EC 535: Introduction to Embedded Systems

Dataflow Modeling and Implementation - Continued

Dataflow

- General theme: model the *flow of data* in processes.
- Data flow graph: a directed acyclic graph that shows data dependencies.
- Applications: signal processing, distributed systems, computer architecture, software, etc.

SDF Load Balancing

- How many times should we execute each actor so that all of the intermediate tokens that are produced get consumed?
- *Load balancing* is implemented by an algorithm that is linear in time and memory with the size of the SDF graph.

- To load balance the SDF in previous slide, we must:
    - Fire (execute) A 1 time
    - Fire B 2 times
    - Fire C 4 times

```
A   ------------>  B   ------------>  C
  20           10    20           10
```

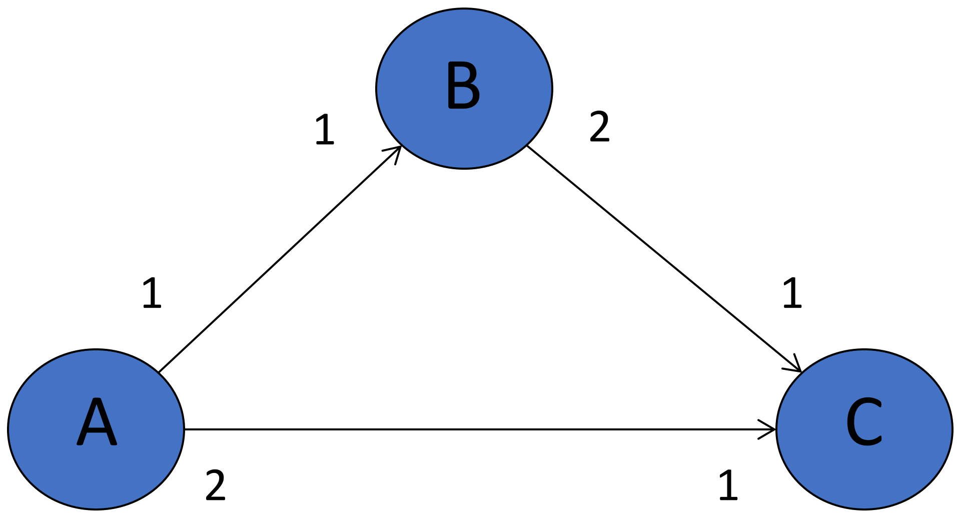
- Load balancing algorithms:
    - List and loop scheduling

A Brief Summary So Far

- An SDF graph expresses a concurrent system in which individual behaviors are captured inside of **actors** that communicate using **FIFO communication queues**.
- A schedule of an SDF graph is the execution order in which actors can execute (read tokens from input FIFO's, transform them with computation, and write the result to output FIFO's).
- An **admissible** SDF graph is one that does not cause deadlock or an infinite amount of tokens on a FIFO. Such a system can be statically scheduled.
- A periodic admissible schedule is called a **PASS.**
- SDF is convenient to describe regular data-processing. However, not all concurrent systems can be captured in SDF. It is not possible to express non-determinate behavior.

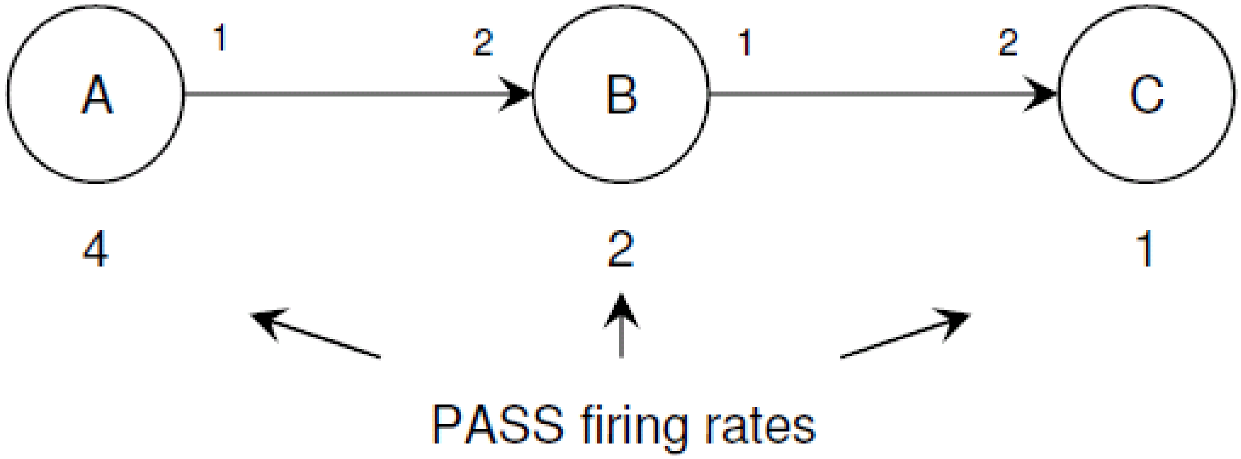
Production/Consumption Matrix

- What is the production/consumption (topology) matrix of the following system?
- Write down the load balance equation (don't have to solve).



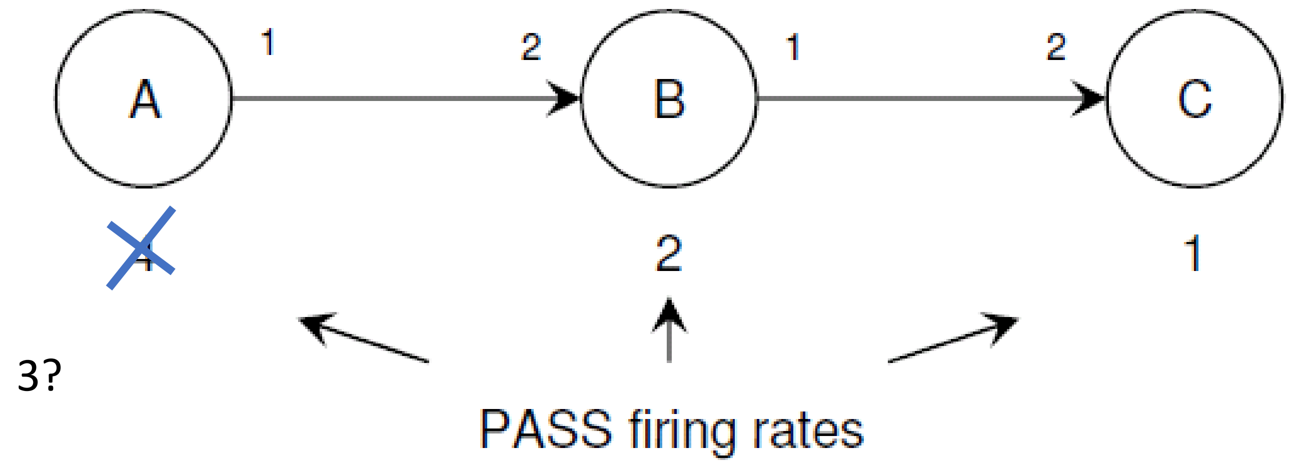
System Schedule

- When you write the system schedule of the entire system, you still have to take the relative firing rates into account.



