Introduction to Embedded Systems
EHB326E
Lectures

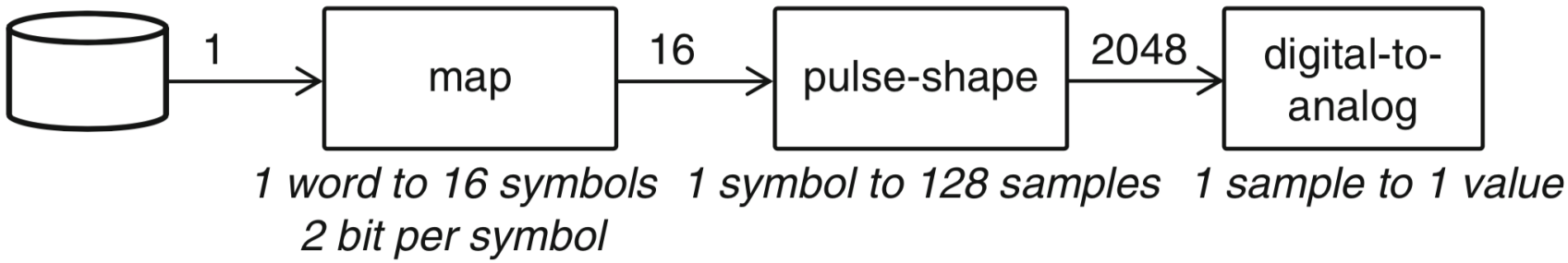
Prof. Dr. Müştak E. Yalçın

Istanbul Technical University

mustak.yalcin@itu.edu.tr

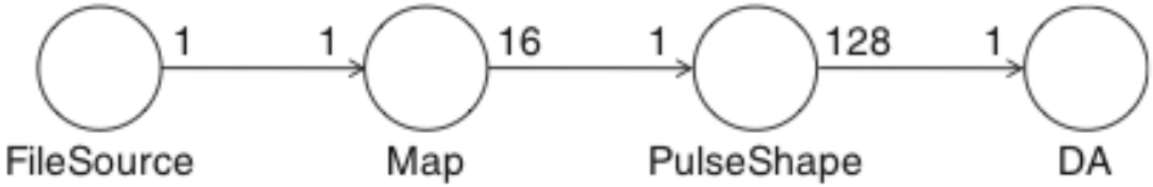
Data Flow Modeling and Implementation

Input Output Functionality of Digital Signal Processing System is described using block diagrams ! A block diagram only shows a chain of signal processing algorithms and the data samples to send to each other. Pulse-Amplitude Modulation (PAM) system



$[01110010....]_{32}$ –> $[-1\ 3\ -3\ 1\ ...]_{16}$ –> $[0...030...0]_{128}$ –> *h(n)–> DAC
Chapter 2, A Practical Introduction to Hardware/Software Codesign, Patrick R. Schaumont

The block diagram only specifies the flow of data in the system but not the execution order of the functions.




```
extern int read_from_file();
extern int map_to_symbol(int, int);
extern int pulse_shape(int, int);
extern void send_to_da(int);
int main() {
int word, symbol, sample;
int i, j;
while (1) {
          word = read_from_file();
          for (i=0; i < 16; i++) {
               symbol = map_to_symbol(word, i);
               for (j=0; j<128; j++)
                    sample = pulse_shape(symbol, j);
                    send_to_da(sample); }
                   }
              }
}
```

a sequential implementation, but it does not encourage a parallel implementation

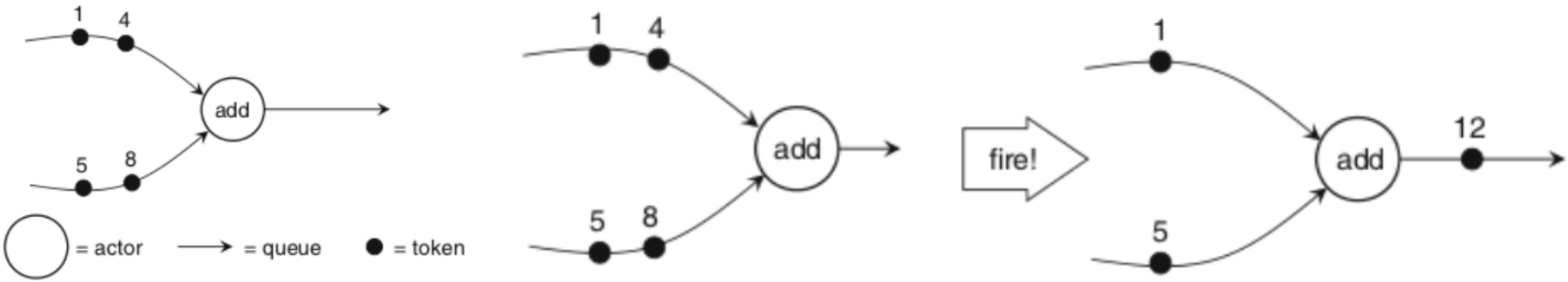
The major differences of this modeling style compared to modeling in C.

