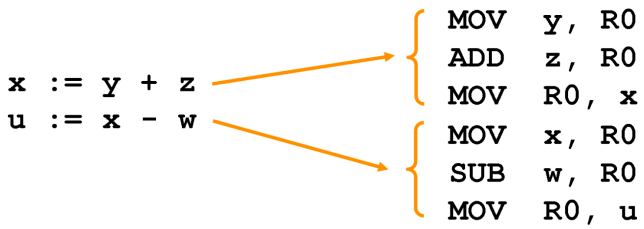


Abbildung 20: 3-address to assembler

Scheduling

Goal: efficient instruction execution sequences, as short as possible, using few registers

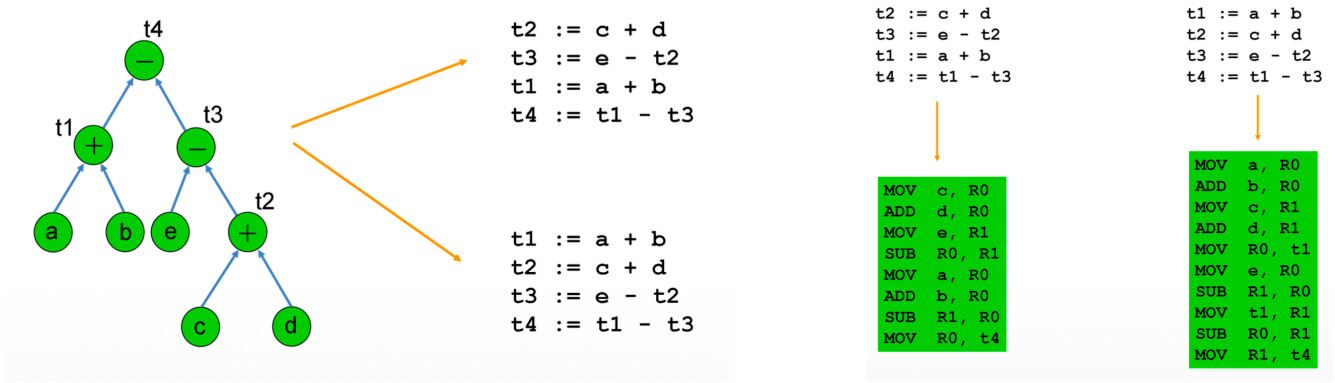


Abbildung 21: Scheduling differences

Register allocation

- global register allocation
  - reserve a certain number of registers
    - for global variables
    - for loop variables
    - for variables in basic blocks
  - user-defined register allocation
    - E.g. in programming language C:
      `register int i;`
- Usage counters
- Register assignment through graph coloring

Usage Counters

- Let a loop `L` be composed of multiple basic blocks
- In case a variable `a` is kept in a register during execution of Loop `L`, we obtain cost savings
  - 1 cost unit for each reference to `a`
    - `ADD R0, R1` (cost 1) instead of `ADD a, R1` (cost 2)
  - 2 cost units for each basic block, if `a` is defined in the basic block and active still thereafter
    - no store necessary (`MOV R0, a`)
- Cost savings for the whole loop

$$\sum_{B \in L} (\text{verwendet}(a, B) + 2 \cdot \text{aktiv}(a, B))$$

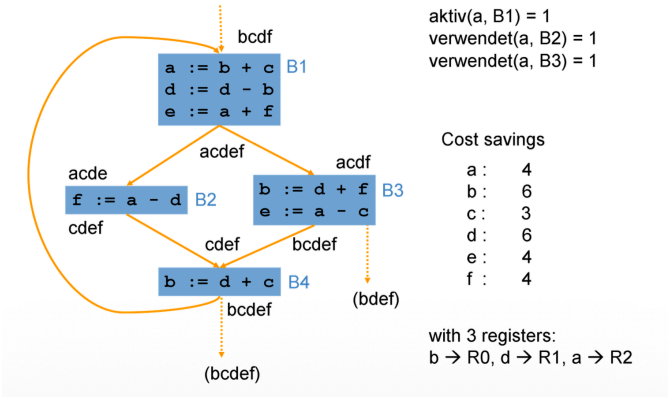


Abbildung 22: Usage Counter - Example
Variables in brackets are given. Start from bottom. Variables below basic block remain active. Basic block depends on variables above.