System Schedule

- When you write the system schedule of the entire system, you still have to take the relative firing rates into account.



Euclid’s GCD

```
while (b != 0) {
   int temp = b;
   b = a % b;   // remainder
   a = temp;
}
```

vs.

```
while (a != b) {
   if (a > b)
      a = a - b;
   else
      b = b - a;
}
```

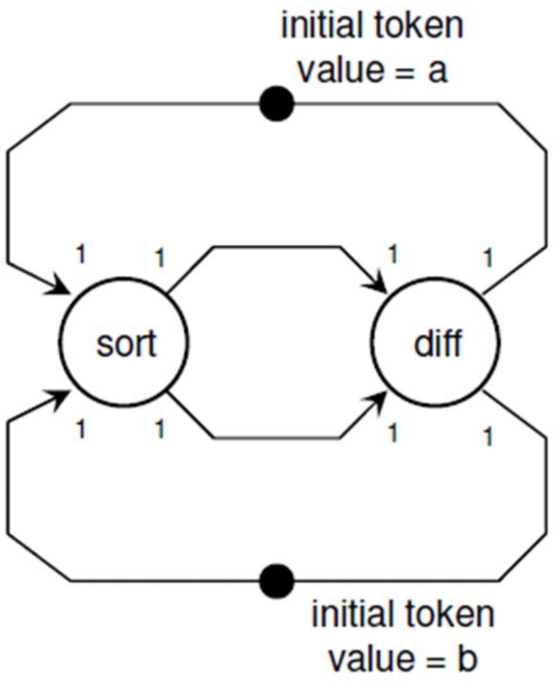
Euclid’s GCD

sort

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

diff

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



Topology Matrix:

$$
:\left[\begin{array}{cc} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \end{array}\right]\begin{array}{l} edge(sort, diff) \\ edge(sort, diff) \\ edge(diff, sort) \\ edge(diff, sort) \end{array}
$$

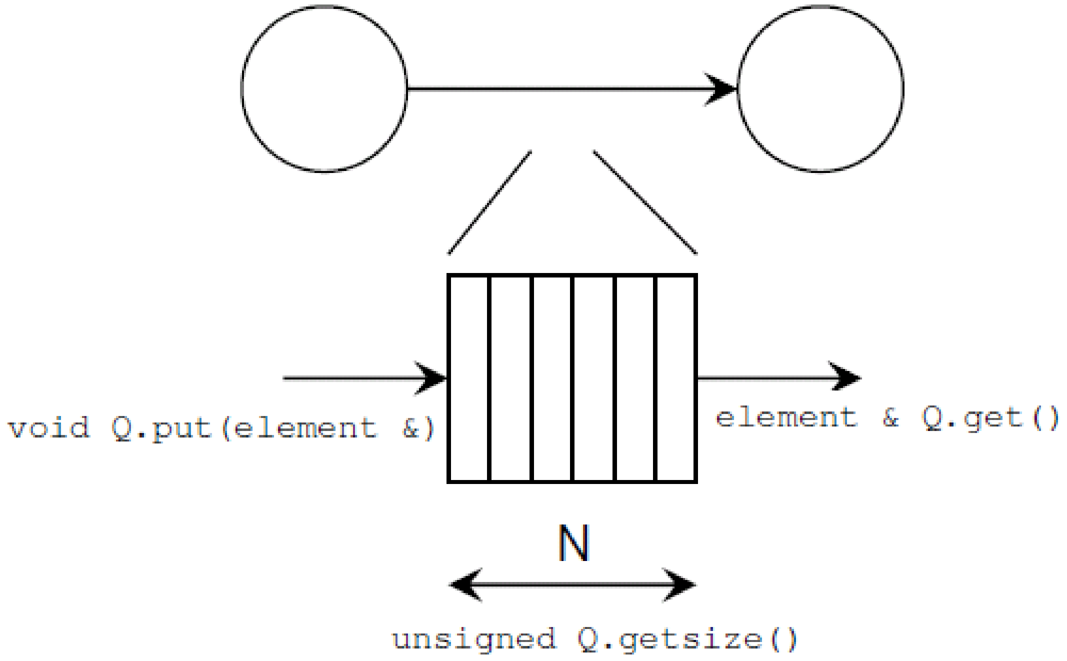
- Rank of the matrix?
- PASS condition:

Rank(T)=nodes-1

SW Implementation of Dataflow

- The typical software interface of a **FIFO queue** has two parameters and three methods:

1. The number of elements N that can be stored by the queue. (parameter)
2. The data type *element* of a queue of elements. (parameter)
3. A method to ***put*** elements into the queue.
4. A method to ***get*** elements from the queue.
5. A method to ***test*** the number of elements in the queue.



SW Implementation of the Euclid GCD

The actors are interconnected in the main program through queues. The main program executes the PASS in the form of a while loop.

```
void main() {
  fifo_t F1, F2, F3, F4;
  actorio_t sort_io, diff_io;
  sort_io.in1  = &F1;
  sort_io.in2  = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4;
  diff_io.in1  = &F3;
  diff_io.in2  = &F4;
  diff_io.out1 = &F1;
  diff_io.out2 = &F2;

  // initial tokens
  put_fifo(&F1, 16);
  put_fifo(&F1, 12);

  // system schedule
  while (1) {
    sort_actor(&sort_io);
    diff_actor(&diff_io);
  }
}
```

Optimization for Embedded Systems

1. Replacing FIFO queues with variables:

```
loop {
        ...
        F1.put(value1);
        F1.put(value2);
        ...
        .. = F1.get();
        .. = F1.get();
}
```

$$\longrightarrow$$

```
loop {
...
r1 = value1;
r2 = value2;
...
.. = r1;
.. = r2;
}
```

2. Inlining actor code inside of the main program and the main scheduling loop. In combination with the above optimization, this will allow to drop the firing rules and to collapse an entire dataflow graph in a single function.

