Implementing FIFO Operations Using TIE Queues

Application Note
Implementing FIFO Operations Using TIE Queues

© 2006 Tensilica, Inc.

Printed in the United States of America

All Rights Reserved

This publication is provided "AS IS." Tensilica, Inc. (hereafter "Tensilica") does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Information in this document is provided solely to enable system and software developers to use Tensilica processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereinunder to design or fabricate Tensilica integrated circuits or integrated circuits based on the information in this document. Tensilica does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Tensilica is a registered trademark of Tensilica, Inc. The following terms are trademarks of Tensilica, Inc.: FLIX, OShKit, Sea of Processors, TurboXim, Vectra, Xenergy, Xplorer, and XPRES. All other trademarks and registered trademarks are the property of their respective companies.

Dolby, AC-3, and the double-D symbol are trademarks of Dolby Laboratories. All other trademarks and registered trademarks are the property of their respective companies.

Supply of this Implementation of Dolby Technology does not convey a license nor imply a right under any patent, or any other Industrial or Intellectual Property Right of Dolby Laboratories, to use this Implementation in any finished end user or ready-to-use final product. Companies planning to use this Implementation in products must obtain a license from Dolby Laboratories Licensing Corporation before designing such products.

Notice

Tensilica, Inc. reserves the right to make changes to its products or discontinue any of its products or offerings without notice.

Tensilica warrants the performance of its products to the specifications applicable at the time of sale in accordance with Tensilica’s standard warranty.

Document Change History:

Published December 2006
## Contents

1. Introduction ................................................................................................................................. 1

2. FIFO Operations for a 32-bit Queue......................................................................................... 1
   - Output Queue .......................................................................................................................... 1
     - Full ......................................................................................................................................... 1
     - Push ....................................................................................................................................... 2
     - Non-Blocking Push ................................................................................................................ 2
   - Input Queue ............................................................................................................................... 3
     - Empty .................................................................................................................................... 3
     - Pop ....................................................................................................................................... 4
     - Non-Blocking Pop .................................................................................................................. 4
   - Example q32test.c ...................................................................................................................... 5

3. Operations for a 16-bit Queue................................................................................................... 5
   - Output Queue ............................................................................................................................ 6
   - Input Queue .............................................................................................................................. 6
     - Pop ....................................................................................................................................... 6
     - POPsshort.InQ ....................................................................................................................... 6
     - POPushort.InQ ...................................................................................................................... 7
     - POPschar.InQ ....................................................................................................................... 8
     - POPuchar.InQ ....................................................................................................................... 8
   - Example q16test.c ..................................................................................................................... 9

4. Memory ......................................................................................................................................... 9
   - Output Queue ............................................................................................................................ 9
     - Full ....................................................................................................................................... 9
     - Push ..................................................................................................................................... 10
     - Non-Blocking Push .............................................................................................................. 10
   - Input Queue ............................................................................................................................ 11
     - Empty .................................................................................................................................... 11
     - Pop ..................................................................................................................................... 11
     - Non-Blocking Pop ................................................................................................................. 11
   - Example q32memtest.c ........................................................................................................... 12

5. User-Defined Register Files and State.................................................................................. 12

6. Implementing Non-Blocking Queue Operations .................................................................. 13
   - Boolean Registers .................................................................................................................. 14
     - Non-Blocking Push .............................................................................................................. 14
     - Non-Blocking Pop ............................................................................................................... 14
   - Example q32booltest.c ......................................................................................................... 15
User-Defined Registers ........................................................................................ 15
User-Defined State ............................................................................................... 16
7 Example Code ...................................................................................................... 16
Abstract
This application note provides TIE and C code that demonstrate how to implement standard FIFO (First In, First Out) operations using TIE queues. The standard FIFO operations described in this document include testing for a full or empty FIFO, pushing a value onto a FIFO, and popping a value from a FIFO. This application note also shows how to describe the C data type of the values carried by a FIFO to ensure correct and efficient code generation by the Xtensa C/C++ compiler.
1 Introduction

This application note provides TIE and C code that demonstrate how to implement standard FIFO (First In, First Out) operations using TIE queues. The standard FIFO operations described in this document include testing for a full or empty FIFO, pushing a value onto a FIFO, and popping a value from a FIFO. Both blocking and non-blocking versions of the push and pop operations are described (the non-blocking push and pop operations can only be implemented on Xtensa LX2 or later cores).

