Chapter 3. Data Flow Implementation in SW and HW
**Software Implementation of Data Flow**

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

**Actor firing** only depends on the availability of data (tokens) in the FIFO queues; the firing conditions are captured in the firing rule for that actor.

When implementing an **SDF graph in software**, we have to map all elements of the **SDF graph in software**: actors, queues, and firing rules.

**Figure 3.1** demonstrates several different approaches to map dataflow into software.
Fig. 3.1. Approaches to map dataflow into software.

- **Sequential (on a single CPU)**
  - Using a Dynamic Schedule
    - Single-thread executive
    - Multithreading
  - Using a Static Schedule
    - Single-thread executive
    - Inlined

- **Parallel (on multiple CPU)**
  - Processor Networks
The key objective of a single-processor implementation is the efficient implementation of a sequential schedule. There are two methods to implement such a sequential schedule.

- We can use a *dynamic schedule*, which evaluates the execution order of actors during the execution of the SDF graph. Thus, at runtime, the software will evaluate the actors’ firing rule and decide if the actor body should execute or not. A dynamic schedule can be implemented using a single-thread executive or a multi-thread executive.

- We can also use a *static schedule*, which means that the execution order of the actors is determined at design-time. A static schedule can be implemented using a single-threaded executive.
Figure 3.2 shows a software interface to a queue object. The software interface has two parameters and three methods.

- The **number of elements** $N$ that can be stored by the queue (parameter).
- The **data type element** of a queue elements (parameter).
- 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**.
Fig. 3.2. A software queue.

```c
void Q.put(element &)
```

```c
element & Q.get()
```

```c
unsigned Q.getsize()
```
A circular queue is a data structure consisting of an array of memory locations, a write-pointer and a read-pointer.

Figure 3.3 illustrates the operation of a two-element circular queue.

Listing 3.1 shows the definition of a FIFO object in C.
Fig. 3.3. Operation of the circular queue.

Initialization

After 'put(5)'

After 'put(6)'

'put(2)'? No! Queue is Full

After 'get( )'
(return 5)

After 'put(2)'

After 'get( )'
(return 6)

etc   ●   ●   ●
Listing 3.1 FIFO object in C (1)

```c
#define MAXFIFO 1024
typedef struct fifo {
    int data[MAXFIFO]; // token storage
    unsigned wptr;    // write pointer
    unsigned rptr;    // read pointer
} fifo_t;

void init_fifo(fifo_t *F) {
    F->wptr = F->rptr = 0;
}

void put_fifo(fifo_t *F, int d) {
    if (((F->wptr + 1) % MAXFIFO) != F->rptr) {
        F->data[F->wptr] = d;
        F->wptr = (F->wptr + 1) % MAXFIFO;
        assert(fifo_size(F) <= 10);
    }
}
```
Listing 3.1 FIFO object in C (2)

```c
int get_fifo(fifo_t *F) {
    int r;
    if (F->rptr != F->wptr) {
        r = F->data[F->rptr];
        F->rptr = (F->rptr + 1) % MAXFIFO;
    }
    return r;
}
return -1;
}

unsigned fifo_size(fifo_t *F) {
    if (F->wptr >= F->rptr)
        return F->wptr - F->rptr;
    else
        return MAXFIFO - (F->rptr - F->wptr) + 1;
}
```
int main() {
    fifo_t F1;
    init_fifo(&F1);       // resets wpotr, 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
}

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

Figure 3.4 shows that the firing rule logic is implemented as a small, local controller inside of the actor. The local controller goes through two states.

- In the wait state the actor remains idle, but it tests the firing rule upon each invocation of the actor.
- When the firing rule evaluates true, the actor proceeds to the work state.

SDF actor firing implies that the actor has to read all input queues according to the specified consumption rates, and it has to write all output queues according to the specified production rates.
Fig. 3.4. Software implementation of dataflow actor.
Let’s say we will support up to eight inputs and outputs per actor, then we could define a `struct` to collect the input/output per actor as follows.

```c
#define MAXIO 8
typedef struct actorio {
    fifo_t *in[MAXIO];
    fifo_t *out[MAXIO];
} actorio_t;
```

Next, we use `actorio_t` to model actors as functions.