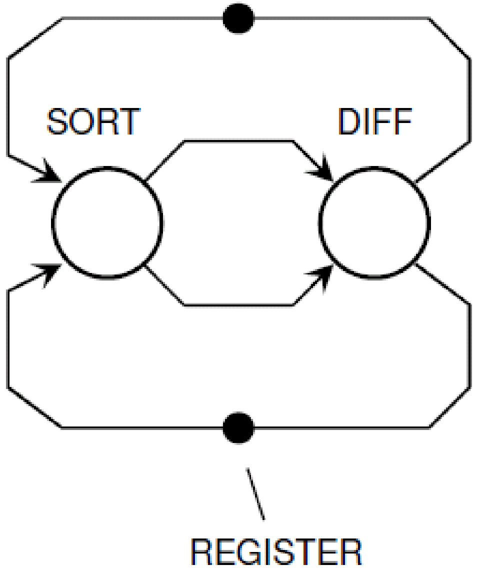
Optimized Euclid GCD Implementation

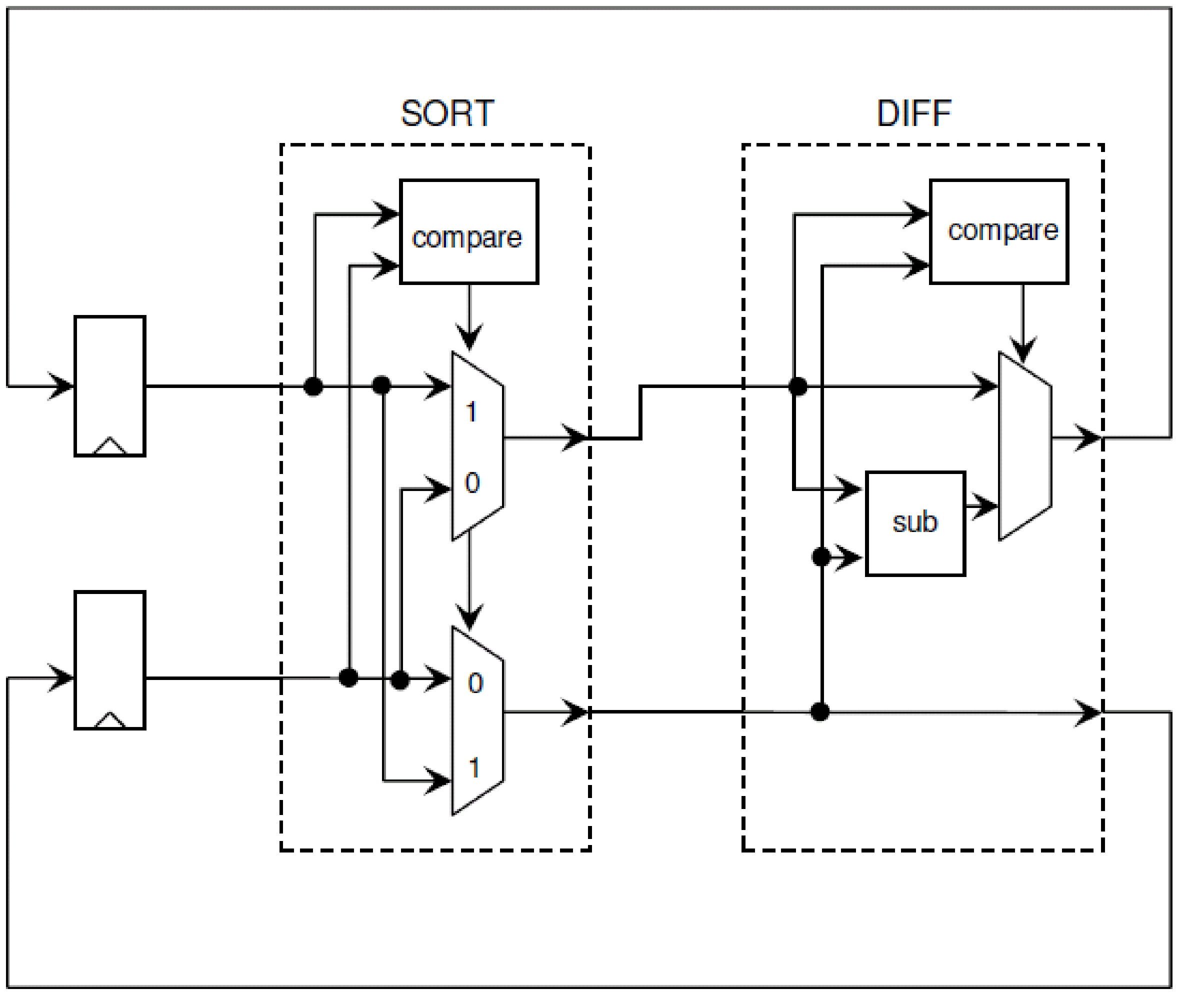
```
void main() {
    int f1, f2, f3, f4; // queues
    // initial token
    f1 = 16;
    f2 = 12;
    // system schedule
    while (1) {
            // code for actor 1
            f3 = (f1 > f2) ? f1 : f2;
            f4 = (f1 > f2) ? f2 : f2;
            // code for actor 2
            f1 = (f3 != f4) ? f3 - f4;
            f2 = f4;
    }
}
```

- We have dropped the following:
  - Testing of the firing rules
  - FIFO manipulations
  - Function boundaries

- This is possible because we have determined a valid PASS for the initial data-flow system, and we have chosen a fixed schedule to implement that PASS.

