Hardware-Software-Codesign

Felix Leitl

9. September 2025

Inhaltsverzeichnis

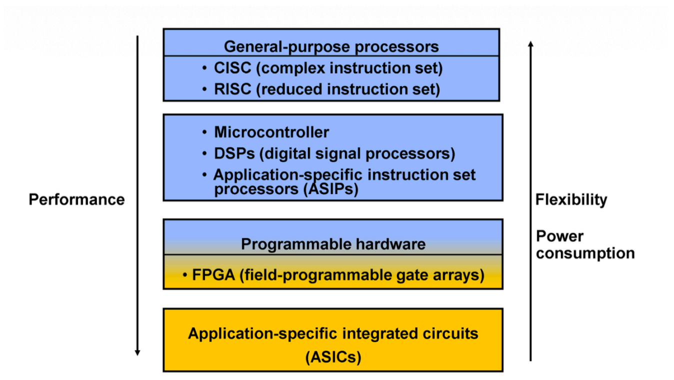

General-purpose processors
• CISC (complex instruction set)
• RISC (reduced instruction set)
• Microcontroller
• DSPs (digital signal processors)
• Application-specific instruction set processors (ASIPs)
Programmable hardware
• FPGA (field-programmable gate arrays)
Application-specific integrated circuits (ASICs)
Performance
Flexibility
Power consumption

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

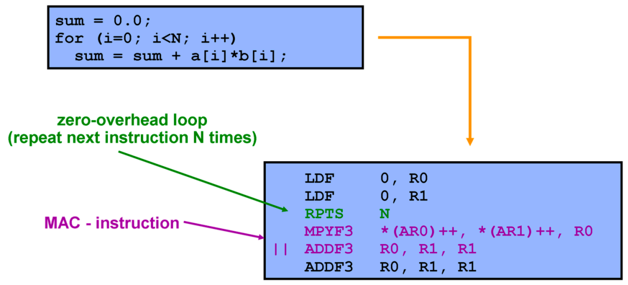
- Multimedia Applications
    - 8 / 16 Bit data types
    - many arithmetic operations
    - huge data sets $\Rightarrow$ Huge I/O-bandwidth
    - high data parallelism
- Sub-word execution
    - split 32/64-bit registers and ALUs into smaller sub-units
    - instructions execute in parallel on these 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)



```
sum = 0.0;
for (i=0; i<N; i++)
  sum = sum + a[i]*b[i];
```

```
     LDF     0, R0
     LDF     0, R1
     RPTS    N
     MPYF3   *(AR0)++, *(AR1)++, R0
||   ADDF3   R0, R1, R1
     ADDF3   R0, R1, R1
```

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 → profiling → 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

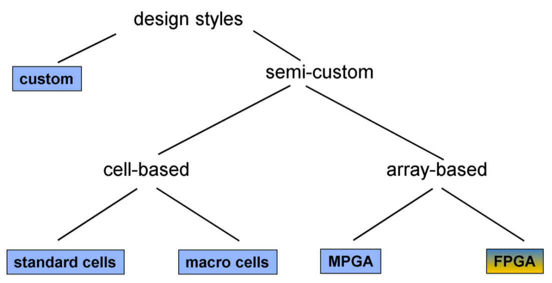
Integrated Circuit

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

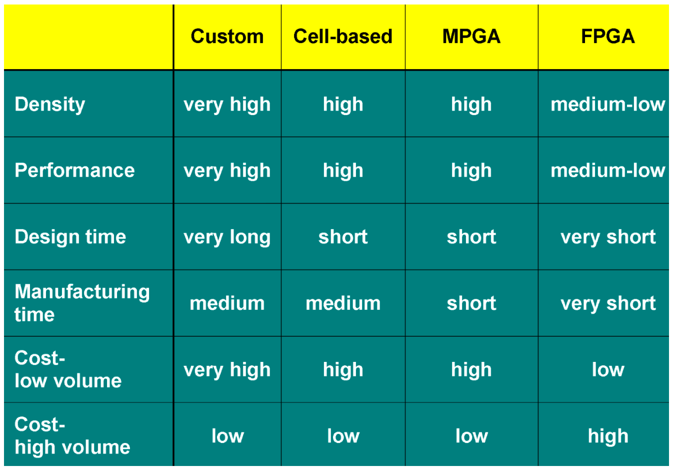
Design Styles for ICs



• MPGA: mask-programmable gate array (beim Hersteller “programmiert”)
• FPGA: field-programmable gate array (beim Anwender “programmiert”)

