Embedded Systems
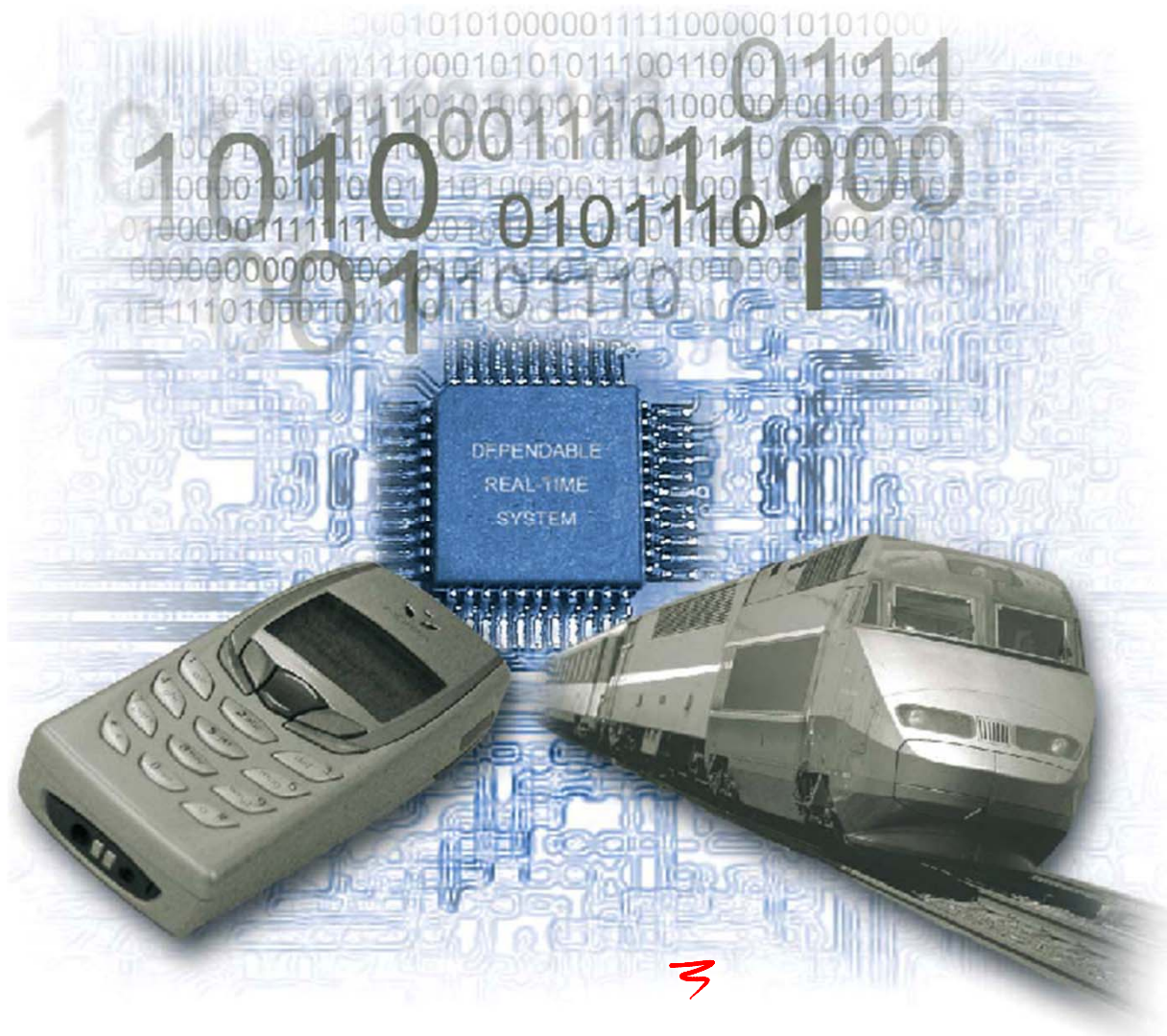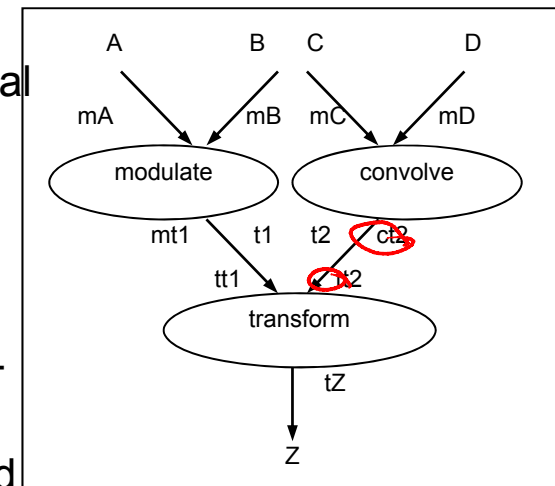
Synchronous dataflow

- Multiple tokens consumed and produced per firing

- Synchronous dataflow model takes advantage of this
  - Each edge labeled with number of tokens consumed/produced each firing
  - Can statically schedule nodes, so can easily use sequential program model
    - Don't need real-time operating system and its overhead

- Algorithms developed for scheduling nodes into "single-appearance" schedules
  - Only one statement needed to call each node's associated procedure
    - Allows procedure inlining without code explosion, thus reducing overhead even more

Synchronous dataflow

CS - ES

- 2

2

Synchronous DataFlow

SDF firing rules:

- **Actor enabling** = each incoming arc carries at least *weight* tokens
- **Actor execution** = atomic consumption/production of tokens by an enabled actor
  - i.e., **consume weight tokens on each incoming arcs and produce weight tokens on each outgoing arc**
- **Delay** is an initial token load on an arc.
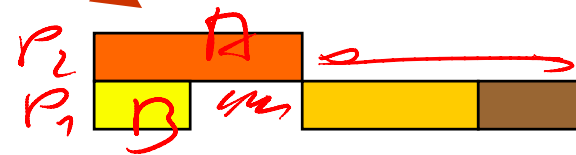
Parallel Scheduling of SDF Models

SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.
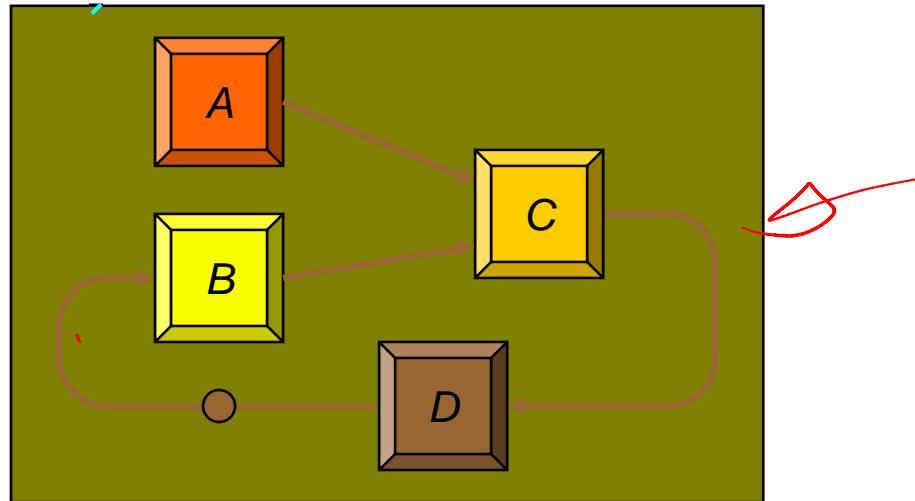
Sequential
periodic admissible sequential schedule (PASS)

Parallel
periodic admissible parallel schedule (PAPS)

(admissible = correct schedule, finite amount of memory required)

Delays

- Kahn processes often have an initialization phase

- SDF doesn't allow this because rates are not always constant

- Alternative: an SDF system may start with tokens in its buffers

- These behave like delays (signal-processing)

- Delays are sometimes necessary to avoid deadlock

SDF Scheduling

- Schedule can be determined completely before the system runs

- Two steps:

1. Establish relative execution rates by solving a system of linear equations

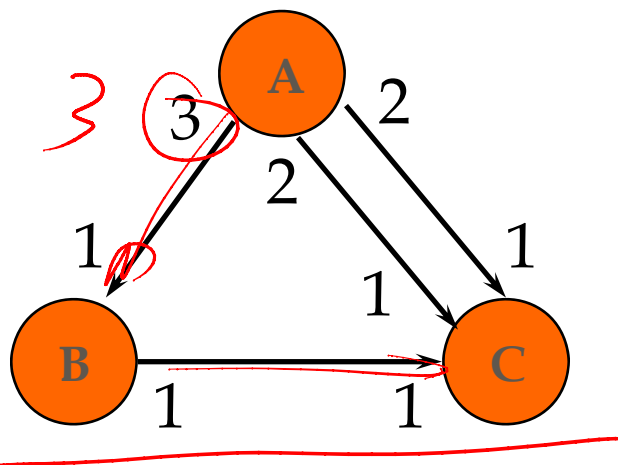2. Determine periodic schedule by simulating system for a single round