The following illustrates an actor with a single input and a single output. The actor reads two tokens, and produces their sum and difference.
Data flow actor implementation

```c
void fft2(actorio_t *g) {
    int a, b;
    if ( fifo_size(g->in[0]) >= 2 ) {
        a = get_fifo(g->in[0]);
        b = get_fifo(g->in[0]);
        put_fifo(g->out[0], a+b);
        put_fifo(g->out[0], a-b);
    }
}
```

Finally, the `actorio_t` and queue objects can be instantiated in the main program, and the actor functions can be called using a system scheduler.
A system schedule is implemented as a function that instantiates all actors and queues, and next calls the actors in a round-robin fashion.

```c
void main() {
    fifo_t  q1, q2;
    actorio_t  fft2_io = {{&q1}, {&q2}};
    ...
    init_fifo(&q1);
    init_fifo(&q2);
    ...
    while (1) {
        fft2_actor(&fft2_io);
        // .. call other actors
    }
}
```
In Fig. 3.5a, the \texttt{snk\_actor} will only fire every other time the main function invokes it.

In Fig. 3.5b, the \texttt{src\_actor} will produce two tokens each time the main function invokes it, but the \texttt{snk\_actor} will only read one of these tokens per invocation. This means that, sooner or later, the queue between \texttt{src} and \texttt{snk} will overflow.

The problem of the system schedule in Fig. 3.5b is that the firing rate provided by the system schedule is different from the required firing rate for a PASS.

This problem can be addressed in several ways.
Fig. 3.5. Simulation under single rate system schedule.

(a)  

(b)  

System Schedule

```c
void main() {
  ..
  while (1) {
    src_actor(&src_io);
    snk_actor(&snk_io);
  }
}
```
PASS satisfaction

- Solution 1: We could adjust the system schedule to reflect the firing rate predicted by the PASS.

```c
void main() {
    ...
    while (1) {
        src_actor(&src_io);
        snk_actor(&snk_io);
        snk_actor(&snk_io);
    }
}
```
Solution 2: We could adjust the code for the `snk_actor` to continue execution as long as there are tokens present.

```c
void snk_actor(actorio_t *g) {
    int r1, r2;
    while ((fifo_size(g->in[0]) > 0)) {
        r1 = get_fifo(g->in[0]);
        ... // do processing
    }
}
```
Four-point FFT Example

Figure 3.6a shows a four-point Fast Fourier Transform (FFT).

An equivalent set of operations corresponding to the graph from Fig. 3.6a is shown in Fig. 3.6b.

The twiddle factor $W(k,N)$, or $W_N^k$, is a complex number defined as $e^{-j2\pi k/N}$. Obviously, $W(0,4)=1$ and $W(1,4)=-j$. The FFT thus produces complex numbers at the output.
Fig. 3.6. Flow diagram for a four-point FFT.

\[
\begin{align*}
  a &= t[0] + W(0,4) \cdot t[2] = t[0] + t[2] \\
  b &= t[0] - W(0,4) \cdot t[2] = t[0] - t[2] \\
  f[0] &= a + W(0,4) \cdot c = a + c \\
  f[1] &= b + W(1,4) \cdot d = b - j.d \\
  f[2] &= c - W(0,4) \cdot c = a - c \\
  f[3] &= b - W(1,4) \cdot d = b + j.d
\end{align*}
\]
Figure 3.7 shows a data flow model for the same flow diagram. It consists of three data flow actors: reorder, fft2, and fft4mag.

- **reorder** reads four tokens and reshuffles them according to the requirements of an FFT.

- **fft2** calculates the butterflies for the left half of Fig. 3.6a. This actor reads two tokens, computes the butterfly, and produces two tokens.

