

# Esterel Studio 5.2 Example Series

## Fifo11: Concurrent Read / Write Fifo

G. Berry, Esterel Technologies\*

October 13, 2004

### 1 Introduction

We program a fifo in Esterel v7\_20 for Esterel Studio 5.2. The fifo has two concurrent ports which may act at the same cycle, a read port and a write port. It is of fixed size.

The fifo accepts write data of a type `ObjType`, set to `unsigned<[8]>` for the demo. It reads a write command using a valued `Write` signal, and a read command using a pure input `Read`, in which case the value is returned by a valued output signal `DataOut`. The fifo is implemented using a memory. There is a direct *value bypass* from the `Write` input to the `DataOut` output if a `Read` request is issued at the same time as a `Write` request with the fifo currently empty. The fifo reports being empty or full at the end of a reaction by emitting outputs `Empty` or `Full`, and it reports errors when one tries to read from an empty fifo or to write in a full fifo by emitting outputs `EmptyError` or `FullError`.

We assume that there is an initial empty instant (boot) to start the fifo. At that instant, no `Read` or `Write` request should be issued. This is common in designs and makes life simpler. If one wants the fifo to be already active at first tick, one only needs to replace all ‘every’ statements by “every immediate” in the code below.

The program models the fifo using a fifo controller and a memory model. We provide Esterel Studio simulation of the whole fifo, optimized HDL generation for the controller only, and formal property verification.

This fifo is mostly intended to explain Esterel v7 programming, synthesis, and verification. In particular, we discuss the use of `temp` to avoid generating extra hardware registers, a problem which is meaningless for C code generation. We suggest to ignore `temp` at first reading and to carefully read Section ?? if one is interested in efficient hardware generation. In practical hardware implementation of comparatively large fifos, only the controller matters, since the memory model should be replaced by a physical memory.

### 2 Data definition

The three relevant data objects are the type `ObjType` of stored elements, the size `FifoSize` of the fifo, and the derived type `Address` defined as `unsigned<FifoSize>`. For the demo, we fix `ObjType = unsigned<[8]>` and `FifoSize = 4`.

```
data MemData :
  constant MemSize : unsigned<> = 4;
  type Address = unsigned<MemSize>;
  type ObjType = unsigned<[8]>;
end data
```

---

\*The program was originally designed by M. Kishinevsky, Intel Strategic Cad Lab

To make the `fifo` generic for your own usage, simply make `ObjType` and `FifoSize` generic as follows, without changing the rest of the code:

```
data MemData :
  generic constant MemSize : unsigned;
  type Address = unsigned<MemSize>;
  generic type ObjType;
end data
```

### 3 The memory model

Remember that this is only a simulation model, not something to be synthesized in hardware (synthesis is actually possible from our memory specification, but it would clearly yield large register-based hardware compared to well-crafted library memory devices).

#### 3.1 The memory write interface

The memory write interface has an address input and a data input. Writing should occur when `WriteAddress` is present. The address and data must then be defined:

```
interface MemWriteIntf :
  extends MemData;
  input WriteAddress : temp Address;           % address
  input DataIn : temp ObjType;                % data bus
end interface
```

The `DataIn` input must be available when `WriteAddress` is present.

#### 3.2 The memory read interface

The memory read interface has an address input and a data output. Reading should occur when `ReadAddress` is present and return `DataOut` in the same cycle.

```
interface MemReadIntf :
  extends Memdata;
  input ReadAddress : temp Address;           % address bus
  output DataOut : temp ObjType;            % data bus
end interface
```

#### 3.3 The memory interface

We use extension to define the memory interface from the read and write interfaces. Notice that `MemData` is automatically imported.

```
interface DualMemIntf :
  extends MemReadIntf;
  extends MemWriteIntf;
end interface
```

### 3.4 The memory module

The memory module simply writes and reads an array of the appropriate size.

```
module DualMem :
  extends DualMemIntf;

  var MemArray : ObjType[MemSize] in
    every WriteAddress do
      emit ?MemArray [?WriteAddress] = ?DataIn
    end every
  ||
    every ReadAddress do
      emit ?DataOut = ?MemArray[?ReadAddress]
    end every
  end signal
end module
```

## 4 The Fifo11Ctrl controller

This is the most interesting module. The fifo controller decides when to bypass the memory, when to read or write the memory, and when to issue empty / full messages or errors. It uses a mix of equational and imperative styles.

The `FifoSize` signal holds the current fifo size *at end of reaction*, which means that the fifo size at the beginning of a reaction is `pre(?FifoSize)` (see Section ?? for a version of the controller using registered signals). The `DeltaSize` signal is used to compute the memory usage variation. A write operation augments the memory size by 1, a read operation diminishes it by 1. The write and read operations being optional and concurrent, they may respectively emit `DeltaSize(1)` and `DeltaSize(-1)`. The `DeltaSize` signal is declared combined with `+`, which means that the value are appropriately added to compute the global memory change. The `FifoSize` signal is updated whenever the size changes.