This application note also shows how to describe the C data type of the values carried by a FIFO to ensure correct and efficient code generation by the Xtensa C/C++ compiler.

A TIE queue can be used in many ways that are beyond the scope of this application note. For example, a queue can be accessed conditionally, a queue access can be combined with computation, and multiple queues can be accessed in a single operation. For more information on other TIE queue uses, see the TIE Reference Manual and the TIE User’s Guide.

This application note assumes the reader is familiar with TIE queues as described in the TIE Reference Manual and with the Xtensa C/C++ compiler as described in the Xtensa C and C++ Compiler User’s Guide.

2 FIFO Operations for a 32-bit Queue

This section describes a TIE implementation of the operations that enable access to a 32-bit queue. The TIE described in this section is provided in file q32.tie. The C code showing usage examples for the TIE is provided in file q32test.c. See Section 7 for directions on installing the example TIE and C files.

Output Queue

File q32.tie defines a 32-bit wide output queue interface called OutQ:

```c
/*
 * Queue OutQ : 32-bit wide output queue interface
 */
queue OutQ 32 out
```

Full

The FULL.OutQ operation checks if the queue connected to the OutQ interface is full. The operation simply assigns the OutQ_NOTRDY signal to the output AR register.

```c
/*
 * FULL.OutQ
 *
 * Check if queue 'OutQ' is full. Return 1 in 'a' if 'OutQ' is full.
 * Return 0 in 'a' if 'OutQ' is not full.
 */
operation FULL.OutQ { out AR a } { in OutQ_NOTRDY } {
    assign a = { 31'b0, OutQ_NOTRDY };}
```

You can use the FULL.OutQ operation in your C code to check if a queue is full, as demonstrated in the following example.
Implementing FIFO Operations Using TIE Queues

if (FULL_OutQ())
    printf("OutQ is full.\n");
else
    printf("OutQ is not full.\n");

Push

The PUSH_OutQ operation pushes a 32-bit value onto the queue connected to the OutQ interface. The operation stalls until the queue connected to OutQ is not full and then performs the push. The operation simply assigns the input AR register value to the OutQ interface.

/*
* PUSH.OutQ
*
* Push 32-bit value 'a' onto queue 'OutQ'. If 'OutQ' is full, stall
* until 'OutQ' is not full and then push 'a'.
*/
operation PUSH.OutQ { in AR a } { out OutQ }
    assign OutQ = a;
}

You can use the PUSH_OutQ operation in your C code to push a value onto the queue connected to the OutQ interface, as demonstrated in the following example. In this example, the data type of the value being pushed is an unsigned int (on Xtensa an unsigned int is a 32-bit value). Section 3 describes how other C data types can be pushed onto a queue.

unsigned int v = …;
PUSH_OutQ(v);

Non-Blocking Push

The NBPUSH_OutQ operation pushes a 32-bit value onto the queue connected to the OutQ interface if the queue is not full. The NBPUSH_OutQ operation does not stall if the queue is full; instead, NBPUSH_OutQ returns an error code to indicate that the push failed. The operation assigns the OutQ_NOTRDY interface value to the OutQ_KILL interface so that the push is cancelled if the queue connected to OutQ is full (cancelling a queue operation by assigning to the queue KILL signal is not available in the LX core, and so the non-blocking push operation can only be implemented in LX2 or later cores).

/*
* NBPUSH.OutQ
*
* Attempt to push 32-bit value 'a' onto queue 'OutQ'. If 'OutQ' is
* not full, push 'a' and return value 1 in 'p' to indicate that the
* push succeeded. If 'OutQ' is full, do not push 'a' and return value
* 0 in 'p' to indicate that the push failed.
*/
operation NBPUSH.OutQ { out AR p, in AR a }
    { out OutQ, in OutQ_NOTRDY, out OutQ_KILL }
    assign OutQ = a;
    assign OutQ_KILL = OutQ_NOTRDY;
    assign p = { 31'b0, ~OutQ_NOTRDY };
You can use the `NBPUSH.OutQ` operation in your C code to push a value onto the queue connected to the `OutQ` interface, as demonstrated in the following example. In this example, an array of values is pushed onto the queue until the queue is full.

```c
unsigned int *v;
for (i = 0; i < N; i++)
    if (!NBPUSH_OutQ(v[i])) break;
```