- **fft4mag** calculates the butterflies of the right half of Fig. 3.6a. This actor reads four tokens, computes two butterflies, and produces four tokens. The **fft4mag** actor computes the magnitude vector \( \text{real}(V[0]*V[0]), \text{real}(V[1]*V[1]), \text{real}(V[2]*V[2]), \text{real}(V[3]*V[3]) \).
It’s easy to see that a **stable firing vector** for this set of actors is 
\[ q\text{reorder}, q\text{ft2}, q\text{ft4mag} \] = [1,2,1].

**Listing 3.2** shows a description for the **reorder** and **fft4mag** actors, as well as a **main** program to implement this schedule.
Listing 3.2 4-point FFT as an SDF system (1)

void reorder(actorio_t *g) {
    int v0, v1, v2, v3;
    while (fifo_size(g->in[0]) >= 4) {
        v0 = get_fifo(g->in[0]);
        v1 = get_fifo(g->in[0]);
        v2 = get_fifo(g->in[0]);
        v3 = get_fifo(g->in[0]);
        put_fifo(g->out[0], v0);
        put_fifo(g->out[0], v2);
        put_fifo(g->out[0], v1);
        put_fifo(g->out[0], v3);
    }
}

void fft2(actorio_t *g) {
    int a, b;
    while (fifo_size(g->in[0]) >= 2) {
        a = get_fifo(g->in[0]);
        b = get_fifo(g->in[0]);
        put_fifo(g->out[0], a+b);
        put_fifo(g->out[0], a-b);
    }
}
# Listing 3.2 4-point FFT as an SDF system (2)

```c
void fft4mag(actorio_t *g) {
    int a, b, c, d;
    while (fifo_size(g->in[0]) >= 4) {
        a = get_fifo(g->in[0]);
        b = get_fifo(g->in[0]);
        c = get_fifo(g->in[0]);
        d = get_fifo(g->in[0]);
        put_fifo(g->out[0], (a+c)*(a+c));
        put_fifo(g->out[0], b*b + d*d);
        put_fifo(g->out[0], (a-c)*(a-c));
        put_fifo(g->out[0], b*b + d*d);
    }
}
```

![Diagram of the 4-point FFT as an SDF system](image)
The use of the `actorio_t` in the main program simplifies the interconnection of FIFO queues to actors.

```c
int main() {
    fifo_t  q1, q2, q3, q4;
    actorio_t reorder_io = {{&q1}, {&q2}};
    actorio_t fft2_io = {{&q2}, {&q3}};
    actorio_t fft4_io = {{&q3}, {&q4}};
    init_fifo(&q1);
    init_fifo(&q2);
    init_fifo(&q3);
    init_fifo(&q4);
    // test vector fft([1 1 1 1])
    put_fifo(&q1, 1);
    put_fifo(&q1, 1);
    put_fifo(&q1, 1);
    put_fifo(&q1, 1);
```
The actor descriptions use while loops, as discussed earlier, to ensure that the dynamic scheduler can achieve the PASS firing rate for each actor.

```c
int main() {
    fifo_t q1, q2, q3, q4;
    ...
    // test vector fft([1 1 1 1])
    ...

    // test vector fft([1 1 1 0])
    put_fifo(&q1, 1);
    put_fifo(&q1, 1);
    put_fifo(&q1, 1);
    put_fifo(&q1, 0);
    while (1) {
        reorder(&reorder_io);
        fft2 (&fft2_io);
        fft4mag(&fft4_io);
    }
    return 0;
}
```
Another approach to create dynamic schedules is to use multi-threaded programming. A **multi-threaded C program** is a program that has two concurrent threads of execution.

In a **cooperative multithreading model**, the threads of control indicate at which point they release control back to the scheduler. The scheduler then decides which thread can run next.

**Figure 3.8** shows an example with two threads. Initially, the user has provided the starting point of each thread using `create()`.

Assume that the **upper thread (thread1)** is running and arrives at a `yield()` point. This is a point where the thread returns control to the scheduler.

When the **lower thread (thread2)** is ready to run, the scheduler allows thread2 to run until that thread, too, comes at a `yield` point.
Fig. 3.8. Cooperative multi-threading.

