System-on-Chip Design
Data Flow Modeling

(Based on slides at ECE 522 at UNM)

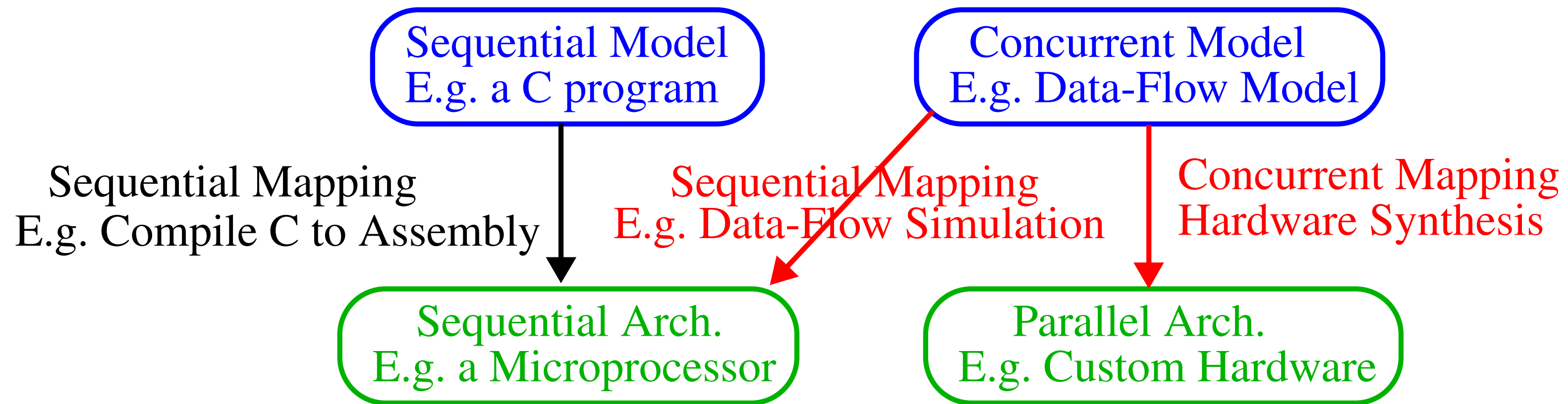
Hao Zheng
Comp Sci & Eng
U of South Florida

Data-Flow Modeling (A Practical Introduction to HW/SW Codesign, P. Schaumont)

As we discussed, hardware models are used to describe parallel systems while software models target sequential systems

Fortunately, we can use **concurrent** models to describe systems that are *potentially* parallel, and are not forced to opt at the start of a design for one or the other

Concurrent models can be implemented as either *parallel* or *sequential* processes



Data-flow models are introduced as a *classic* and *often-used* mechanism of concurrent application modeling

Data-Flow Modeling

Data-flow models have several nice features that are not offered by C:

- Data-flow models are **concurrent**
  - They can describe hardware and software and can be implemented in hardware or in software
- Data-flow models are also **distributed**
  - Components are interconnected *without* the need for a *centralized* controller to synchronize the individual components
- Data-flow models are **modular**
  - It is possible to develop a design library of data-flow components and to use that library in a *plug-and-play* fashion to construct systems
- Data-flow models are well suited for **regular data processing**
  - They are often used in signal processing applications

Data Flow systems are easy to analyze, and properties such as **deadlock** and **stability** can be evaluated based on inspection of the model

- This is difficult to do with, e.g. C programs or HDLs

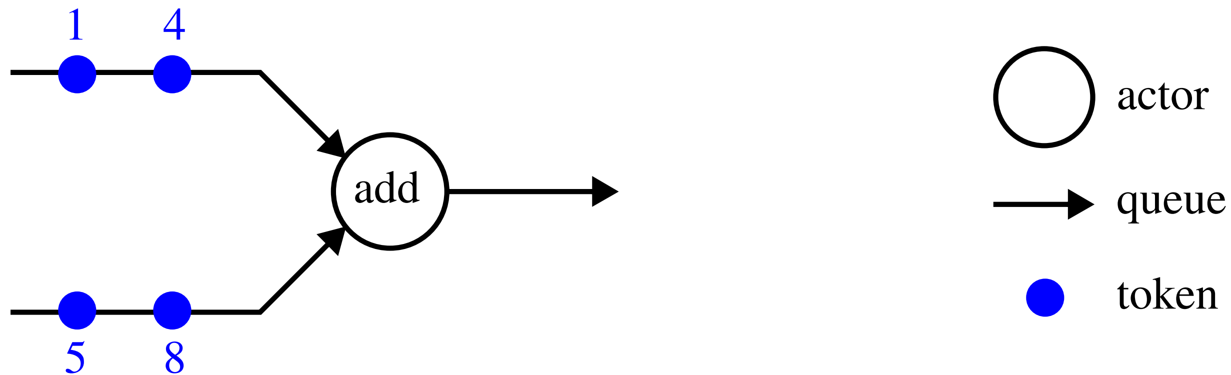
Data-Flow Modeling

We first consider the elements that make up a data flow model, and discuss a technique for formal analysis of data flow models called SDF graphs

We then look into systematic conversion of SDF graphs into a hardware or software implementation