The non-blocking push operation combines the functionality of a queue full operation followed by a queue push operation, as demonstrated by the following example implementing the same function as the loop above.

```c
unsigned int *v;
for (i = 0; i < N; i++)
    if (!FULL_OutQ())
        PUSH_OutQ(v[i]);
    else
        break;
```

Implementing the loop with the non-blocking push operation requires fewer instructions than the loop implemented using the full and push operations, and so will typically result in more efficient code.

**Input Queue**

File `q32.tie` defines a 32-bit wide input queue interface called `InQ`:

```c
/*
* Queue InQ : 32-bit wide input queue interface
*/
queue InQ 32 in
```

**Empty**

The `EMPTY.InQ` operation checks if the queue connected to the `InQ` interface is empty. The operation simply assigns the `InQ_NOTRDY` signal to the output AR register.

```c
/*
* EMPTY.InQ
* 
* Check if queue 'InQ' is empty. Return 1 in 'a' if 'InQ' is empty.
* Return 0 in 'a' if 'InQ' is not empty.
* /
operation EMPTY.InQ { out AR a } { in InQ_NOTRDY } {
    assign a = { 31'b0, InQ_NOTRDY };}
```

You can use the `EMPTY.InQ` operation in your C code to check if a queue is empty, as demonstrated in the following example.

```c
if (EMPTY_InQ())
    printf("InQ is empty.\n");
else
    printf("InQ is not empty.\n");
```
Pop

The POP.InQ operation pops a 32-bit value from the queue connected to the InQ interface. The operation stalls until the queue connected to InQ is not empty and then performs the pop. The operation simply assigns the InQ interface value to the output AR register.

```
/*
 * POP.InQ
 * * Pop 32-bit value from queue 'InQ' and assign to 'a'. If 'InQ'
 * is empty, stall until 'InQ' is not empty and then pop.
 */
operation POP.InQ { out AR a } { in InQ } {
    assign a = InQ;
}
```

You can use the POP.InQ operation in your C code to pop a value from the queue connected to the InQ interface, as demonstrated in the following example. In this example, the data type of the value being popped is an unsigned int (on Xtensa an unsigned int is a 32-bit value). Section 3 describes how other data types can be popped from a queue.

```
unsigned int v = POP_InQ();
```

Non-Blocking Pop

The NBPOP.InQ operation pops a 32-bit value from the queue connected to the InQ interface if the queue is not empty. The NBPOP.InQ operation does not stall if the queue is empty; instead, NBPOP.InQ sets the 1-bit NBQRES state to indicate that the pop failed. The operation assigns the InQ_NOTRDY interface value to the InQ_KILL interface so that the pop is cancelled if the queue connected to InQ is empty (cancelling a queue operation by assigning to the queue KILL signal is not available in the LX core, and so the non-blocking pop operation can only be implemented in LX2 or later cores). The NBQRES state is used to record pop success or failure because a TIE operation can have at most one output to each register file. For the NBPOP.InQ operation, the popped value is written to the AR register file, and so the pop success/failure status cannot also be written to the AR register file (as is done for the NBPUSH.OutQ operation). Section 6 discusses several alternatives to using the NBQRES state for the non-blocking pop operation.

```
/*
 * NBPOP.InQ
 * * Attempt to pop a 32-bit value from queue 'InQ'. If 'InQ' is not empty,
 * pop the queue and assign the value to 'a', and set NBQRES to 1 to
 * indicate that the pop succeeded. If 'InQ' is empty, set NBQRES to 0 to
 * indicate that the pop failed (in this case the value of 'a' is
 * unchanged).
 */
operation NBPOP.InQ { out AR a } { out NBQRES, in InQ, in InQ_NOTRDY, out InQ_KILL } {
    assign a = InQ;
    assign a_kill = InQ_NOTRDY;
    assign InQ_KILL = InQ_NOTRDY;
    assign NBQRES = ~InQ_NOTRDY;
}
```
You can use the `NBPOP_InQ` operation in your C code to pop a value from the queue connected to the `InQ` interface, as demonstrated in the following example. In this example, an array of values is popped from the queue until the queue is empty.

```c
unsigned int *v;
for (i = 0; i < N; i++) {
    NBPOP_InQ(v[i]);
    if (RUR_NBQRES() == 0) break;
}
```