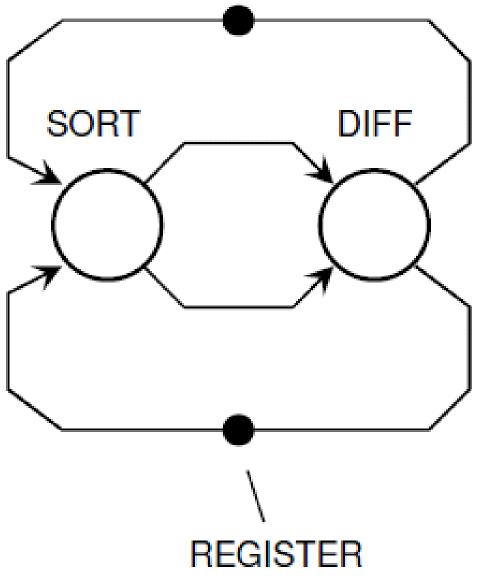
HW Implementation of Dataflow

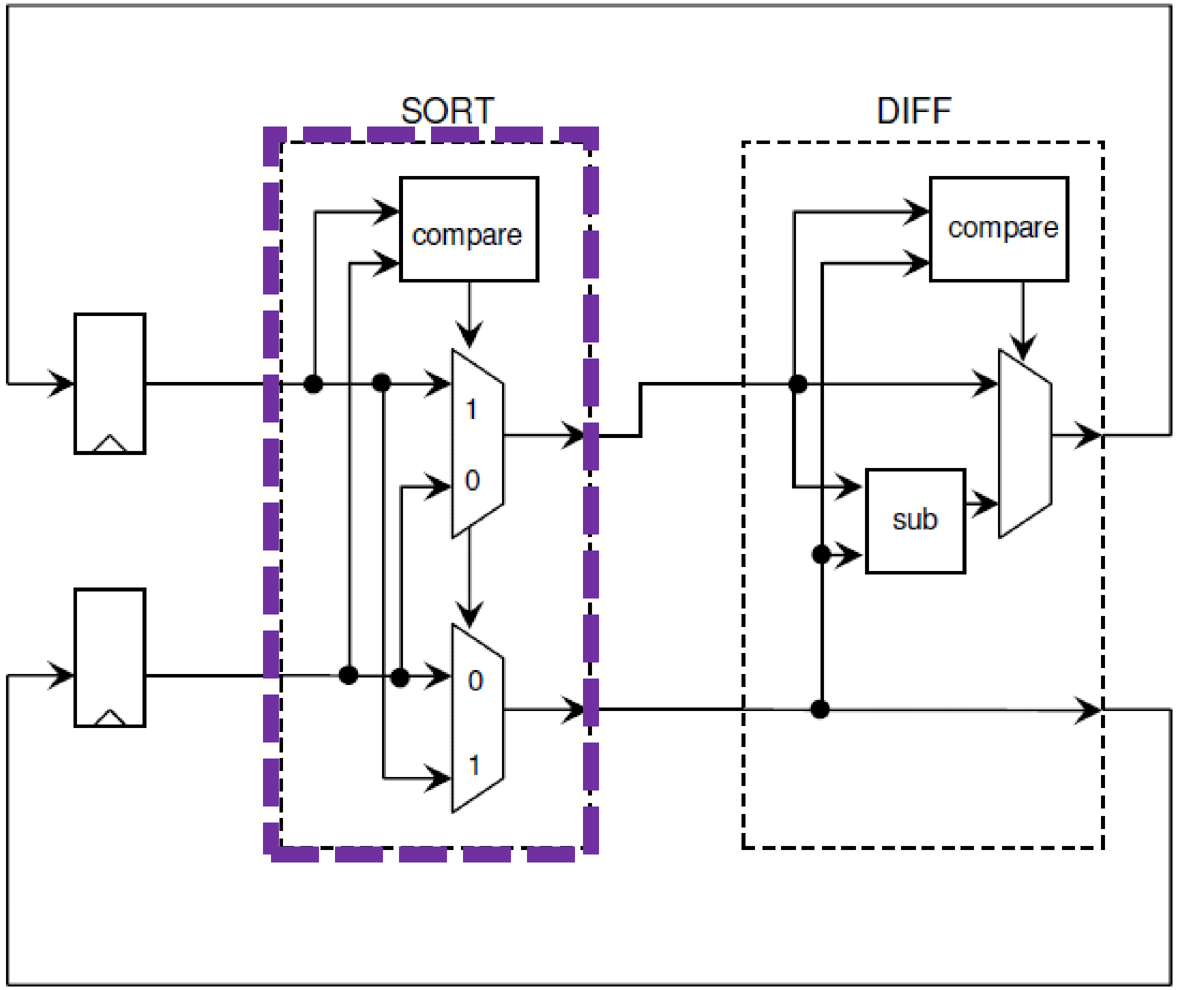




HW Implementation of Dataflow

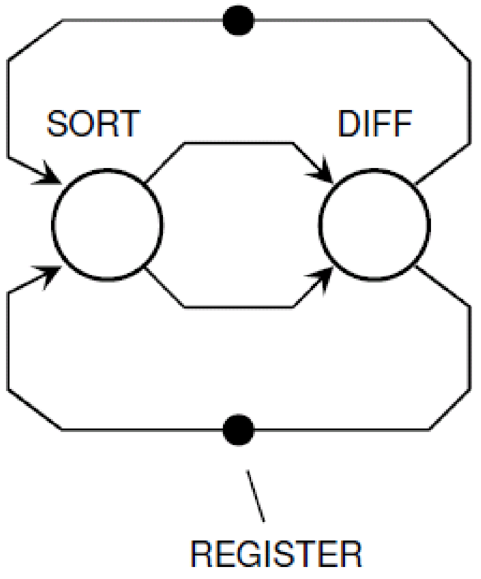
```
if (a > b) {
    out1 = a; out2 = b;
} else {
    out1 = b; out2 = a;
}
```