Basics of Data-Flow Modeling

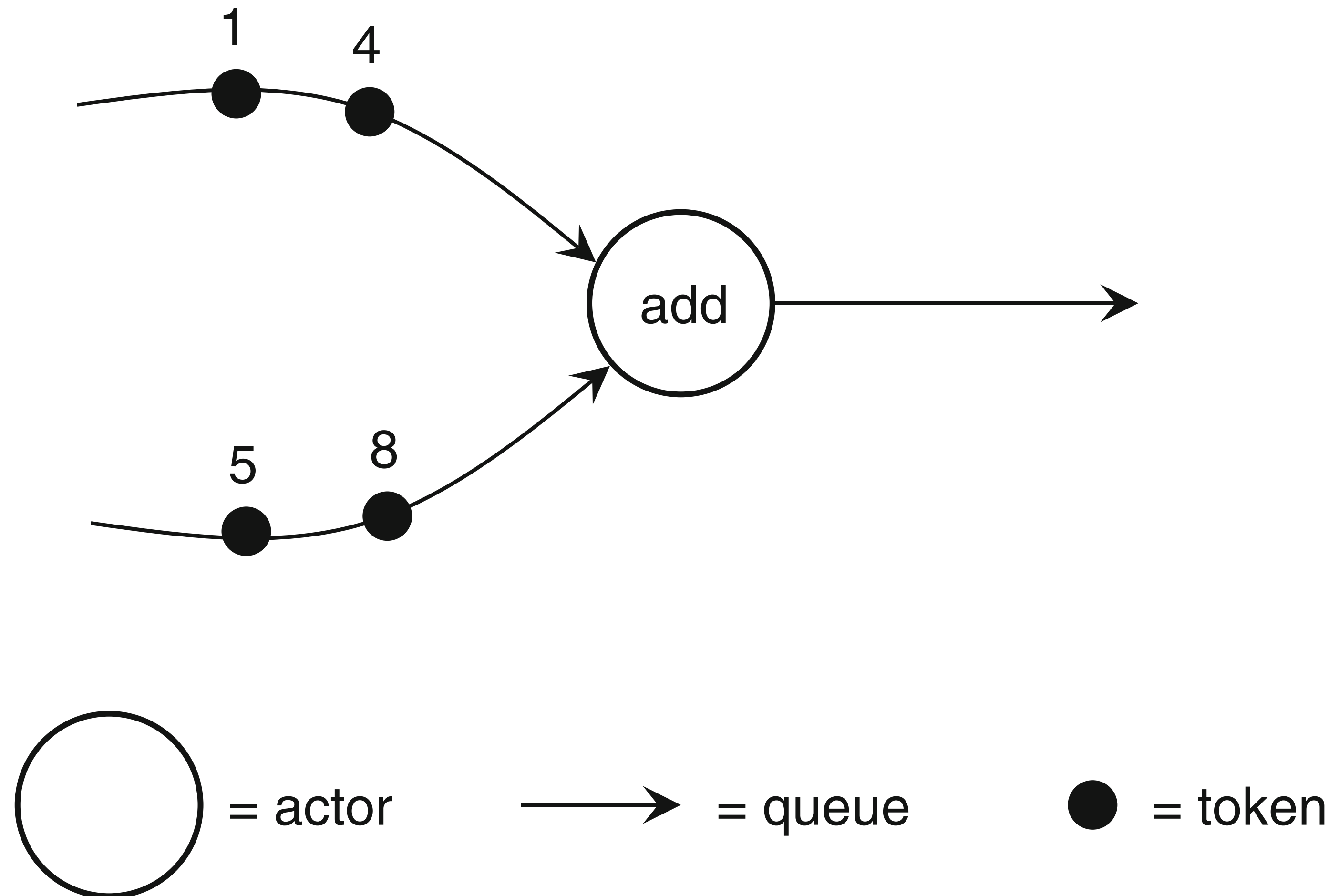
A simple example:



Basic Elements of DF Models

Actors contain the actual operations: bounded behavior with beginning and ending.

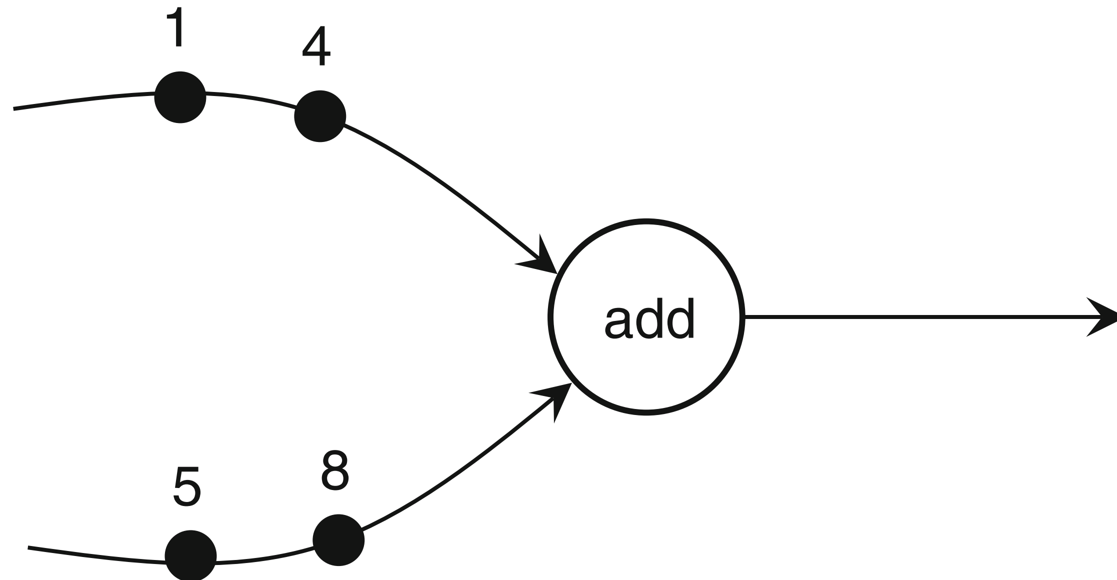
Actors iterate the behavior from beginning to the end.



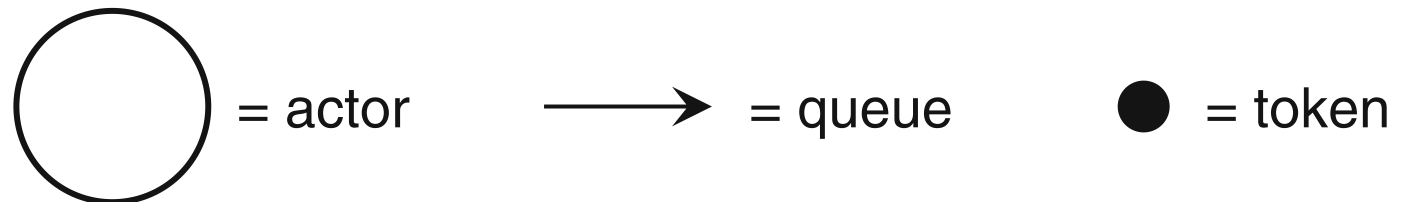
Each iteration is called a **firing.**

Basic Elements of DF Models

Tokens carry information from one actor to another.

