FIFO Modelica

Simone Sestito
sestito.1937764@studenti.uniroma1.it

Gennaio 2023

Indice

1 Convenzioni per le variabili **2**
1.1 Parametri del block FIFO 2
1.1.1 Lato lettura 2
1.1.2 Lato scrittura 2

2 Come comunicano **3**
2.1 Sender 3
2.2 Receiver 4

3 Comunicazione bidirezionale **5**
3.1 Client 5
3.2 Server 7
3.3 System 8

1 Convenzioni per le variabili

1.1 Parametri del block FIFO

All'inizio, ci sono i parametri M e N.

```
parameter Integer M = 1; // size IO array
parameter Integer N = 100; // fifo size
```

Ha senso modificare M se devo salvare un array di più di un solo elemento, come una tripla. Poi ci sono le variabili che andranno connesse (`connect(..., ...)`) con la parte del sender e del receiver. Fortunatamente, il resto della FIFO è sempre copia incolla.

1.1.1 Lato lettura

```
InputBoolean readsignal;
OutputInteger readfifodata[M];
OutputBoolean dataavailable;
```

- `readsignal` serve per il trigger dell'evento e deve essere assegnata con `readsignal := not(pre(readsignal))`.
- `readfifodata` sarà connesso al buffer dentro il ricevente, e verrà assegnato all'evento di lettura.
- `dataavailable` serve per comunicare se c'è qualcosa da leggere, e sarà controllata dal ricevente: altrimenti *non invia neanche il segnale* per leggere

1.1.2 Lato scrittura

```
InputBoolean writesignal;
InputInteger writefifodata[M];
OutputBoolean spaceavailable;
```

- `writesignal` analoga della `readsignal`, assegnata con `not(pre(...))`
- `writefifodata` deve essere riempita dallo scrivente *prima* di scatenare l'evento di scrittura
- `spaceavailable` dovrebbe *prevenire lo scrivente* dall'aggiungere dati - deve attendere

→ File fifo.mo

2 Come comunicano

Una FIFO è monodirezionale, per cui per una comunicazione bidirezionale avrò bisogno di 2 FIFO distinte. Possono essere lo stesso **block** FIFO ma sicuramente 2 istanze.

2.1 Sender

In tutti gli esempi, si vede sempre come chi usa le FIFO (vale anche per il receiver), dovrà lavorare a metà clock (**parameter** Real T) rispetto al resto del programma per agevolare l'uso della FIFO che, ricordiamo, ha bisogno di due clock per trasmettere il dato a destinazione.

Un sender internamente avrà il parallelo delle variabili che ha la FIFO per la scrittura, chiaramente a connettore invertito (**input** ⇄ **output**) e andrà prevista una **connect**(..., ...) per ogni coppia di variabili nel **class** System. In aggiunta, ci sarà anche lo **stato**:

```
OutputInteger writefifodata[M];
OutputBoolean writesignal;
InputBoolean spaceavailable;

// STATO!
Integer state;
```

Dopo aver inserito queste variabili, possiamo procedere alla logica in **algorithm**. Prima di tutto inizializziamo le variabili di output e interne.

```
when initial() then
    writesignal := false;
    state := 0;
elsewhen sample(0, T) then
```

Poi avremo la parte di codice vera e propria. Qui viene l'importanza dello stato: proprio perché dobbiamo prima scrivere il dato da inviare, poi successivamente inviarlo, è fondamentale dividere questo in due step (⇢ due stati!).

0. Ho dei dati da inviare alla FIFO, intanto li metto nel buffer e vado in 1.

```
if pre(state) == 0 then
    writefifodata := myrandom(); // TODO: Qualcosa da inserire
    // oppure writefifodata[1] se singolo elemento
    state := 1;
```

1. Ho messo già dei dati nel buffer, e **se c'è spazio** invio scatenando la scrittura.

```
elseif pre(state) == 1 and pre(spaceavailable) then
    writesignal := not(pre(writesignal)); // Scatena la write
    state := 0; // Torna alla prossima scrittura
end if;

end when;
```

→ File sender.mo monodirezionale

2.2 Receiver

Vale anche qui la storia del clock dimezzato citata nel sender. E anche qui serve lo stato, che andrà spezzato (similmente) come di seguito.