Balance equations

- Number of produced tokens must equal number of consumed tokens on every edge
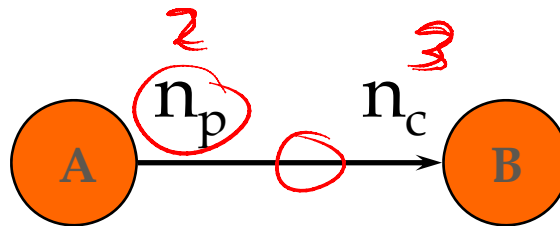
- Repetitions (or firing) vector v_S of schedule S: number of firings of each actor in S
- $v_S(A) \, n_p = v_S(B) \, n_c$
  must be satisfied for each edge

Balance equations

topology matrix

$$M = \begin{vmatrix} 3 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{vmatrix}$$

the (c, r)th entry in the matrix is the amount of data produced by node c on arc r each time it is involved
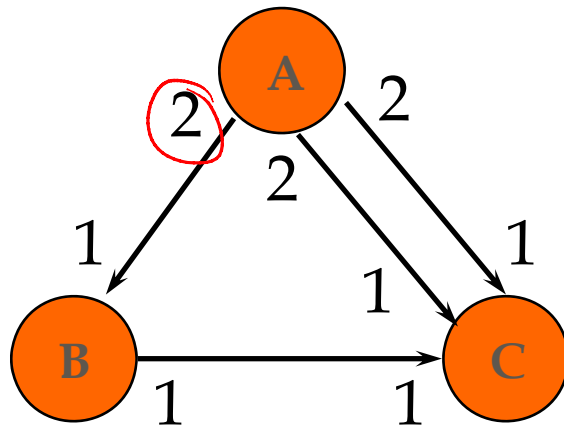
- $M\, v_S = 0$

  iff S is periodic

- Full rank (as in this case)

  - no non-zero solution
  - no periodic schedule

  (too many tokens accumulate on A->B or B->C)

CS - ES

- 8 -

Balance equations

$$M = \begin{vmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{vmatrix}$$
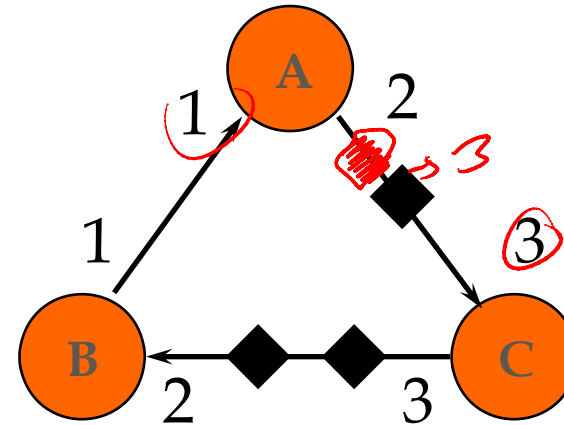
- **Non-full rank**
  - infinite solutions exist
- **Any multiple of $v_S = |1 \quad 2 \quad 2|^T$ satisfies the balance equations**
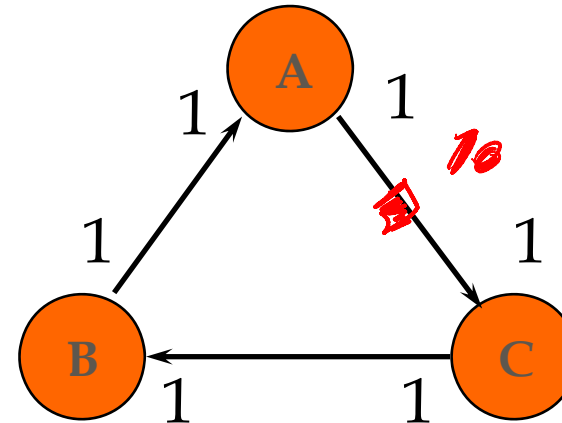- **ABCBC and ABBCC are minimal valid schedules**

Admissibility of schedules

- No admissible schedule:
  BACBA, then deadlock…
- Adding one token on A->C makes
  BACBACBA valid
- Making a periodic schedule admissible is always possible, but changes specification...

Admissibility of schedules

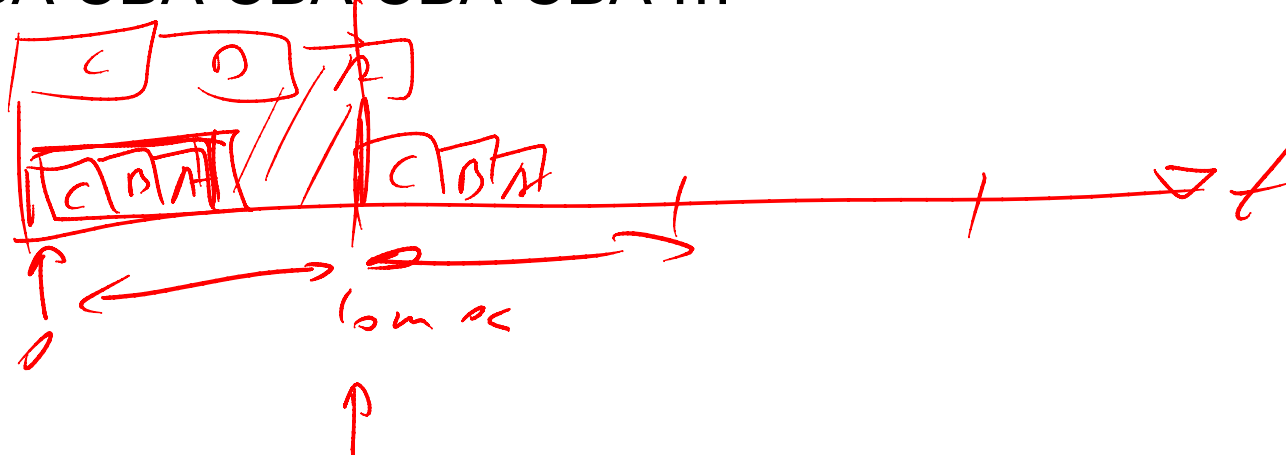- No admissible schedule:

  Adding one token on
  e.g., A->C makes

  CBA valid

  CBA CBA CBA CBA CBA …

SDF Compiler

Task for an SDF compiler:

- Allocation of memory for the passing of data between nodes
- Scheduling of nodes onto processors in such a way that data is available for a block when it is invoked

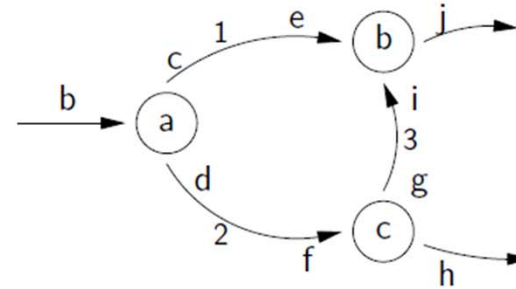Assumptions on the SDF graph:

- The SDF graph is nonterminating and does not deadlock
- The SDF graph is connected

Goal:

- Development of a periodic admissible parallel schedule (PAPS)
- or a periodic admissible sequential schedule (PASS)

(admissible = correct schedule, finite amount of memory required)

Does a PASS exist?
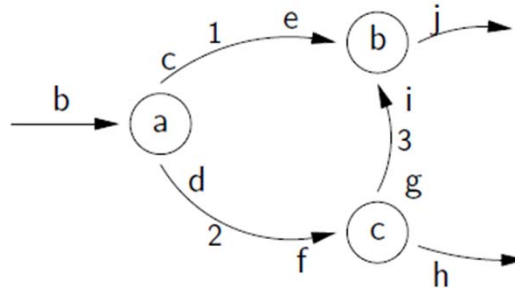
- The SDF graph is described by the topology matrix

$$M = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & -i & g \end{bmatrix}$$

- The entry of row r and column c is the number of tokens produced (positive number) or consumed (negative number) by node c on arc r .

- Connections to the outside world are not considered.

CS - ES

- 13 -

Does a PASS exist?

- A PSS (periodic sequential schedule) exists if rank (M) = s -1, where s is the number if nodes in a graph.

- There is a **v** such that **Mv = O** where **O** is a vector full of zeros. v describes the number of firings in each scheduling period.