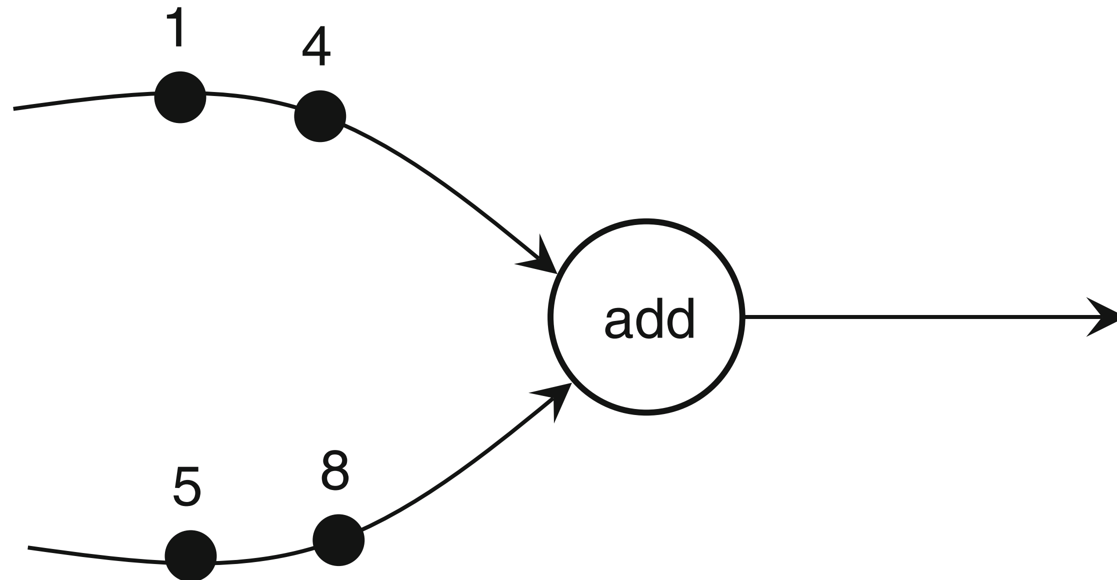


Tokens can be labeled with values

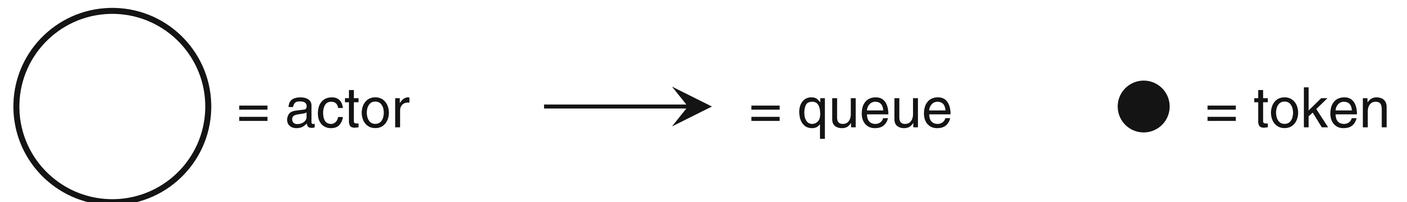


Basic Elements of DF Models

Arcs are **queues,** unidirectional communication links.



Queues are organized as FIFOs



Basics of Data-Flow Modeling

When a data-flow model executes, actors read tokens from their queues and transform *input* token values to *output* token values

The execution of a data-flow model is expressed as a sequence of *possibly concurrent* actor ***firings***

Data-flow models are **untimed**

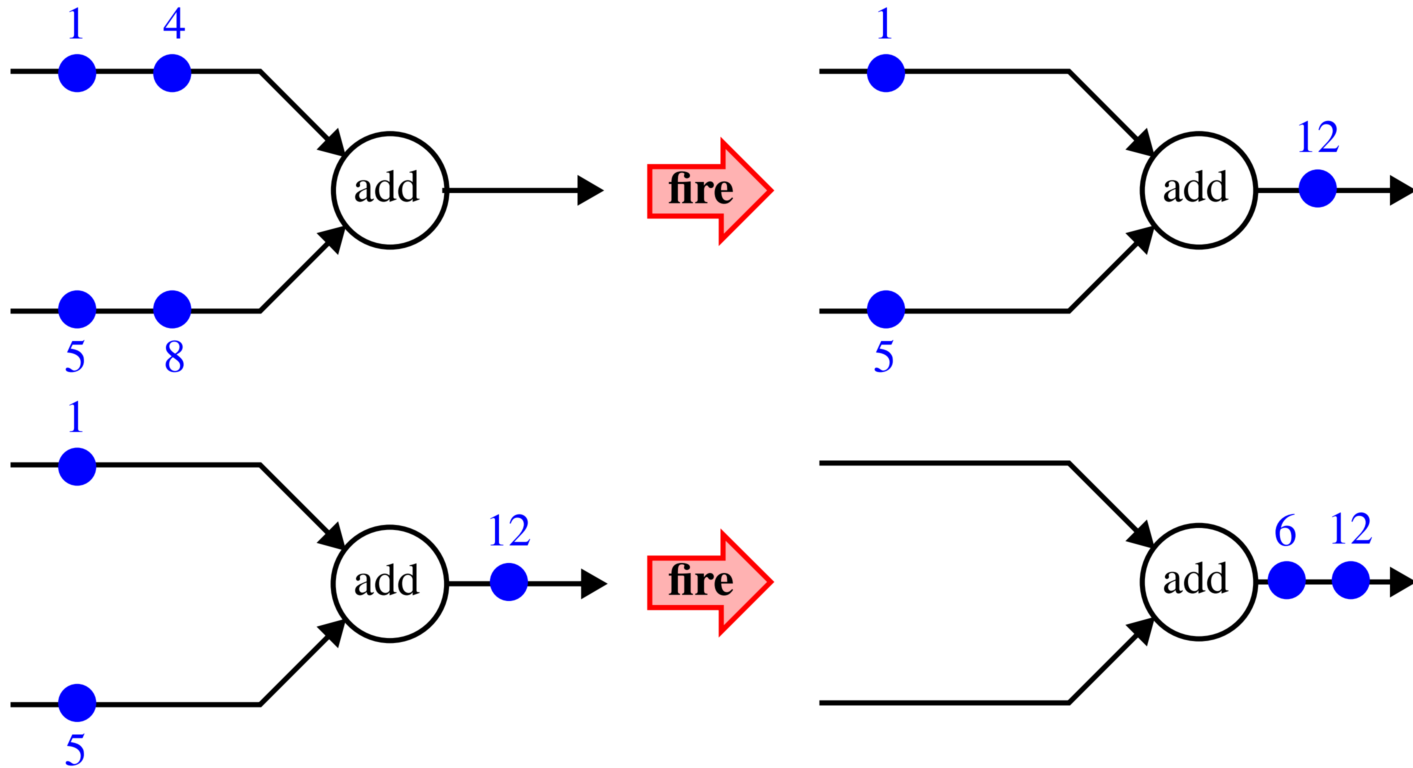
The firing of an actor takes zero time (obviously a real implementation requires a finite amount of time), i.e., time is **irrelevant**