Variabili:

```
InputBoolean datavailable;
InputInteger readfifodata[M];
OutputBoolean readsignal;
```

Qui in più avremo sia lo stato che una variabile per conservare il valore letto, spesso utile altrimenti o lo usiamo in altro modo o va perso...

```
Integer x; // Dato letto
Integer state;
```

Prima di tutto facciamo la solita inizializzazione:

```
algorithm

when initial() then
    x := 0;
    state := 0;
    readsignal := 0;
elsewhen sample(0, T) then
```

0. Stato in cui posso leggere, a patto che **ci siano dati**. Andrò a richiedere di scrivermi i dati nel buffer. L'evento si scatena come sempre invertendo il `Boolean`.

```
    if pre(state) == 0 and pre(datavailable) then
        readsignal := not(pre(readsignal));
        state := 1;
```

1. Stato in cui ho precedentemente richiesto la lettura, i dati erano disponibili, e ora mi aspetto di averlo nel buffer. Devo estrarlo e riprendere il loop

```
    elseif pre(state) == 1 then
        x := pre(readfifodata[1]); // o direttamente []
        state := 0;
    end if;

    end when;
```

→ File receiver.mo monodirezionale

3 Comunicazione bidirezionale

Fin'ora abbiamo visto Sender e Receiver monodirezionali, ma sull'esame non c'è mica monodirezionale, ma bensì bidirezionale. Come detto, serve fare una *coppia* di FIFO, una per direzione della comunicazione. Anche i vari Sender e Receiver si dovranno evolvere, e prenderanno il nome di `Client` e `Server`. Un esempio dato anche a lezione (esercizio 2250) è un client che invia due interi e riceve la loro differenza come risposta.

Nell'esercizio vero 2250, venivano inviati prima un intero, poi un altro, e infine attesa risultato. Qui per semplicità facciamo che inviamo una coppia di interi ($M = 2$, e questo può essere impostato come visto nella sezione 3.3).

3.1 Client

Il client prende come base il Receiver della sezione 2.2 e si evolve perchè ora avrà anche una parte da Sender (2.1). Come ci si può immaginare, anche gli stati aumenteranno.

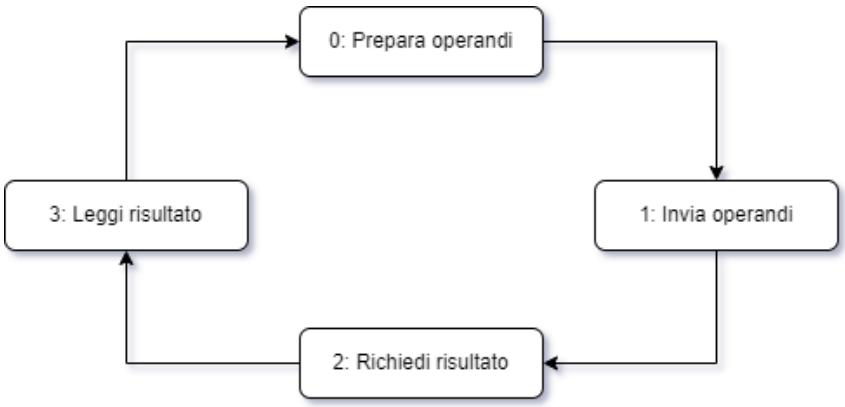

Figura 1: Stati del client

Prima di tutto, le variabili:

```
block Client
parameter Real T = 1; // O meno!

// Variabili lato lettura (server -> client)
OutputBoolean readsignal;
InputInteger readfifodata[1];
InputBoolean datavailable;

// Variabili lato scrittura (client -> server)
OutputInteger writefifodata[2]; // M = 2
OutputBoolean writesignal;
InputBoolean spaceavailable;

// Stato e variabili
```

```
Integer result;
Integer state;
```

Successivamente bisogna procedere con le inizializzazioni:

```
algorithm
when initial() then
    // server -> client
    readsignal := false;

    // client -> server
    writesignal := false;
    writefifodata[1] := 0;
    writefifodata[2] := 0;

    // Stato
    result := 0;
    state := 0;
```

E ora la logica degli stati come rappresentata in figura 1:

```
elsewhen sample(0, T) then
    // TODO: Logica degli stati
end when;
```

0. Prepara operandi nel buffer per essere scritti

```
if pre(state) == 0 then
    writefifodata[1] := myrandint();
    writefifodata[2] := myrandint();
    state := 1;
```

1. Se c'è spazio, invia i dati al server

```
elseif pre(state) == 1 and pre(spaceavailable) then
    writesignal := not(pre(writesignal));
    state := 2;
```

