Dual LIFO

Table of Contents

The `lifo2` module implements a dual Last In First Out stack data structure. The two stack structures share a single block RAM, or may optionally be constructed from shift registers. This module forms the heart of a stack based micro-controller. One of the stacks is used as a data stack while the other is used as a subroutine call stack. By implementing these in a single BRAM we can save a lot of space.

1. Interface

The module interface consists of a clock and reset signal, and a control interface for each stack.

1.1. Module declaration

§1.1.1: §7.1

```
module lifo2
  §1.3.1. Parameters
  (
   input clk,
   input reset,
   §1.2.1.1. Stack port A,
   §1.2.2.1. Stack port B
   );
```

1.2. Stack ports

The two ports use the suffix a and b to distinguish interface signals. A *push* input indicates that data from the *push_data* input should be pushed onto the stack. A *pop* input indicates that data should be popped from the stack. Popped data should be read from the *tos* output during the same cycle that *pop* is valid. It is completely normal to push and pop data on the same clock cycle.

Status signals *empty* and *full* indicate if the stack is empty or full. The *count* output provides the number of valid entries currently on the stack.

1.2.1. Stack port A

§1.2.1.1: §1.1.1

```
   input push_a,
   input pop_a,
   output reg empty_a,
   output reg full_a,
   output reg `CNT_T count_a,
   output `DATA_T tos_a,
   input `DATA_T push_data_a
```

1.2.2. Stack port B

§1.2.2.1: §1.1.1

```
input push_b,
input pop_b,
output reg empty_b,
output reg full_b,
output reg `CNT_T count_b,
output `DATA_T tos_b,
input `DATA_T push_data_b
```

1.3. Parameters

A *depth* parameter specifies the depth for each of the stacks. For the BRAM implementation, the size of the RAM is just the sum of the two *depth* parameters. For the shift register implementation each stack uses its *depth* parameter independently. The *width* parameter specifies the data width. This is the same for both stacks.

The *implementation* parameter may be set to "BRAM" or "SRL" to select between BRAM and shift register implementations. Note that the quotes are required.

The *full_checking* specifies whether the logic that generates the *full* outputs should be generated. In the case of the shift register implementation the concept of full is easy to implement and sensible. With this implementation, there really are two independent stacks. However, with the BRAM implementation the concept of full is a little harder to pin down. Is the stack full when the number of elements is equal to its *depth* parameter? This seems overly restrictive since there may still be plenty of room for it to grow. But if it does grow then it does so at the expense of the other stack. There is a further difficulty if both stacks are almost full. If you push to both stacks then you will overflow one of the stacks. Which one? The one that is already over its posted limit? The logic to implement this seems needlessly complex. So the compromise here is to implement the strict full checking for the BRAM implementation when the *full_checking* parameter is true and to not generate any full checking logic at all if it is false.

§1.3.1: §1.1.1

```
#(
  parameter depth_a = 512,
  parameter depth_b = 512,
  parameter width = 32,
  parameter implementation = "SRL",
  parameter full_checking = 0,
  parameter depth = depth_a+depth_b
  )
```

1.4. Definitions

Here are some definitions that we can make use of later. DATA_T is used for data values. PTR_T is used for pointers like the top of stack pointer. The CNT_T definition is suitable for holding a count. Note that this value needs to hold a value of 0 as well as a value of *depth* so it may be larger than the PTR_T value.

§1.4.1: §7.1

```
`define DATA_T [width-1:0]
`define PTR_T [log2(depth)-1:0]
`define CNT_T [log2(depth+1)-1:0]
```

2. Functions

The log2 function computes the log base 2 of its input value. This function is used to computer widths of pointer and count values.

§2.1: §7.1

```
  function integer log2;
     input [31:0] value;
     begin
value = value-1;
for (log2=0; value>0; log2=log2+1)
  value = value>>1;
     end
  endfunction
```

3. Logic common to all implementations

3.1. Determine read and write signals

The *writing* signals indicate that we are writing to the corresponding stack. This occurs when the push signal is asserted and there is space on the stack or if the push signal is asserted and a pop is asserted as well. In this case we can perform the operation even though the stack is full.