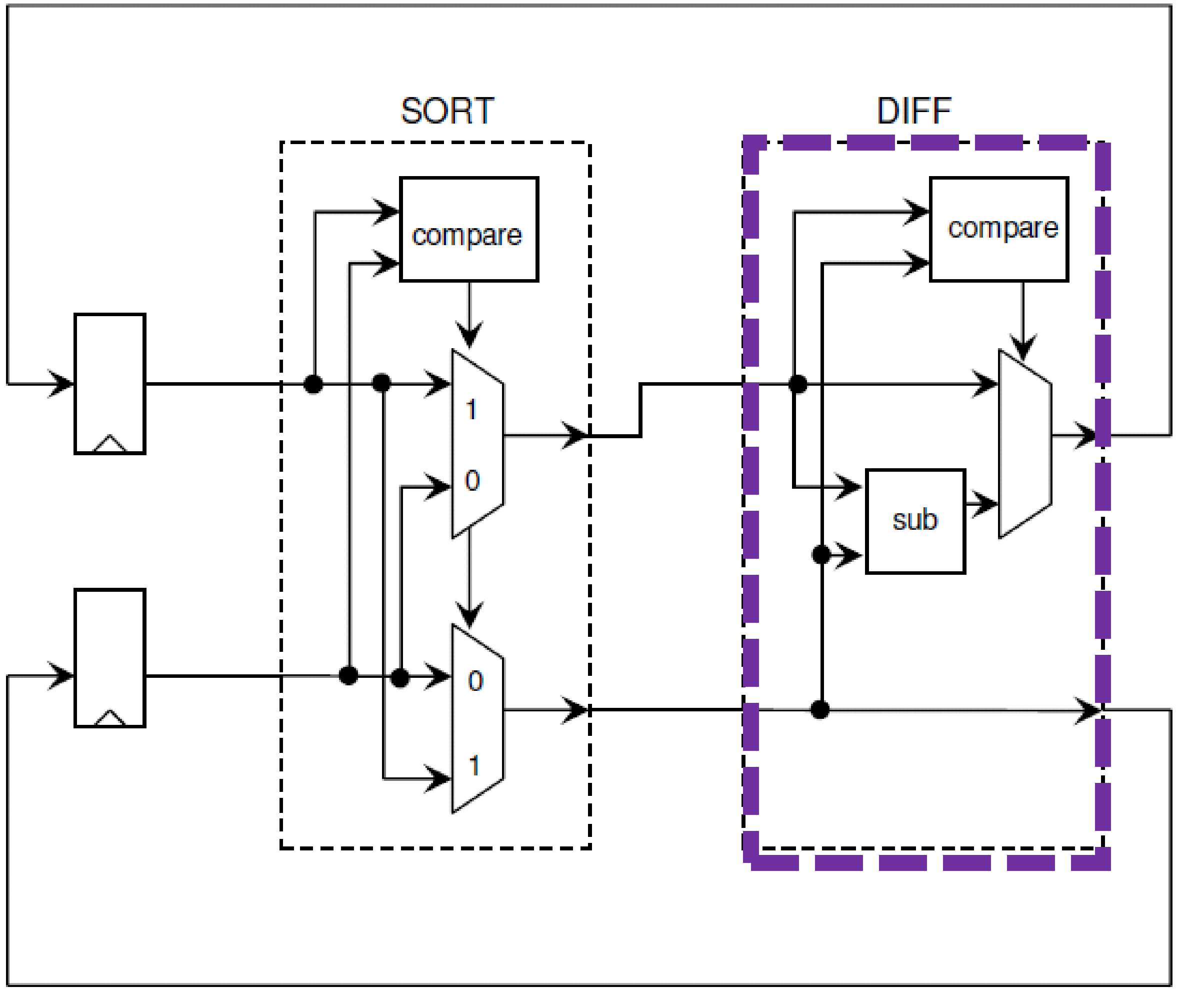




HW Implementation of Dataflow

```
if (a > b) {
   out1 = a; out2 = b;
} else {
   out1 = b; out2 = a;
}
```





Translation Procedure

1. Map each ***queue*** to a ***wire***.

Translation Procedure

1. Map each ***<u>queue</u>*** to a ***<u>wire</u>***.
2. Map each queue containing a ***<u>token</u>*** to a ***<u>register</u>***. The initial value of the register must equal the value of the initial token.

Translation Procedure

1. Map each ***queue*** to a ***wire***.
2. Map each queue containing a ***token*** to a ***register***. The initial value of the register must equal the value of the initial token.
3. Map each actor to a ***combinational circuit***, which completes a firing within a clock cycle. Both the sort and diff actors require no more than a comparator module, a few multiplexers and a subtractor.

Translation Procedure

1. Map each ***<u>queue</u>*** to a ***<u>wire</u>***.
2. Map each queue containing a ***<u>token</u>*** to a ***<u>register</u>***. The initial value of the register must equal the value of the initial token.
3. Map each actor to a ***<u>combinational circuit</u>***, which completes a firing within a ***<u>clock cycle</u>***. Both the sort and diff actors require no more than a comparator module, a few multiplexers and a subtractor.

Operations Limit Clockcycles

- register -> some logic -> register

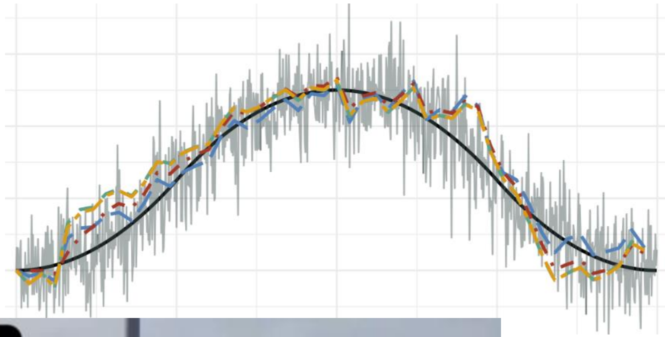
- You’d need to guess exactly when data is ready. But different logical operations can take different speeds.
- Timing bugs become nightmares.

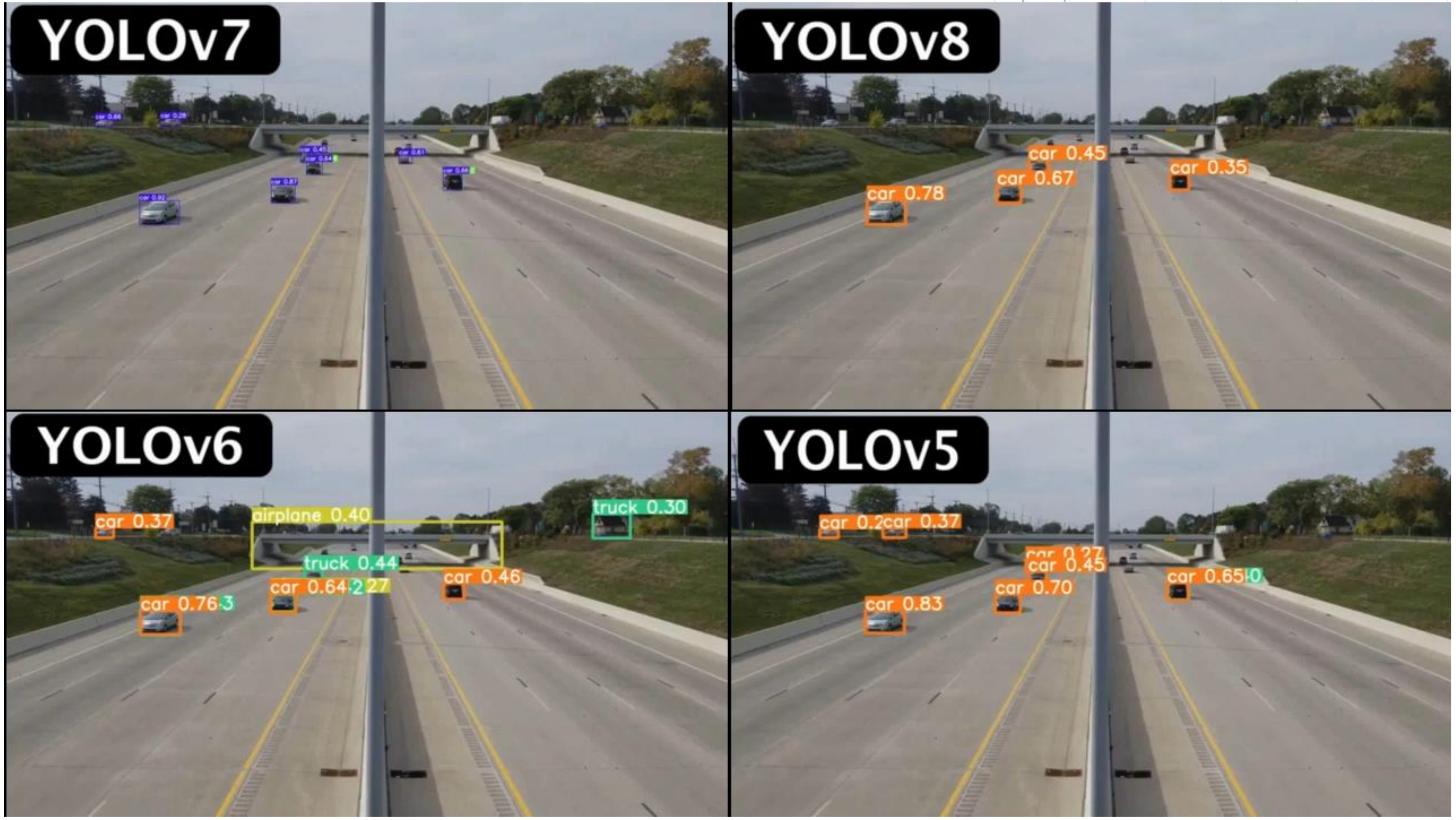
- So - Everything updates with the **clockcycle**

