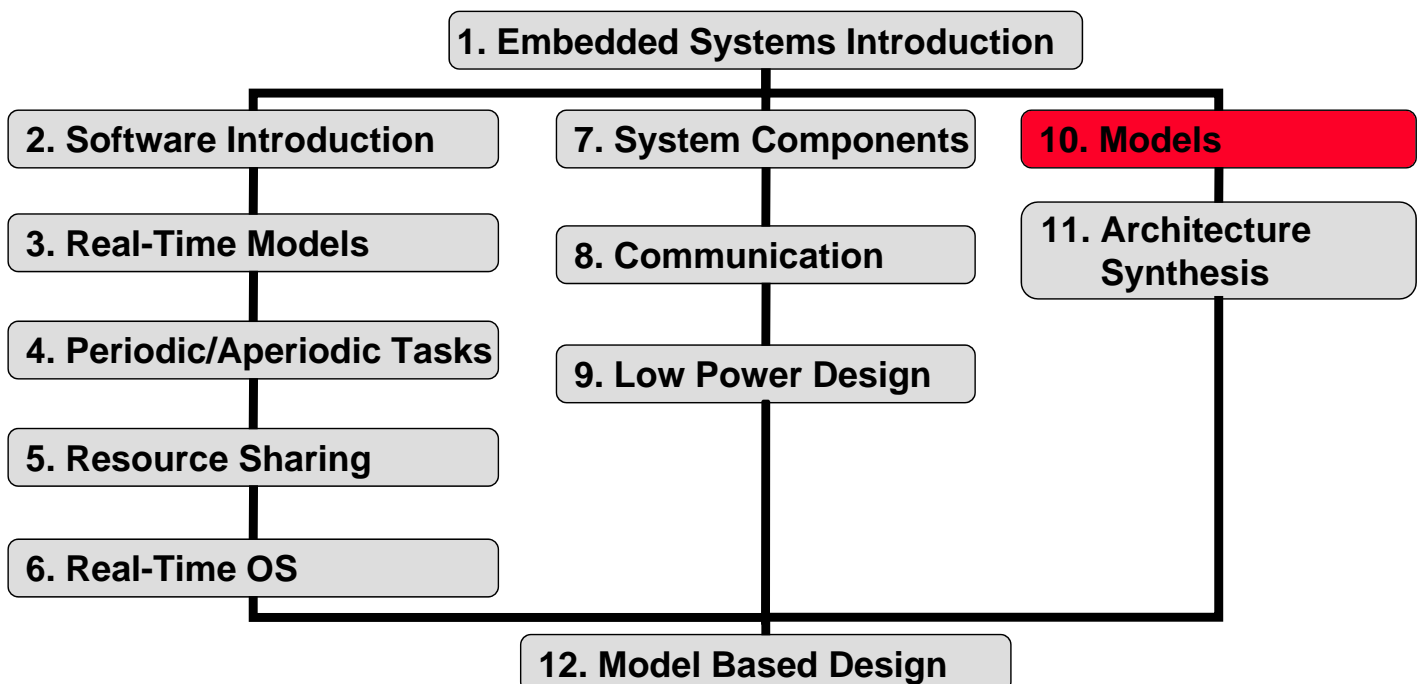


Embedded Systems

10. Architecture Design – Models

Lothar Thiele

Contents of Course



*Software and
Programming*

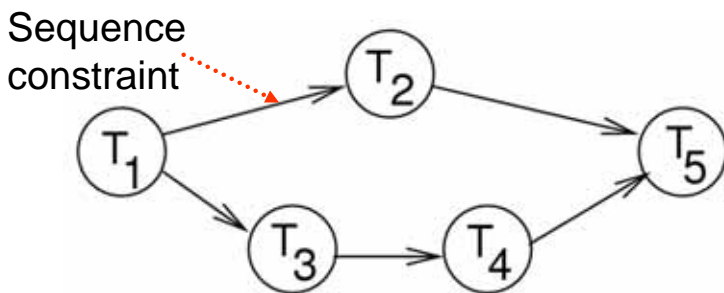
*Processing and
Communication*

Hardware

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, see also the models used for real-time software (chapter 3) and general specification models for the whole system.
- ▶ **Now:**
 - Relevant models for the architecture level (hardware).