A PASS exists
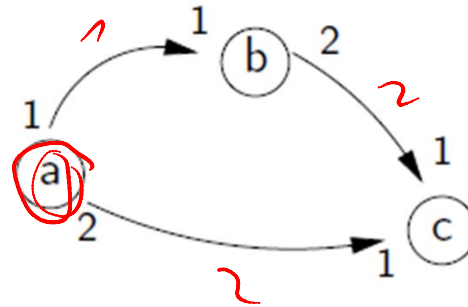
- The rank of the matrix $M =$ $\begin{bmatrix} & & \\ & & \end{bmatrix}$ is $s - 1 = 2$ and $v =$ $\begin{bmatrix} \\ \end{bmatrix}$

- A valid schedule is $\Phi = \{a, b, c, c\}$, but not $\Phi = \{b, a, c, c\}$

- The maximum buffer sizes for the arcs are $b = < 1, 2, 2 >$

A PASS does not exist
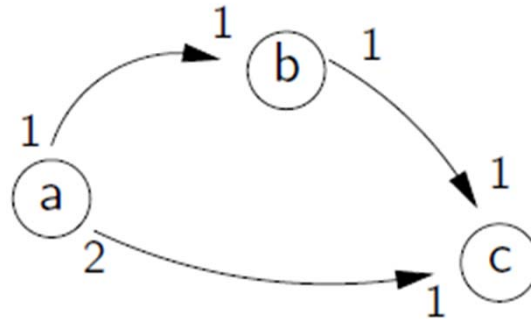
- The graph has sample rate inconsistencies.

- A schedule for the graph will result in unbounded buffer sizes.

- No PASS can be found (rank (M) = s = 3).
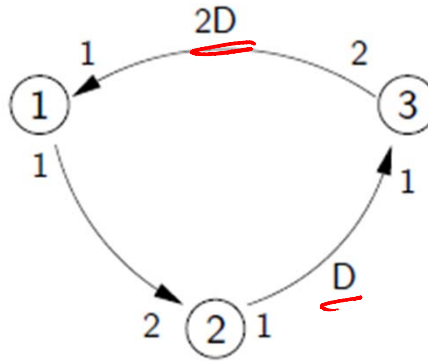
- Assumption: Block 1 : 1 time unit
  Block 2 : 2 time units
  Block 3 : 3 time units
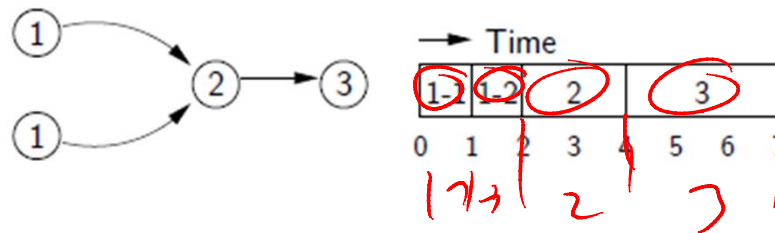
Trivial Case - All computations are scheduled on same processor
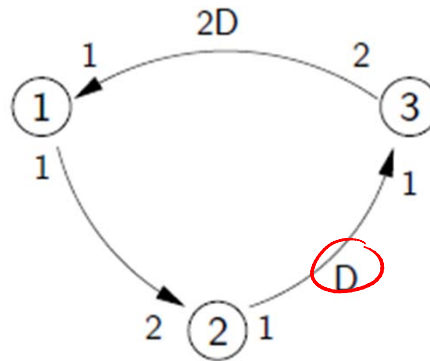
- The performance can be improved, if a schedule is constructed that exploits the potential parallelism in the SDF-graph. Here the schedule covers one single period.
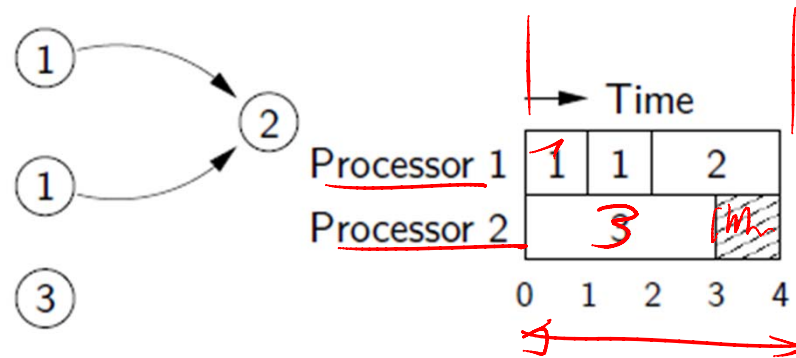
Single Period Schedule

PAPS
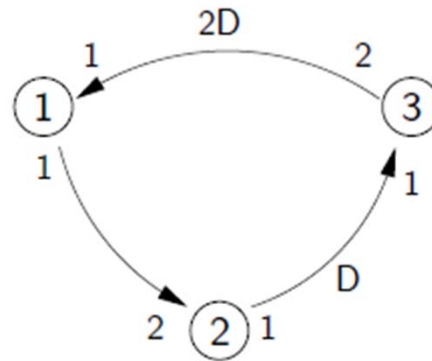
- The performance can be further improved, if the schedule is constructed over two periods.
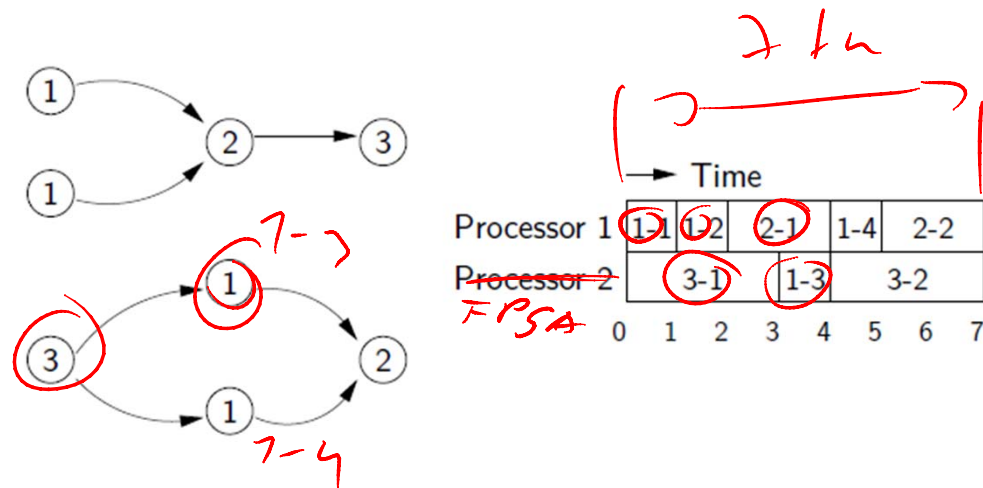
Double Period Schedule

CS - ES

Limitations of the Model

- Asynchronous graphs do exist in digital signal processing. A solution is to divide the graph into synchronous subgraphs and to schedule these graphs.

- The SDF model does not reflect the real-time nature of the connections to the real time world.

- In the construction of a PAPS the run-time of each node is assumed to be data-independent, which may not be the case. In hard real-time systems scheduling must also perform with worst case data, which thus can be taken as time for the scheduling of a node.

Scheduling Choices

- SDF Scheduling Theorem guarantees a schedule will be found if it exists

- Systems often have many possible schedules

- How can we use this flexibility?
  - Reduced code size
  - Reduced buffer sizes

SDF Code Generation

- Often done with prewritten blocks

- For traditional DSP, handwritten implementation of large functions (e.g., FFT)

- One copy of each block's code made for each appearance in the schedule

Code Generation

- In this simple-minded approach, the schedule

  BBBCDDDDAA

would produce code like

```
B;
B;
B;
C;
D;
D;
D;
D;
A;
A;
```

Looped Code Generation

- Obvious improvement: use loops

- Rewrite the schedule in "looped" form:

$$(3\ B)\ C\ (4\ D)\ (2\ A)$$

- Generated code becomes

        for (i = 0 ; i < 3; i++) B;
        C;
        for (i = 0 ; i < 4 ; i++) D;
        for (i = 0 ; i < 2 ; i++) A;
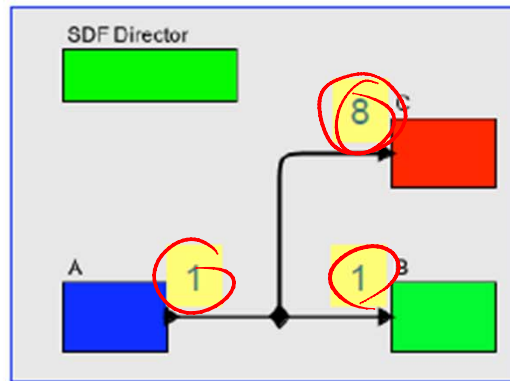
Suppose that C requires 8 data values from A to execute. Suppose further that C takes much longer to execute than A or B. Then a schedule might look like this:

CS - ES
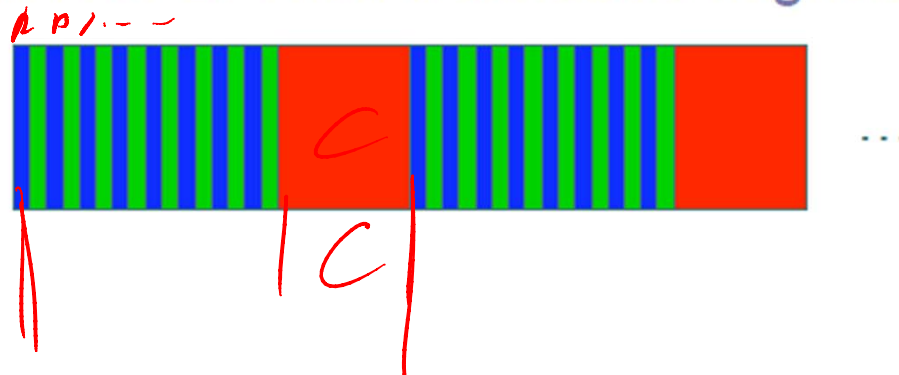
Conclusion SDF

- The SDF model is very useful for regular DSP applications

- Used for: simulation, scheduling, memory allocation, code generation for Digital Signal Processors (HW and SW)

- There is a mathematical framework to calculate a PASS or a PAPS and to determine the maximum size of buffers, if a PASS/PAPS exists

- The work on SDF can be used to derive single and multiple processor implementations

Summary dataflow

- Communication exclusively through FIFOs

- Kahn process networks
  - blocking read, nonblocking write
  - deterministic
  - schedulability undecidable
  - Parks' scheduling algorithm

- SDF
  - fixed token consumption/production
  - compile-time scheduling (no OS): balance equations

Selected Models of computation

Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets	C/E nets, P/T nets, …		
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative (Von Neumann) model	C, C++, Java	C, C++, Java with libraries CSP, ADA	

* Classification based on the **implementation** of HDLs

Imperative (von-Neumann) model

- The von-Neumann model reflects the principles of operation of standard computers:

    - Sequential execution of instructions (sequential control flow, fixed sequence of operations)
    - Possible branches
    - Partitioning of applications into threads
    - In most cases:
        - Context switching between threads, frequently based on pre-emption
        - Access to shared memory

From implementation concepts to programming models

- Example languages
  - Machine languages (binary)
  - Assembly languages (mnemonics)
  - Imperative languages providing a limited abstraction of machine languages (C, C++, Java, ...)

- Threads/processes
  - Initially available only as entities managed by the operating system
  - Made available to the programmer as well
  - Languages initially not designed for communication, availability of threads made synchronization and communication a must.

Models vs. languages

- How can we (precisely) capture behavior?
  - We may think of languages (C, C++), but *computation model* is the key

Models	Poetry	Recipe	Story		State machine	Sequent. program	Data-flow
Languages	English	Spanish	Japanese		C	C++	Java

Recipes vs. English   **Sequential programs vs. C**
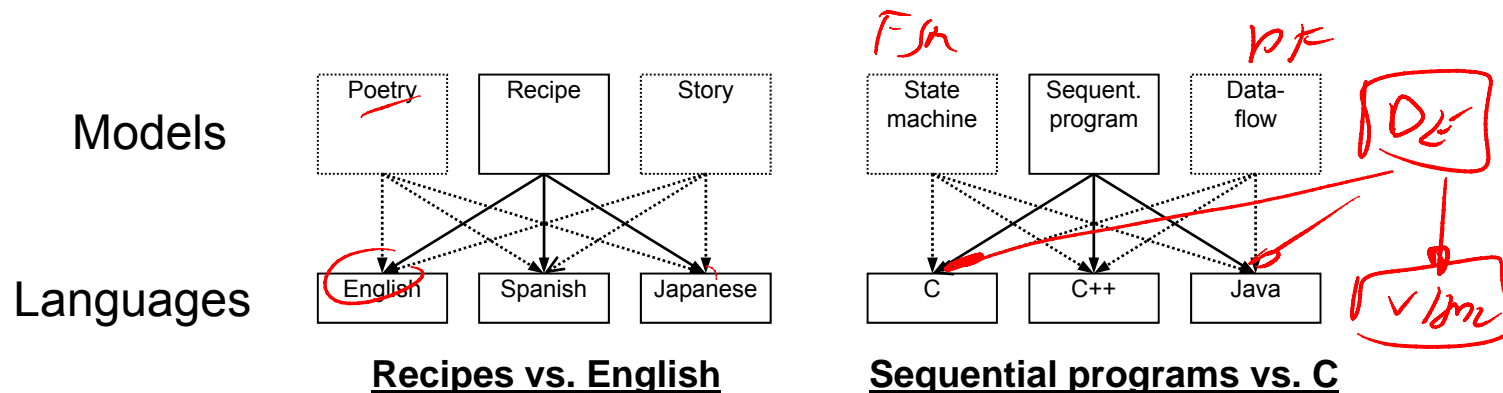
- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program

- Languages capture models
  - Concrete form, e.g., English, C

Models vs. languages

Models

Poetry	Recipe	Story

Languages

English	Spanish	Japanese

Recipes vs. English

State machine	Sequent. program	Data-flow

C	C++	Java

Sequential programs vs. C
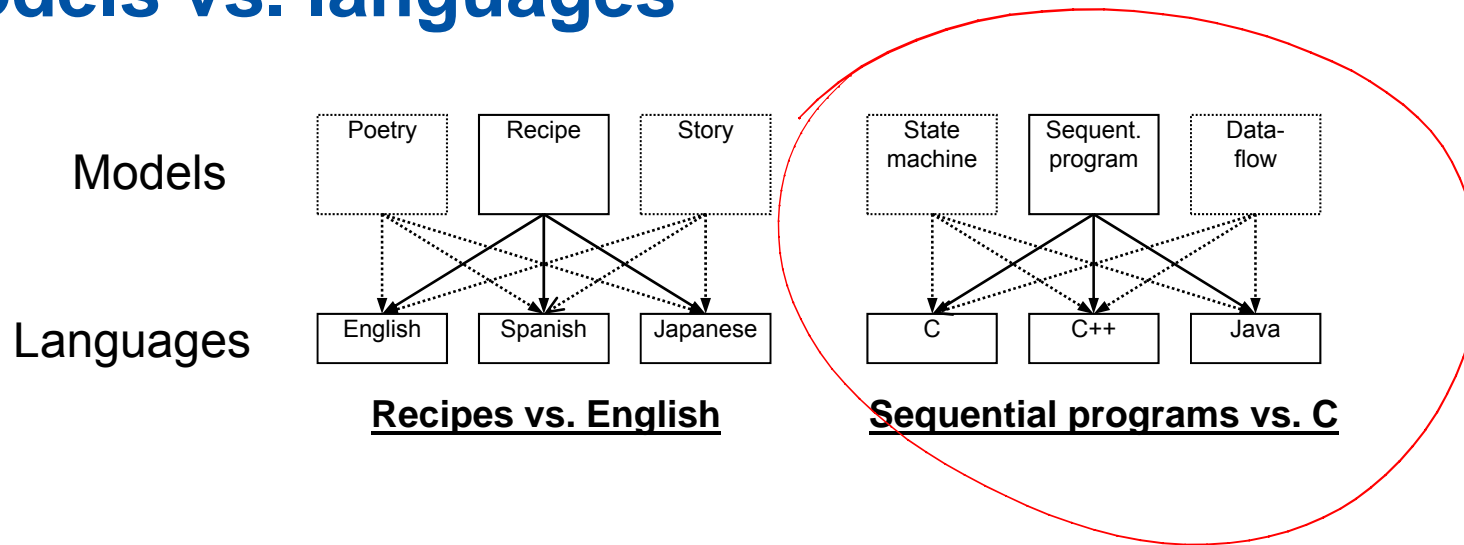
- Variety of languages can capture one model
  - E.g., sequential program model → C, C++, Java

- One language can capture variety of models
  - E.g., C++ → sequential program model, object-oriented model, state machine model

- Certain languages better at capturing certain computation models

(Other) Languages and Models

- **UML (Unified Modelling Language) [Rational 1997]**
"systematic" approach to support the first phases of the
design process

  - UML 1.xx not designed for embedded systems
  UML 2.xx supports real-time applications

  - several diagram types included
  9 (UML 1.4)
  13 (UML 2.0)
  **in particular variants of
  StateCharts, MSCs, Petri Nets (called acticity diagrams)**

Java

Java 2 Micro Edition (J2ME)

CardJava

Real-time specification for Java (JSR-1), see //www.rtj.org

Verilog

- HW description language competing with VHDL
- More popular in the US (VHDL common in Europe)

Many other languages

- **Pearl:** Designed in Germany for process control applications. Dating back to the 70s. Popular in Europe.

- **Chill**: Designed for telephone exchange stations. Based on PASCAL.