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

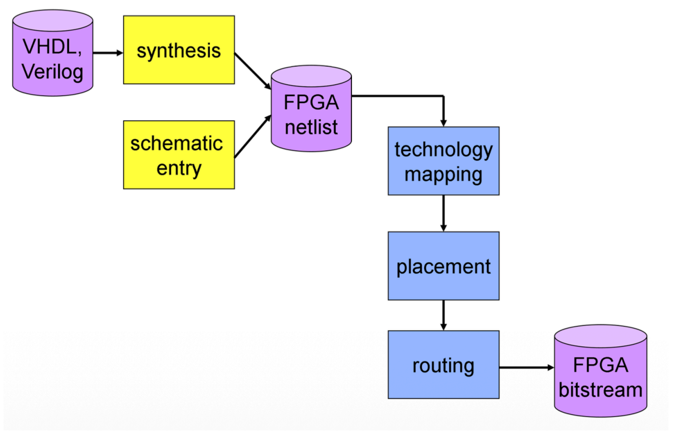


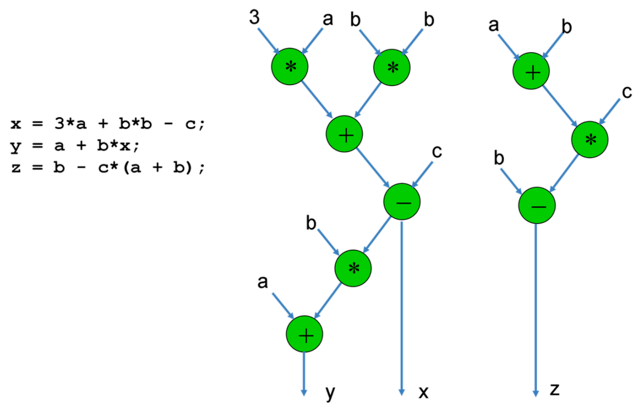
Abbildung 5: FPGA design steps

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

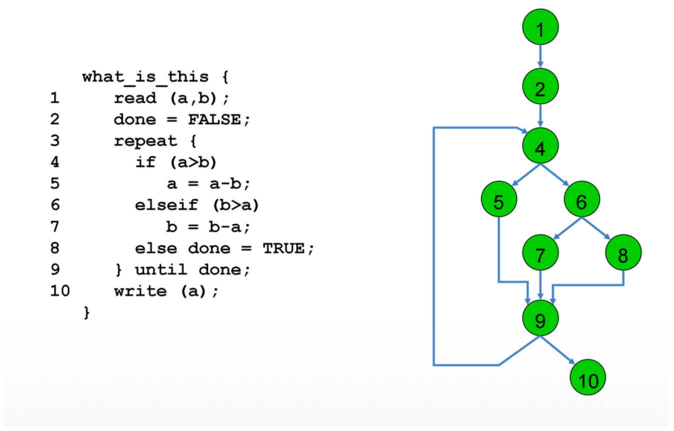
Dataflow Graph (DFG)



```
x = 3*a + b*b - c;
y = a + b*x;
z = b - c*(a + b);
```

Abbildung 6: Data Flow Graph

Control flow graph (CFG)