A Practical Introduction to Hardware/Software Codesign, 2nd Ed © Springer
A cooperative multithreading library, called Quickthreads, has a Quickthreads API (Application Programmers’ Interface) consisting of four function calls.

- `stp_init( )` initializes the threading system.
- `stp_create(stp userf_t *F, void *G)` creates a thread that will start execution with user function F. The function will be called with a single argument G. The thread will terminate when that function completes, or when the thread aborts.
- `stp_yield( )` releases control over the thread to the scheduler.
- `stp_abort( )` terminates a thread, so that it will be no more scheduled.
Listing 3.3 Example of QuickThreads (1)

```c
#include "../qt/stp.h"
#include <stdio.h>

void hello(void *null) {
    int n = 3;
    while (n-- > 0) {
        printf("hello\n");
        stp_yield();
    }
}

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

int main(int argc, char **argv) {
    stp_init();
    stp_create(hello, 0);
    stp_create(world, 0);
    stp_start();
    return 0;
}
```

A Practical Introduction to Hardware/Software Codesign, 2nd Ed © Springer
Listing 3.3 Example of QuickThreads (2)

```
>gcc -c ex1.c -o ex1 ../qt/libstp.a ../qt/libqt.a
./ex1
hello
world
hello
world
hello
world
world
World

The printing of hello and world are interleaved for the first three iterations, and then the world thread runs through completion.
```
void fft2(actorio_t *g) {
    int a, b;
    while (1) {
        while (fifo_size(g->in[0]) >= 2) {
            a = get_fifo(g->in[0]);
            b = get_fifo(g->in[0]);
            put_fifo(g->out[0], a+b);
            put_fifo(g->out[0], a-b);
        }
        stp_yield();
    }
}

int main() {
    fifo_t q1, q2;
    actorio_t fft2_io = {{&q1}, {&q1}};
    ...
    stp_create(fft2, &fft2_io);
    // create thread
    ...
    stp_start(); // run the schedule
}
In Fig. 3.9, the relative firing rates of A, B, and C must be 4, 2, and 1 to yield a PASS.

In a steady state condition, the queue AB will carry a maximum of four tokens, while the queue BC will contain a maximum of two tokens.

By calling the actors in the sequence (A,A,B,A,A,B,C), the maximum amount of tokens on any queue is reduced to two.

Finding an optimal interleaving in an SDF graph is an optimization problem.
Fig. 3.9. System schedule for a multirate SDF graph.

```
while(1) {
    // call A four times
    A(); A(); A(); A();

    // call B two times
    B(); B();

    // call C one time
    C();
}
```
Optimization of single-thread SDF systems with a static schedule can be shown by using the example of 4-point FFT (Listing 3.2).

1. Because the firing order of actors can be completely fixed, the access order on queues can be completely fixed as well. In the following example, only two positions of FIFO q1 are occupied at a time. Hence, FIFO q1 can be replaced by two single variables.

```c
loop {
    ...
    q1.put(value1);
    q1.put(value2);
    ...
    .. = q1.get();
    .. = q1.get();
}
```

```c
loop {
    ...
    r1 = value1;
    r2 = value2;
    ...
    .. = r1;
    .. = r2;
}
```
2. As a second optimization, we can **inline actor code inside of the main program** and the main scheduling loop (Listing 3.4).

The runtime of this system to decrease significantly: there are no firing rules, no FIFO manipulations and no function boundaries.