Issue of Long Combinational Paths

Temporal Filter Example

c2*x0 + c1*x1 + c0*x2





Issue of Long Combinational Paths

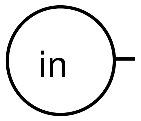
Digital filter with
c0, c1, c2
as weights

c2*x0 + c1*x1 + c0*x2

Issue of Long Combinational Paths

Digital filter with
c0, c1, c2
as weights

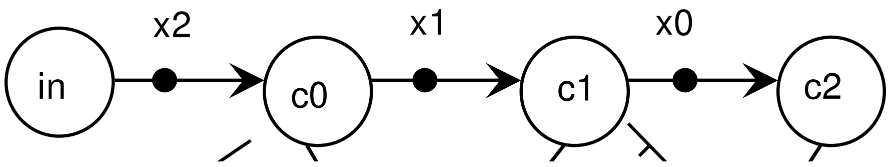
c2*x0 + c1*x1 + c0*x2



Issue of Long Combinational Paths

Digital filter with
c0, c1, c2
as weights

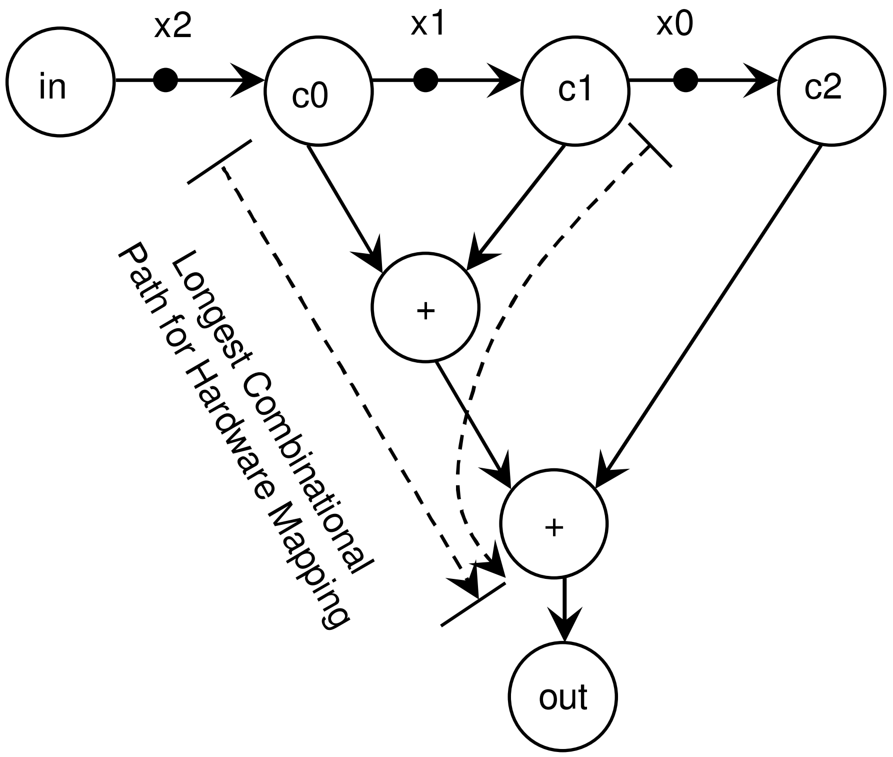
c2*x0 + c1*x1 + c0*x2



Issue of Long Combinational Paths

Digital filter with
c0, c1, c2
as weights

c2*x0 + c1*x1 + c0*x2

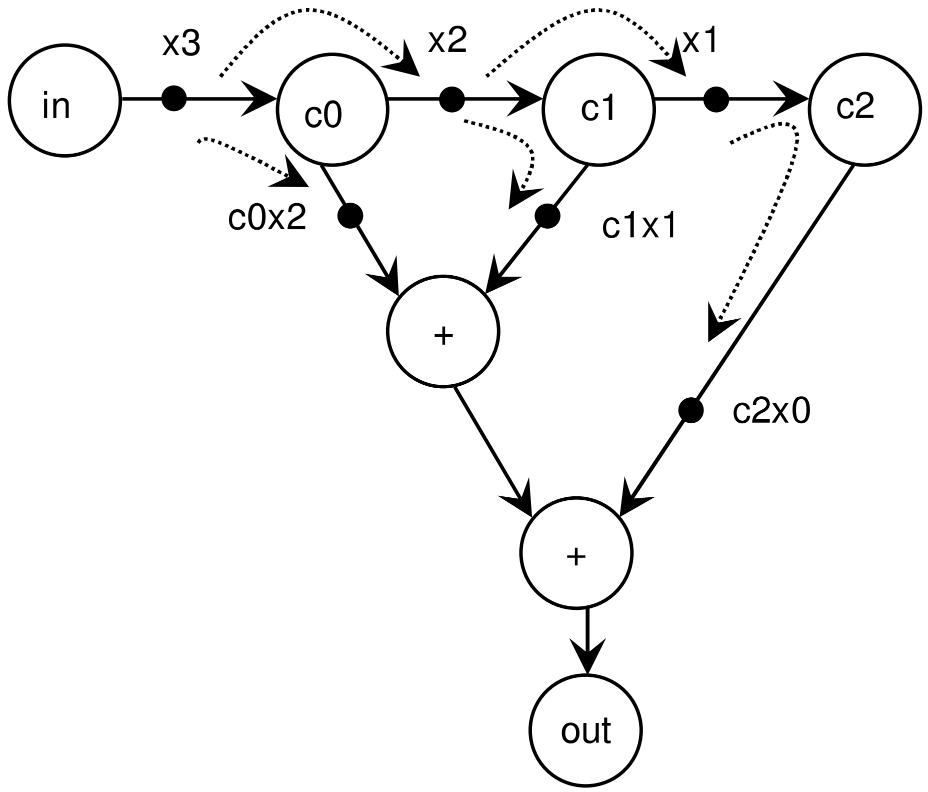


Hardware Optimization Methods

- Pipelining
  - Helps to break long combinational paths that may exist in dataflow circuits.
- Multi-rate expansion
  - Concerns multi-rate dataflow graphs.
  - It is possible to transform multi-rate graphs into single-rate dataflow graphs. These single-rate dataflow graphs then can be directly mapped into hardware circuits.