The *reading* indicates that we are reading from the corresponding stack. It is just the pop signal qualified with not empty.

§3.1.1: §7.1

```
wire writing_a = push_a && (!full_a || pop_a);
wire writing_b = push_b && (!full_b || pop_b);
wire reading_a = pop_a && !empty_a;
wire reading_b = pop_b && ! empty_b;
```

3.2. Determine count value for stack A

We first generate the *next_count_a* value. Incrementing if we are writing but not reading, decrementing if we are reading and not writing and leaving the count the same if we are both reading and writing or neither reading or writing. We use a separate combinational always block to generate a next value signal *next_count_a*. This signal is used by later code to determine empty and full conditions.

§3.2.1: §7.1

```
reg `CNT_T next_count_a;
always @(*)
  if (writing_a && !reading_a)
    next_count_a = count_a + 1;
  else if (reading_a && !writing_a)
    next_count_a = count_a - 1;
  else
    next_count_a = count_a;

always @(posedge clk)
  if (reset)
    count_a <= 0;
  else
    count_a <= next_count_a;
```

3.3. Determine count value for stack B

Here we generate the count values for stack B.

§3.3.1: §7.1

```
reg `CNT_T next_count_b;
always @(*)
  if (writing_b && !reading_b)
    next_count_b = count_b + 1;
  else if (reading_b && !writing_b)
    next_count_b = count_b - 1;
  else
    next_count_b = count_b;

always @(posedge clk)
  if (reset)
    count_a <= 0;
  else
    count_b <= next_count_b;
```

3.4. Generate empty

Here we generate the logic for determining if either stack is empty. The stack is empty if on the next clock its count will be zero.

§3.4.1: §7.1

```
always @(posedge clk)
  if (reset)
    empty_a <= 1;
  else
    empty_a <= next_count_a == 0;

always @(posedge clk)
  if (reset)
    empty_b <= 1;
  else
    empty_b <= next_count_b == 0;
```

3.5. Generate full

Full is pretty simple as well. We go full when the next count value will be equal to the *depth* parameter of the stack. Writes will not occur when the stack is full so there can never be a case when the next count value is greater than the depth.

§3.5.1: §3.6.1

```
always @(posedge clk)
  if (reset)
    full_a <= 0;
  else
    full_a <= next_count_a == depth_a;

always @(posedge clk)
  if (reset)
    full_b <= 0;
  else
    full_b <= next_count_b == depth_b;
```

3.6. Conditionally generate the full logic

As discussed in Section 1.3, “Parameters”, based on the value of the *full_checking* parameter we either create the checking logic or just stub the full signals out to always false.

§3.6.1: §7.1

```
 generate
   if (full_checking)
     begin
       §3.5.1. Generate full
     end
   else
     begin
initial full_a = 0;
initial full_b = 0;
     end
 endgenerate
```

4. BRAM implementation

This section describes the BRAM implementation of the dual LIFO.

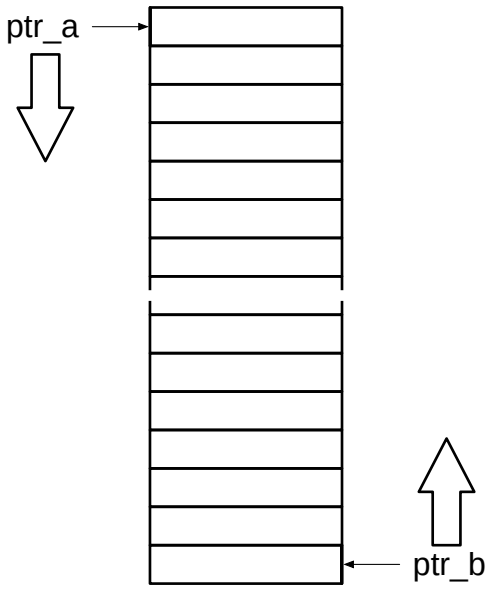
4.1. The data memory

§4.1.1: §4.6.1

```
 reg `DATA_T mem [depth-1:0];
```

4.2. Maintain address pointers

Figure 1. The pointer to the top of the A stack grows down while the pointer to the top of the B stack grows up



Here we maintain the address pointers into the RAM. The pointer for stack A starts at address zero and works up, while the pointer for stack B starts at the last address and works down.