- A Data Flow model is a concurrent model.
- Data Flow models are distributed, and there is no need for central controller
- Data Flow models are modular.
- Data Flow systems are easy to analyze (deadlock, stability)

Tokens, Actors, and Queues

The major differences of this modeling style compared to modeling in C.

- Actors contain the actual operations.
- Tokens carry information from one actor to the other.
- Queues are unidirectional communication links that transport tokens from one actor to the other. Data Flow queues have an infinite amount of storage so that tokens will never get lost in a queue. Data Flow queues are first-in first-out.

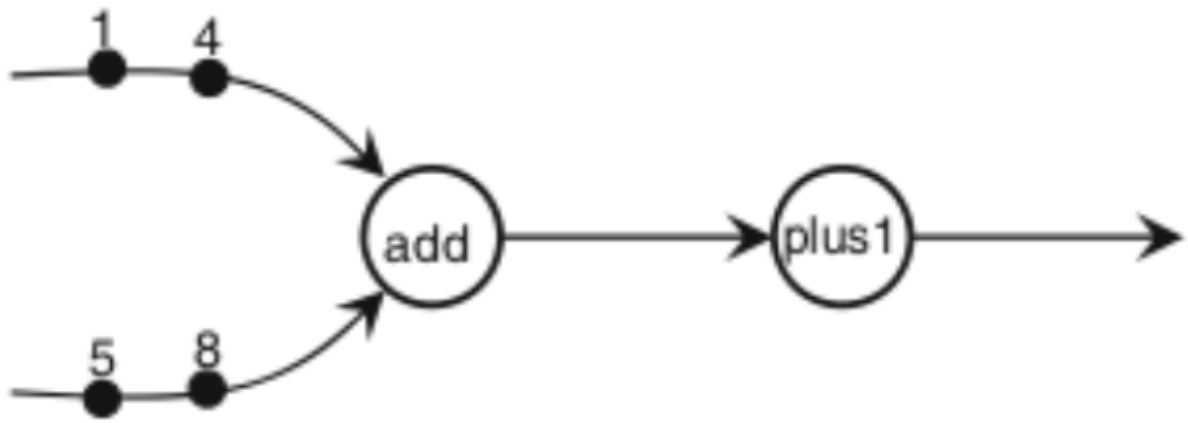


Each single execution of an actor is called the firing of that actor. An actor will never fire if there is no input data, but instead it will wait until data becomes available at its inputs.

The token consumption rate (at the actor inputs) and token production rate (at the actor outputs).

When the number of tokens consumed/produced per actor firing is a fixed and constant value, the resulting class of systems are called synchronous data flow graphs or SDF graphs

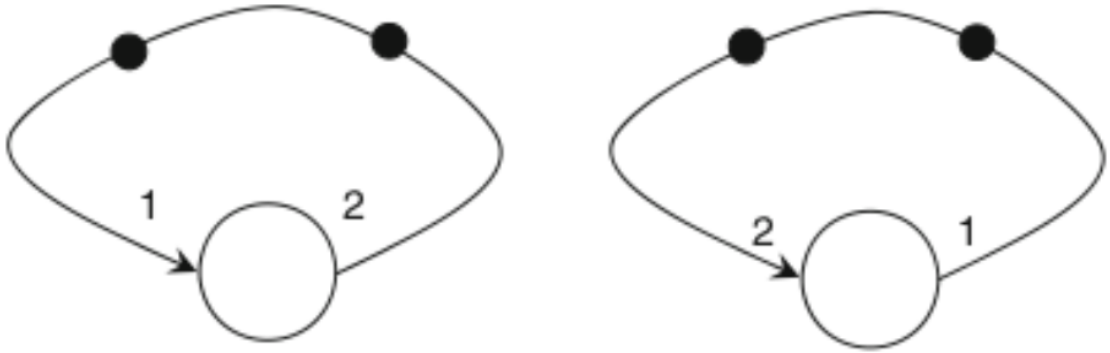
Assuming that each SDF actor implements a deterministic function, then the entire SDF graph is determinate ?



the system will work as specified as long as we implement the firing rules correctly!

Analyzing Synchronous Data Flow Graphs

Deadlock and unstable ?



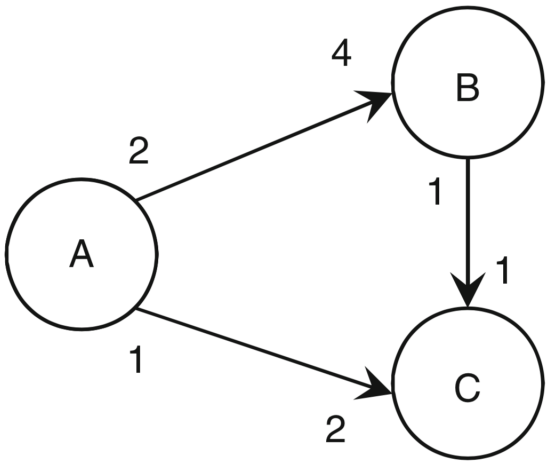
Periodic Admissible Sequential Schedules (PASS) (can continue forever):

- A schedule is the order in which the actors must fire.
- An admissible schedule is a firing order that will not cause deadlock or token- build-up.