  - verwendet(a, B) denotes the number of uses of a in basic block B before potentially being defined therein
  - aktiv(a, B) equals 1, if a has been defined in B and is active at the end of B; else 0
- Approximation of cost savings for assumption that
  - all basic blocks are executed equally often
  - loop will be executed often

Register Binding using Graph Coloring

- Flow
  1. Apply code generation assuming unbounded number of available registers, i.e. each variable is assigned to a unique symbolic register
  2. Determine the life time of each variable
  3. Construct a conflict graph
  4. Mapping of symbolic registers onto physical registers through graph coloring
- Nodes represent the symbolic registers and edges represent conflict between variables
- Heuristics: Is a graph G colorable with I colors?
  1. Determine a node $v_i \in G$ of degree Grad$(v_i) < I$
  2. Eliminate v_i and all incident edges to obtain a graph G'
  3. If $G' = \emptyset$:
     - I-coloring possible

If all nodes in G' have degree $\geq I$:

– I-coloring not possible

$G = G'$ and goto 1

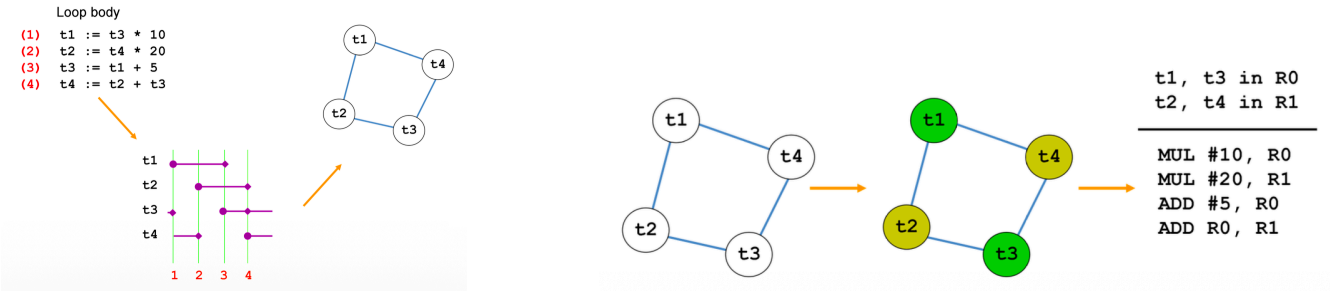


Abbildung 23: Register conflict graph - Example

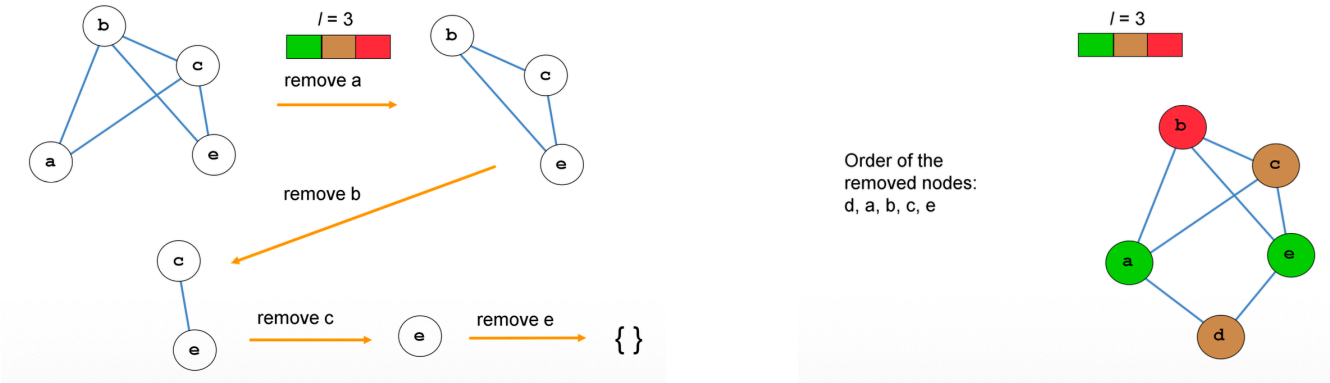


Abbildung 24: Graph coloring - Example

Code Generation for DAGs

- The order of evaluation of the nodes pf a DAG may have a great influence on the number of required instructions
- Heuristic for determination of a feasible evaluation order:

```
while there exists any non-ranked inner node do
    choose a node n whose parents have been ranked and rank it
    while the furthest left child m of n has no unranked parents and is no leaf do
        rank node m
```

```
      n <- m
   end while
end while
```

- For a DAG with n nodes that is a tree there is an algorithm that determines optimal code in time $\mathcal{O}(n)$
- common subexpressions
  - If the DAG is not a tree → split the DAG at nodes that represent common subexpressions and generate optimal code for each part