The `ReadEmpty` local signal tells whether the user tries to read from an empty fifo. The `Bypass` local signal is used to decide when bypass should occur.

```
module Fifo11Ctrl :
  extends MemData;

  // read interface

  input  Read; // from user
  output DataOut : temp ObjType; // to user
  output ReadAddress : temp Address; // to memory

  // write interface

  input  Write : temp ObjType; // from user
  output WriteAddress : temp Address; // to memory

  // additional user control interface

  output Empty; // after reaction %
  output Full; // after reaction
  output EmptyError;
  output FullError;
```

```

signal FifoSize : unsigned<MemSize+1> init 0, // fifo size after reaction
      DeltaSize : temp signed<2> combine + , // fifo size variation
      ReadEmpty, // read from empty fifo
      Bypass // bypass write->read
in
  sustain {
    // compute ReadEmpty and Bypass
    ReadEmpty = Read and pre1(Empty), // initially empty
    Bypass = ReadEmpty and Write,
    // compute errors
    EmptyError = ReadEmpty and not Write,
    FullError = pre(Full) and Write,
    // compute empty and full at the tick end
    Empty = (?FifoSize = 0),
    Full = (?FifoSize = MemSize)
  }
||
  // update fifo size at each change
  every DeltaSize do
    emit FifoSize(pre(?FifoSize) + ?DeltaSize)
  end every
||
  // read section
  var ReadPointer : Address := 0 in
    always
      if
        case EmptyError // no action, prevents changing
                        // the pointer by Read below
        case Bypass do // direct write->read transfer
          emit ?DataOut = ?Write
        case Read do // read from memory
          emit ?ReadAddress = ReadPointer;
          ReadPointer := (ReadPointer + 1) mod MemSize;
          emit ?DeltaSize <= -1 // one less element in memory
        end if
      end always
    end var
||
  // write section
  var WritePointer : Address := 0 in
    every Write and not FullError and not Bypass do
      emit ?WriteAddress = WritePointer;
      WritePointer := (WritePointer + 1) mod MemSize;
      emit ?DeltaSize <= 1 // one more element in memory
    end every
  end var
||
  // assertion section
  sustain {

    assert NoReadAndWrite = (WriteAddress and ReadAddress)
                        => (?WriteAddress <> ?ReadAddress),
    // bypass exclusive with memory access
    assert BypassNoMemAccess =
      not (Bypass and (WriteAddress or ReadAddress))
  }

```

```

    }
end signal
end module

```

## 4.1 Variants using registered signals

An alternative way to write the controller is to use a registered signal for `FifoSize`. This is simply a shift in time which can make the code more readable. Then, `?FifoSize` is the fifo size at the beginning of a reaction and `next(?FifoSize)` is the fifo size at the end of the instant. The declaration has to be changed as follows:

```

    signal FifoSize : reg integer init 0 in % fifo size before reaction

```

The equations of `Empty` and `Full` become:

```

    next Empty = :(next(?FifoSize) = 0)
    next Full  = :(next(?FifoSize) = MemSize)

```

And the update of `FifoSize` becomes:

```

    emit next ?FifoSize <= ?FifoSize + ?DeltaSize;

```

## 5 The main Fifo11 module

The main `Fifo11` module simply consists of the memory and fifo put in parallel:

```

main module Fifo11 :
extends MemData;
// read interface
input Read;
output DataOut : temp ObjType;
// write interface
input Write : temp ObjType;
// user empty / full info and errors
output Empty;
output Full;
output EmptyError;
output FullError;

// body : put control and memory in parallel
signal ReadAddress : temp Address,
        WriteAddress : temp Address in
    run Fifo11Ctrl
||
    run DualMem [signal Write / DataIn]
end signal
end module

```

## 6 Running the Fifo11 example

There are four source Esterel files, `data.str1`, `memory.str1`, `fifo11ctrl.str1` and `fifo11.str1`, plus a simulation input file `fifo11.esi` for Esterel Studio5.2. Change `fifo11.esi` to run other simulations; we tried to make this trivial.

## 7 Formal verification

For formal verification, there are two assertions. The first assertion states that there is no memory access in case of bypass. The second assertion states that there is no memory Read/Write conflict. Run bug-chasing on the assertions in the Fifo to verify that the Esterel Studio Verifier finds no counter-example no matter how long you leave it running. Since the fifo is short, any counter-example should be short anyway. Then, purposely introduce a bug, for instance by changing

```
WritePointer := (WritePointer + 1) mod MemSize;
```

into

```
WritePointer := (WritePointer + 2) mod MemSize;
```

Call the Esterel Studio Verifier and ask for verification of the assertions, using bug chasing strategy. Then the following counter-example is instantaneously generated:

```
% assertion "Fifo11/Fifo11Ctrl/NoMemoryConflict" is violated at depth 3  
!reset ;  
Read ;  
Write=0 ;  
Write=0 ;  
Read Write=0 ;
```

## 8 Hardware generation

For hardware generation, in Esterel Studio configuration “VHDL”, the controller alone is optimized and compiled into VHDL or Verilog synthesizable code, respectively in files `fifo11.vhd` and `fifo11.v`. As we mentioned before, the memory model should not be implemented as is in HDLs but replaced by a memory designware. Nevertheless, our memory is synthesizable and one can compile the full model into HDL for simulation or prototyping purposes.

Hardware generation is where the `temp` declarations are useful, to avoid creating extra registers. All I/O data and addresses are declared `temp` since they are used only within one cycle. The same holds for the `DeltaSize` signal. On the contrary, the `FifoSize` signal and the `ReadPointer` and `WritePointer` variables must be persistent over time and cannot be declared `temp`.