Such a schedule requires only a single actor to fire at a time. A PASS would be used, for example, to execute an SDF model on top of a microprocessor.

We can create a PASS for an SDF graph:

1. Create the topology matrix G of the SDF graph
2. Verify the rank of the matrix to be one less than the number of nodes in the graph
3. Determine a firing vector
4. Try firing each actor in a round robin fashion, until it reaches the firing count as specified in the firing vector

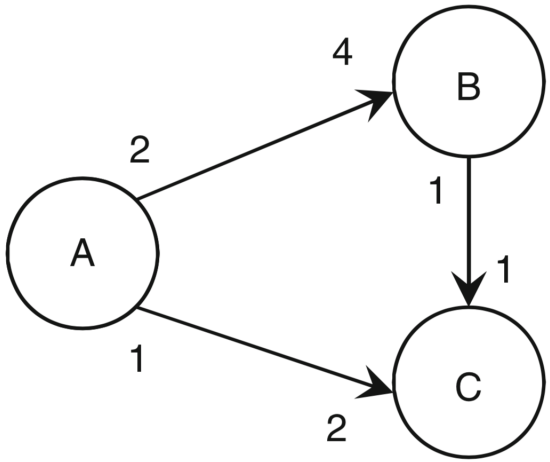


$$G = \begin{bmatrix} A & B & C & \\ 2 & -4 & 0 & < -\text{edgeAtoB} \\ 1 & 0 & -2 & < -\text{edgeAtoC} \\ 0 & 1 & -1 & < -\text{edgeBtoC} \end{bmatrix}$$

Step$_1$: The entry (i,j) of this matrix will be positive if the node j produces tokens into graph edge i . The entry (i,j) will be negative if the node j consumes tokens from graph edge i.

Step$_2$: Verifies that tokens cannot accumulate on any of the edges of the graph!

Tokens left on the edges after the firings!



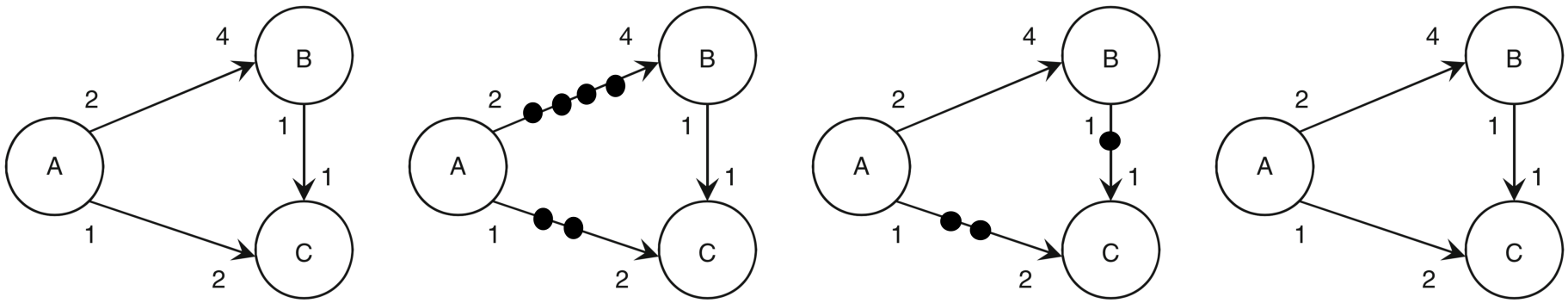
$$G = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 0 \end{bmatrix}$$

Step$_3$: Determine a periodic firing vector q_{pass}.

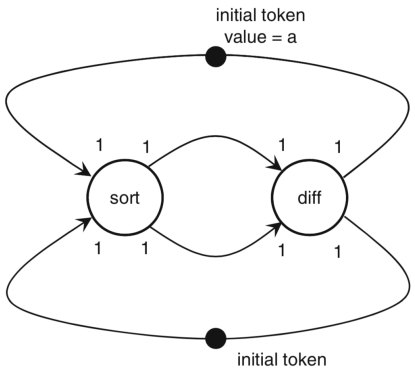
$$Gq_{\text{pass}} = 0$$

Step$_4$: Each node which has the adequate number of tokens on its input queues will fire when tried