- **IEC 60848, STEP 7**: Process control languages using graphical elements

Introduction - Assembly
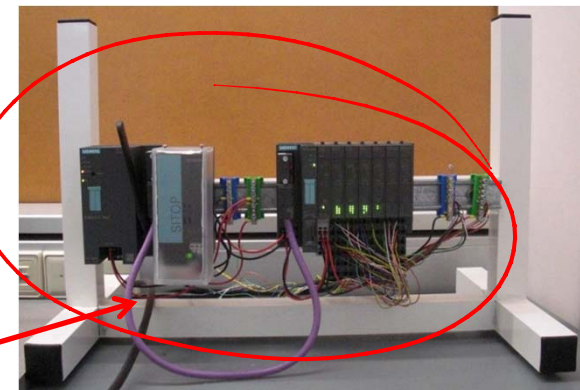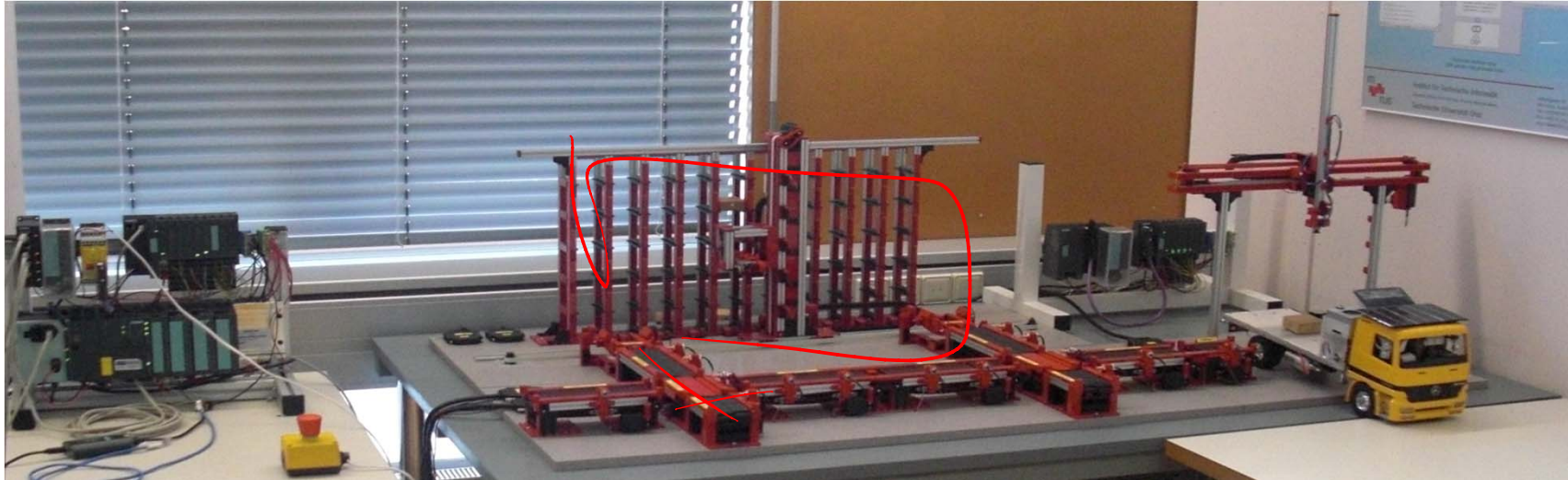
IEC 60848, STEP 7:

Process control languages using graphical elements

CS - ES

Architecture Design – Models

Specification

- Formal specification of the desired functionality and the structure (architecture) of an embedded systems is a necessary step for using computer aided design methods.

- There exist many different formalisms and models of computation

- *Now*:

    Relevant models for the architecture level (hardware synthesis).

# Task graphs or dependency graph (DG)

Sequence
constraint
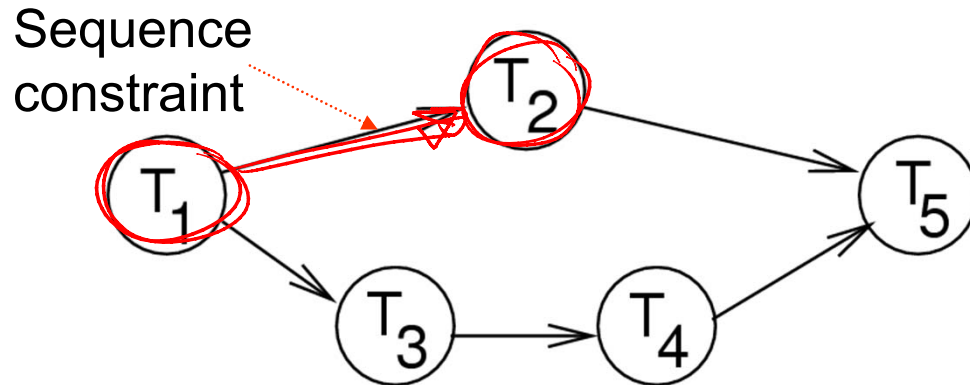
Nodes are assumed to be a „program" described in some programming language, e.g. C or Java.

- **Def.:** A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.

- If $(v1, v2) \in E$, then $v1$ is called an **immediate predecessor** of $v2$ and $v2$ is called an **immediate successor** of $v1$.

Dependence Graph (DG)

- A dependence graph describes **order relations** for the execution of single operations or tasks. **Nodes** correspond to **tasks or operations, edges** correspond to **relations** („executed after").

- Usually, a dependence graph describes a **partial ordering** between operations and therefore, leaves freedom for scheduling (parallel or sequential). It represents **parallelism** in a program **but no branches** in control flow.
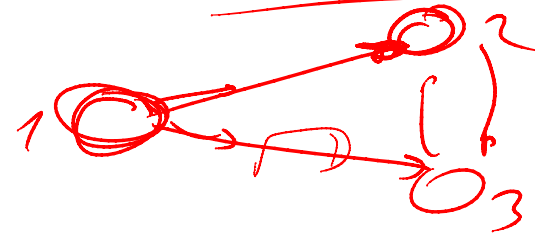
- A dependence graph is acyclic.

- Often, there are additional quantities associated to edges or nodes such as
    - **execution times, deadlines, arrival times**
    - **communication demand**
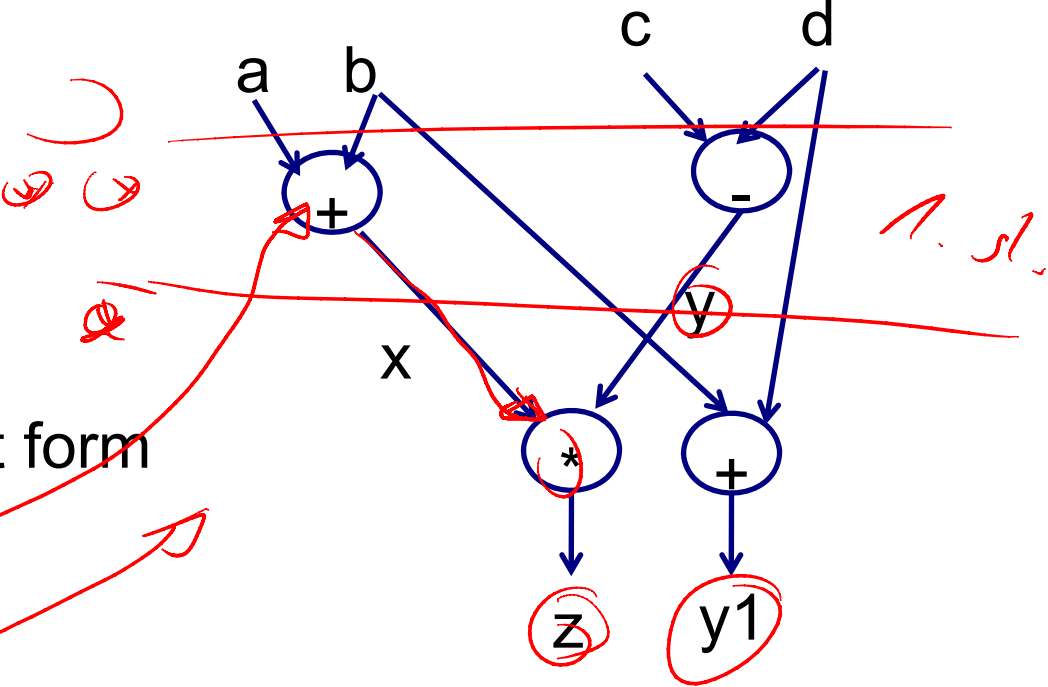
Single Assignment Form

Basic block

x = a + b;
y = c - d;
z = x * y;
y = b + d;