The execution of data-flow models is *guided* only by the presence of data, i.e., an actor can **not** fire until data becomes available on its inputs

A data-flow graph with tokens is called a **marking** of a data-flow graph

A data-flow graph goes through a series of *marking* when it is executed

Basics of Data-Flow Modeling



Each marking refers to a different **state** of the system

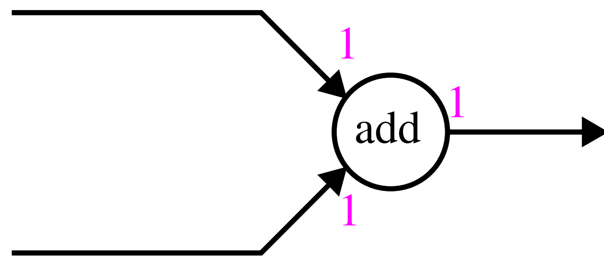
Actors do not have internal states.

Basics of Data-Flow Modeling **Firing Rates, Firing Rules, and Schedules**

Simple actors, e.g., the *add* actor, fire when there is a token on each of its queues

A firing rule involves testing the number of tokens present on the input queues

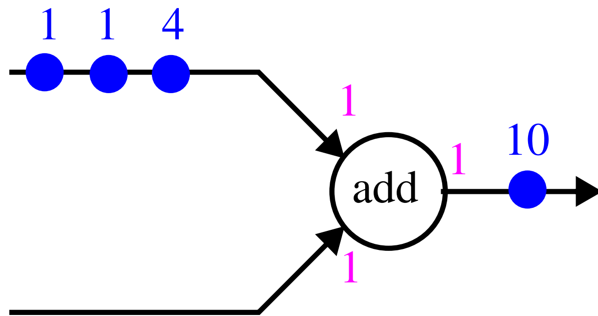
The **required** number of tokens consumed and produced can be annotated on the actors *inputs* and *outputs*, respectively



Inputs: consumption rate

Outputs: production rate

With this information, it becomes clear whether or not an actor can fire under a given marking



Synchronous Data-Flow Graphs

Data-flow actors can also consume *more than one* token per firing

This is referred to as a **multi-rate** data-flow graph

Synchronous data-flow (SDF) graphs refer to systems where the number of tokens consumed/produced per actor firing is ***fixed*** and ***constant***