Dynamic Programming

- Machine model extended to more complex instructions
  - n Register `R0...Rn-1`
  - Instructions `Ri:=E`
    E is an arbitrary expression including registers and memory locations as operants
  - In case E contains multiple registers, then Ri must be one of them
  - Load: `Ri:=M`, Store: `M:=Ri`, Move/Copy: `Ri:=Rj`
- Examples
  - `ADD R0, R1` → `R1:=R1+R0`
  - `ADD *R0, R1` → `R1:=R1 + ind R0`
  - `SUB a, R0` → `R0:=R0-a`
- Optimal code for `E=(T1 op T2)`:
  1. Optimal code for `T1`, `T2`
  2. Either `T1, T2, op` or `T2, T1, op`
- Method runs in 3 phases
  1. Computation of cost vectors
  2. Determination of execution orders
  3. Generation of target code
- Determination of cost vectors for each node n (bottom up)
  - $C[i]$ optimal cost for computing n with i available registers
  - $C[0]$ optimal cost for computing n when result is stored in memory

Code Optimization

- Transformations possible on either intermediate code or on target code
- Peephole Optimization
  - small window is moved over code
  - multiple runs, as one optimization may trigger anotherone
- Local Optimization
  - Transformation on basic blocks
- Global Optimization
  - Transformations affecting multiple basic blocks

Peephole Optimization

- Elimination of redundant assignments

```
(1) MOV R0, a
(2) MOV a, R0
```

↓

```
(1) MOV R0, a
```

- Algebraic simplification
  `x := y + 0 * a;` → `x := y;`
- Control flow optimization

```
(1)     goto L1
(2) L2  goto L2
```

↓

```
(1)     goto L2
(2) L1  goto L2
```

- Operator (strength) reduction
  `x := y*8;` → `x := y << 3;`
  `x := y**2;` → `x := y*y;`
- Common subexpression elimination

```
(1) a := b + c
(2) b := a - d
(3) c := b + c
(4) d := a - d
```

↓

```
(1) a := b + c
(2) b := a - d
(3) c := a
(4) d := b
```

- Variable renaming
  `t := b + c` → `u := b + c`
- Instruction interchange

```
t1 := b + c
t2 := x + y
```

↓

```
t2 := x + y
t1 := b + c
```

Global Optimization

- Passive code elimination
  - An assignment that defines x can be eliminated, if x is not used thereafter
- Copy propagation

```
(1) x := t1
(2) a[t2] := t3
(3) a[t4] := x
(4) goto L
```

↓

```
(1) x := t1
(2) a[t2] := t3
(3) a[t4] := t1
(4) goto L
```

- Code motion

```
while (i <= limit*4+2) {...}
```

↓

```
t = limit*4+2;
while (i <= t)
```

- Induced variables and operators

```
     j := n
(1)  j := j - 1
(2) t4 := 4 * j
(3) t5 := a[t4]
(4) if t5 > v goto (1)
```

↓

```
     j := n
    t4 := 4 * j
(1)  j := j - 1
(2) t4 := t4 - 4
(3) t5 := a[t4]
(4) if t5 > v goto (1)
```

Code generation for special-purpose processors

- Software design for embedded systems
  - transition from assembly to High-Level Language programming
- Major requirements on code generation
  - correctness
  - speed of generation
  - compact code
- Further requirements on HLL and compiler
  - safety: formal verification of compiler
  - specification of real-time constrains
  - support for DSP algorithms/architectures
  - retargetablility: can the compiler be retargeted easily?
- Non-homogeneous register architectures, irregular data paths
  - Tight coupling of the phases of register binding, code selection and scheduling

- Assignment of memory addresses and address registers
  - Efficient usage of address registers and specialized address generation units
- Code compression
  - Reduction of memory footprint, important in cost-sensitive applications

Register Transfer Graph

- Definition: The register transfer graph of a processor is a directed graph, in which each node corresponds to a location in the data path where data may be stored. An edge between two nodes r_i and r_j is labeled with those instructions that read from r_i and write r_j

RTG - Criterion

- Definition: The RTG criterion is satisfied, if for all nodes r_1, r_2 and r_3 of the RTG for which
  1. r_3 has incoming register nodes r_1 and r_2 with the same labeling
  2. There exists at least a cycle between r_1 and r_2

  In any cycle including r_1 and r_2 exists a memory node
  $\rightarrow$ Processors satisfying the RTG criterion, an optimal schedule may be generated in $\mathcal{O}(n)$ (n is the number of DAG nodes)