**Listing 3.4** Inlined data flow system for the four-point FFT

```c
void dfsystem(int in0, in1, in2, in3,
              *out0, *out1, *out2, *out3) {
    int reorder_out0, reorder_out1, reorder_out2, reorder_out3;
    int fft2_0_out0, fft2_0_out1, fft2_0_out2, fft2_0_out3;
    int fft2_1_out0, fft2_1_out1, fft2_1_out2, fft2_1_out3;
    int fft4mag_out0, fft4mag_out1, fft4mag_out2, fft4mag_out3;
    reorder_out0 = in0;
    reorder_out1 = in2;
    reorder_out2 = in1;
    reorder_out3 = in3;
```
Listing 3.4  Inlined data flow system for the four-point FFT (cont’d)

```c
fft2_0_out0 = reorder_out0 + reorder_out1;
fft2_0_out1 = reorder_out0 - reorder_out1;
fft2_1_out0 = reorder_out2 + reorder_out3;
fft2_1_out1 = reorder_out2 - reorder_out3;
fft4mag_out0 = (fft2_0_out0 + fft2_1_out0)*
                (fft2_0_out0 + fft2_1_out0);
fft4mag_out1 = fft2_0_out1*fft2_0_out1 +
                fft2_1_out1*fft2_1_out1;
fft4mag_out2 = (fft2_0_out0 - fft2_1_out0)*
                (fft2_0_out0 - fft2_1_out0);
fft4mag_out3 = fft2_0_out1*fft2_0_out1 +
                fft2_1_out1*fft2_1_out1;
```
Mapping a single-rate graph into hardware by using the following three rules.

1. All actors are implemented as combinational circuits.
2. All communication queues are implemented as wires (without storage).
3. Each initial token on a communication queue is replaced by a register.

An SDF system for Euclid’s Greatest Common Divisor (GCD) algorithm is shown in Fig. 3.10. The SDF evaluates the GCD of two numbers $a$ and $b$. It uses two actors: sort and diff.

The sort actor reads two numbers, sorts them and copies them to the output. The diff actor subtracts the smallest number from the largest one, as long as they are different.
Fig. 3.10. Euclid’s GCD as an SDF graph.

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

diff
out1 = (a != b) ? a - b : a;
out2 = b;
```
PASS for the GCD SDF graph

Assume \((a_0, b_0) = (16, 12)\), then we see the following sequence of token values:

\((a_1, b_1) = (4, 12), (a_2, b_2) = (8, 4), (a_3, b_3) = (4, 4), \ldots\)

\(a_i\) and \(b_i\) are the token values upon iteration \(i\) of the PASS. Since this sequence converges to the tuple \((4, 4)\), the greatest common divisor of 12 and 16 is 4.

The topology matrix \(G\) for this graph is

\[
G = \begin{bmatrix}
+1 & -1 \\
+1 & -1 \\
-1 & +1 \\
-1 & +1
\end{bmatrix}
\leftarrow\text{edge}(\text{sort}, \text{diff})
\leftarrow\text{edge}(\text{sort}, \text{diff})
\leftarrow\text{edge}(\text{diff}, \text{sort})
\leftarrow\text{edge}(\text{diff}, \text{sort})
\]

A valid firing vector for this system is one in which each actor fires exactly once per iteration.

\[
q_{\text{PASS}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}
\]
Hardware implementation of the Euclid design can be done by using the following transformation.

1. Map each communication 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. Both the sort and diff actors require no more than a comparator module, a few multiplexers and a subtractor.

Figure 3.11 illustrates how this works out for the Euclid example.
Fig. 3.11. HW implementation of Euclid’s algorithm.
Figure 3.12 shows a data flow specification of a digital filter. It evaluates a weighted sum of samples of an input stream, with the sum defined as $\text{out} = x_0.c_2 + x_1.c_1 + x_2.c_0$.

The critical path is equal to a constant multiplication (with $c_0$ or $c_1$) and two additions.

Assume the in actor fires additional tokens, and the $c_0$, $c_1$, $c_2$ and add actors fire as well so that additional tokens start to appear on queues that have no such tokens. For example, assume that the in actor produces a single additional token $x_3$. Then the resulting graph looks as in Fig. 3.13, where the critical path is reduced to only two additions.

By letting the in actor produce another token, we will be able to reduce the critical path to a single addition (see Fig. 3.14).
Fig. 3.12. SDF of a moving-average application.
Fig. 3.13. Pipelining the moving-average filter by inserting additional tokens (1).
Fig. 3.14. Pipelining the moving-average filter by inserting additional tokens (2).
Fig. 3.15. HW implementation of moving-average filter.
Using a single token in the feedback loop of an add actor will accumulate all input samples (Fig. 3.16a).

Using two tokens in the feedback loop will accumulate the odd samples and even samples separately (Fig. 3.16b).
Figure 3.17 shows a single rate data flow system with two actors and an initial token in between them. We map this system such that the first actor, \texttt{ctr}, is implemented in hardware, while the second actor, \texttt{snk}, is implement in software. We are using 8051 microcontroller ports to connect hardware and software.

Listing 3.5 (in the book) shows a GEZEL system description of the data flow design of Fig. 3.17. The hardware actor is included on lines 1–24; the rest of the listing includes an 8051 processor, and communication ports to connect the hardware actor to the 8051.

Listing 3.6 shows the 8051 software to interface the hardware actor. The schedule in the main function invokes two functions, \texttt{collect} and \texttt{snk}. 
Fig. 3.17. Hybrid HW-SW implementation of a dataflow graph

Hardware Design

8051 Microcontroller

A Practical Introduction to Hardware/Software Codesign, 2nd Ed © Springer
Listing 3.5 GEZEL hardware description of data flow example of Fig. 3.17