Task Graph or Dependence Graph (DG)



Nodes are assumed to be a „program“ described in some programming language, e.g. C or Java; or just a single operation.

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 $v1$.

Suppose E^* is the transitive closure of E .

If $(v1, v2) \in E^*$, then $v1$ is called a **predecessor** of $v2$ and $v2$ is called a **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

given basic block:

$x = a + b;$

$y = c - d;$

$z = x * y;$

$y = b + d;$

single assignment form:

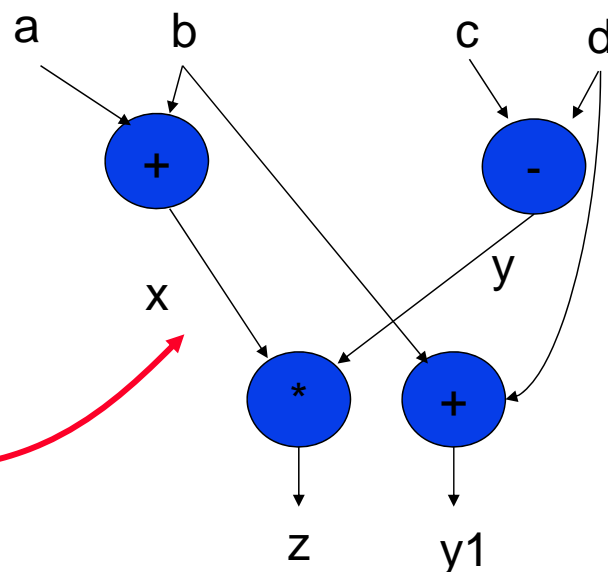
$x = a + b;$

$y = c - d;$

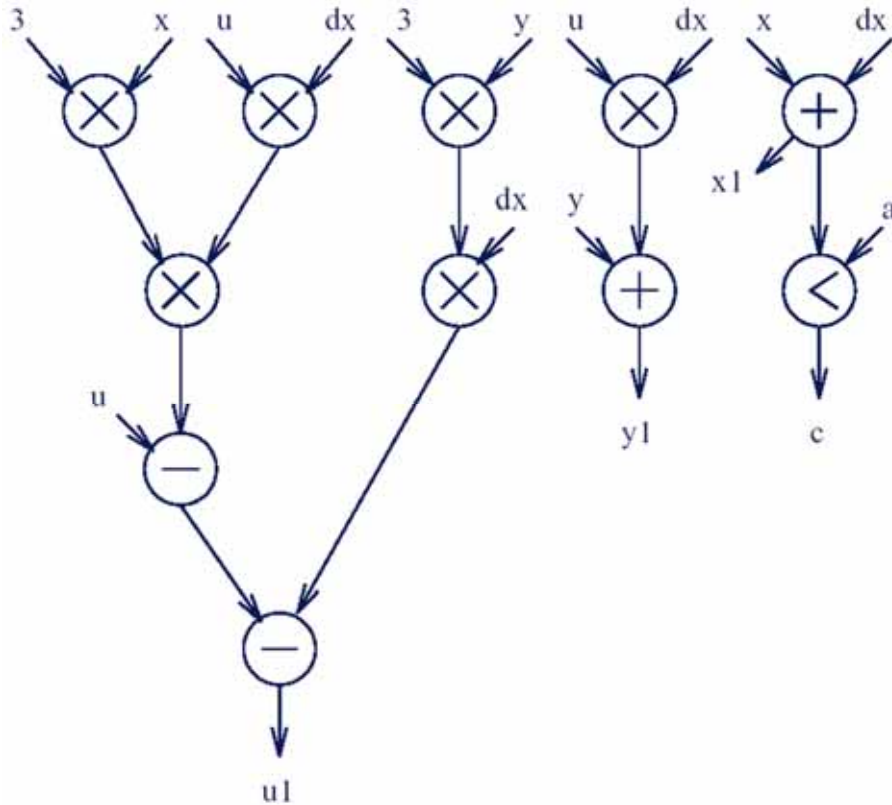
$z = x * y;$

$y1 = b + d;$

dependence graph



Dependence Graph (DG)



