# EC 535: Introduction to Embedded Systems

Dataflow Modeling and Implementation

# Class Exam

- **4/15 Tuesday**

# Time-Aware Assistive Navigation

# TIMELI Benchmark

# Accessibility Requests

# Wrap-up from earlier lectures

- QEMU and Linux basics
- Kernel modules
- Embedded processor architectures
- RTOS
- Performance metrics, profiling, simulation
- Power & energy management

# Models of Computation

Actors: react to stimuli at input ports, produce stimuli on output ports

# 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.

# Synchronous Dataflow (SDF)

- First introduced: Edward A. Lee, 1986
- SDF:
  - All computation and data communication are scheduled statically.
  - Algorithms expressed as SDF graphs can always be converted into an implementation that is guaranteed to take **finite-time** to complete all tasks and use **finite memory**.
  - Periodic execution without additional resources
  - Application:
    - Digital signal processing and communications systems

# SDF Overview

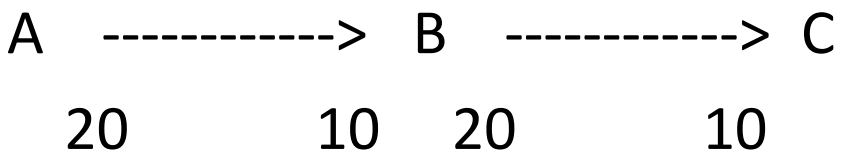
- Nodes and arcs
  - Nodes represent operations which are called *actors*.
  - Arcs represent data values called *tokens* which stored in first-in first-out (FIFO) queues.
    - Token: Each data value can represent any data type (e.g., integer or real) or any data structure (e.g., matrix or image).

# SDF Rules

- An actor is enabled for execution when enough tokens are available on all of the inputs, and source actors are always enabled.
- When an actor executes, it always produces and consumes the same fixed amount of tokens.
- The flow of data through the graph does not depend on values of the data.
- Delay is a property of an arc, and a delay of n samples means that n tokens are initially in the queue of that arc.

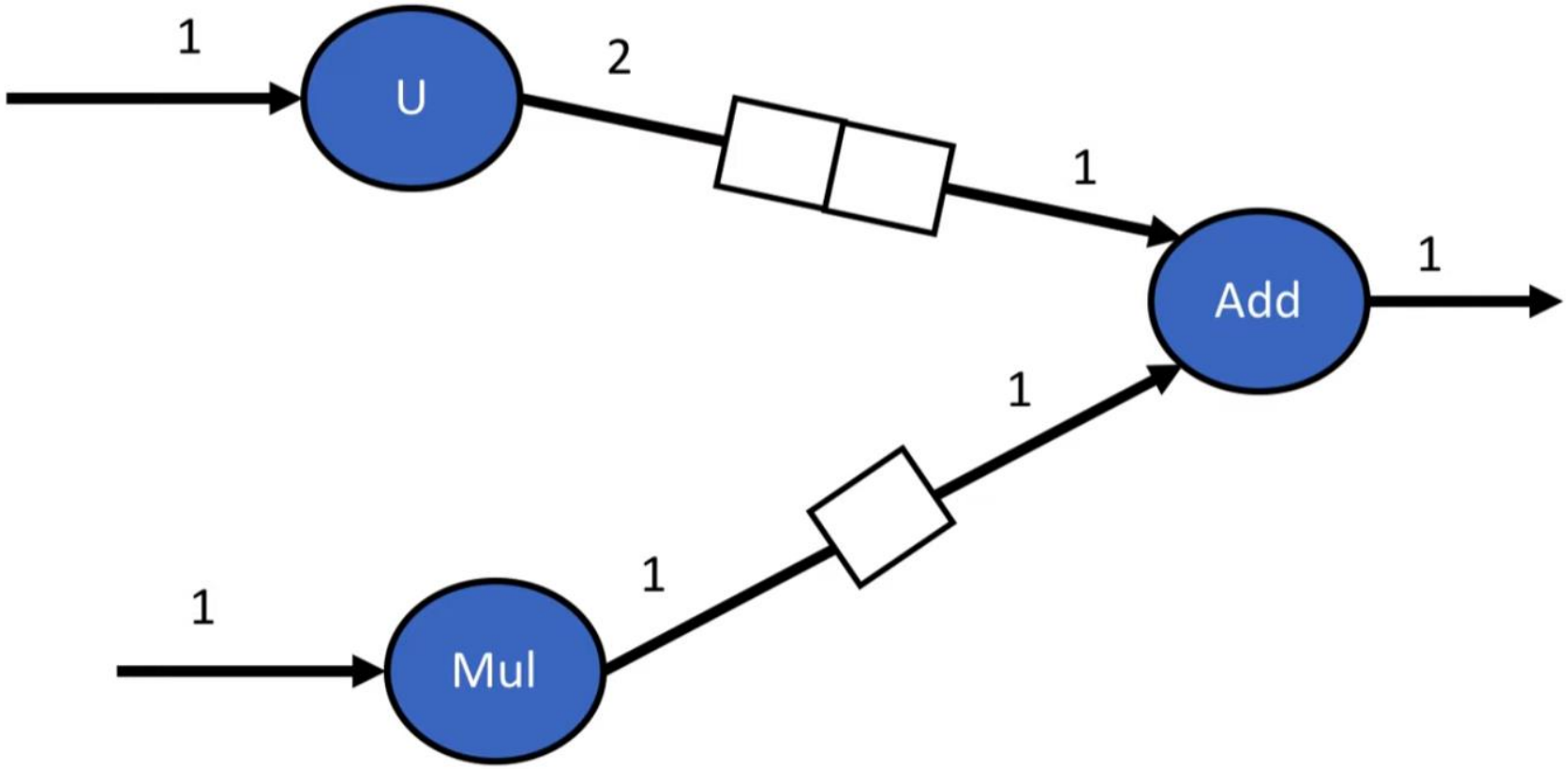
# Example

- Feed-forward (acyclic) SDF:

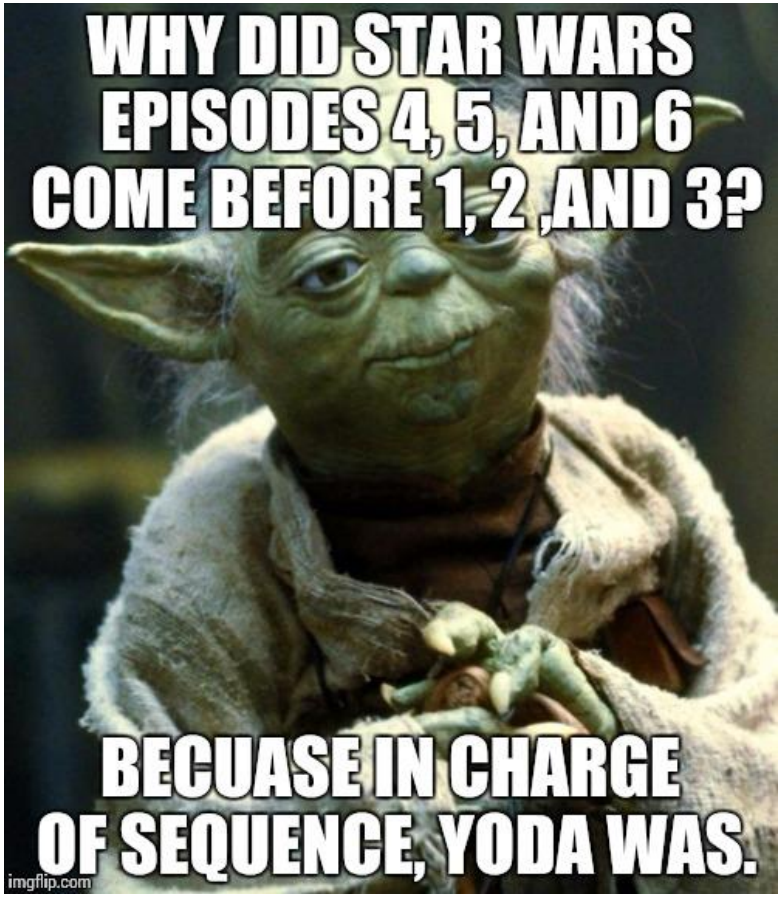



- When A executes, it produces 20 tokens.
- When B executes, it consumes 10 tokens and produces 20 tokens.
- When C executes, it consumes 10 tokens.