Single assignment form

x = a + b;
y = c - d;
z = x * y;
y1 = b + d;

dependence graph

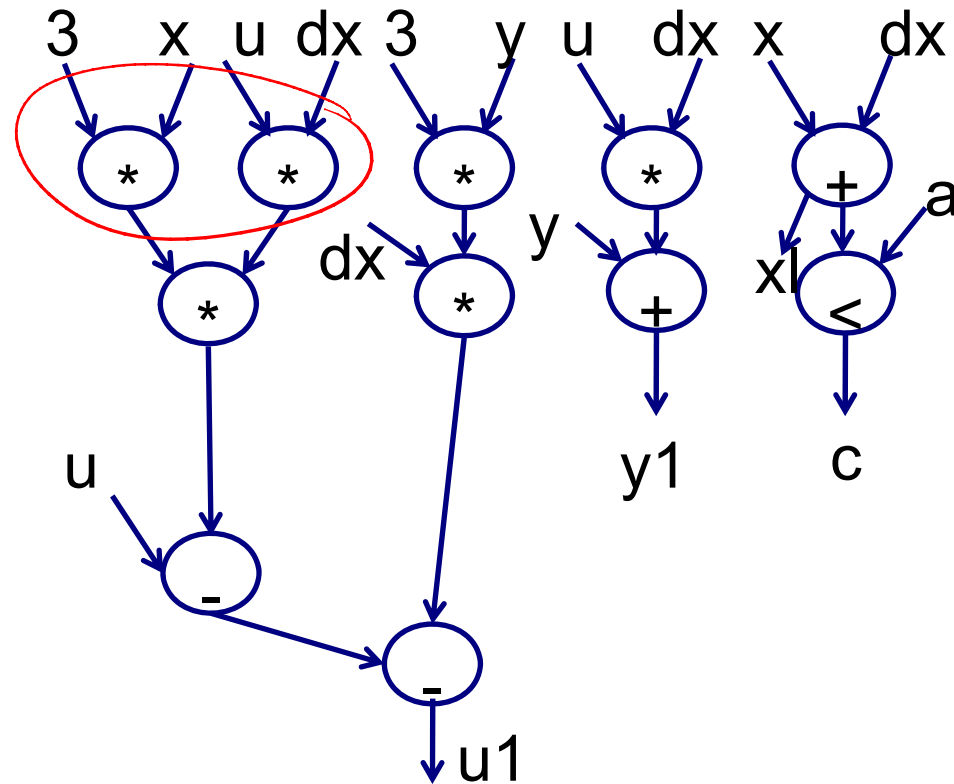sequential program → optimized hardware

Dependence graph (DG)

```
x1 = x + dx;
u1 = u - (3*x*u*dx) - (3*y*dx);
y1 = y + u*dx;
c = x1 < a;
```

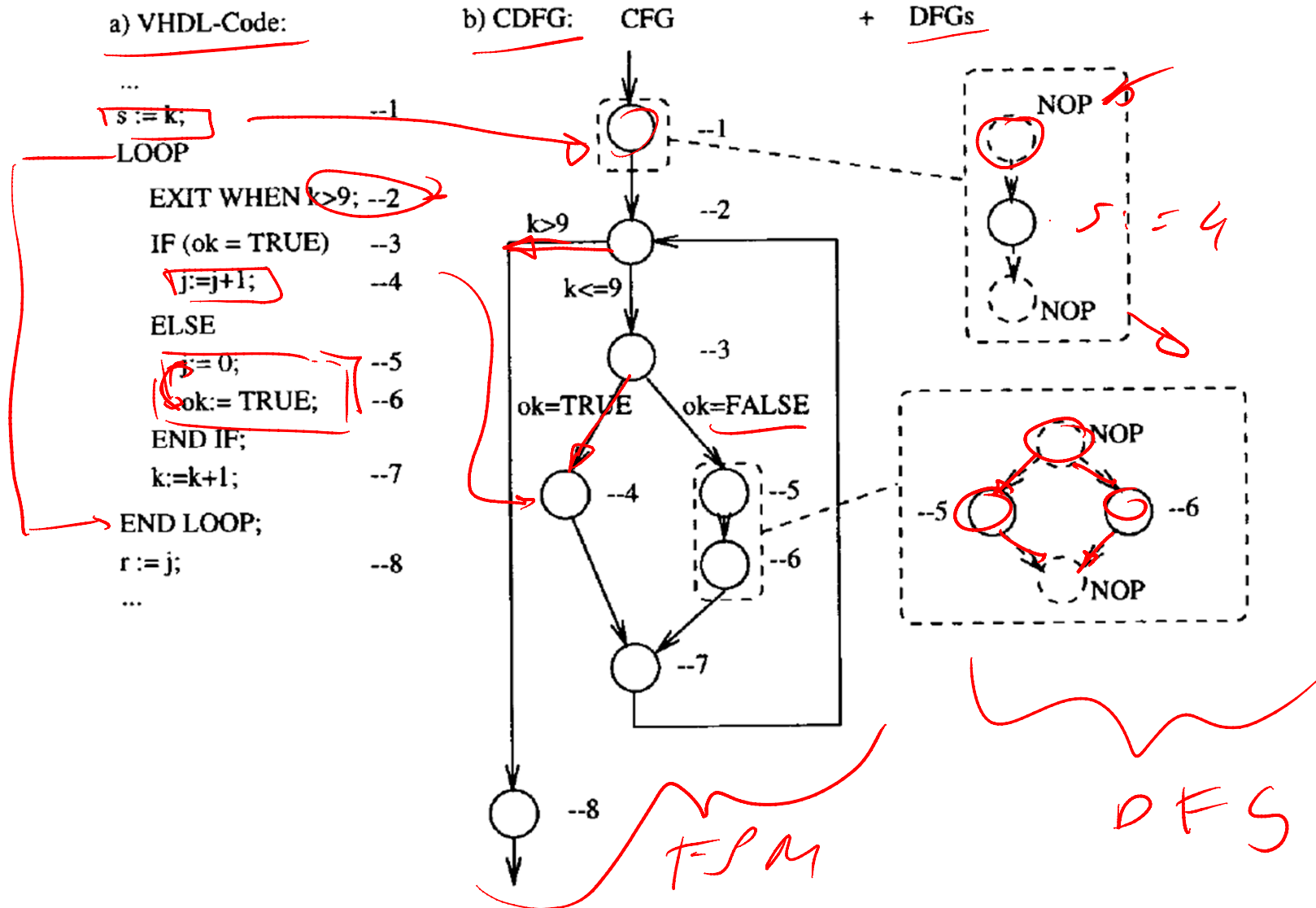# Control-Data Flow Graph (CDFG)

- Goal:
    - Description of control structures (for example branches) and data dependencies.

- Applications:
    - Describing the semantics of programming languages.
    - Internal **representation in compilers** for hardware and software.

- Representation:
    - Combination of control flow (sequential state machine) and dependence representation.
    - Many variants exist.

CDFG

a) VHDL-Code:

```
...
s := k;                         --1
LOOP
    EXIT WHEN k>9;              --2
    IF (ok = TRUE)             --3
        j:=j+1;                 --4
    ELSE
        := 0;                   --5
        ok:= TRUE;              --6
    END IF;
    k:=k+1;                     --7
END LOOP;
r := j;                         --8
...
```

b) CDFG: CFG   +   DFGs

k>9

k<=9

ok=TRUE   ok=FALSE

--1
--2
--3
--4   --5
--6
--7
--8

NOP

s = k

NOP

NOP

--5   --6

NOP

TSM

DFS

Control-Data Flow Graph (CDFG)

- Control Flow Graph:
  - It corresponds to a **finite state machine**, which represents the sequential control flow in a program.

  - **Branch conditions** are very often associated to the outgoing edges of a node.

  - The operations to be executed within a state (node) are associated in form of a **dependence graph.**

- Dependence Graph (also called Data Flow Graph DFG):

  - NOP (no operation) operations represent the start point and end point of the execution. This form of a graph is called a **polar graph**: it contains two distinguished nodes, one without incoming edges, the other one without outgoing edges.

Sequence Graph (SG)

- A **sequence graph** is a **hierarchy** of directed graphs. A generic element of the graph is a **dependence graph** with the following properties:

    - It contains **two kinds** of nodes: (a) **operations or tasks** and (b) **hierarchy nodes**.

    - Each graph is **acyclic and polar** with two distinguished nodes: the start node and the end node. No operation is assigned to them (NOP).