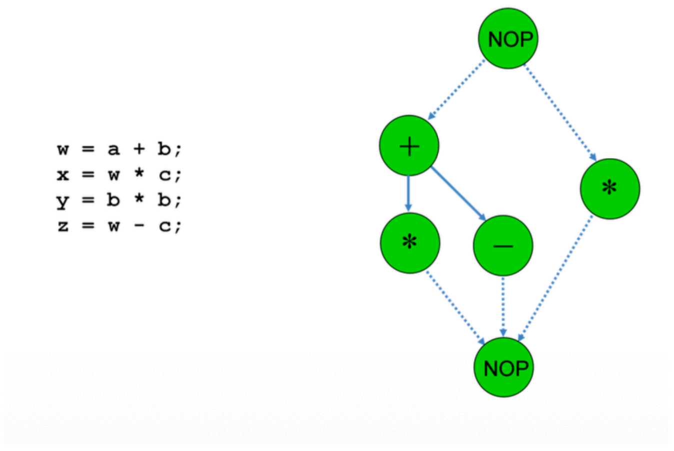


```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```

Abbildung 7: Control Flow Graph

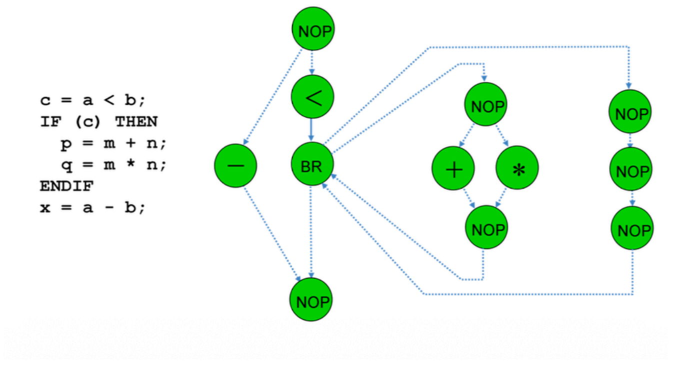
Sequencing Graphs

- 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



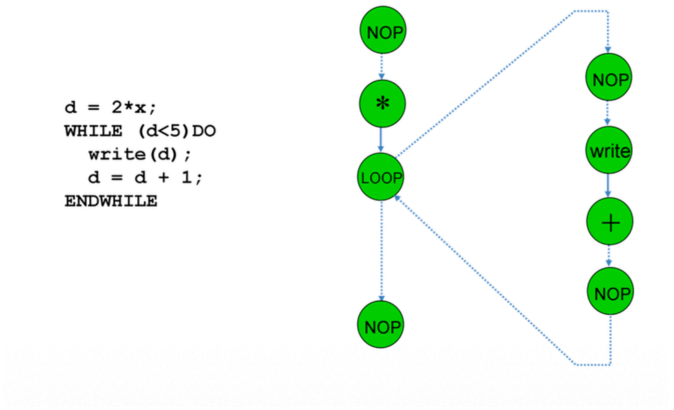
```
w = a + b;
x = w * c;
y = b * b;
z = w - c;
```

Abbildung 8: Sequencing Graph - Entity



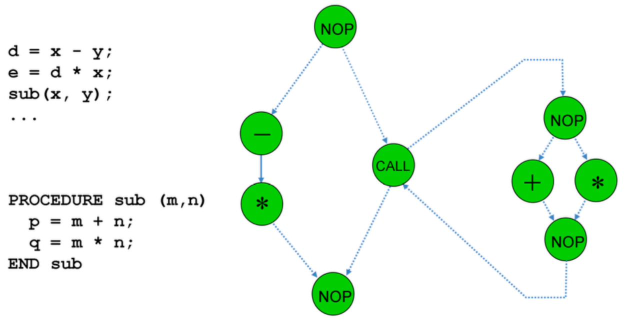
```
c = a < b;
IF (c) THEN
  p = m + n;
  q = m * n;
ENDIF
x = a - b;
```

Abbildung 9: Sequencing Graph - Branch



```
d = 2*x;
WHILE (d<5)DO
  write(d);
  d = d + 1;
ENDWHILE
```

Abbildung 10: Sequencing Graph - Loop



```
d = x - y;
e = d * x;
sub(x, y);
...


PROCEDURE sub (m,n)
  p = m + n;
  q = m * n;
END sub
```

Abbildung 11: Sequencing Graph - Call

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, . . .

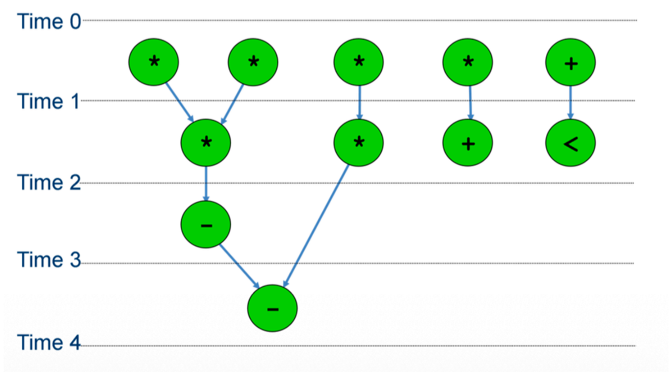
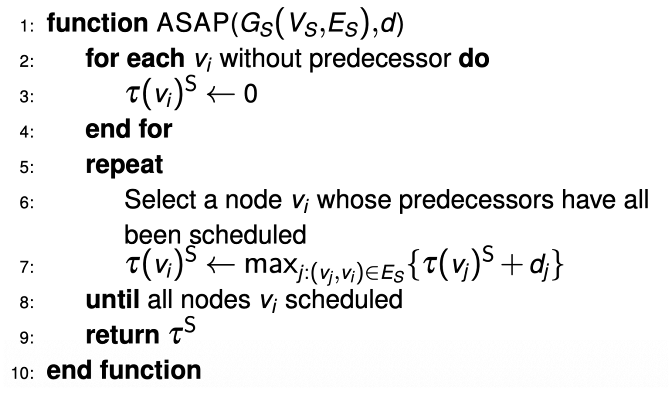


Abbildung 12: latency-optimal schedule: $L = 4$