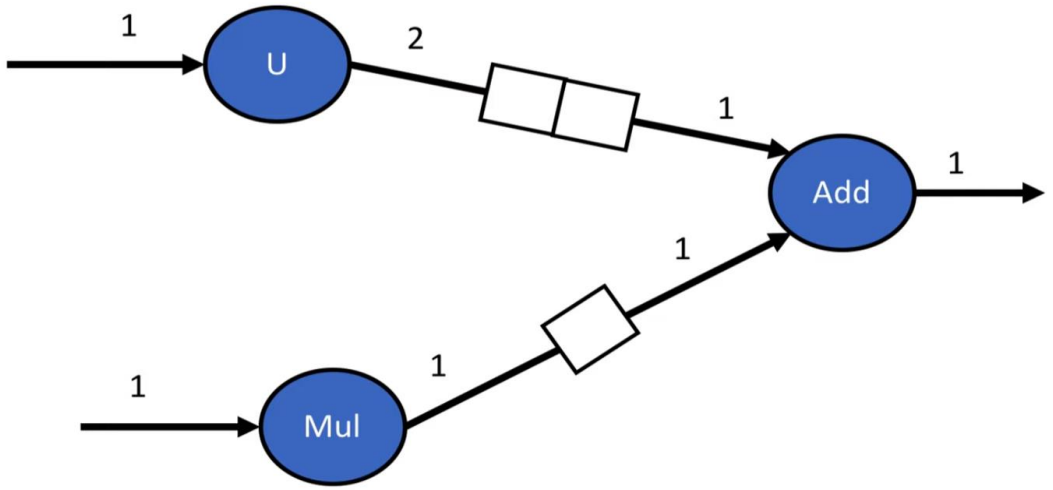
# Data Flow Graph Example



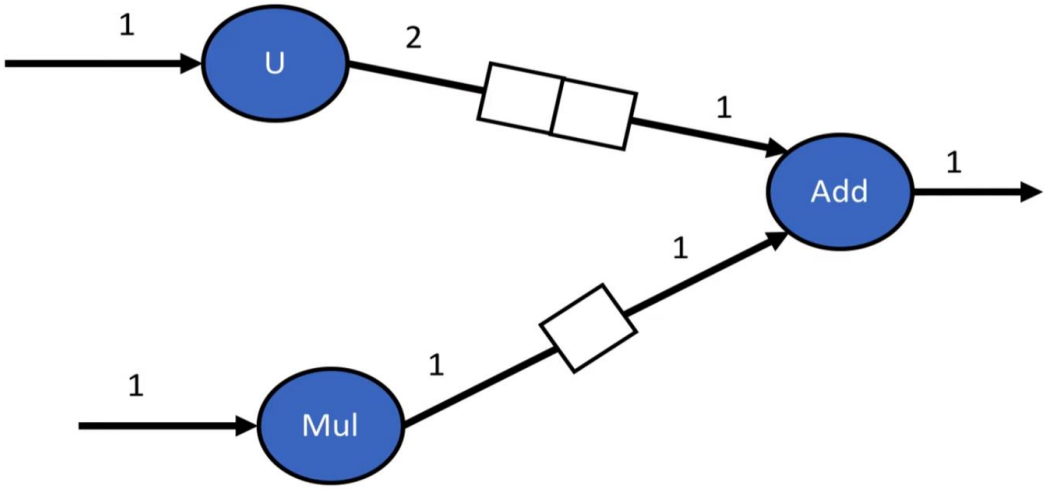
# How to Find the Sequence of Firings?