2. Ora che ho inviato, sono pronto ad accettare una risposta *(se è disponibile)*

```
elseif pre(state) == 2 and pre(dataavailable) then
    readsignal := not(pre(writesignal));
    state := 3;
```

3. Ho chiesto la risposta, e quando l'ho chiesta era disponibile. Ora devo ottenerla ed elaborarla

```
elseif pre(state) == 3 then
    result := pre(readfifodata[1]);
    state := 0; // Ricomincia
end if;
```

→ File client.mo bidirezionale con coppia di operandi

3.2 Server

Il server in maniera analoga dovrà leggere la coppia, calcolare il risultato e inviarli al client in risposta. Possiamo vedere graficamente questi stati.

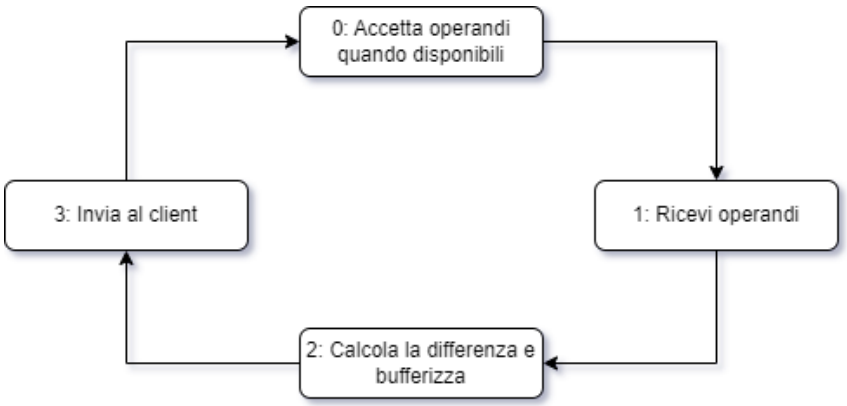

Figura 2: Stati del server

Anche qui le variabili sono parallele alla FIFO, ma con un'eccezione! Infatti, siccome ora devo leggere $M = 2$ e scrivere un solo dato, le dimensioni degli array saranno speculari al client.

```
block Server
parameter Real T = 1; // O meno, soprattutto con piu' client!

// Variabili lato lettura (client -> server)
OutputBoolean readsignal;
InputInteger readfifodata[2]; // M = 2
InputBoolean datavailable;

// Variabili lato scrittura (server -> client)
OutputInteger writefifodata[1];
OutputBoolean writesignal;
InputBoolean spaceavailable;

// Stato corrente e variabili
Integer state;
Integer operands[2];
```

Successivamente bisogna procedere con le inizializzazioni:

```
algorithm
when initial() then
    // client -> server
    readsignal := false;

    // server -> client
```

```
writesignal := false;
writefifodata[1] := 0;

// Stato
state := 0;
operands[1] := 0;
operands[2] := 0;
```

E ora la logica degli stati come rappresentata in figura 1:

```
elsewhen sample(0, T) then
    // TODO: Logica degli stati
end when;
```

0. Accetta gli operandi, se disponibili

```
if pre(state) == 0 and pre(datavailable) then
    readsignal := not(pre(readsignal));
    state := 1;
```

1. Leggi la coppia di operandi richiesta al clock precedente

```
elseif pre(state) == 1 then
    operands[1] := readfifodata[1];
    operands[2] := readfifodata[2];
    state := 2;
```

2. Calcola il risultato e bufferizzalo

```
elseif pre(state) == 2 then
    writefifodata[1] := operands[1] - operands[2];
    state := 3;
```

3. Ho calcolato la risposta. Quando c'è spazio, inviala

```
elseif pre(state) == 3 and pre(spaceavailable) then
    writesignal := not(pre(writesignal));
    state := 0; // Ricomincia
end if;
```

→ File server.mo bidirezionale

3.3 System

Il System deve avere le classiche **connect**(..., ...) dentro **equation** che non scriveremo. Ma cosa importante è l'istanziazione delle variabili, in particolare delle FIFO.

Abbiamo detto che la FIFO client → server ha $M = 2$ e quella server → client ha $M = 1$ (valore di default). Questo andrà impostato tramite **parameter**:

```
class System
FIFO clientToServer(M = 2), serverToClient(M = 1);
Client c;
Server s;

equation

// Solite connect
```