The non-blocking pop operation combines the functionality of a queue empty operation followed by a queue pop operation, as demonstrated by the following example implementing the same function as the loop above.

```c
unsigned int *v;
for (i = 0; i < N; i++)
    if (!EMPTY_InQ())
        v[i] = POP_InQ();
    else
        break;
```

As presented, the non-blocking pop operation does not require fewer instructions to implement the example loop, because a `RUR_NBQRES` operation is required to read the status of the non-blocking pop (that is, if the pop succeeded or failed). Section 6 discusses alternate implementations of non-blocking pop that avoid the need for `RUR_NBQRES`.

### Example q32test.c

File `q32test.c` contains example code that demonstrates the use of all the 32-bit queue operations. The TIE required by the example is contained in `q32.tie`. See Section 7 for directions on installing the example TIE and C files.

### 3 Operations for a 16-bit Queue

This section describes a TIE implementation of the operations that enable access to a 16-bit queue. The implementation specifically illustrates how a queue can be used to carry C `short` and `char` data type values. More generally, this section discusses the requirements of the Xtensa C/C++ compiler in regards to C data type values pushed onto and popped from queues. The TIE described in this section is provided in file `q16.tie`. The C code showing usage examples for the TIE is provided in file `q16test.c`. See Section 7 for directions on installing the example TIE and C files.

The Xtensa C/C++ compiler (XCC) uses the 32-bit AR registers to hold signed and unsigned versions of `int`, `short`, and `char` data type values. An `int` value’s size is 32 bits, a `short` value’s size is 16 bits, and a `char` value’s size is 8 bits. For values of these data types, XCC maintains a 32-bit representation of the value when it is stored in an AR register. For example, the 8 bits representing a `signed char` value are stored in the least-significant 8 bits of an AR register, and the most-significant 24 bits of the AR register are filled with the sign-extension of the 8-bit value. The sign-extension is performed automatically by XCC as necessary.

For efficient and correct code, any queue operations that write `short` or `char` data type values to an AR register must perform the correct sign- or zero-extension as expected by XCC. The operations described in this section illustrate how to satisfy this requirement.
Output Queue

File q16.tie defines a 16-bit wide output queue interface called OutQ. To simplify the example, the TIE file contains only a push operation for OutQ. As described in Section 2, a full and non-blocking push operation can also be defined for OutQ.

```c
/*
 * Queue OutQ : 16-bit wide output queue interface
 */
queue OutQ 16 out
```

Push

The PUSH.OutQ operation pushes the least-significant 16 bits of an AR register onto the queue connected to the OutQ interface. As described above, the least-significant bits of an AR register hold the significant bits of a short or char data type value. So by pushing the least-significant 16 bits of the AR register, the PUSH.OutQ operation pushes the entire 16-bit short or 8-bit char value.

```c
/*
 * PUSH.OutQ
 *
 * Push the least-significant 16 bits of 'a' onto queue 'OutQ'.
 * If 'OutQ' is full, stall until 'OutQ' is not full and then push.
 */
operation PUSH.OutQ { in AR a } { out OutQ } {
    assign OutQ = a[15:0];
}
```

Input Queue

File q16.tie defines a 16-bit wide input queue interface called InQ. To simplify the example, the TIE file contains only pop operations for InQ. As described in Section 2, an empty and non-blocking pop operation can also be defined for InQ.

```c
/*
 * Queue InQ : 16-bit wide input queue interface
 */
queue InQ 16 in
```

Pop

As described above, a short or char data type value must be represented in an AR register as a full 32-bit value. So, depending on the required data type of the popped value, a pop operation must zero- or sign-extend the value appropriately.