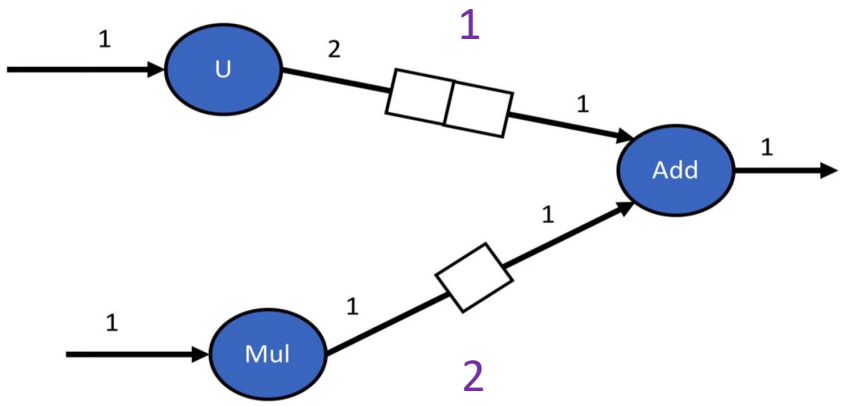
# Data Flow Graph Example



# Topology Matrix

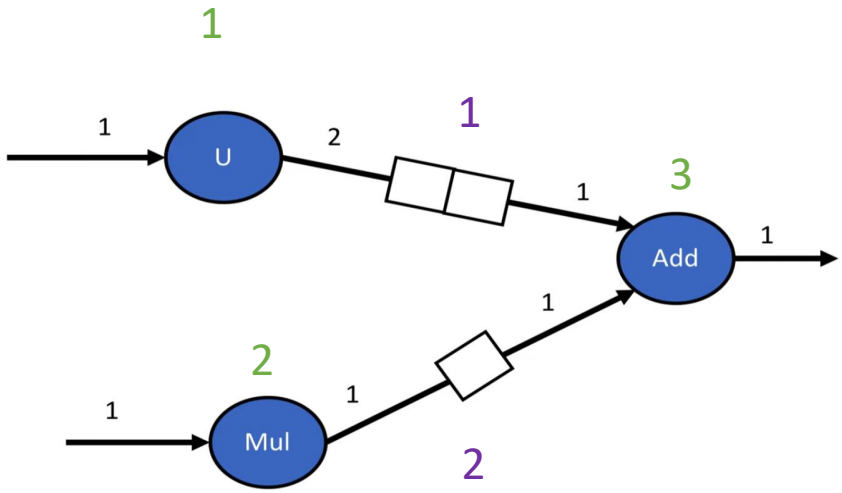


# Topology Matrix



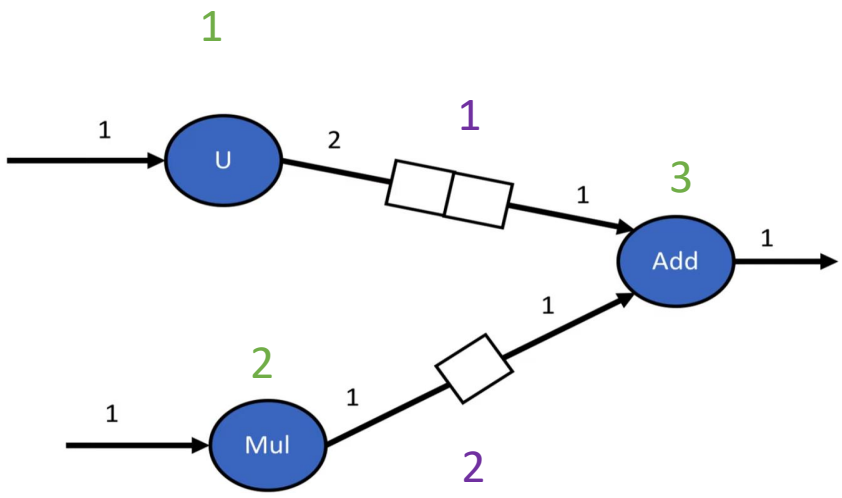
# Topology Matrix

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | | | |
| 2 | | | |



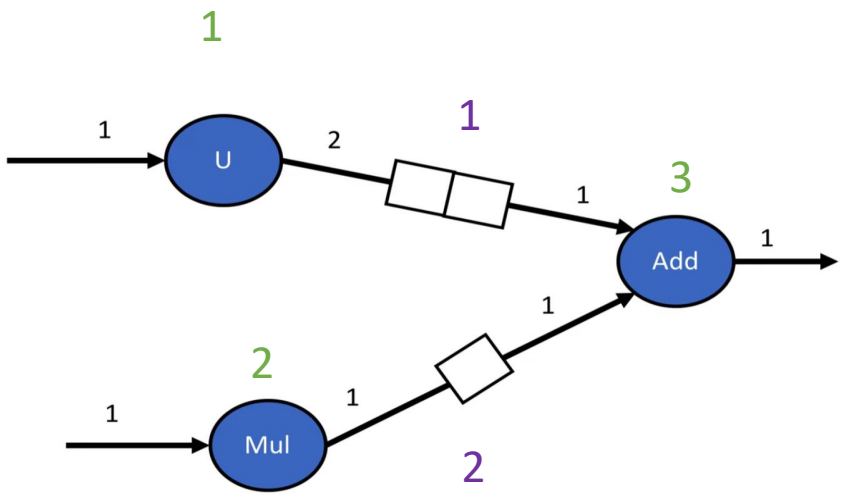
# Topology Matrix

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 2 | | |
| 2 | | | |



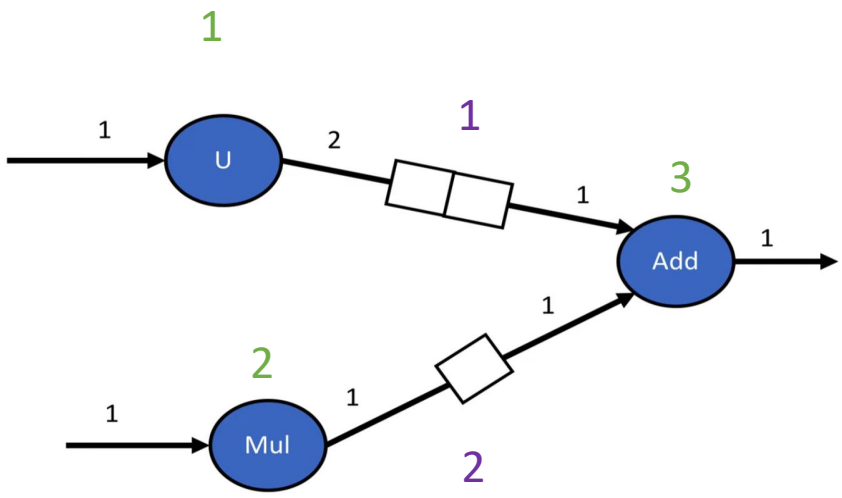
# Topology Matrix

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 2 | 0 | −1 |
| 2 | 0 | 1 | −1 |



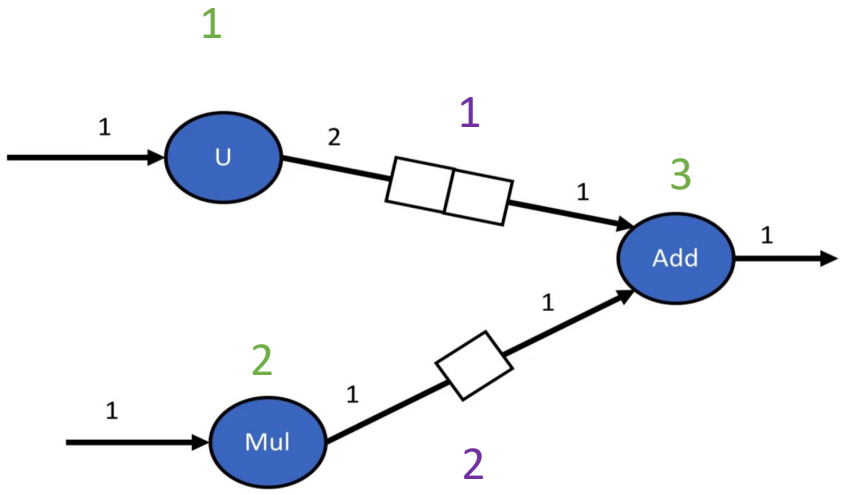
# Firing Vector

$$\begin{array}{c} \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \end{array} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$



# Firing Vector

$$\begin{array}{c} \\ 1 \\ 2 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \end{array} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$



# Firing Vectors

$$\begin{matrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{matrix} \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} + \begin{matrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{matrix} \begin{matrix} 1 \\ 0 \\ 0 \end{matrix}$$

# Firing Vectors

$$\begin{matrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{matrix} \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} + \begin{matrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{matrix} \begin{matrix} 1 \\ 0 \\ 0 \end{matrix}$$

T*v1 + T*v2 + …

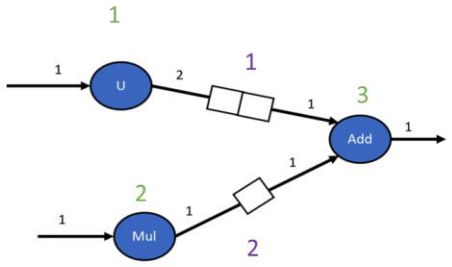
T*v = 0

# Firing Vectors

$$\begin{matrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{matrix} \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} + \begin{matrix} 2 & 0 & -1 \\ 0 & 1 & -1 \end{matrix} \begin{matrix} 1 \\ 0 \\ 0 \end{matrix}$$