$$q_{\text{pass}} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

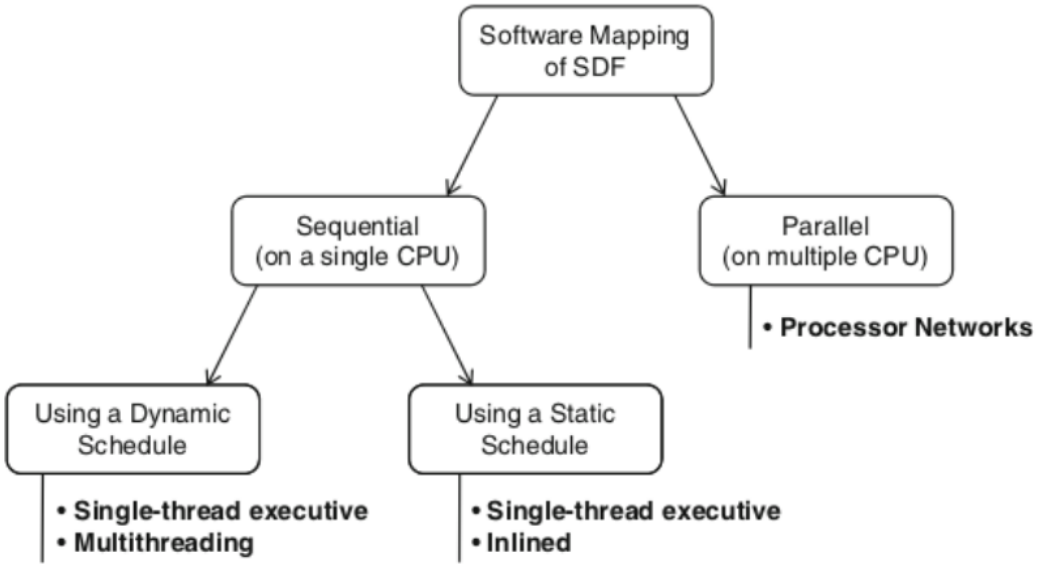


Example:



G=? and q_{pass}

Converting Queues and Actors into Software

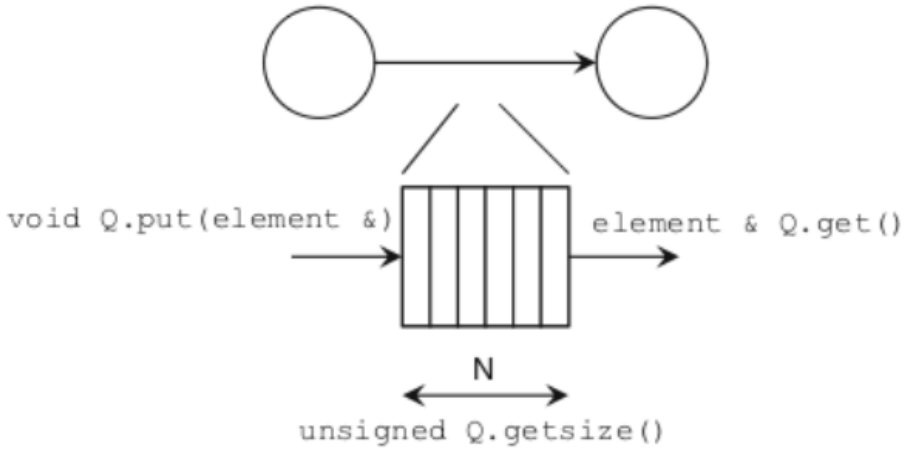


All elements of the SDF graph are mapped in software!

Converting Queues and Actors into Software: Queue

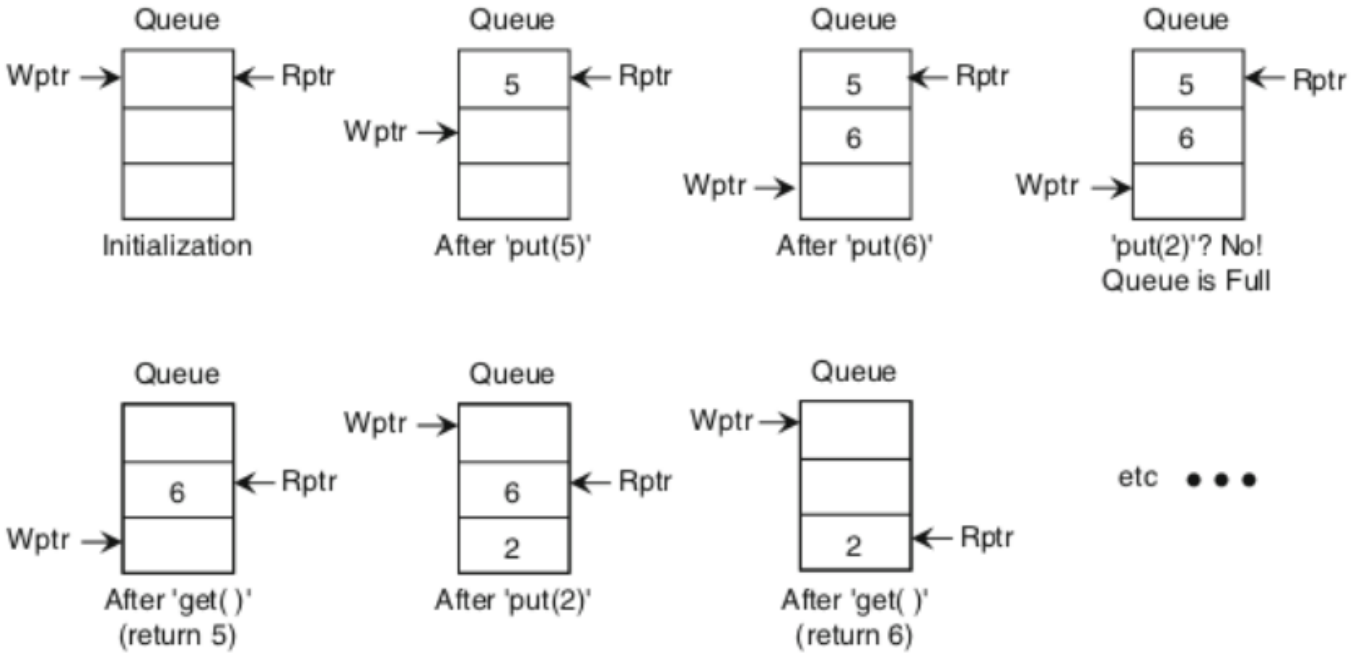
SDF graphs represent concurrent systems and use actors which communicate over FIFO queues.