§4.2.1: §4.6.1

```
wire `PTR_T ptr_a = writing_a ? count_a `PTR_T : (count_a `PTR_T)-1;
wire `PTR_T ptr_b = (depth-1)-(writing_b ? count_b `PTR_T
                                         : (count_b `PTR_T)-1);
```

4.3. Top-of-stack

One limitation of the BRAM implementation is that we only have two ports. One port must be used for each stack since both stacks can operate simultaneously. The solution to this problem is to recognize that if we are both pushing and popping the stack then we are really only operating on the top element. This element can be stored in a register and no actual memory operation needs to occur.

4.3.1. Top-of-stack shadow registers

The *tos_shadow* registers hold the latest value pushed onto the stack.

§4.3.1.1: §4.6.1

```
reg `DATA_T tos_shadow_a;
always @(posedge clk)
  if (reset)
    tos_shadow_a <= 0;
  else if (writing_a)
    tos_shadow_a <= push_data_a;

reg `DATA_T tos_shadow_b;
always @(posedge clk)
  if (reset)
    tos_shadow_b <= 0;
  else if (writing_b)
    tos_shadow_b <= push_data_b;
```

4.3.2. Select between top-of-stack shadow registers and memory read values

Here we determine if we should use the memory read value or the top-of-stack shadow register to provide the top-of-stack value. If we are popping the stack then the new top-of-stack value should come from the memory read. Otherwise we should just use the value in the top-of-stack shadow register.

§4.3.2.1: §4.6.1

```
reg use_mem_rd_a;
always @(posedge clk)
  if (reset)
    use_mem_rd_a <= 0;
  else if (writing_a)
    use_mem_rd_a <= 0;
  else if (reading_a)
    use_mem_rd_a <= 1;

reg use_mem_rd_b;
always @(posedge clk)
  if (reset)
    use_mem_rd_b <= 0;
  else if (writing_b)
    use_mem_rd_b <= 0;
  else if (reading_b)
    use_mem_rd_b <= 1;

assign tos_a = use_mem_rd_a ? mem_rd_a : tos_shadow_a;

assign tos_b = use_mem_rd_b ? mem_rd_b : tos_shadow_b;
```

4.4. Perform memory reads

Here we read from the stack. The values read will be the top-of-stack outputs if we are reading from the stack but not writing to it.

§4.4.1: §4.6.1

```
reg `DATA_T mem_rd_a;
always @(posedge clk)
  if (reading_a)
    mem_rd_a <= mem[ptr_a];

reg `DATA_T mem_rd_b;
always @(posedge clk)
  if (reading_b)
    mem_rd_b <= mem[ptr_b];
```

4.5. Perform memory writes

Conversely if we are writing to the stack but not reading then we perform a memory write.

§4.5.1: §4.6.1

```
always @(posedge clk)
  if (writing_a && !reading_a)
    mem[ptr_a] <= tos_a;

always @(posedge clk)
  if (writing_b && !reading_b)
    mem[ptr_b] <= tos_b;
```

4.6. BRAM implementation body

Here we collect all the components of the BRAM implementation.

§4.6.1: §7.1

```
§4.1.1. The data memory
§4.2.1. Maintain address pointers
§4.4.1. Perform memory reads
§4.5.1. Perform memory writes
§4.3.1.1. Top-of-stack shadow registers
§4.3.2.1. Select between top-of-stack shadow registers and memory read values
```

5. Shift register implementation

5.1. Shift register implementation of stack A

There may be situations where two stacks are needed but we don't want to use a block RAM, or if both stacks don't need to be very large and we want to save the overhead of the RAM implementation. This alternate shift register implementation may be used instead.

The basic idea of the shift register implementation is to use a shift register for each bit of the data word. We then concatenate the head of the shift registers together to form the top of the stack.

5.1.1. Pop data off the shift register

Here we shift the register left, shifting in a zero value.

§5.1.1.1: §5.1.1

```
srl <= {srl[depth-2:0],1'b0}
```

5.1.2. Push data onto the shift register

Here we shift the push data onto the shift register. This will only happen when the shift register is not full.