    - There are the following **hierarchy nodes**: (a) module call (CALL) (b) branch (BR) and (c) iteration (LOOP).
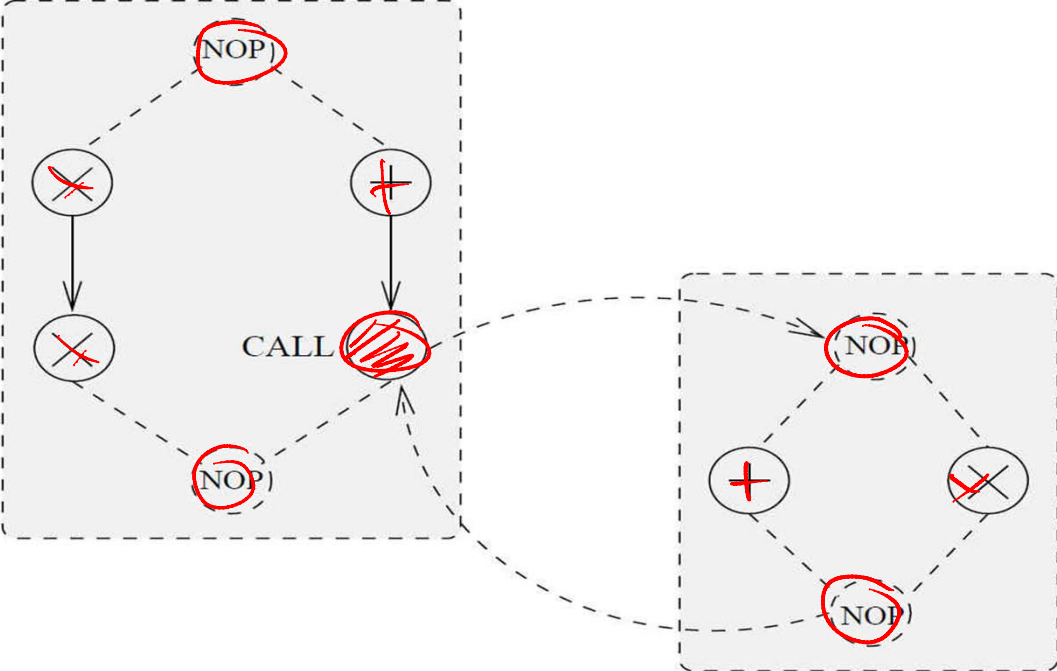
CS - ES

Sequence Graph (SG)

Example - CALL:

```
x := a * b;
y := x * c;
z := a + b;
submodul(a, z);
```

```
PROCDEDURE submodul(m, n) IS
p := m + n;
q := m * n;
END submodul
```

Sequence Graph (SG)
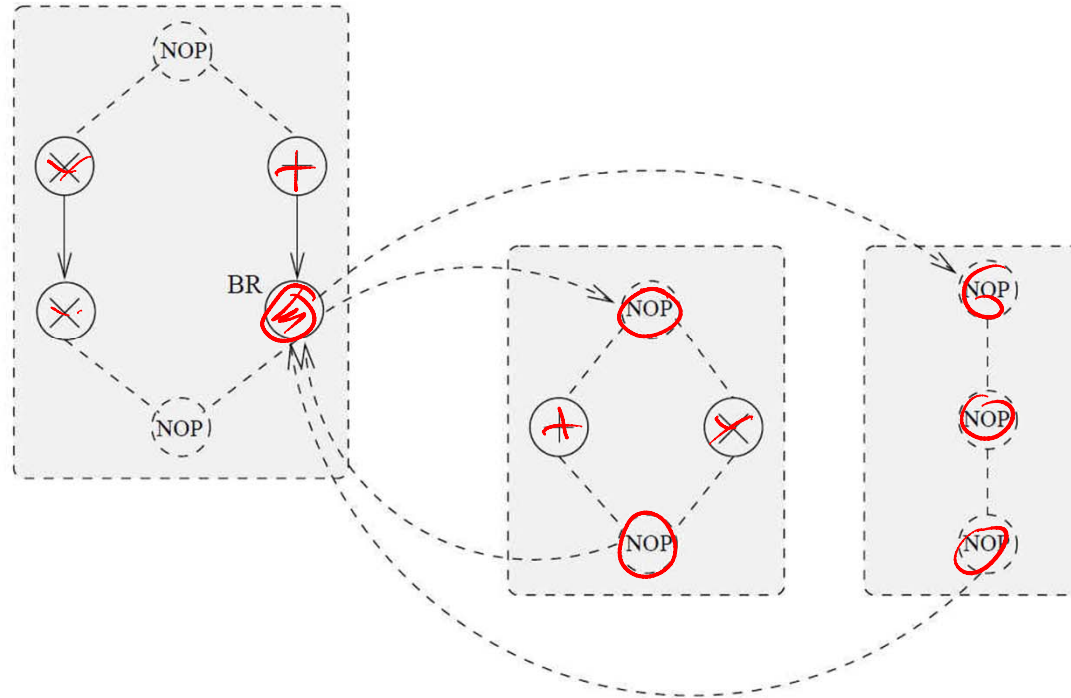
Example - BR:

x := a * b;

y := x * c;

z := a + b;

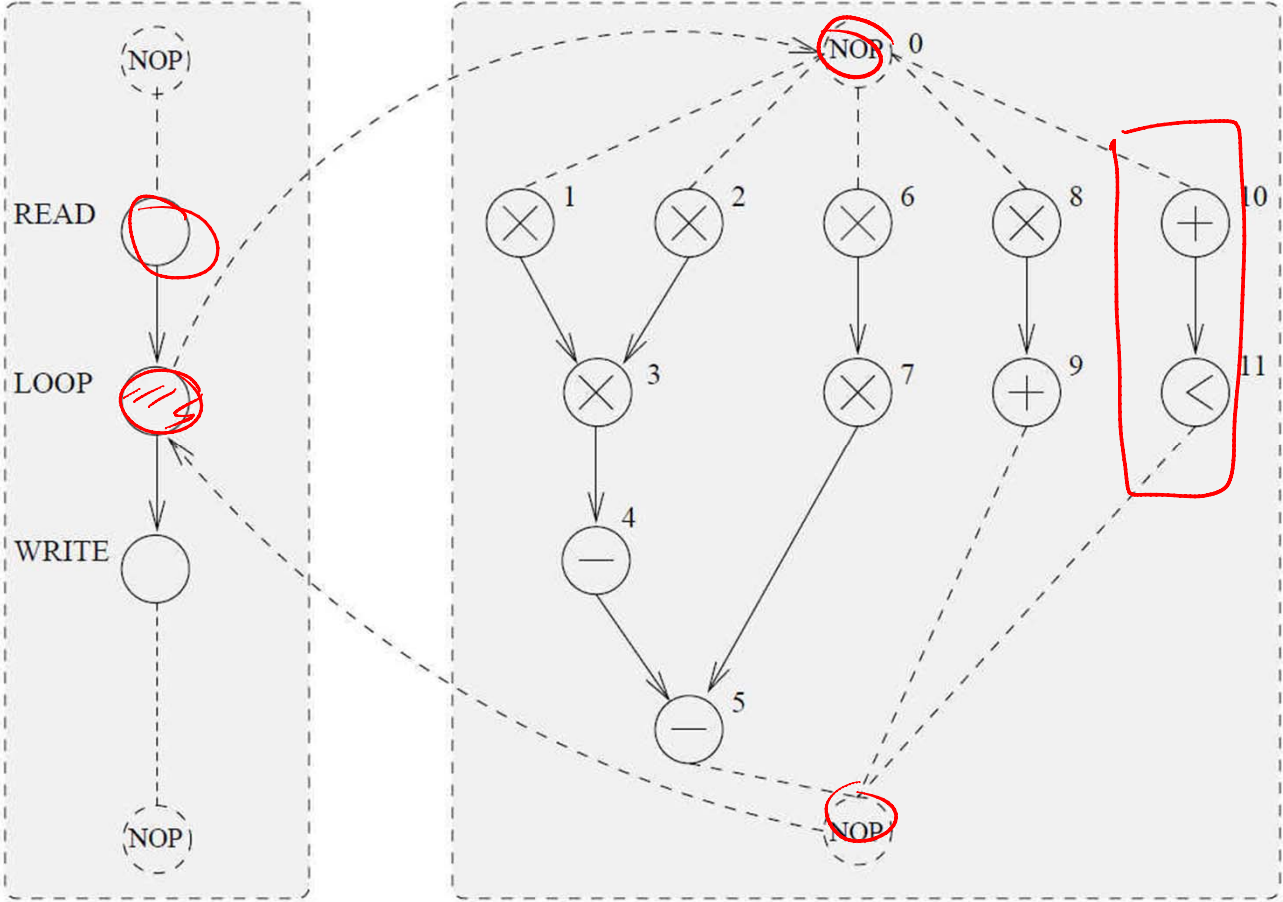IF z > 0 THEN

     p := m + n;

     q := m * n;

END IF

Sequence Graph (SG)