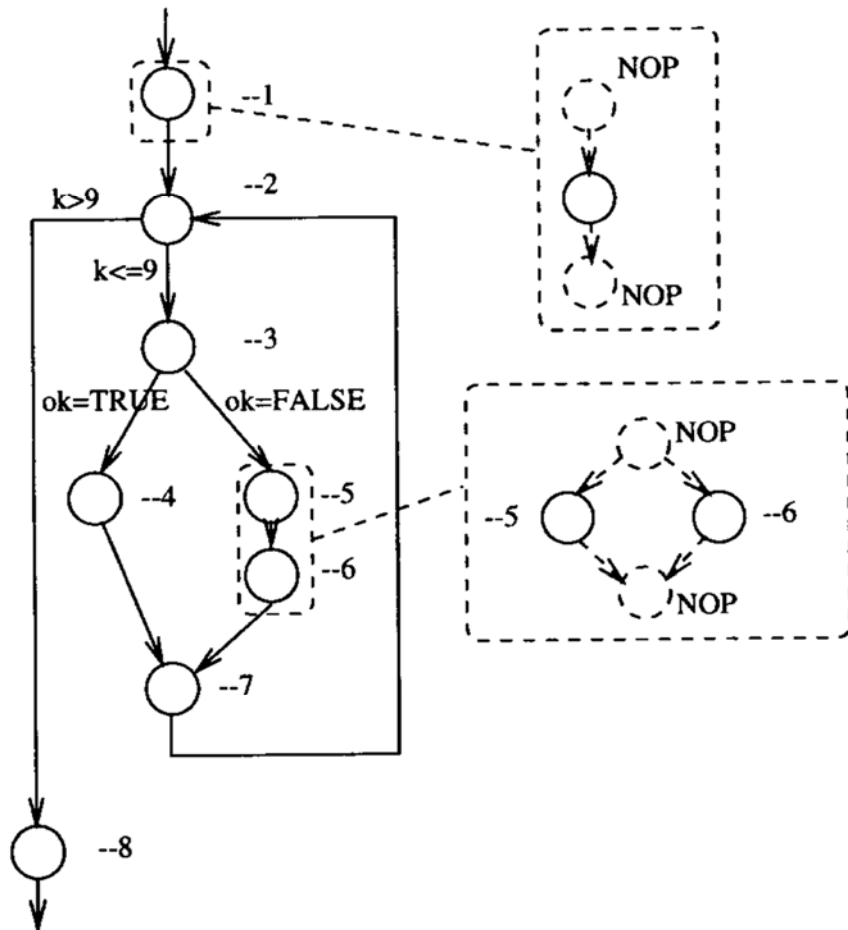
Control-Data Flow Graph (CDFG)

- ▶ **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.

```

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

```



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**).