```
function ASAP($G_S(V_S, E_S), d$)
    for each $v_i$ without predecessor do
        $\tau(v_i)^S \leftarrow 0$
    end for
    repeat
        Select a node $v_i$ whose predecessors have all
        been scheduled
        $\tau(v_i)^S \leftarrow \max_{j:(v_j,v_i)\in E_S}\{\tau(v_j)^S + d_j\}$
    until all nodes $v_i$ scheduled
    return $\tau^S$
end function
```

Abbildung 13: ASAP algorithm

Periodic scheduling

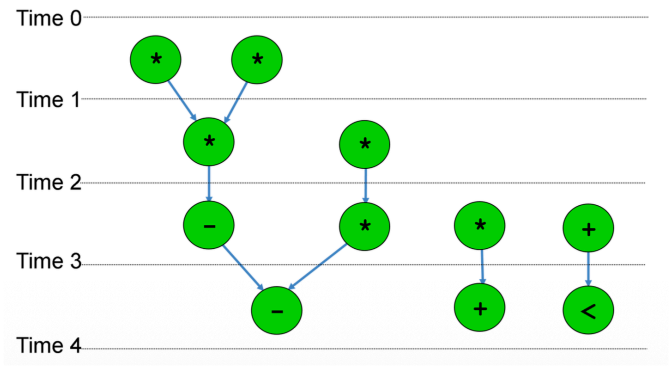


Abbildung 14: latency bound: $\overline{L} = 4$

```
function ALAP($G_S(V_S, E_S), d, \overline{L}$)
    for each $v_i$ without successor do
        $\tau(v_i)^L \leftarrow \overline{L} - d_i$
    end for
    repeat
        Select a node $v_i$ whose successors have all
        been scheduled
        $\tau(v_i)^L \leftarrow \min_{j:(v_i,v_j) \in E_S}\{\tau(v_j)^L\} - d_i$
    until all nodes $v_i$ scheduled
    return $\tau^L$
end function
```

Abbildung 15: ALAP algorithm

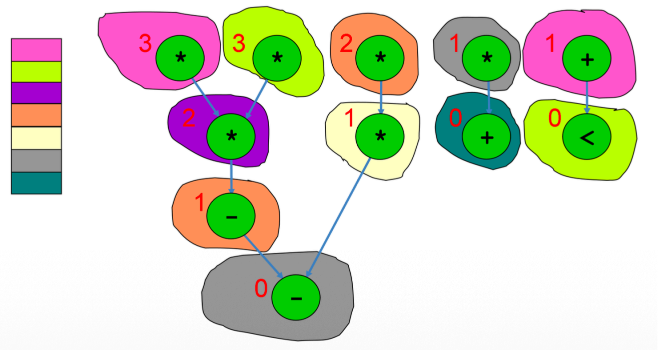


Abbildung 16: Priority: Number of successor nodes
Resources: 1 multiplier, 1 ALU (+, -, <)

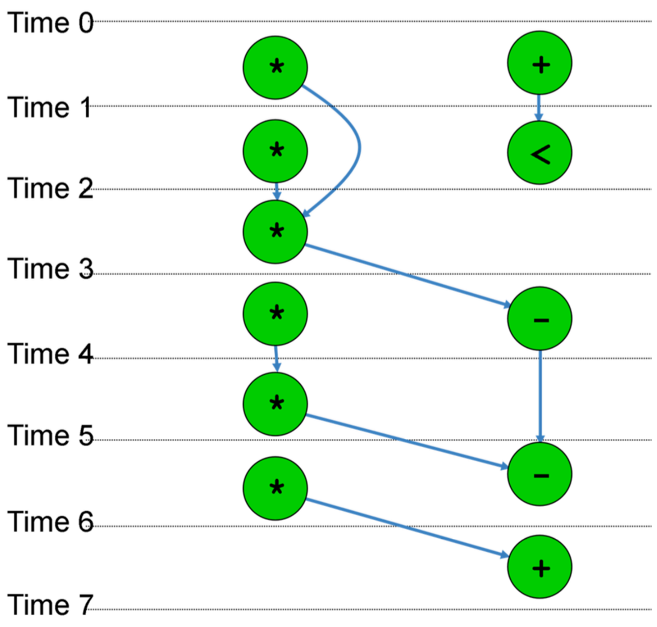


Abbildung 17: List scheduling

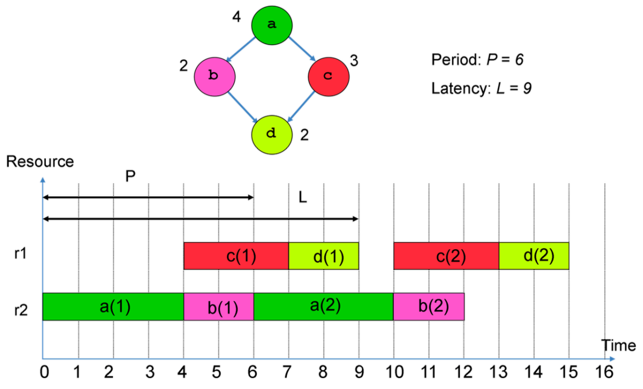


Abbildung 18: Periodic scheduling