- The number of elements N that can be stored by the queue.
- The data type element of a queue elements.
- A method to put elements into the queue.
- A method to get elements from the queue.
- A method to test the number of elements in the queue.



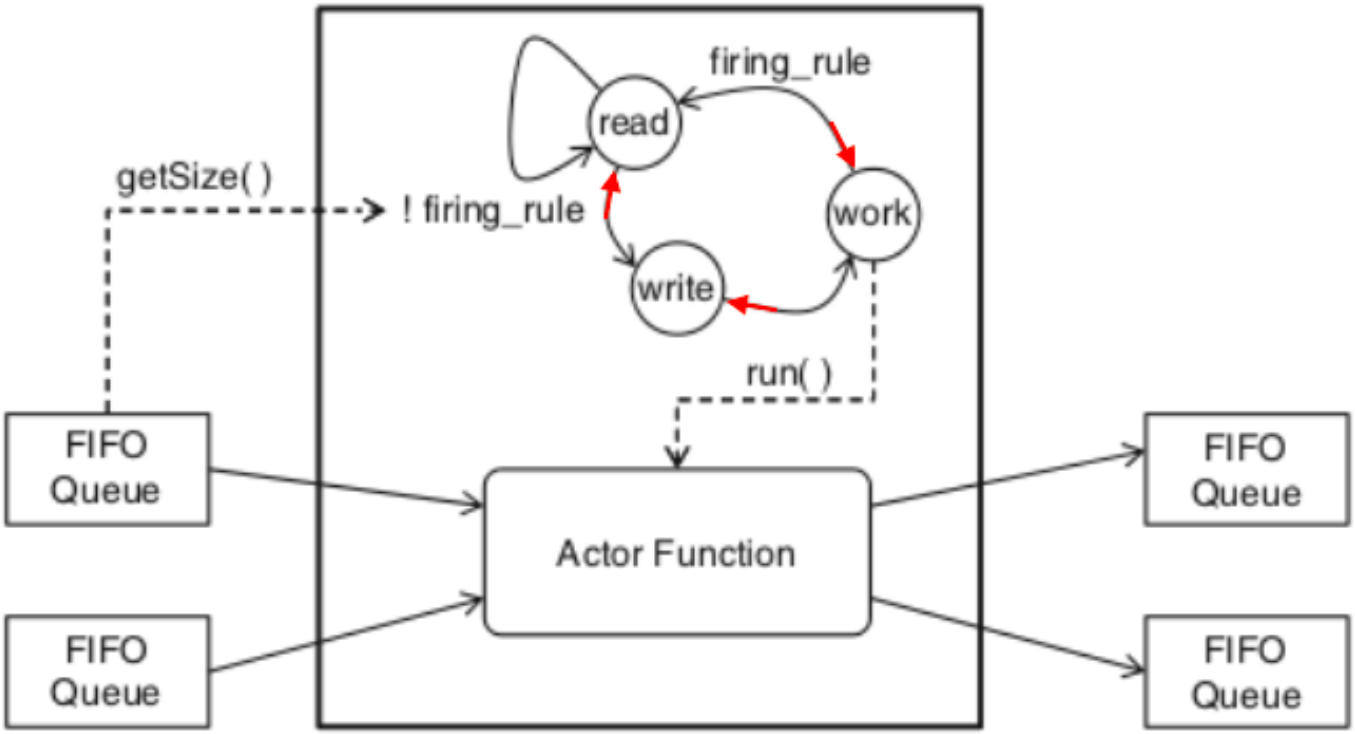
Converting Queues and Actors into Software: Queue