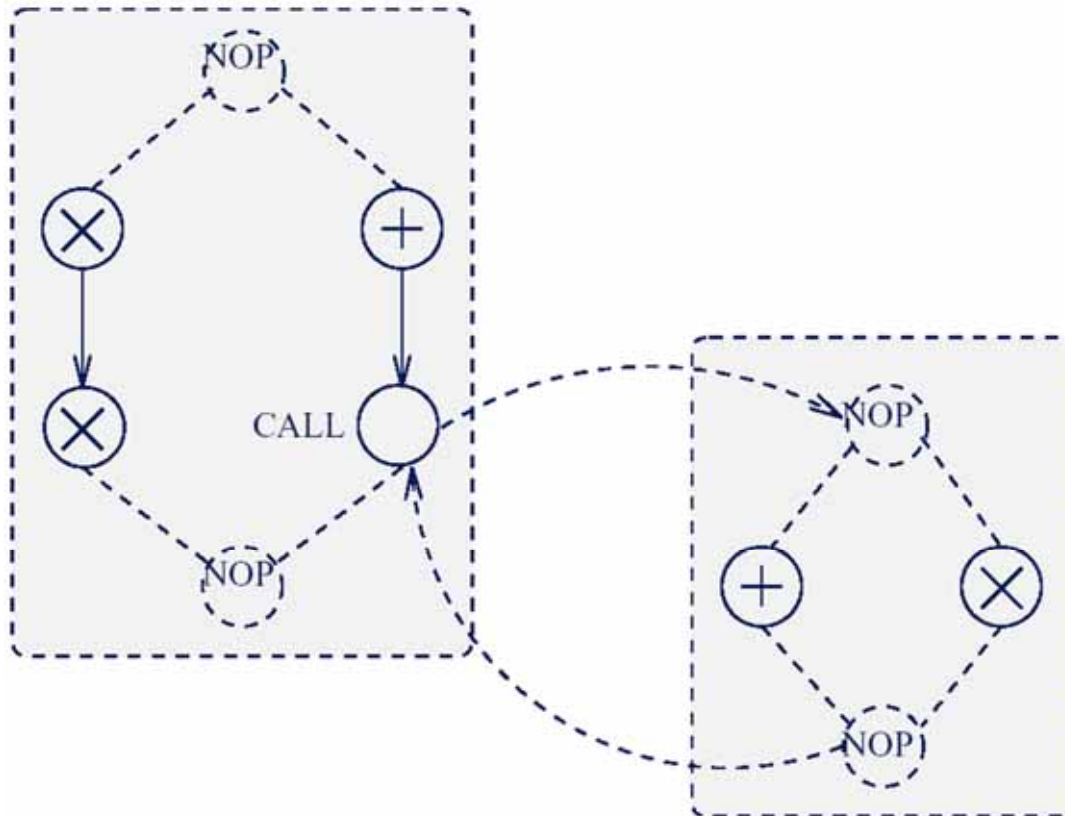
Sequence Graph (SG)

▶ **Example:**

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

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

Sequence Graph (SG)