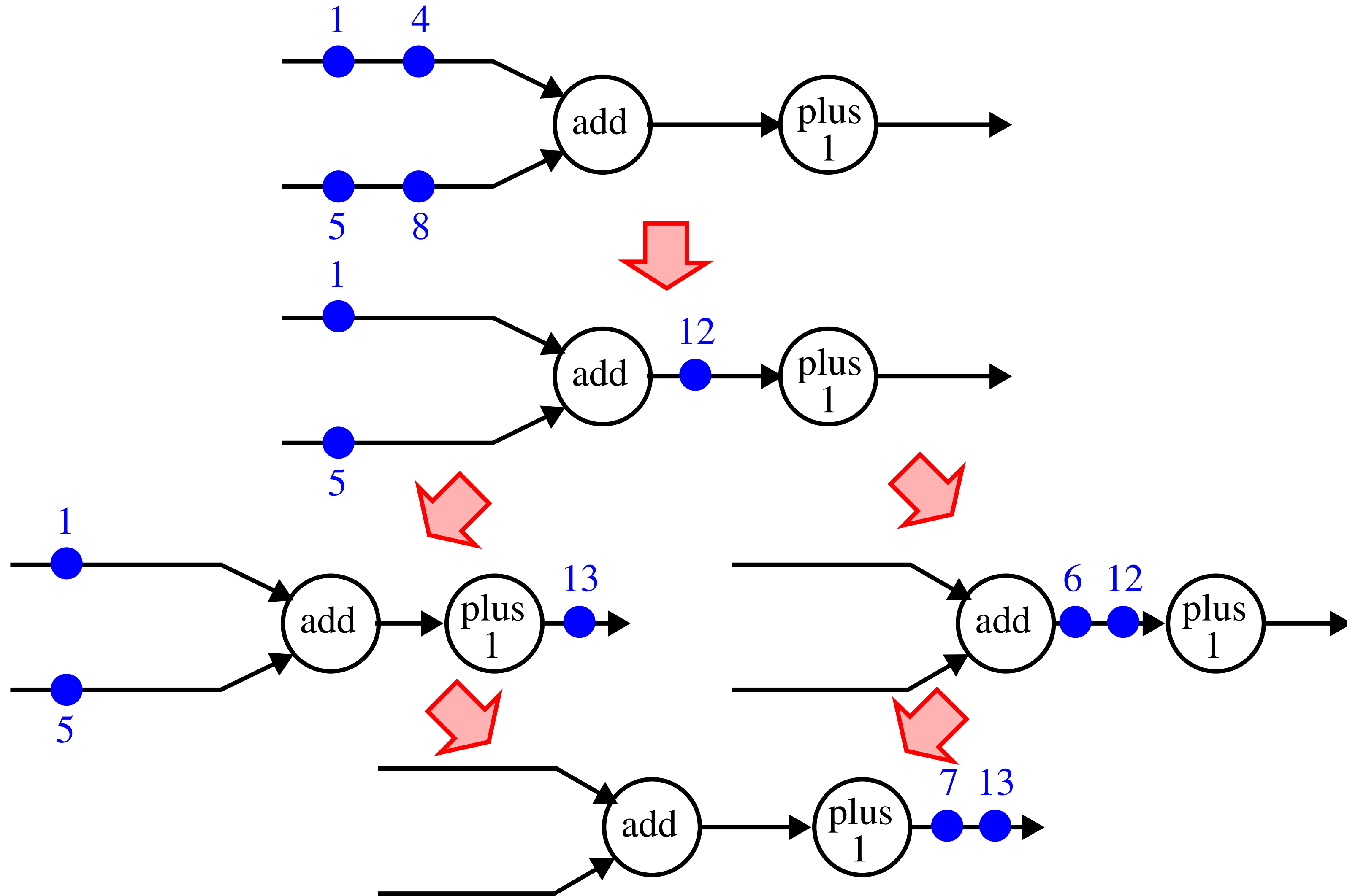
SDFs are the most popular form of data-flowing modeling because of certain properties

- An **admissible** SDF is one that can run forever without *deadlock* or without storing an infinite number of tokens on a communication queue
- An admissible SDF is **determinate**, which means the results produced are independent of the actual *firing order* of the actors in the SDF graph

  The dataflow computation is *independent* of the marking sequence

Synchronous Data-Flow Graphs

An example:



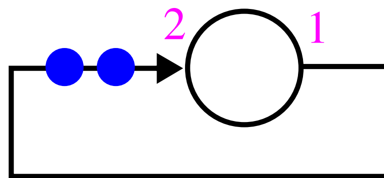
Synchronous Data-Flow Graphs

The **determinate** property is very important, especially for safety-critical embedded system applications

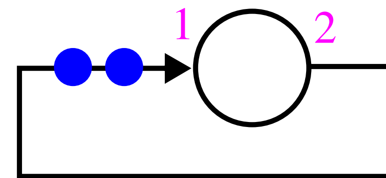
It makes the results independent of the implementation

Given the determinism property, it does **not** matter if, e.g., the ’add’ actor executes on a fast processor and the ’plus 1’ actor on a slow processor

The first property, **admissible**, can be determined by looking only at the graph topology and the actor production/consumption rates



Graph is deadlocked



Infinite # of tokens produced

There is also a systematic method to determine whether a graph is *admissible*

The method developed by Lee is called *Periodic Admissible Schedules*

E. Lee, "Static Scheduling of Synchronous Data Flow Graphs"

Synchronous Data-Flow Graphs

First some definitions:

- A *schedule* is the order in which the actors must fire
- An *admissible schedule* is a firing order that will not cause deadlock nor token build-up
- A *periodic admissible schedule* is a schedule that can continue forever (is periodic and therefore will restart)

We consider *Periodic Admissible Sequential Schedules* (PASS), which requires that only one actor at a time fires

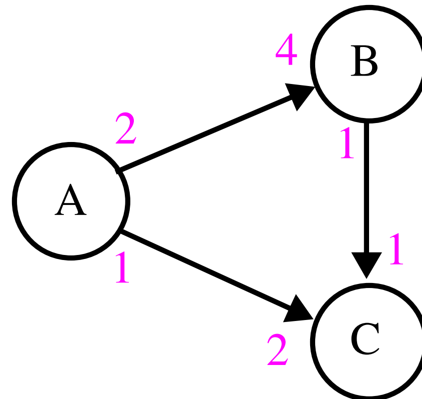
A PASS can be used to execute an SDF model on top of a microprocessor

There are four steps to creating a PASS for an SDF graph (this also tests to see if one exists):

- Create the topology matrix **G** of the SDF graph
- Verify the *rank* of the matrix to be one less than the number of nodes in the graph
- Determine a firing vector
- Try firing each actor in a *round robin* fashion, until the *firing count* given by the firing vector is reached

Synchronous Data-Flow Graphs

Consider the following example:



Step 1: Create a topology matrix for this graph:

The topology matrix has as many rows as there are *edges* (FIFO queues) and as many columns as there are *nodes*

The entry (*i,j*) will be positive if the node *j* **produces** tokens onto the edge *i* and negative if it consumes tokens

$$G = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{matrix} \leftarrow edge(A,B) \\ \leftarrow edge(A,C) \\ \leftarrow edge(B,C) \end{matrix}$$

NOTE: This matrix do NOT need to be square

Synchronous Data-Flow Graphs

Step 2: The condition for a PASS to exist is that the *rank* of **G** has to be one less than the number of nodes in the graph (see Lee's paper for proof)

The *rank* of the matrix is the number of **independent equations** in **G**

For our graph, the rank is 2 -- verify by multiplying the first column by -2 and the second column by -1, and adding them to produce the third column

$$G = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \Rightarrow G = \begin{bmatrix} -4 & +4 & 0 \\ -2 & 0 & -2 \\ 0 & -1 & -1 \end{bmatrix}$$