Example iteration (differential equation) - LOOP:

```
int diffeq(int x, int y, int u, int dx, int a)
    { int x1, u1, y1;
    while ( x < a ){
        x1 = x + dx;
        u1 = u - (3 * x * u * dx) - (3 * y * dx);
        y1 = y + u * dx;
        x = x1; u = u1; y = y1;
    }
    return y;
    }
```
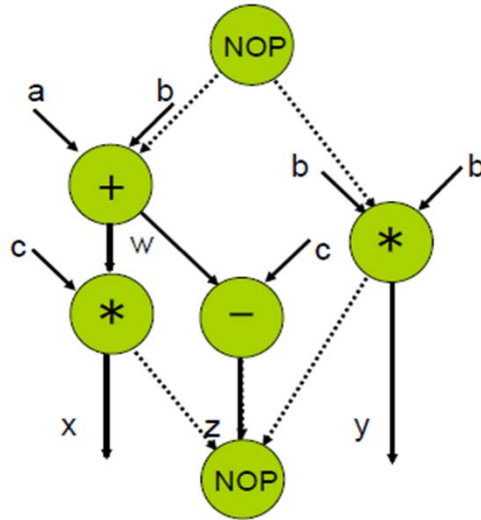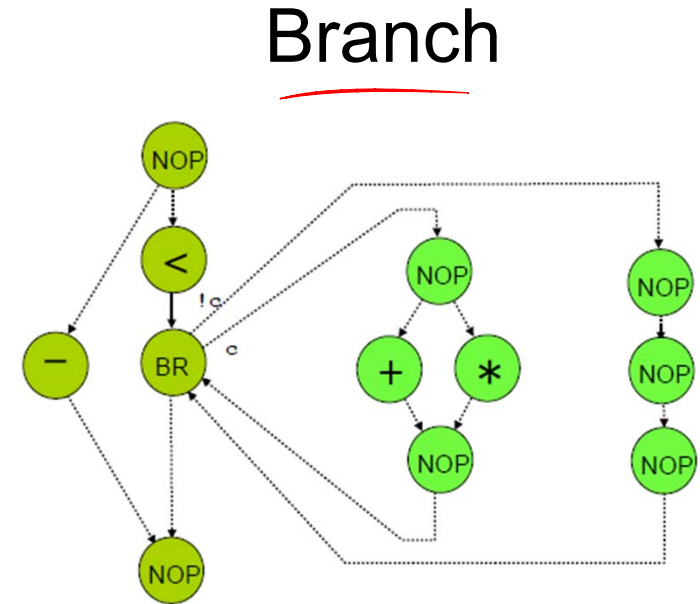
Sequence Graph (SG)

Sequence Graph (SG)
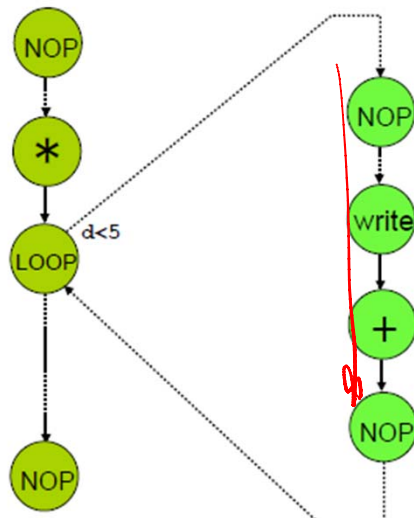
Unit


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

Branch


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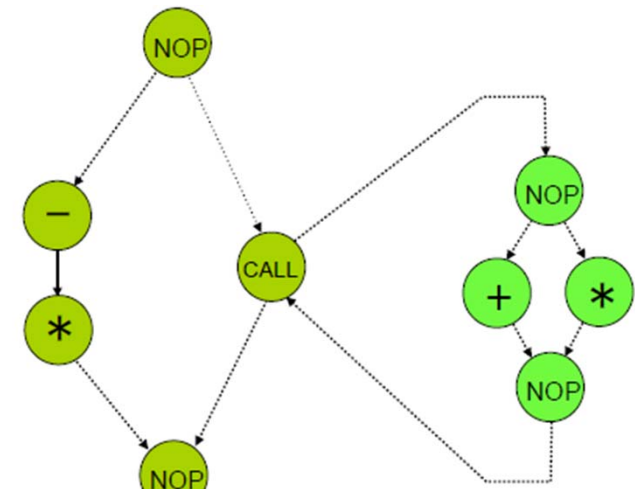
Loop


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

Call


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

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

Sequence graph

- Hierarchy of chained units
    - units model data flow
    - hierarchy models control flow
- Special nodes
    - start/end nodes: NOP (no operation)
    - branch nodes (BR)
    - iteration nodes (LOOP)
    - module call nodes (CALL)
- Attributes
    - nodes: computation times, cost, ...
    - edges: conditions for branches and iterations

Selected Models of computation

| Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA	

* Classification based on implementation

Hardware/System description languages

- ## VDHL
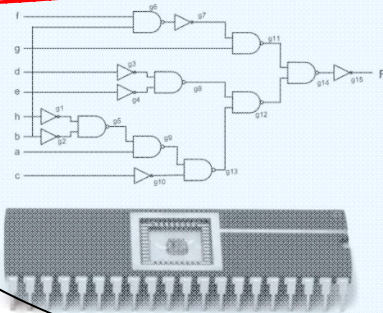 - ### VHDL-AMS

- ## SystemC
  - ### TLM

Discrete event semantics

- **Basic discrete event (DE) semantics**
  - Queue of future actions, sorted by time
  - Loop:
    - Fetch next entry from queue
    - Perform function as listed in entry
      - May include generation of new entries
  - Until termination criterion = true

queue

time	5	10	13	15	19	𝑢
action	a:=5	b:=7	c:=8	a:=6	a:=9	c:=9

6 a
7 b
8 c

time
action
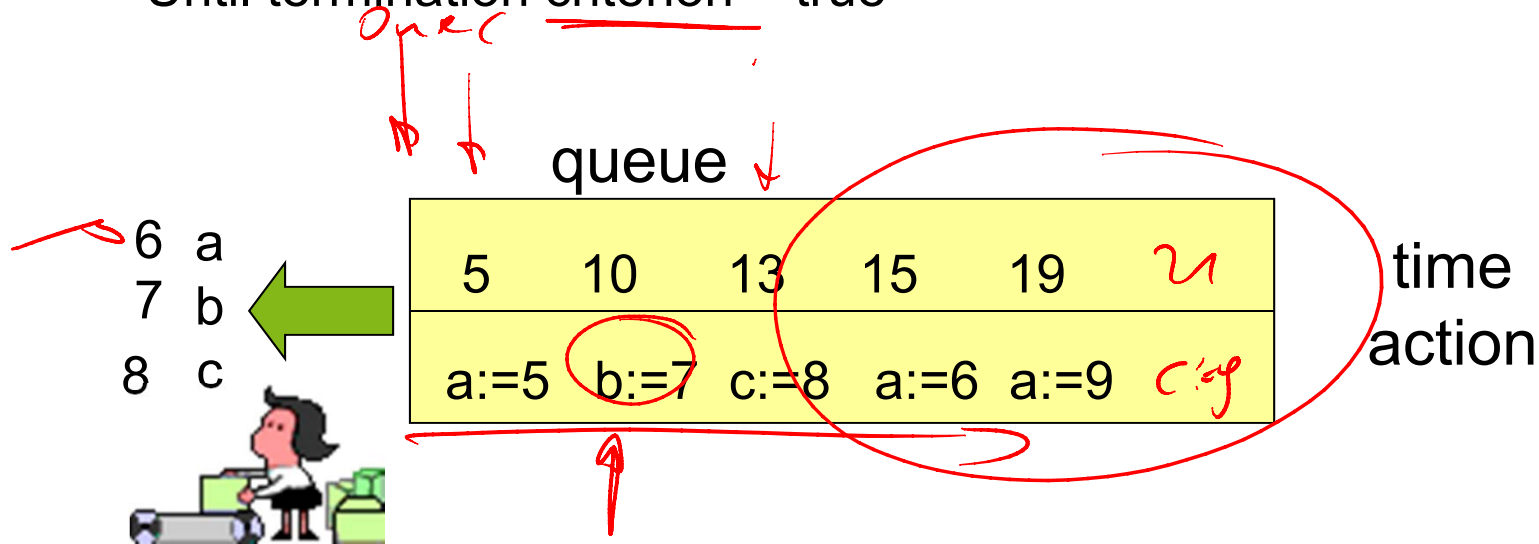
Methods for executing algorithms

Hardware
(Application Specific Integrated Circuits)

Reconfigurable computing

Software-programmed processors

Advantages:
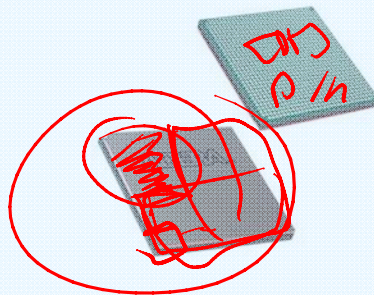- very high performance and efficient

Disadvantages:
- not flexible (can't be altered after fabrication)
- expensive

Advantages:
- fills the gap between hardware and software
- much higher performance than software
- higher level of flexibility than hardware

Advantages:
- software is very flexible to change

Disadvantages:
- performance can suffer if clock is not fast
- fixed instruction set by hardware

Basic Design Methodology