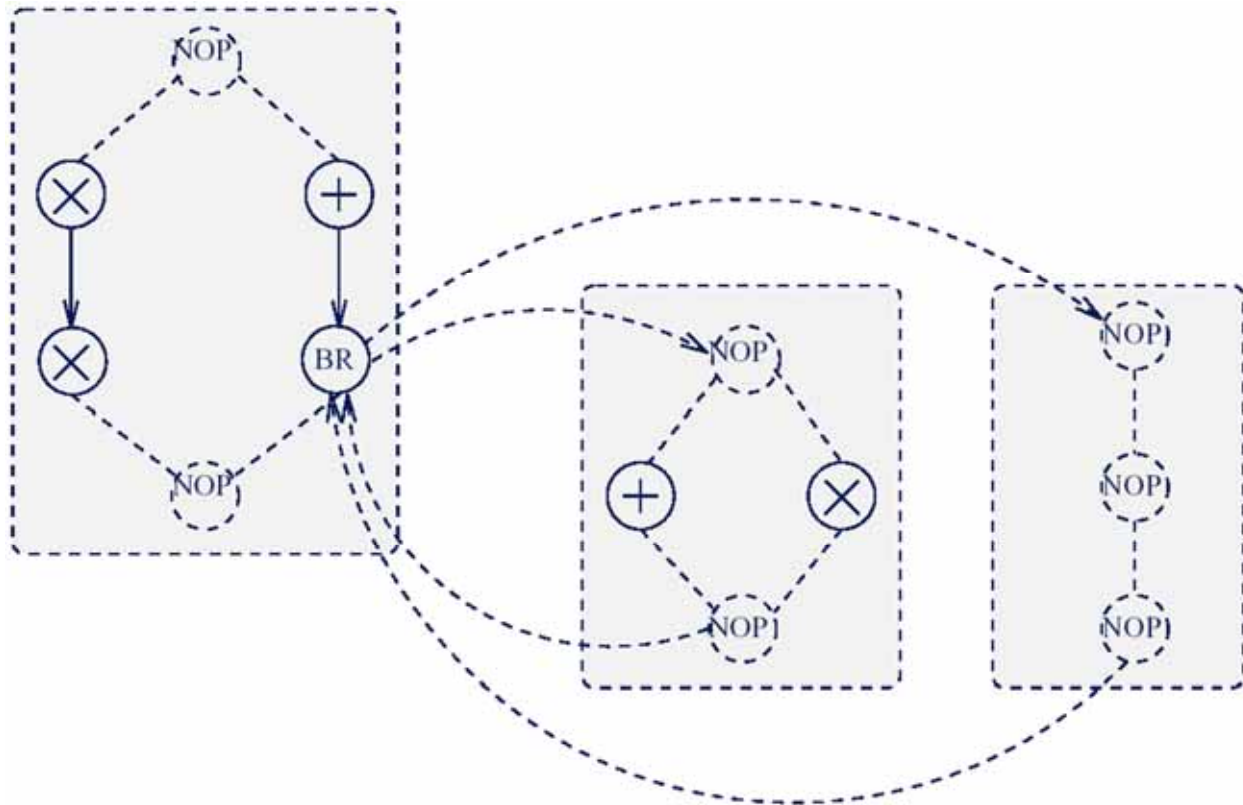


Sequence Graph (SG)

► **Example:**

```
x := a * b;  
y := x * c;  
z := a + b;  
IF z > 0 THEN  
    p := m + n;  
    q := m * n;  
END IF
```

Sequence Graph (SG)

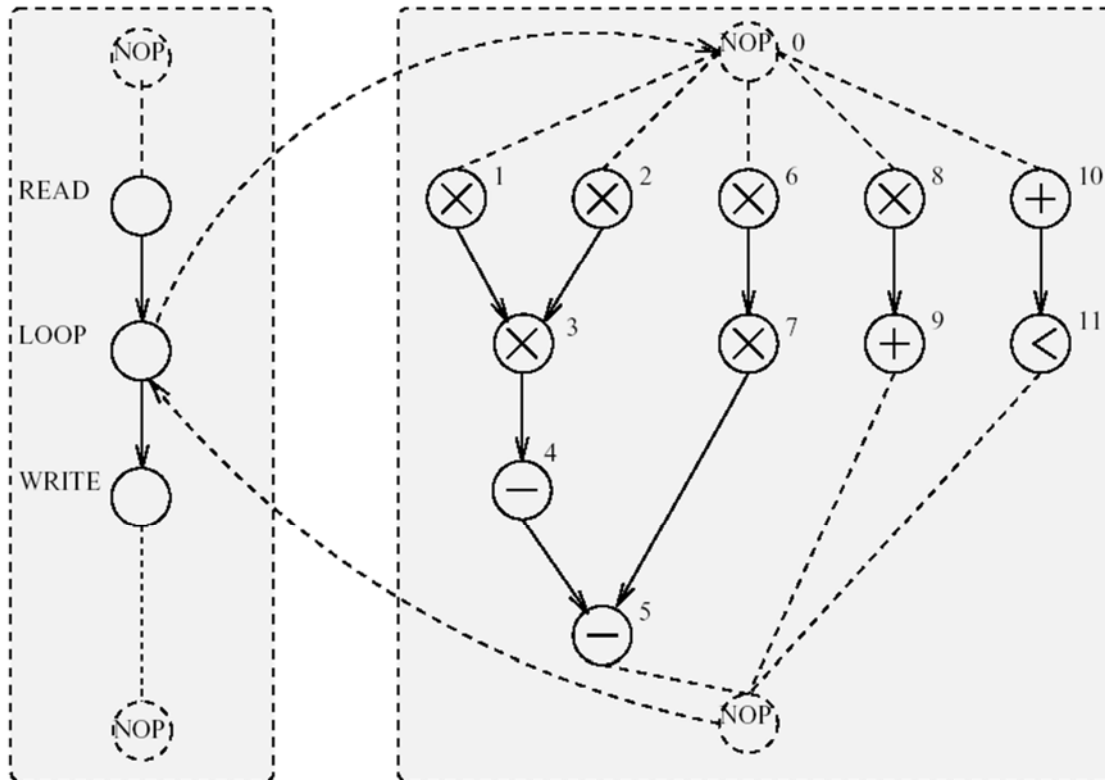


Sequence Graph (SG)

- **Example** iteration (differential equation):

```
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;
}
```

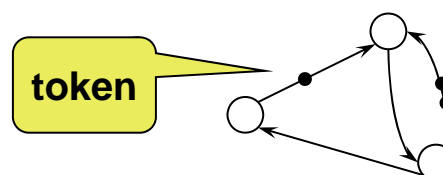
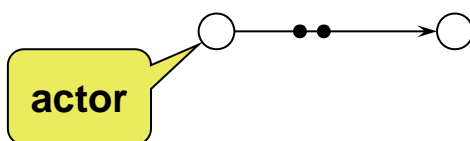
Sequence Graph (SG)