**POPsshort.InQ**

The POPsshort.InQ operation pops the 16-bit value from the head of the queue connected to the InQ interface and then sign-extends the value to form the 32-bit signed short representation of the value.
IMPLEMENTING FIFO OPERATIONS USING TIE QUEUES

* POPsshort.InQ *
* Pop value from queue 'InQ', sign-extend it to form the signed short *
* representation of the value, and assign to 'a'. If 'InQ' is empty, *
* stall until 'InQ' is not empty and then pop. *
*
operation POPsshort.InQ { out AR a } { in InQ } {
  assign a = { 16{InQ[15]}, InQ[15:0] }; 
}

You can use the POPsshort.InQ operation in your C code to pop a signed short value from the head of the queue connected to the InQ interface.

signed short ss = POPsshort_InQ();

For this example code, XCC will generate assembly code like the following. By default, TIE instructions that write the AR register file are assumed to write an unsigned int value into the register. And so XCC inserts a SEXT instruction to convert the unsigned int value to a signed short value (if your processor configuration does not include the SEXT instruction, then XCC will insert a pair of shifts that performs the equivalent operation).

popsshort.inq a2
sext a2, a2, 15

The SEXT instruction is unnecessary because POPsshort.InQ is already converting the 16-bit value into a signed short representation. We use the TIE proto construct to indicate that the POPsshort.InQ operation writes an int16 value (int16 is the TIE ctype that represents the signed short data type) into the AR register.

proto POPsshort.InQ { out int16 a } { } {
  POPsshort.InQ a;
}

Including the POPsshort.InQ proto in the TIE file allows XCC to recognize that POPsshort.InQ produces a signed short value, and so XCC does not generate the SEXT instruction.

POPushort.InQ

The POPushort.InQ operation pops the 16-bit value from the head of the queue connected to the InQ interface and then zero-extends the value to form the 32-bit unsigned short representation of the value. The TIE proto construct indicates that the POPushort.InQ operation writes a uint16 value (uint16 is the TIE ctype that represents the unsigned short data type) into the AR register.
/*
 * POPushort.InQ
 *
 * Pop value from queue 'InQ', zero-extend to form the unsigned short
 * representation of the value, and assign to 'a'. If 'InQ' is empty,
 * stall until 'InQ' is not empty and then pop.
 */
operation POPushort.InQ { out AR a } { in InQ } {
    assign a = { 16'b0, InQ[15:0] }; }

proto POPushort.InQ { out uint16 a } { }
{ POPushort.InQ a; }

You can use the POPushort.InQ operation in your C code to pop an unsigned short value from the head of the queue connected to the InQ interface.
unsigned short us = POPushort_InQ();

**POPschar.InQ**

The POPschar.InQ operation pops the 16-bit value from the head of the queue connected to the InQ interface and then sign-extends the value to form the 32-bit signed char representation of the value (note that an 8-bit wide queue would be sufficient for a signed or unsigned char value). The TIE proto construct indicates that the POPschar.InQ operation writes an int8 value (int8 is the TIE ctype that represents the signed char data type) into the AR register.

/*
 * POPschar.InQ
 *
 * Pop value from queue 'InQ', sign-extend to form the signed char
 * representation of the value, and assign to 'a'. If 'InQ' is empty,
 * stall until 'InQ' is not empty and then pop.
 */
operation POPschar.InQ { out AR a } { in InQ } {
    assign a = { 24{InQ[7]}, InQ[7:0] }; }

proto POPschar.InQ { out int8 a } { }
{ POPschar.InQ a; }

You can use the POPschar.InQ operation in your C code to pop a signed char value from the head of the queue connected to the InQ interface.
signed char sc = POPschar_InQ();

**POPuchar.InQ**

The POPuchar.InQ operation pops the 16-bit value from the head of the queue connected to the InQ interface and then zero-extends the value to form the 32-bit unsigned char...
representation of the value. The TIE proto construct indicates that the POPuchar.InQ operation writes an uint8 value (uint8 is the TIE ctype that represents the unsigned char data type) into the AR register.

```c
/*
 * POPuchar.InQ
 *
 * Pop value from queue 'InQ', zero-extend to form the unsigned char
 * representation of the value, and assign to 'a'. If 'InQ' is empty,
 * stall until 'InQ' is not empty and then pop.
 */
operation POPuchar.InQ { out AR a } { in InQ } {
    assign a = { 24'b0, InQ[7:0] };}

proto POPuchar.InQ { out uint8 a } { }
{
    POPuchar.InQ a;
}
```