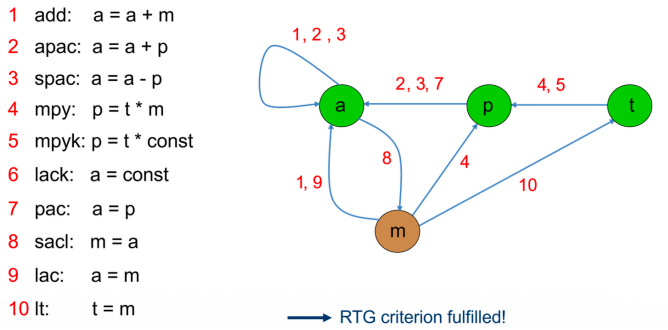


Abbildung 25: RTG for TMS320C25

Retargetable compiler

- Machine-independent compiler (automatically retargetable)
  - Compiler has built-in code generators for multiple targets
  - Often used in parametric architecture
- Compiler-Compiler (user retargetable)

  - Compiler is generated from a description of target architecture
- portable Compiler (developer retargetable)

Processor Models

- Behavioral
  - Describes the instruction set
  - Relatively fast simulation possible
  - Inaccuracy
- Structural
  - Describes the processor at RTL
  - Accurate
  - Simulation much slower
  - Not always available
- Mixed models

Tree-translation schemes

- Rules for transforming a syntax tree (resp. DAG)

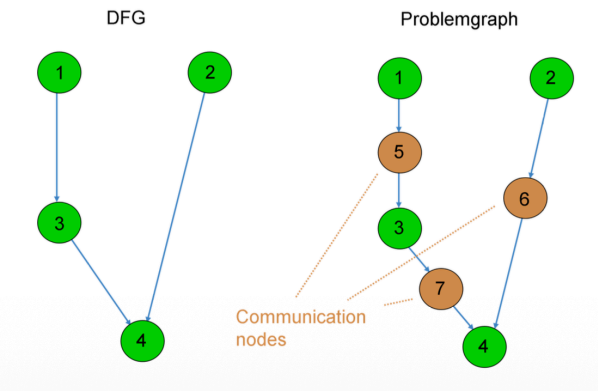
Partitioning

Models for System Synthesis

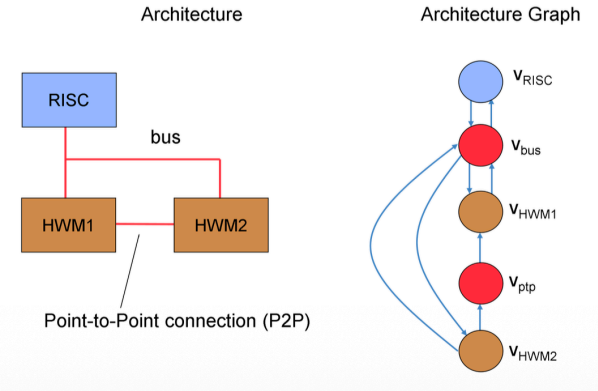
- Allocation + Binding = Partitioning
- Problem graph
  - Nodes: functional and communication task
  - Edges: dependecies
- Architecture graph
  - Nodes: functional and communication resources
  - Edges: directed communication resources
- Specification graph
  - Problem graph + Architecture graph + Mapping edges

Partitioning

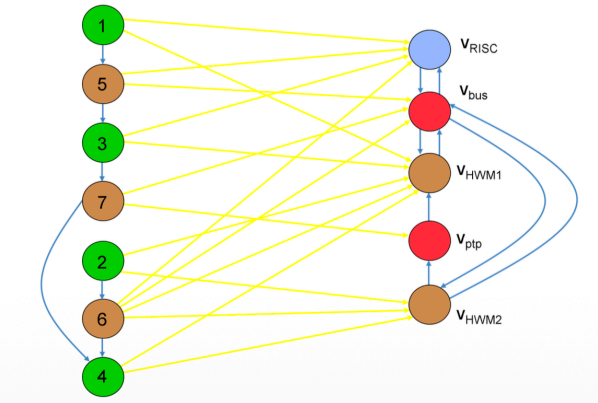
- Abstraction level
  - Structural partitioning: RTL, netlists
    - System relatively well known
    - No more comparison of design alternatives possible
  - Functional partitioning: system level
    - Comparison of design alternatives possible
    - Quality of designs still not accurate $\rightarrow$ Estimation, Rapid Prototyping
- Cost(objective) function - Example:
  - $f(C, L, P) = k_1 \cdot h_c(C, \overline{C}) + k_2 \cdot h_L(L, \overline{L}) + k_3 \cdot h_p(P, \overline{P})$
  - System cost C, Latency L and Power consumption P
- Problem definition: Group n objects $O = \{o_1, \ldots, o_n\}$ into m blocks $P = \{p_1, \ldots, p_m\}$ such that
  - $p_1 \cup \cdots \cup p_m = O$
  - $p_i \cap p_j = \emptyset \quad \forall i, j : i \neq j$
  - while minimizing the cost $c(P)$
  - The general partitioning problem is NP-complete