Circular queue



Converting Queues and Actors into Software: Actor

A data flow actor can be captured as a function, with some additional support to interface with the FIFO queues



```
typedef struct actorio {
fifo_t *in1;
fifo_t *in2;
fifo_t *out1;
fifo_t *out2;
} actorio_t;


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);
          }

}
```

Listing 2.2 FIFO object in C (P. Schaumont, Page 52)

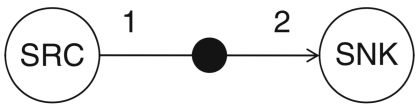
Sequential Targets with Dynamic Schedule

In a dynamic system schedule, the firing rules of the actors will be tested at runtime;

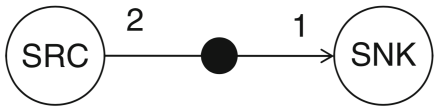
Single-Thread Dynamic Schedules: Next calls the actors in a round-robing fashion

```
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

} }
```

It is impossible to call the actors in the 'wrong' order ! why ?




```
void main() {

...

while(1){

src_actor(&src_io);

snk_actor(&src_io);

} }
```



Using PASS

```
 void main() {
...
while(1){
src_actor(&src_io);
snk_actor(&src_io);
snk_actor(&src_io);
} }
```

Check tokens present

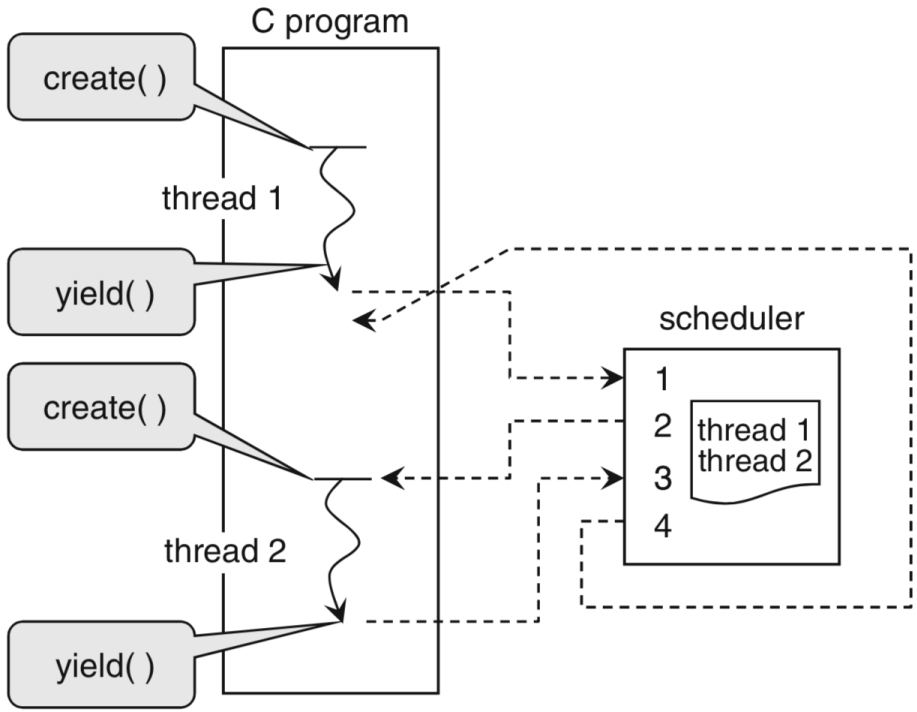
```
void snk_actor(actorio_t *g) {
int r1, r2;
while ((fifo_size(g->in1) > 0))
{
r1 = get_fifo(g->in1);
...  // do processing
} }
```

MultiThread Dynamic Schedules

To switch the processor back and forth between the two threads of control. How ? A thread scheduler will switch between threads!

Cooperative multithreading

The threads of control indicate at which point they release control back to the scheduler. The scheduler then decides which thread can run next.



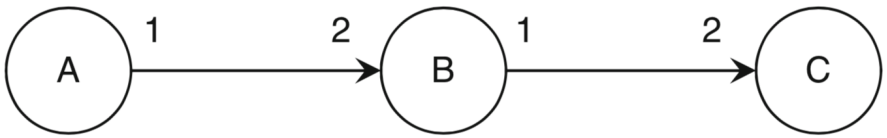
stp_init() initializes the theading system.
stp_create(stp_userf_t *F, void *G) creates a thread that will start execution with user function F.
stp_yield() releases control over the thread to the scheduler.
stp_abort() terminates a thread so that it will be no more scheduled.

```
void world(void *null) {
int n = 5;
while (n-- > 0) {
printf("world");
stp_yield();
}
}
```

See: Listing 2.3: and ▸ Link: Quick thread programming

Static schedule

Determine upfront exactly in what order the actors need to run (fire) and no longer test firing rules when calling actors.