You can use the POPuchar.InQ operation in your C code to pop an unsigned char value from the head of the queue connected to the InQ interface.

```c
unsigned char uc = POPuchar_InQ();
```

**Example q16test.c**

File q16test.c contains example code that demonstrates how to push and pop char and short data type values onto a 16-bit queue. The TIE required by the example code is contained in q16.tie.

### 4 Memory

The TIE operations described in Section 2 and Section 3 push values from an AR register and pop values into an AR register. Values can also be pushed from and popped to memory.

This section describes a TIE implementation of the operations that enable access to a 32-bit queue where values are pushed directly from memory and values are popped directly to memory. The TIE described in this section is provided in file q32mem.tie. The C code showing usage examples for the TIE is provided in file q32memtest.c. See Section 7 for directions on installing the example TIE and C files.

**Output Queue**

File q32mem.tie defines a 32-bit wide output queue interface called OutQ:

```c
/*
 * Queue OutQ : 32-bit wide output queue interface
 */
queue OutQ 32 out
```

**Full**

The FULL.OutQ operation checks if the queue connected to the OutQ interface is full. The implementation is identical to that described in Section 2.
Implementing FIFO Operations Using TIE Queues

Push

The PUSH.OutQ operation loads a 32-bit value directly from memory and pushes the value onto the queue connected to the OutQ interface. The operation stalls until the queue connected to OutQ is not full and then performs the push.

```
/*
  * PUSH.OutQ
  *
  * Load a 32-bit value from the memory address specified by 'a' and push
  * the value onto queue 'OutQ'. If 'OutQ' is full, stall until 'OutQ' is
  * not full and then push the value.
  */
operation PUSH.OutQ { in AR *a } { out VAddr, in MemDataIn32, out OutQ } {
  assign VAddr = a;
  assign OutQ = MemDataIn32;
}
```

You can use the PUSH.OutQ operation in your C code to push a value onto the queue connected to the OutQ interface, as demonstrated in the following example. Notice that the PUSH.OutQ operation takes a pointer value argument indicating the memory address of the value to be pushed.

```c
unsigned int *v = …;
PUSH_OutQ(&v[3]);
```

Non-Blocking Push

The NBPUSH.OutQ operation loads a 32-bit value directly from memory and pushes it onto the queue connected to the OutQ interface if the queue is not full. The NBPUSH.OutQ operation does not stall if the queue is full; instead, NBPUSH.OutQ returns an error code to indicate that the push failed. The operation assigns the OutQ.NOTRDY interface value to the OutQ.KILL interface so that the push is cancelled if the queue connected to OutQ is full (cancelling a queue operation by assigning to the queue KILL signal is not available in the LX core, and so the non-blocking push operation can only be implemented in LX2 or later cores). We cannot use the LoadByteDisable interface to also kill the load when OutQ is full, because the full indication comes too late to be used for LoadByteDisable.

```
/*
  * NBPUSH.OutQ
  *
  * Load a 32-bit value from the memory address specified by 'a' and
  * attempt to push the value onto queue 'OutQ'. If 'OutQ' is not full,
  * push the value and return 1 in 'p' to indicate that the push succeeded.
  * If 'OutQ' is full, do not push the value and return 0 in 'p' to
  * indicate that the push failed.
  */
operation NBPUSH.OutQ { out AR p, in AR *a } {
  out VAddr, in MemDataIn32,
  out OutQ, in OutQ.NOTRDY, out OutQ.KILL } {
  assign VAddr = a;
  assign OutQ = MemDataIn32;
  assign OutQ.KILL = OutQ.NOTRDY;
  assign p = { 31'b0, ~OutQ.NOTRDY };
}
```
Implementing FIFO Operations Using TIE Queues

You can use the NBPUSH.OutQ operation in your C code to push a value onto the queue connected to the OutQ interface, as demonstrated in the following example. In this example, an array of values is pushed onto the queue until the queue is full.

```c
unsigned int *v;
for (i = 0; i < N; i++)
    if (!NBPUSH_OutQ(&v[i])) break;
```

**Input Queue**

File q32mem.tie defines a 32-bit wide input queue interface called InQ:

```c
/*
 * Queue InQ : 32-bit wide input queue interface
 */
queue InQ 32 in
```

**Empty**

The EMPTY.InQ operation checks if the queue connected to the InQ interface is empty. The implementation is identical to that described in Section 2.