§5.1.2.1: §5.1.1

```
srl <= {push_data_a[i],srl[depth_a-1:1]}
```

5.1.3. Swap data at the top of the shift register

Here we just replace the leftmost value in the shift register with the new pushed value.

§5.1.3.1: §5.1.1

```
srl <= {push_data_a[i],srl[depth_a-2:0]}
```

We use a generate for loop to implement each shift register and a case statement to update it based on the push and pop conditions.

§5.1.1: §5.1

```
for (i=0; i<width; i=i+1)
  begin : srl_a
    reg [depth_a-1:0] srl;
    always @(posedge clk)
      case ({writing_a,reading_a})
 2'b01: §5.1.1.1. Pop data off the shift register;
 2'b10: §5.1.2.1. Push data onto the shift register;
 2'b11: §5.1.3.1. Swap data at the top of the shift register;
      endcase

      assign tos_a[i] = srl[depth_a-1];
  end
```

5.2. Shift register implementation of stack B

The implementation for stack B is the same as for stack A but with the appropriate name changes.

§5.2.1: §5.1

```
for (i=0; i<width; i=i+1)
  begin : srl_b
    reg [depth_b-1:0] srl;
    always @(posedge clk)
      case ({writing_b,reading_b})
 2'b01: srl <= {srl[depth_b-2:0],1'b0};
 2'b10: srl <= {push_data_b[i],srl[depth_b-1:1]};
 2'b11: srl <= {push_data_b[i],srl[depth_b-2:0]};
      endcase
      assign tos_b[i] = srl[depth_b-1];
  end
```

For the shift register implementation we just declare the generate loop variable and then instantiate the two stacks.

§5.1: §7.1

```
  genvar i;
  §5.1.1. Shift register implementation of stack A
  §5.2.1. Shift register implementation of stack B
```

6. Undefined implementation

§6.1: §7.1

```
initial
  begin
    $display("***ERROR: %m: Undefined implementation \"%0s\".",implementation);
    $finish;
  end
```

7. The main module

This is where we tie everything together.

§7.1

```
§1.1. Header
§2.1. Copyright

`timescale 1ns/1ns
§1.4.1. Definitions

§1.1.1. Module declaration

§2.1. Functions

§3.1.1. Determine read and write signals

§3.2.1. Determine count value for stack A
§3.3.1. Determine count value for stack B

§3.4.1. Generate empty
§3.6.1. Conditionally generate the full logic

`ifndef SYNTHESIS
   initial
     $display("***INFO: %m: lifo2.v implementation=\"%0s\".",implementation);
`endif
  generate
    if (implementation == "BRAM")
      begin : bram_implementation
        §4.6.1. BRAM implementation body
      end
    else if (implementation == "SRL")
      begin : srl_implementation
        §5.1. Shift register implementation
      end
    else
      begin
        §6.1. Undefined implementation
      end
  endgenerate
endmodule
```

A. Front matter

1. Header

§1.1: §7.1

```
//-----------------------------------------------------------------------------
// Title         : Dual LIFO stack
// Project       : Common
//-----------------------------------------------------------------------------
// File          : lifo2.v
//-----------------------------------------------------------------------------
// Description :
//
// Two Synchronous LIFO stacks using a single BRAM.
//
// The stacks start at each end and grow to the middle.
//
```

2. Copyright

§2.1: §7.1

```
//-----------------------------------------------------------------------------
// Copyright 2009 Beyond Circuits. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without modification
// are permitted provided that the following conditions are met:
//   1. Redistributions of source code must retain the above copyright notice,
//      this list of conditions and the following disclaimer.
//   2. Redistributions in binary form must reproduce the above copyright notice,
//      this list of conditions and the following disclaimer in the documentation
//      and/or other materials provided with the distribution.
//
// THIS SOFTWARE IS PROVIDED BY THE BEYOND CIRCUITS ``AS IS'' AND ANY EXPRESS OR
// IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVEN
// SHALL BEYOND CIRCUITS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
// EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
// OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRAC
// STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WA
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAM
//-----------------------------------------------------------------------------
```