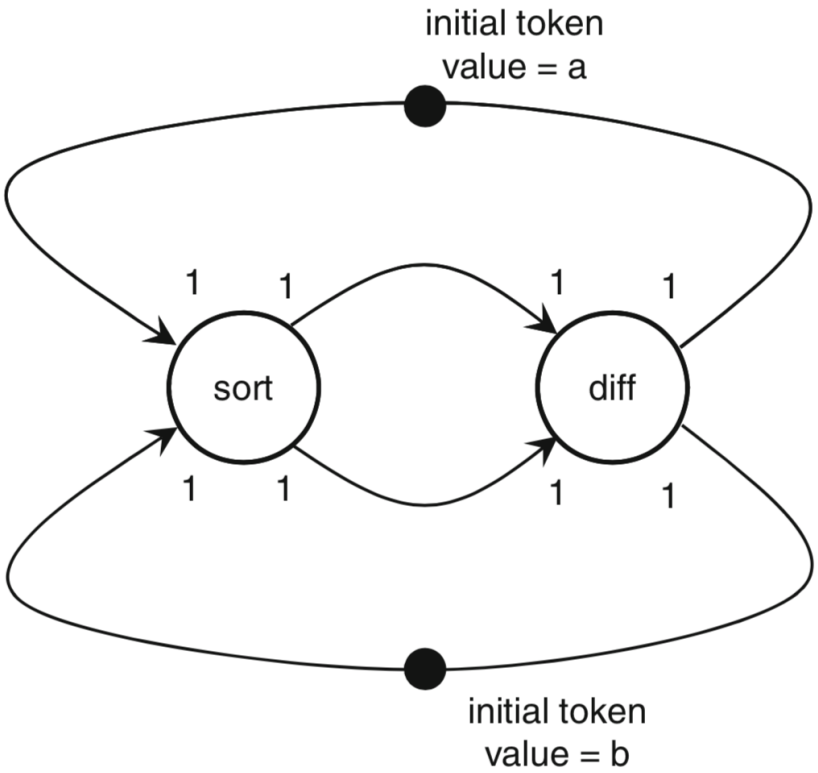


PASS: A();A();A();A();B();B();C(); or A();A();B();A();A();B();C();

```
void main() {
int f1, f2, f3, f4;
// 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;
} }
```

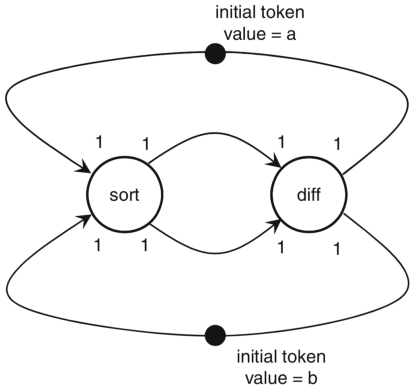
Hardware Implementation of Data Flow

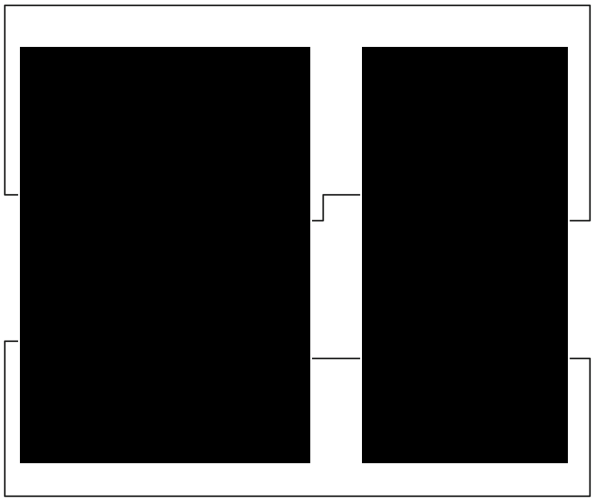
1. Map each queue to a wire.
2. Map each queue containing a token to a register. The initial value of the register must equal the initial value of the token.
3. Map each actor to a combinational circuit, which completes a firing within a clock cycle.