Given that there are *three* nodes in the graph and the rank of the matrix is 2, a PASS is **possible**

This step effectively verifies that tokens canNOT accumulate on any edge of the graph

A firing vector is used to produce/consume tokens

The tokens produced/consumed can be computed using matrix multiplication

Synchronous Data-Flow Graphs

For example, the tokens produced/consumed by firing A twice and B and C zero times is given by:

$$\text{firing vector } q = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \Rightarrow Gq = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 0 \end{bmatrix}$$

This vector produces 4 tokens on edge(A,B) and 2 tokens on edge(A,C)

Step 3: Determine a periodic firing vector

The *firing vector* given above is not a good choice to obtain a PASS because it leaves tokens in the system

We are instead interested in a firing vector that leaves no tokens:

$$Gq_{\text{PASS}} = 0$$

Note that since the *rank* is less than the number of nodes, there are an infinite number of solutions to the matrix equation

Synchronous Data-Flow Graphs

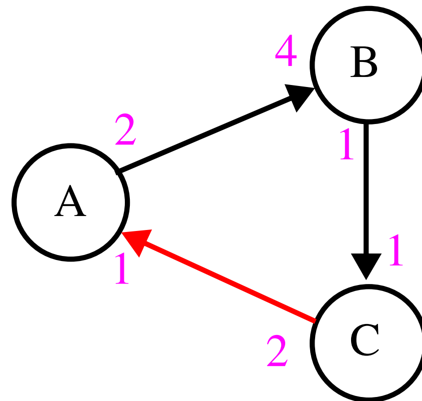
Step 3: Determine a periodic firing vector (cont.)

This is true b/c, intuitively, if *firing vector* (a, b, c) is a PASS, then so should be firing vectors ($2a, 2b, 2c$), ($3a, 3b, 3c$), etc.

Our task is to find the simplest one -- for this example, it is:

$$q_{PASS} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \Rightarrow Gq_{PASS} = \begin{bmatrix} +2 & -4 & 0 \\ +1 & 0 & -2 \\ 0 & +1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that the existence of a PASS firing vector does **not** guarantee that a PASS will also exist



Here, we reversed the (A,C) edge

We would find the same q_{PASS} but the resulting graph is **deadlocked** -- all nodes are waiting for each other

Synchronous Data-Flow Graphs

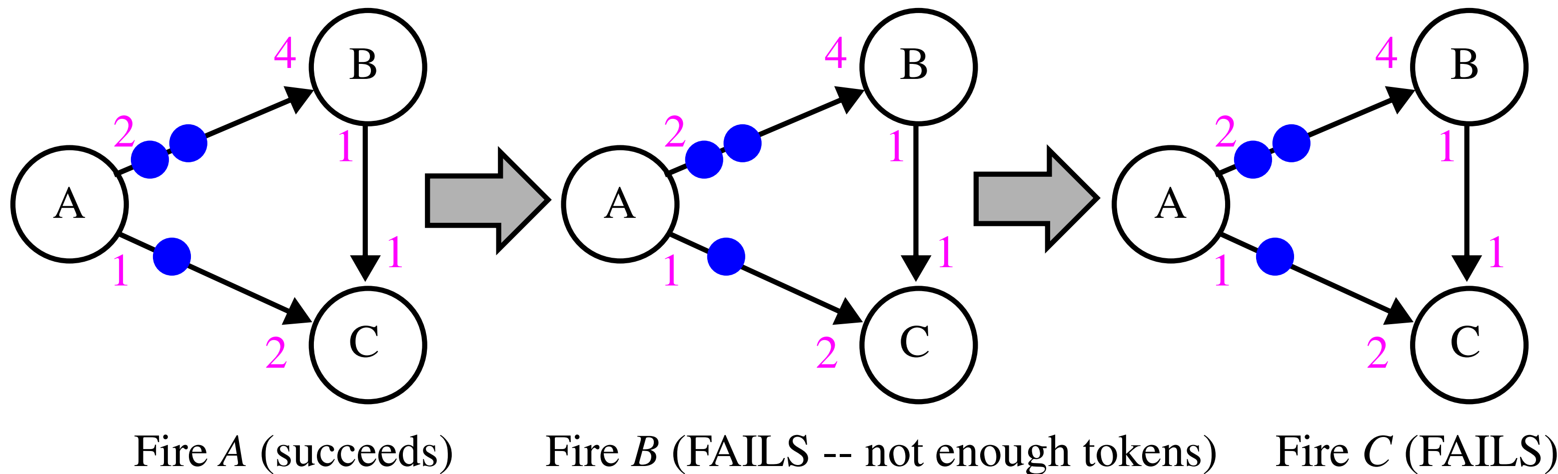
Step 4: Construct a *valid* PASS.

Here, we fire each node up to the number of times specified in q_{PASS}

Each node that is able to fire, i.e., has an adequate number of tokens, will fire

If we find that we can fire NO more nodes, and the firing count is **less** than the number in q_{PASS}, the resulting graph is **deadlocked**