Marked Graphs (MG)

- ▶ A **marked graph** $G = (V, A, del)$ consists of
 - nodes (**actors**) $v \in V$
 - edges $a = (v_i, v_j) \in A, A \subseteq V \times V$
 - number of initial tokens on edges $del : A \rightarrow \mathbf{N}$
- ▶ The **marking** (distribution of tokens) is often represented in form of a vector:

$$del = (... del_k ...) \in \mathbf{Z}^{1 \times |A|}$$



Marked Graphs (MG)

- ▶ The token on the edges correspond to data that are stored in **FIFO queues**.
- ▶ A node (actor) is called **activated** if on every input edge there is at least one token.
- ▶ A node (actor) can **fire** if it is activated.
- ▶ The **firing** of a node v_i (actor operates on the first tokens in the input queues) removes from each input edge a token and adds a token to each output edge. The output token correspond to the processed data.
- ▶ Marked graphs are mainly used for modeling **regular computations**, for example signal flow graphs.

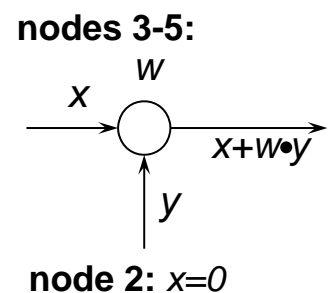
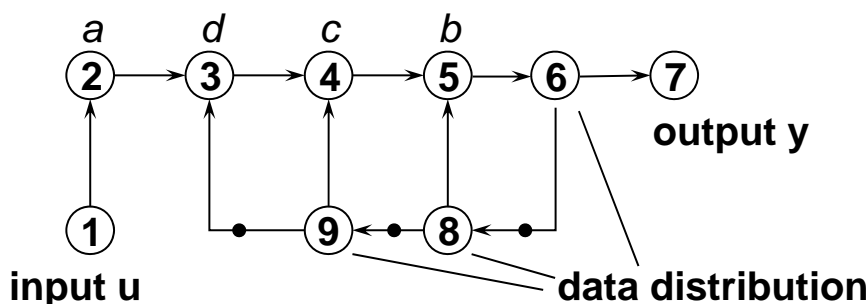
Marked Graphs (MG)

- ▶ **Example** (model of a digital filter with infinite impulse response IIR)

- **Filter equation:**

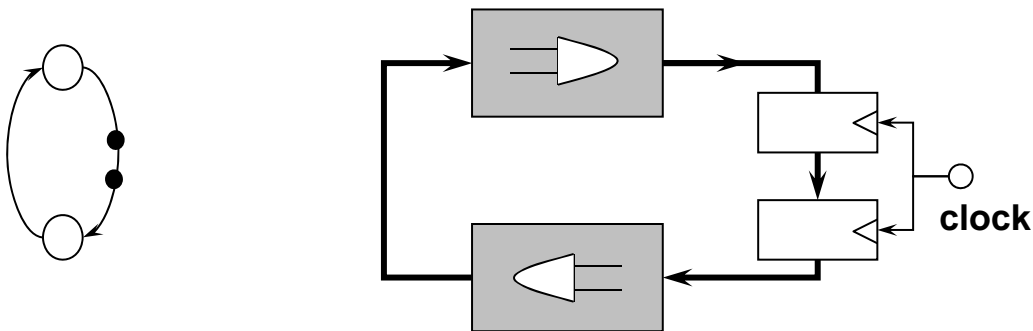
$$y(l) = a \cdot u(l) + b \cdot y(l-1) + c \cdot y(l-2) + d \cdot y(l-3)$$

- Possible model as a **marked graph**:



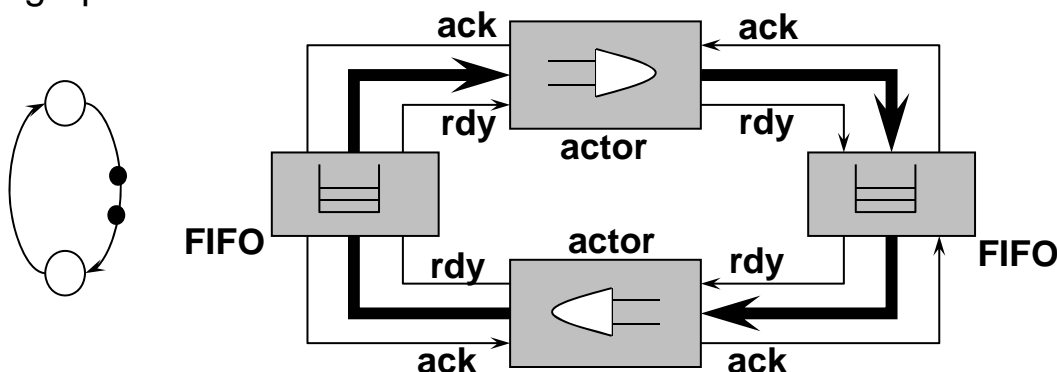
Implementation of Marked Graphs

- ▶ There are *different possibilities* to implement marked graphs in hardware or software directly. Only the most simple possibilities are shown here.
- ▶ Hardware implementation as a *synchronous digital circuit*.
 - Actors are implemented as combinatorial circuits.
 - Edges correspond to synchronously clocked shift registers (FIFOs).



Implementation of Marked Graphs

- Hardware implementation as a *self-timed asynchronous circuit*.
 - Actors and FIFO registers are implemented as independent units.
 - The coordination and synchronization of firings is implemented using a *handshake protocol*.
 - Delay insensitive direct implementation of the semantics of marked graphs.



Implementation of Marked Graphs

- **Software** implementation with *static scheduling*:

- At first, a **feasible sequence** of actor firings is determined which ends in the starting state (initial distribution of tokens).
- This sequence is implemented directly in software.
- **Example** digital filter:

feasible sequence: (1, 2, 3, 9, 4, 8, 5, 6, 7)

program:

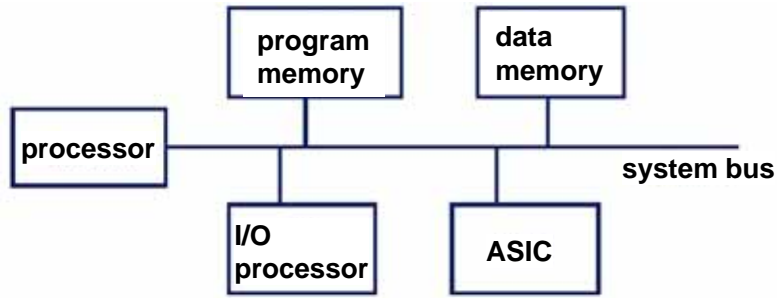
```
while(true) {
    t1 = read(u);
    t2 = a*t1;
    t3 = t2+d*t9;
    t9 = t8;
    t4 = t3+c*t9;
    t8 = t6;
    t5 = t4+b*t8;
    t6 = t5;
    write(y, t6);}
```

Implementation of Marked Graphs

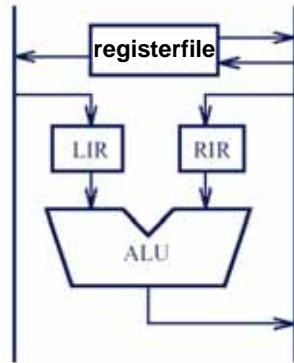
- **Software** implementation with *dynamic scheduling*:

- Scheduling is done using a (real-time) operating system.
- **Actors** correspond to **threads** (or tasks).
- **After firing** (finishing the execution of the corresponding thread) the thread is removed from the set of ready threads and put into **wait state**.
- It is put into the **ready state** if all necessary input data are present.
- This mode of execution directly corresponds to the semantics of marked graphs. It can be compared with the self-timed hardware implementation.

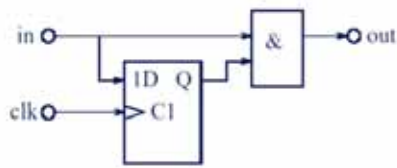
Block Diagrams



a) System Level



b) Architecture Level



c) Logic Level