**Pop**

The POP.InQ operation pops a 32-bit value from the queue connected to the InQ interface and stores it directly to memory. The operation stalls until the queue connected to InQ is not empty and then performs the pop.

```c
/*
 * POP.InQ
 *
 * Pop 32-bit value from queue 'InQ' and store to the memory address
 * specified by 'a'. If 'InQ' is empty, stall until 'InQ' is not
 * empty and then pop.
 */
operation POP.InQ { in AR *a } { out VAddr, out MemDataOut32, in InQ } {
    assign VAddr = a;
    assign MemDataOut32 = InQ;
}
```

You can use the POP.InQ operation in your C code to pop a value from the queue connected to the InQ interface, as demonstrated in the following example.

```c
unsigned int *v = ...;
POP_InQ(&v[3]);
```

**Non-Blocking Pop**

The NBPOP.InQ operation pops a 32-bit value from the queue connected to the InQ interface if the queue is not empty. The popped value is stored directly to memory. The NBPOP.InQ operation does not stall if the queue is empty; instead, NBPOP.InQ returns an error code to indicate that the pop failed. The operation assigns the InQ_NOTRDY interface value to the InQ_KILL interface so that the pop is cancelled if the queue connected to InQ is empty (cancelling a queue operation by assigning to the queue KILL signal is not available in the LX core, and so the non-blocking pop operation can only be implemented in LX2 or later cores). We use the StoreByteDisable interface to also kill the store when InQ is empty so that nothing
Implementing FIFO Operations Using TIE Queues

is stored when the non-blocking pop fails (to use the StoreByteDisable interface the Arbitrary Byte Enables option must be enabled in your processor configuration).

```c
/*
 * NBPOP.InQ
 *
 * Attempt to pop a 32-bit value from queue 'InQ'. If 'InQ' is not empty,
 * pop the queue and store the value to the memory address specified by
 * 'a', and set 'p' to 1 to indicate that the pop succeeded. If 'InQ' is
 * empty, set 'p' to 0 to indicate that the pop failed (in this case
 * nothing is stored to the memory address specified by 'a').
 */
operation NBPOP.InQ { out AR p, in AR *a }
  { out VAddr, out MemDataOut32, out StoreByteDisable,
    in InQ, in InQ_NOTRDY, out InQ_KILL }
  assign VAddr = a;
  assign MemDataOut32 = InQ;
  assign StoreByteDisable = {16{InQ_NOTRDY}};
  assign InQ_KILL = InQ_NOTRDY;
  assign p = { 31'b0, ~InQ_NOTRDY };}
```

You can use the NBPOP.InQ operation in your C code to pop a value from the queue connected to the InQ interface, as demonstrated in the following example. In this example, an array of values is popped from the queue until the queue is empty.

```c
unsigned int *v;
for (i = 0; i < N; i++) {
  if (!NBPOP_InQ(&v[i])) break;
}
```

**Example q32memtest.c**

File q32memtest.c contains example code that demonstrates the use of all the 32-bit queue operations that push from and pop to memory. The TIE required by the example is contained in q32mem.tie. See Section 7 for directions on installing the example TIE and C files.

## 5 User-Defined Register Files and State

The TIE operations described Section 2 and Section 3 push values from an AR register and pop values into an AR register. Values can also be pushed from and popped to user-defined registers and user-defined states. As shown in Section 6, implementing FIFO operations that read and write user-defined registers and user-defined state allows efficient implementation of non-blocking operations because either an AR or BR register can be used to signal success or failure of the operation.