Pipelining Example

- “Push down” initial tokens into the adder tree.

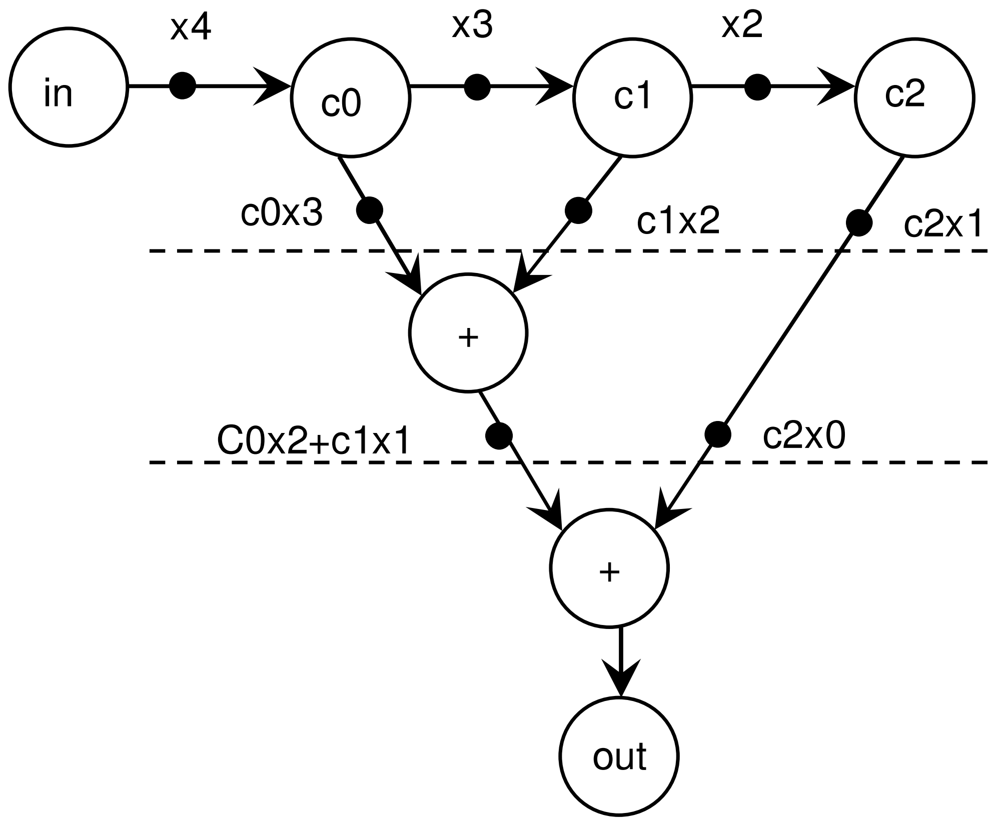


What is the longest combinational path?

Can we do better?

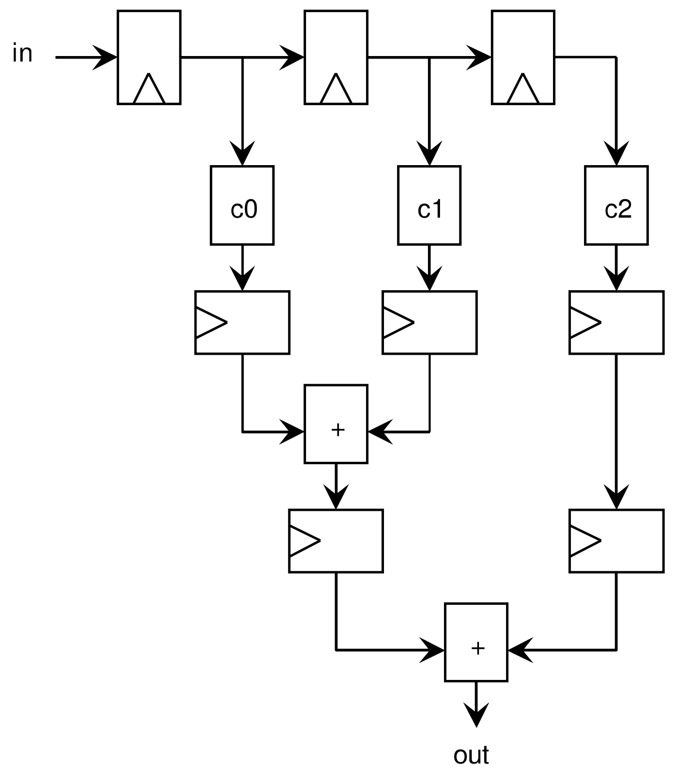
Pipelining Example

- "Push down" again:



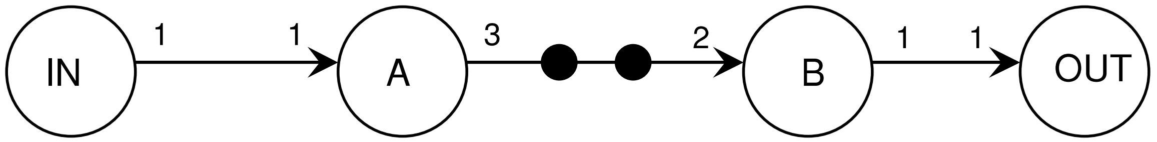
Pipelining Example

- Resulting pipelined implementation:

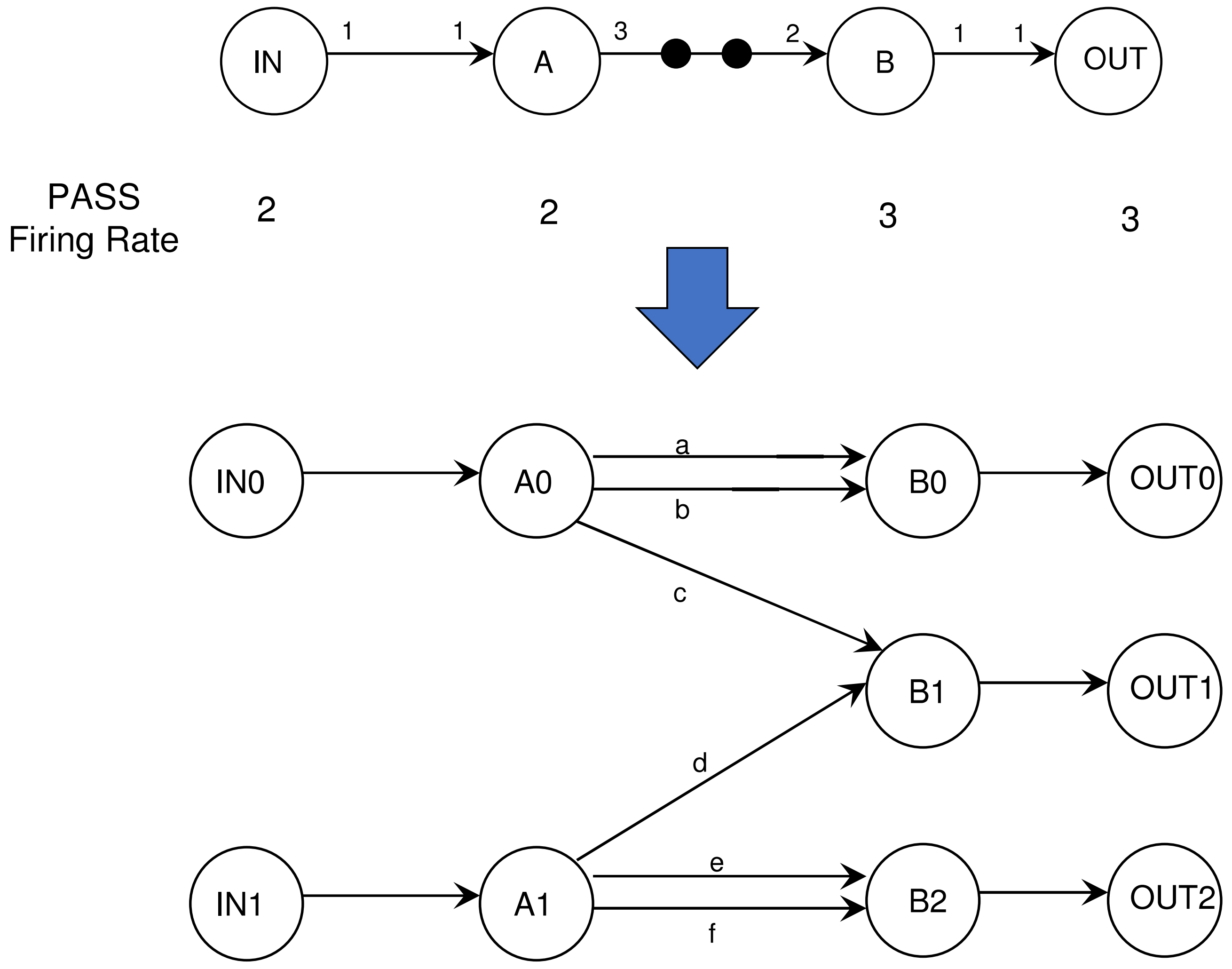


Multirate Example

- Actor A produces 3 tokens per firing, actor B consumes 2 tokens per firing.



Multirate Example



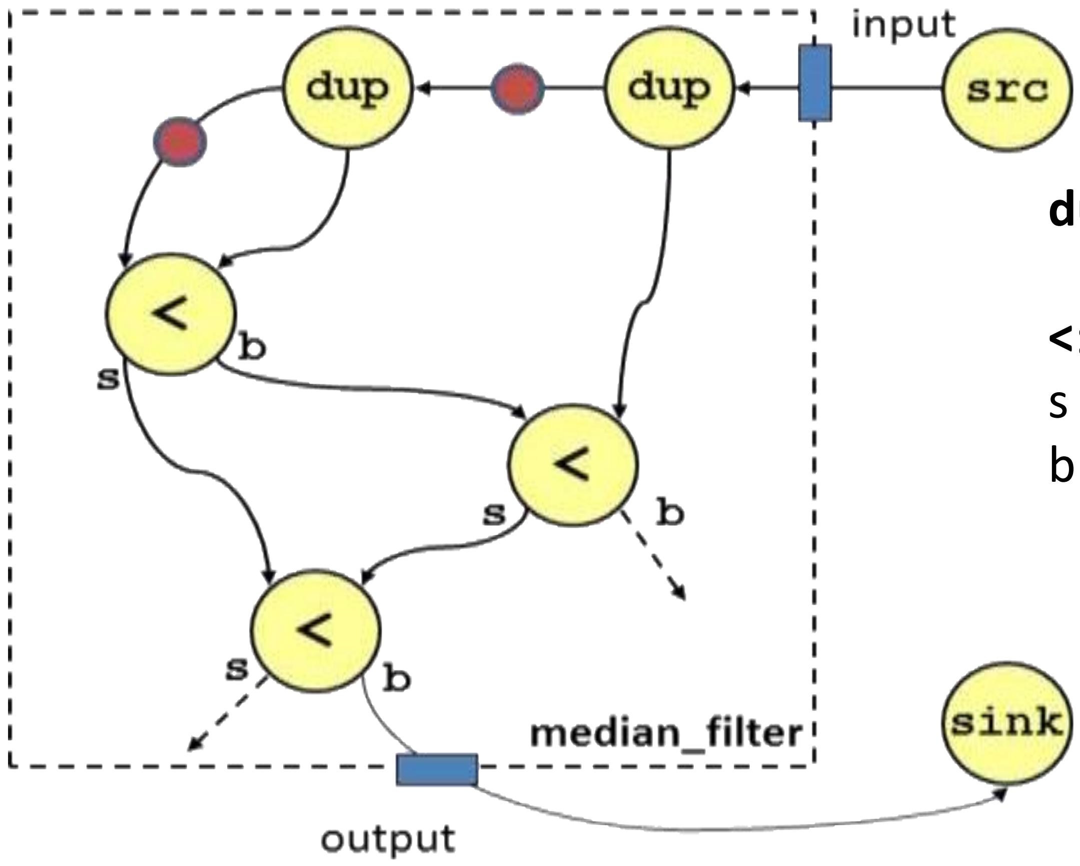
Multirate Expansion

- Convert a multi-rate graph to a single-rate graph:

1. Determine the ***PASS*** firing rates of each actor;
2. Duplicate each actor the number of times indicated by its firing rate. For example, given an actor A with a firing rate of 2, we create A0 and A1. These actors are identical;
3. Convert each multirate actor input/output to multiple single-rate input/outputs. For example, if an actor input has a consumption rate of 3, we replace it with three single-rate inputs;
4. Re-introduce the queues in the dataflow system to connect all actors. Since we are building a PASS system, the total number of actor inputs will be equal to the total number of actor outputs;
5. Re-introduce the initial tokens in the system, distributing them sequentially over the single-rate queues.

Another Example

- Map the SDF into hardware implementation so that the longest combinational path contains no more than one comparator.



dup: copy input to output

<: less than comparator;
s carries the smaller value and
b carries the bigger value.

Use as few registers as possible.