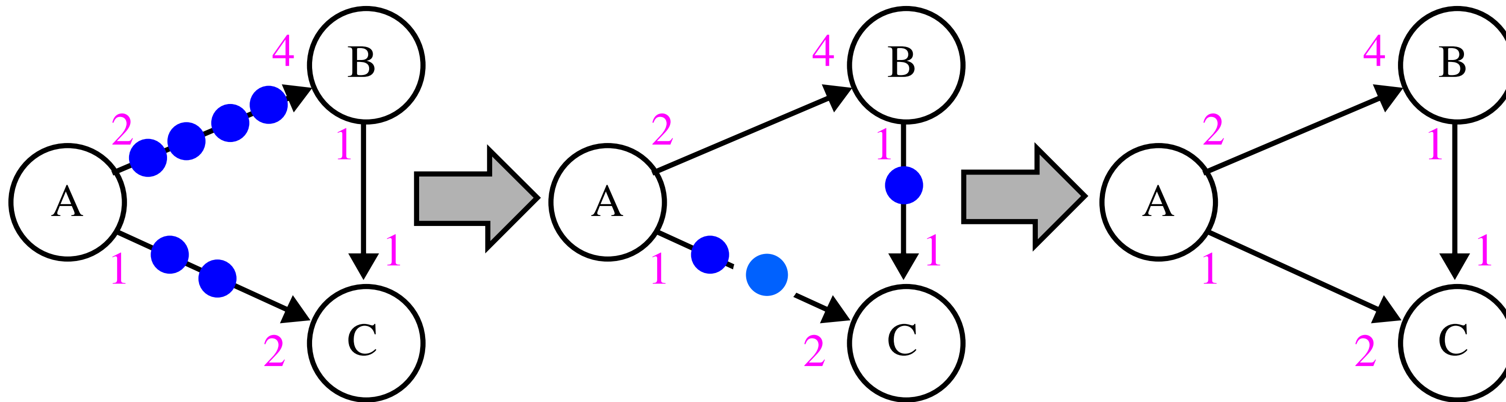
Trying this out on our graph, we fire *A* once, and then B and C



Fire *A* (succeeds) Fire *B* (FAILS -- not enough tokens) Fire *C* (FAILS)

Synchronous Data-Flow Graphs

Step 4: Construct a *valid* PASS.



Fire *A* AGAIN (succeeds) Fire *B* (succeeds) Fire *C* (succeeds)

So the PASS is (*A, A, B, C*)

Try this out on the **deadlocked** graph -- it aborts immediately on the first iteration because no node is able to fire successfully

Note that the **determinate** property allows any ordering to be tried freely, e.g., *B, C* and then *A*

In some graphs (not ours), this may lead to additional PASS solutions

Example

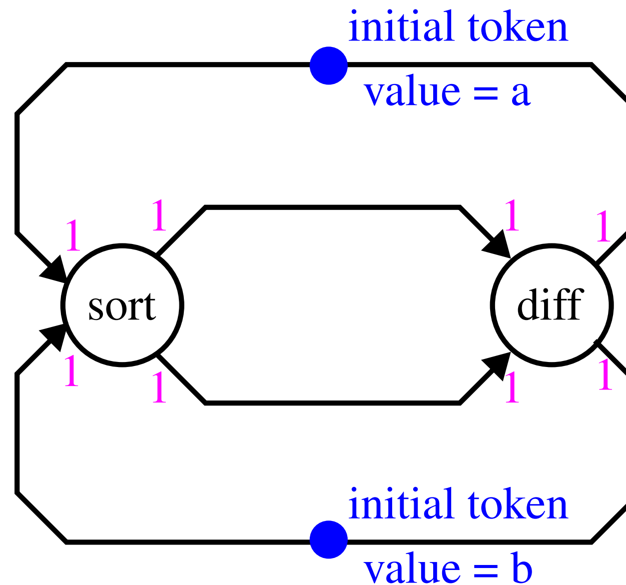
Consider an SDF that models *Euclid's Greatest Common Divisor* (GCD):

sort

```
out1 = (a > b) ? a : b;
out2 = (a > b) ? b : a;
```

diff

```
out1 = (a != b) ? a-b: a;
out2 = b;
```



This SDF evaluates the GCD of two numbers, *a* and *b*

The *sort* actor reads two numbers, sorts them and copies them to the output

The *diff* actor subtracts the smaller number from the larger one (when they are different)

After a couple of iterations, the value of the tokens converge to the GCD

Example

For example, the following sequence is produced when (a,b) = (16,12) are the initial values:

```
(a,b) = (4,12)
(a,b) = (8,4)
(a,b) = (4,4)
(a,b) = (4,4)...
```

Yielding *4* as the GCD of 12 and 16

We will derive a PASS for this system:

left node ↘ ↙ *right node*

$$G = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} \begin{matrix} \leftarrow \textit{edge(sort,diff)} \\ \leftarrow \textit{edge(sort,diff)} \\ \leftarrow \textit{edge(diff,sort)} \\ \leftarrow \textit{edge(diff,sort)} \end{matrix}$$

It is easy to determine that the *rank* is 1 (columns complement each other), so we satisfy condition 1, e.g., rank(G) = nodes - 1

Example

A valid firing vector is one in which each actor fires exactly **once per iteration**

$$q = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

A working *schedule* for this firing vector is to fire each of the actors in sequence using the order (*sort*, *diff*)

Note that in the graph as shown, there is only a single, **strictly sequential** schedule possible

For now, we will also ignore the *stopping condition*, i.e. detecting that *a* and *b* are equal

In conclusion, SDFs have very powerful properties

They allow a designer to determine up-front certain important system properties, such as the *determinism*, *deadlock*, and *storage requirements*

Unfortunately, SDFs are **not** a universal specification mechanism, i.e., they are **not** a good model for any possible hardware/software system.

Control Flow Modeling: Limitations of Data-Flow Models

SDF systems are *distributed*, *data-driven* systems -- they execute when there is data to process and remain idle otherwise

However, SDF have trouble modeling **control** mechanisms

Control appears in many different forms in system design:

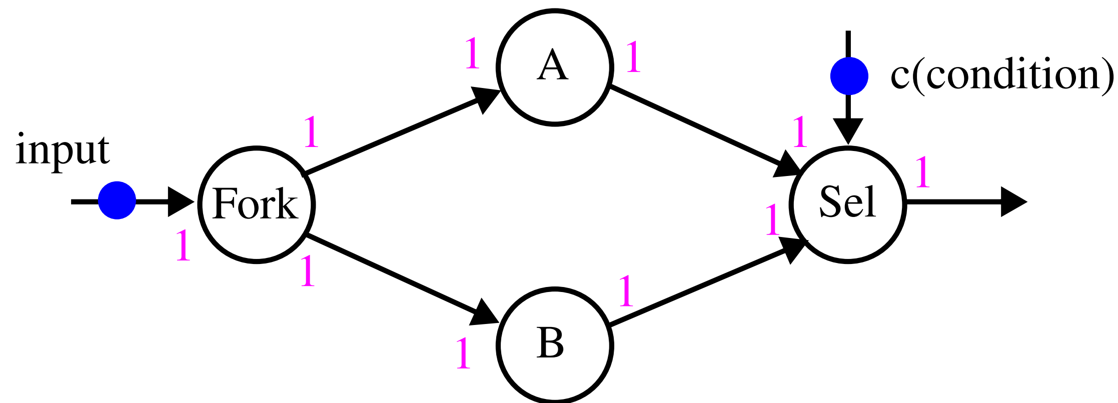
- Stopping and re-starting: An SDF model never terminates -- it keeps running.
  Stopping/re-starting is a control-flow property not addressed well with SDFs
- Mode-switching: When a cell-phone switches from one standard to the other, the baseband processing (modeled as an SDF) needs to be reconfigured
  The topology of an SDF graph is fixed and **cannot** be modified at runtime
- Exceptions: When catastrophic events happen, processing may need to be altered
  SDFs cannot model exceptions that affect the entire graph, e.g., empty queues
- Run-time conditions: A simple *if-then-else* stmt cannot be modeled by SDFs
  An SDF node cannot simply disappear or become *inactive* - it is always there

Control Flow Modeling: Limitations of Data-Flow Models

There are two solutions to the problem of *control flow modeling* in SDFs

Solution 1: *simulate* control flow on top of the SDF semantics at the expense of adding modeling overhead

Consider the stmt *if (c) then A else B*



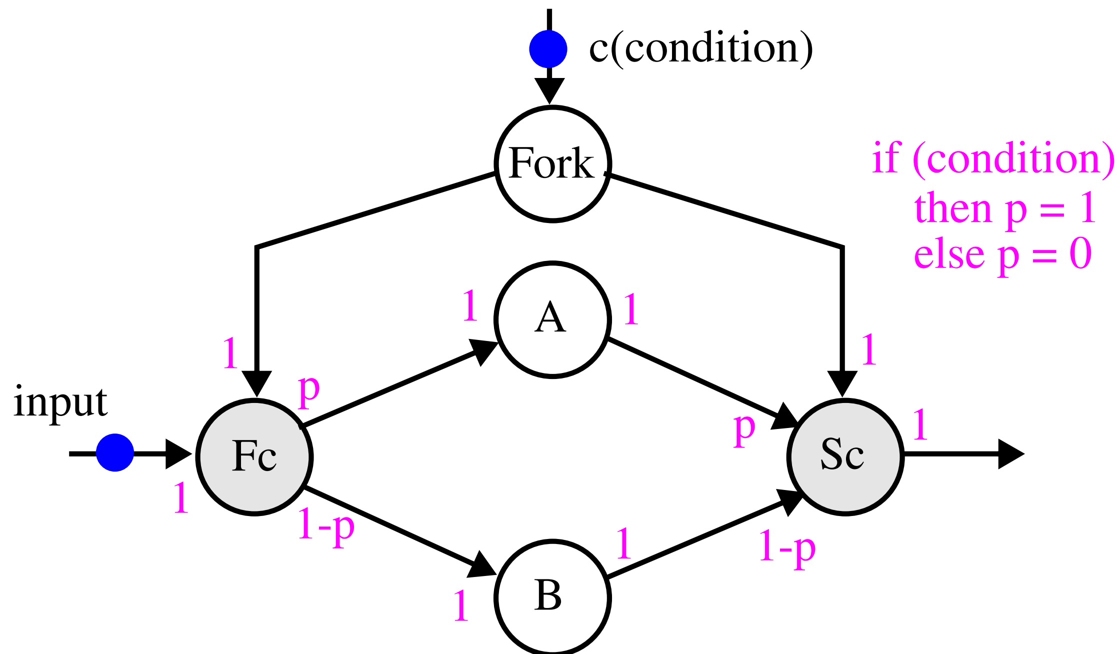
The *selector-actor* on the right chooses either *A* or *B* to output

But note that this does NOT model the *if-then-else* in, for example, C because BOTH the *if* branch (*A*) and the *else* (*B*) must execute

This approach models a *multiplexer* approach in hardware

Control Flow Modeling: Limitations of Data-Flow Models

Solution 2: *extend* SDF semantics -- *Boolean Data Flow* (BDF)



BDFs make the *production* and *consumption* rate of a token **dependent** on the value of an external control token

The *condition* token is distributed to two BDF **conditional fork** and **merge** nodes, *Fc* and *Sc*

Control Flow Modeling: Limitations of Data-Flow Models

The rules are that the *conditional fork* will fire when there is an *input* token AND a *condition* token

A token is produced on EITHER the upper or lower edge, dependent on the *condition* token

- This is indicated by the variable *p* -- a conditional production rate -- which can ONLY be determined at runtime

The *conditional merge* works similarly -- it fires when there is a *condition* token and will consume a token on EITHER the upper or lower edge

Unfortunately, using BDF jeopardizes the basic properties of SDFs

- For example, we now have data-flow graphs that are *conditionally admissible*

- Also, the topology matrix now includes symbolic values, *p*, and become quickly **impractical** to analyze
  - For a graph with 5 conditions, we would have a matrix with 5 symbols or expand the single matrix into 32 variants -- one for each combination

Control Flow Modeling: Limitations of Data-Flow Models

Beyond BDF, other flavors of *control-oriented* data-flow graphs have been proposed, such as:

- Dynamic Data Flow (DDF) which allows variable production and consumption rates
- Cyclo-Static Data Flow (CSDF) which allows a fixed, iterative variation on production and consumption rates

Unfortunately, these extensions reduce the elegance of SDF graphs

SDF remains very popular for modeling in DSP applications

BDF, DDF, etc. have **not** enjoyed widespread acceptance as alternatives

Reading Guide

- Section 2.1 – 2.3, the *CoDesign* book.