T*v1 + T*v2 + …

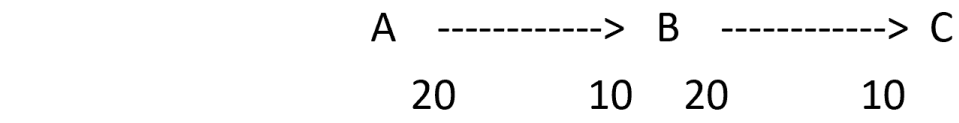
T*v = 0



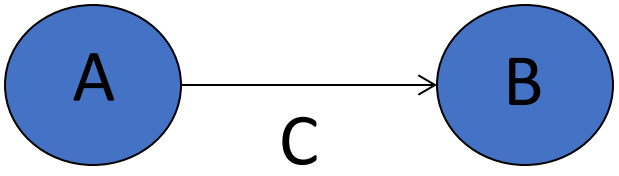
# SDF Load Balancing

- 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
- Load balancing algorithms:
  - List and loop scheduling



# Balance Equations




- Let $q_A$, $q_B$ be the number of firings of actors A and B.
- Let $p_C$, $c_C$ be the number of token produced and consumed on connection C.
- Then the system is *in balance* if for all connections C

$$q_A p_C = q_B c_C$$

# Admissible SDF Graphs

- 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.
  - An admissible SDF can be statically scheduled, i.e., one can find a **fixed schedule**.

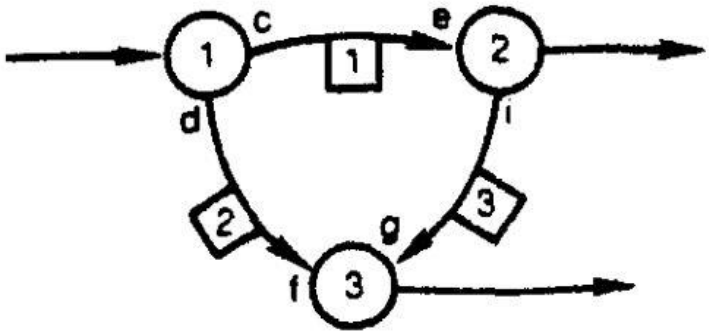
# Consistent Models

- Let N be the number of actors in a connected model. The model is *consistent (there is a PASS)* if the production/consumption (topology) matrix has rank N-1.

- If rank is N, then the balance equations have only a trivial solution.

- If rank is N-1, then the balance equations always have a non-trivial solution.

# Topology Matrix




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

- PASS condition
  - Rank must be #nodes-1

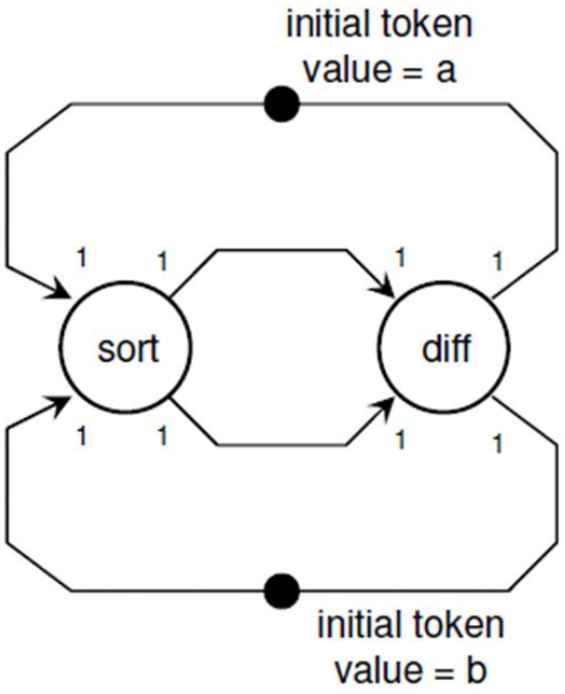
# Example Dataflow Graph: Euclid's Greatest Common Divisor

sort

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

diff

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




(a,b) = (16, 12)

(a,b) = (4,12)

(a,b) = (8,4)

(a,b) = (4,4)

(a,b) = (4,4)

→ GCD of 16 and 12 is 4.

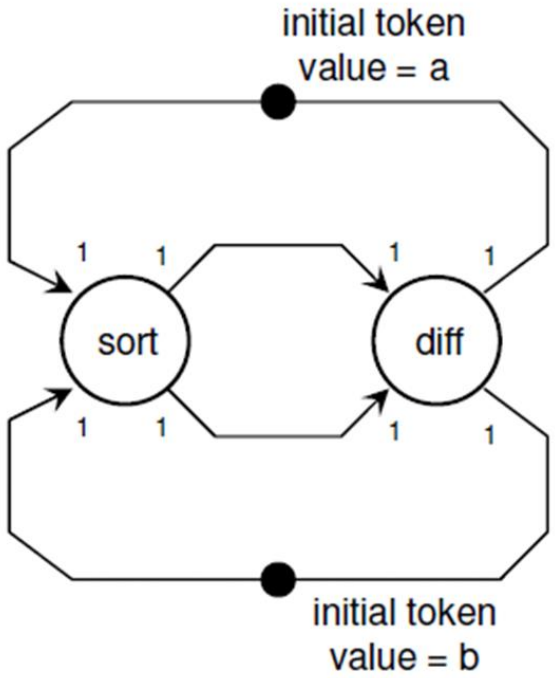
# Deriving a PASS

sort

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

diff

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




Topology Matrix:

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

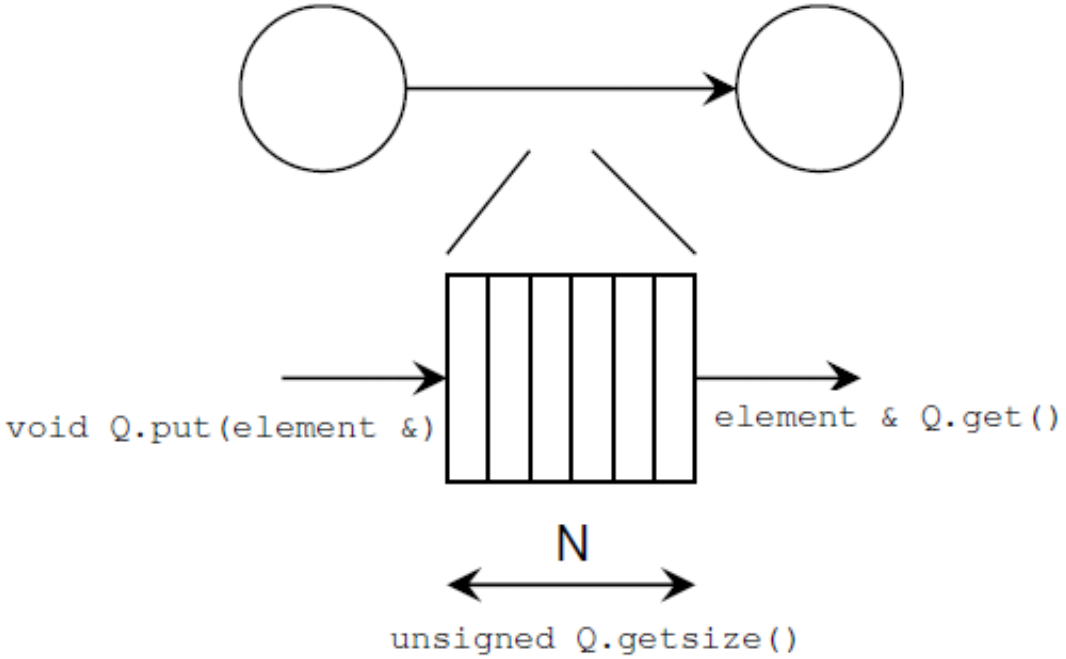
- Rank of the matrix?
- PASS condition:

  Rank(T)=nodes-1

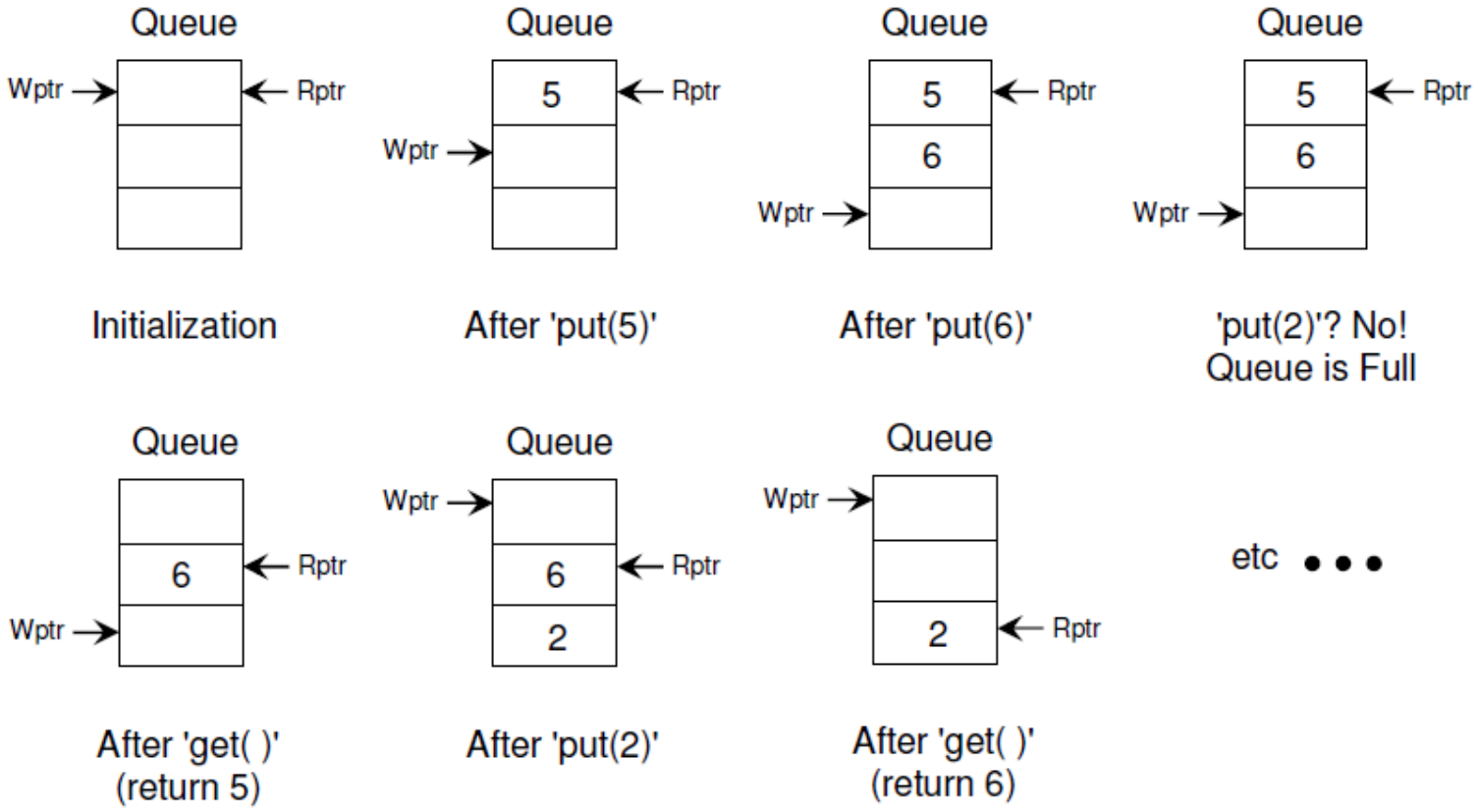
# SW Implementation of Dataflow

- 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.



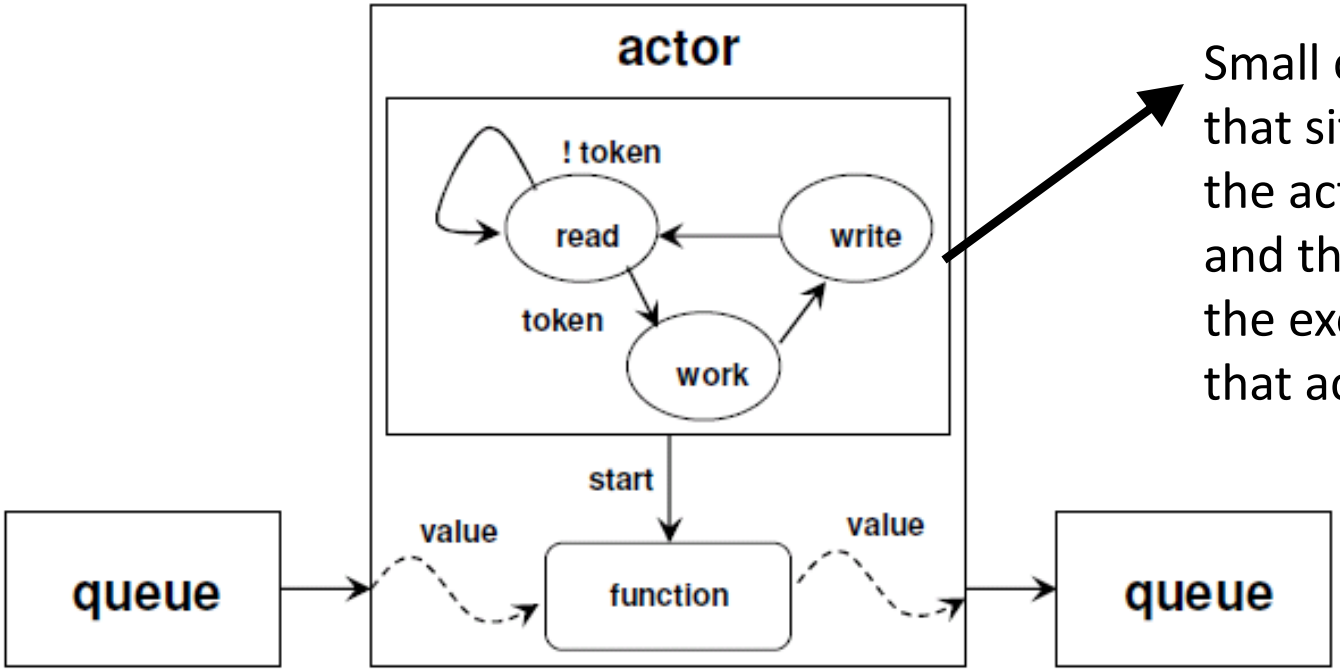
# Storage Organization: Circular Queue



# FIFO in C

```
typedef struct fifo {
  int data[MAXFIFO]; // array
  unsigned wptr;     // write pointer
  unsigned rptr;     // read pointer
} fifo_t;

void     init_fifo(fifo_t *F);
void     put_fifo (fifo_t *F, int d);
int      get_fifo (fifo_t *F);
unsigned fifo_size(fifo_t *F);

int main() {

  fifo_t F1;

  init_fifo(&F1);    // resets wptr, rptr;

  put_fifo(&F1, 5);  // enter 5
  put_fifo(&F1, 6);  // enter 6

  printf("%d %d\n", fifo_size(&F1), get_fifo(&F1)); // prints: 2 5
  printf("%d\n", fifo_size(&F1));                   // prints: 1
}
```

# Dataflow Actors




Small controller that sits on top of the actor function and that controls the execution of that actor

→ How about more complicated actors (e.g., more than one input, multi-rate, etc.)?