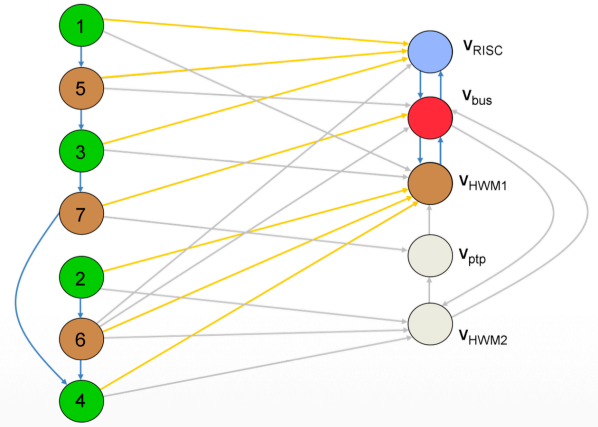
(a) Problem graph



(b) Architecture graph



(c) Specification graph



(d) Allocation and binding

Abbildung 26: Models for System Synthesis

General partitioning methods

- Heuristics
  - Constructive methods
    - Random mapping
    - Hierarchical clustering (!!! Bullshit !!!)
  - Iterative improvement methods
    - Kernighan-Lin algorithm
    - Simulated Annealing
  - Evolutionary algorithms
- Exact techniques
  - Enumeration of solution space
  - Integer Linear Program (ILP)

Constructive Methods

- Random Mapping
  - Each object is mapped randomly to a block
- Hierarchical clustering
  - Stepwise grouping of objects using a closeness function that indicates how beneficial it is to group two objects together
- Constructive methods
  - Are often used to fina an initial partition for methods of iterative improvement
  - Have a problem in defining suitable closeness functions

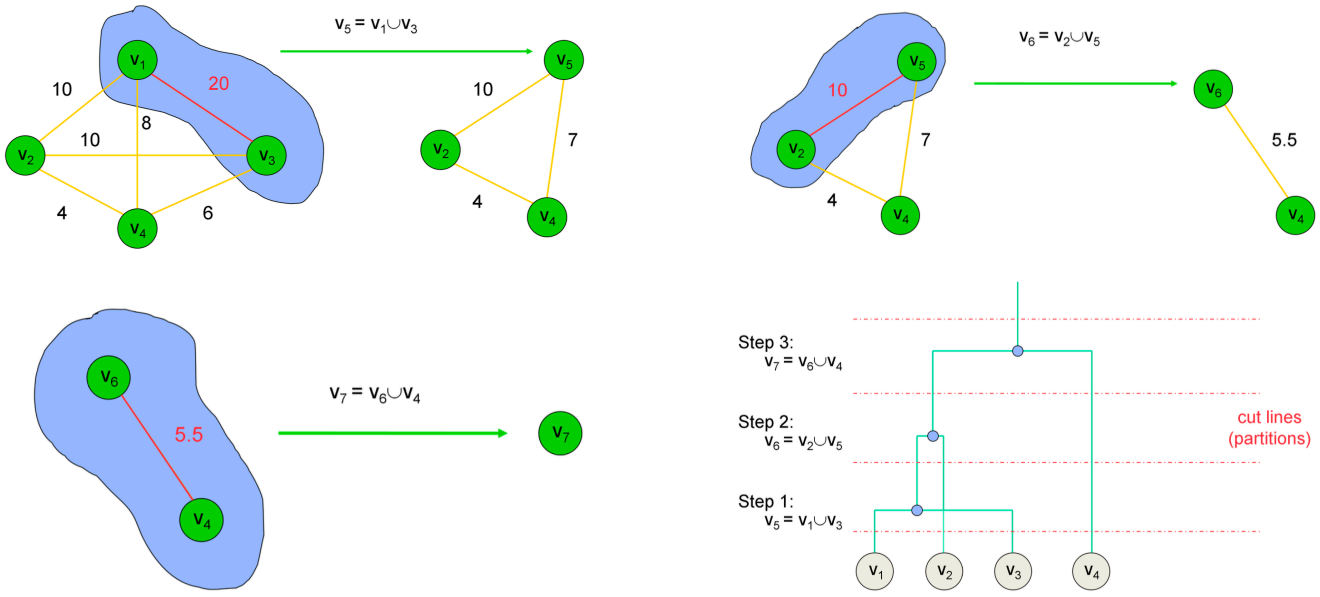


Abbildung 27: Hierarchical Clustering - Example