```vhdl
1  dp send_token(out dout : ns(8);
2       out req : ns(1);
3       in ack : ns(1)) {
4     reg ctr : ns(8);
5     reg rack : ns(1);
6     reg rreq : ns(1);
7     always {
8       rack = ack;
9       rreq = rack ? 0 : 1;
10      ctr = (rack & rreq) ? ctr + 1 : ctr;
11      dout = ctr;
12      req = rreq;
13    }
14    sfg transfer {
15       $display($cycle, " token ", ctr);
16    }
```
Listing 3.5 GEZEL hardware description (2)

17       sfg idle {}
18 }

19   fsm ctl_send_token(send_token) {
20     initial s0;
21     state s1;
22     @s0 if (rreq & rack) then (transfer) -> s0;
23     else (idle) -> s0;
24   }

26   ipblock my8051 {
27     iptype "i8051system";
28     ipparm "exec=df.ihx"
29     ipparm "verbose=1"
30     ipparm "period=1"
Listing 3.5 GEZEL hardware description (3)

33  ipblock my8051_data(in data : ns(8)) {
34    iptype "i8051systemsink";
35    ipparm "core=my8051";
36    ipparm "port=P0";
37  }
38
39  ipblock my8051_req(in data : ns(8)) {
40    iptype "i8051systemsink";
41    ipparm "core=my8051";
42    ipparm "port=P1";
43  }
44
45  ipblock my8051_ack(out data : ns(8)) {
46    iptype "i8051systemsink";
47    ipparm "core=my8051";
48    ipparm "port=P2";
49  }
50
Listing 3.5 GEZEL hardware description (4)

51    dp sys {
52       sig data, req, ack : ns(8);
53       use my8051;
54       use my8051_data(data);
55       use my8051_req (req);
56       use my8051_ack (ack);
57       use send_token (data, req, ack);
58    }
59
60   system S {
61       sys;
62   }
Software description of data flow (1)

Listing 3.6 Software description of data flow example of Fig. 3.17

```c
#include <8051.h>
#include "fifo.c"

void collect(fifo_t *F) {
    if (P1) {    // if hardware has data
        put_fifo(F, P0);  // then accept it
        P2 = 1;   // indicate data was taken
        while (P1 == 1)  // wait until the hardware acknowledges
            P2 = 0;   // and reset
    }
}

unsigned acc;
void snk(fifo_t *F) {
    if (fifo_size(F) >= 1)    // if hardware has data
        acc += get_fifo(F);
}
```

A Practical Introduction to Hardware/Software Codesign, 2nd Ed © Springer
Software description of data flow (2)

18
19  void main() {
20    fifo_t F1;
21
22    init_fifo(&F1);
23    put_fifo(&F1, 0);  // initial token
24    acc = 0;
25
26    while (1) {
27        collect(&F1);
28        snk(&F1);
29    }
30  }