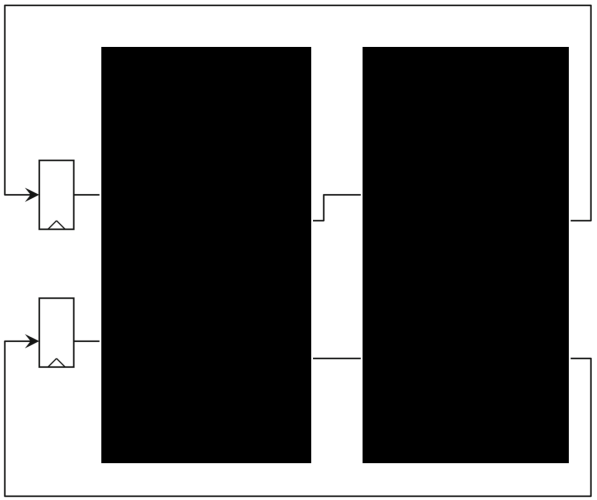


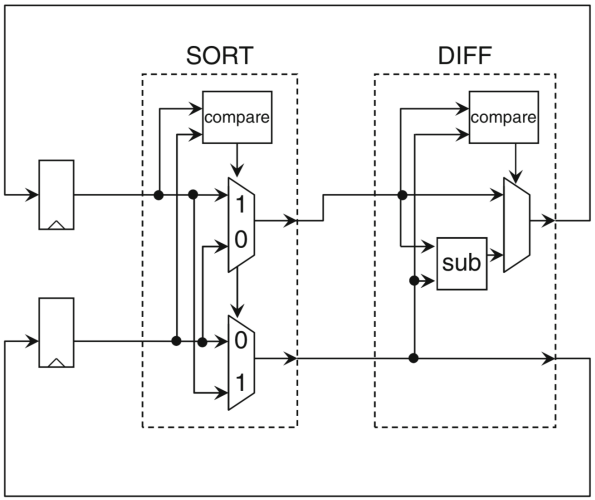

```
sort
out1=(a>b)?  a:b;
out2=(a>b)?  b:a;

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

Hardware Implementation of Data Flow

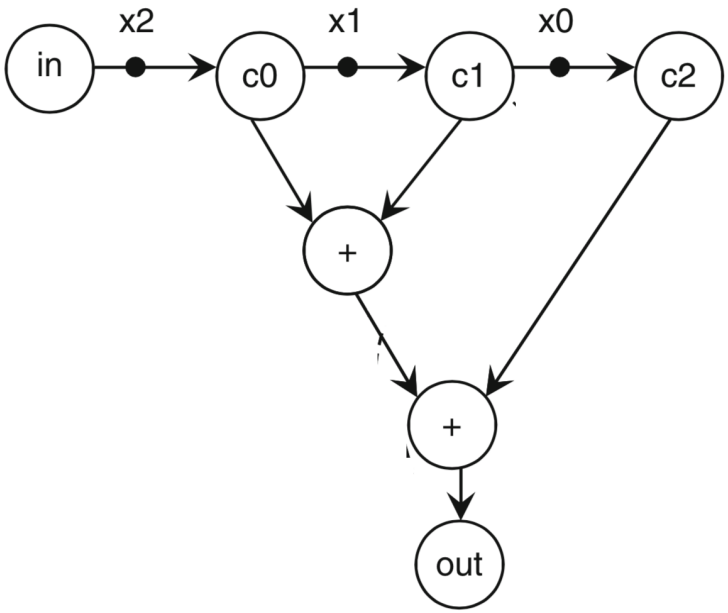




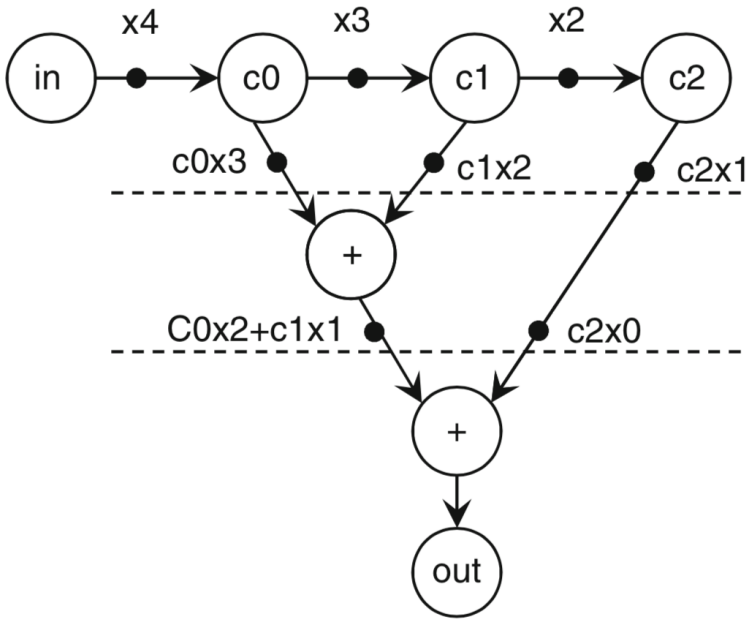




Find critical path !



Not possible to introduce arbitrary initial to
kens in a graph without following the rules fo
actor firing



By letting the in actor produce another token, we will be able to reduce the longest combinational path to a single addition

Multirate Expansion

- Determine the PASS firing rates of each actor.
- Duplicate each actor thenumber of times indicated by its firing rate.
- Convert each multirate actor input/output to multiple single-rate input/outputs.
- Reintroduce the queues in the data flow system to connect all actors.
- Reintroduce the initial tokens in the system, distributing them sequentially over the single-rate queues.