# Implementing actors using cooperative multithreading

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  while (1) {
      if ((fifo_size(g->in1) > 0) &&
          (fifo_size(g->in2) > 0)) {
        r1 = get_fifo(g->in1);
        r2 = get_fifo(g->in2);
        put_fifo(g->out1, (r1 > r2) ? r1 : r2);
        put_fifo(g->out2, (r1 > r2) ? r2 : r1);
      }
    stp_yield();
  }
}

void main() {

  fifo_t F1, F2, F3, F4;
  actorio_t sort_io;
  ..
  sort_io.in1  = &F1;
  sort_io.in2  = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4;                // connect queues
  stp_create(add_actor, &add_io);    // create thread
  stp_start();                       // start system schedule
}
```

# Implementing actors using cooperative multithreading

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  while (1) {
      if ((fifo_size(g->in1) > 0) &&
          (fifo_size(g->in2) > 0)) {
        r1 = get_fifo(g->in1);
        r2 = get_fifo(g->in2);
        put_fifo(g->out1, (r1 > r2) ? r1 : r2);
        put_fifo(g->out2, (r1 > r2) ? r2 : r1);
      }
    stp_yield();
  }
}

void main() {

  fifo_t F1, F2, F3, F4;
  actorio_t sort_io;
  ..
  sort_io.in1  = &F1;
  sort_io.in2  = &F2;
  sort_io.out1 = &F3;
  sort_io.out2 = &F4;                // connect queues
  stp_create(add_actor, &add_io);    // create thread
  stp_start();                       // start system schedule
}
```

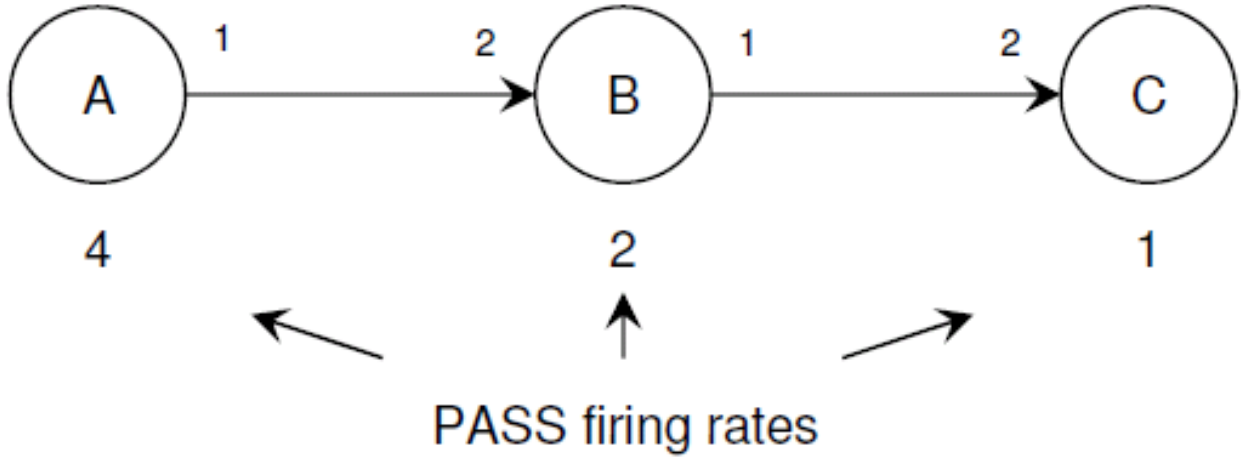
stp_create(sort_actor, &sort_io)

# Implementation without threads

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
      (fifo_size(g->in2) > 0)) {
      r1 = get_fifo(g->in1);
      r2 = get_fifo(g->in2);
      put_fifo(g->out1, (r1 > r2) ? r1 : r2);
      put_fifo(g->out2, (r1 > r2) ? r2 : r1);
  }
}

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

   while (1) {
     sort_actor(&sort_io);
     // .. call other actors
   }
}
```

# System Schedule

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



# SW Implementation of the Euclid GCD

A valid PASS fires each node a single time. Below is a description for each of the actors (sort, diff):

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
      (fifo_size(g->in2) > 0)) {
     r1 = get_fifo(g->in1);
     r2 = get_fifo(g->in2);
     put_fifo(g->out1, (r1 > r2) ? r1 : r2);
     put_fifo(g->out2, (r1 > r2) ? r2 : r1);
  }
}

void diff_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
      (fifo_size(g->in2) > 0)) {
     r1 = get_fifo(g->in1);
     r2 = get_fifo(g->in2);
     put_fifo(g->out1, (r1 != r2) ? r1 - r2 : r1);
     put_fifo(g->out2, r2);
  }
}
```

# SW Implementation of the Euclid GCD

A valid PASS fires each node a single time. Below is a description for each of the actors (sort, diff):

```
void sort_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
      (fifo_size(g->in2) > 0)) {
    r1 = get_fifo(g->in1);
    r2 = get_fifo(g->in2);
    put_fifo(g->out1, (r1 > r2) ? r1 : r2);
    put_fifo(g->out2, (r1 > r2) ? r2 : r1);
  }
}

void diff_actor(actorio_t *g) {
  int r1, r2;
  if ((fifo_size(g->in1) > 0) &&
      (fifo_size(g->in2) > 0)) {
    r1 = get_fifo(g->in1);
    r2 = get_fifo(g->in2);
    put_fifo(g->out1, (r1 != r2) ? r1 - r2 : r1);
    put_fifo(g->out2, r2);
  }
}
```

# 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();
}
```

```
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.

# SDF review

- What is a synchronous data flow graph?
  - Why is it useful?
  - Disadvantages?
- Load balancing
  - Why do we need it?
  - What are some of the algorithms?
- What is an “admissible” graph?
- What is “PASS”?