The following example TIE shows an implementation of a pop operation that writes a 24-bit user-defined register file.
/*
* Regfile XR : 24-bit user-defined register file
*/
regfile XR 24 8 x

/*
* Queue In24Q : 24-bit wide input queue interface
*/
queue In24Q 24 in

/*
* POP.In24Q
*
* Pop 24-bit value from queue 'In24Q' and assign to 'a'. If 'In24Q'
* is empty, stall until 'In24Q' is not empty and then pop.
*/
operation POP.In24Q { out XR a } { in In24Q } {
  assign a = In24Q;
}

The following similar example shows an implementation of a push operation that reads from a
24-bit user-defined state.

/*
* State S24 : 24-bit user-defined state
*/
state S24 24 add_read_write

/*
* Queue Out24Q : 24-bit wide output queue interface
*/
queue Out24Q 24 out

/*
* PUSH.Out24Q
*
* Push 24-bit value from state 'S24' to queue 'Out24Q'. If 'Out24Q'
* is full, stall until 'Out24Q' is not full and then push.
*/
operation PUSH.Out24Q { } { out S24, in In24Q } {
  assign Out24Q = S24;
}

6 Implementing Non-Blocking Queue Operations

Section 2 describes non-blocking push and pop operations. As presented, the non-blocking pop
operation uses a 1-bit state, NBQRES, to indicate success or failure of the operation. The non-
blocking pop operation implemented using NBQRES will work on any LX or LX2 processor
configuration. However, a more efficient implementation is possible when the processor
configuration includes the Boolean register option, when user-defined registers are used to hold queue data, or when user-defined state is used to hold queue data.

**Boolean Registers**

The Boolean registers configuration option adds 16 1-bit BR registers to the processor configuration. The option also adds several new branch instructions that use a BR register for the branch condition. The TIE described in this section is provided in file q32bool.tie. In q32bool.tie, a BR register is used to hold the success or failure of a non-blocking operation (for consistency, a BR register is also used to hold the result of a queue full or empty operation).

**Non-Blocking Push**

In Section 2, the non-blocking push operation writes an AR register to indicate the success or failure status of the operation. This is possible because the non-blocking push operation does not otherwise require an AR register write. When Boolean registers are available in the processor configuration, the non-blocking push operation can be implemented by writing the success or failure status to a BR register. Using a BR register makes the non-blocking push operation consistent with the non-blocking pop operation. Also, because AR registers are a limited resource, using a BR register instead of an AR register can allow the compiler to generate better code.

```c
/*
 * NBPUSH.OutQ
 *
 * Attempt to push 32-bit value 'a' onto queue 'OutQ'. If 'OutQ' is not full, push 'a' and return boolean value 1 (true) in 'p' to indicate that the push succeeded. If 'OutQ' is full, do not push 'a' and return boolean value 0 (false) in 'p' to indicate that the push failed.
 */
operation NBPUSH.OutQ { out BR p, in AR a }
    { out OutQ, in OutQ_NOTRDY, out OutQ_KILL }
    assign OutQ = a;
    assign OutQ_KILL = OutQ_NOTRDY;
    assign p = ~OutQ_NOTRDY;
}
```

**Non-Blocking Pop**

In Section 2, the non-blocking pop operation writes the NBQRES state to indicate the success or failure status of the operation. Reading the NBQRES state to check the status requires an additional operation. Boolean registers allow a more efficient implementation that writes the success of failure status to a BR register.
Implementing FIFO Operations Using TIE Queues

/*
 * NBPOP.InQ
 *
 * Attempt to pop a 32-bit value from queue 'InQ'. If 'InQ' is not empty,
 * pop the queue and assign the value to 'a', and set 'p' to boolean
 * value 1 (true) to indicate that the pop succeeded. If 'InQ' is empty,
 * set 'p' to boolean value 0 (false) to indicate that the pop failed
 * (in this case the value of 'a' is unchanged).
 */
operation NBPOP.InQ { out BR p, out AR a }
  { in InQ, in InQ_NOTRDY, out InQ_KILL }
  
  assign a = InQ;
  assign p = ~InQ_NOTRDY;
  assign a_kill = InQ_NOTRDY;
  assign InQ_KILL = InQ_NOTRDY;
}

Using the Boolean register version of the non-blocking pop in your code is illustrated in the following example.

unsigned int v;
xtbool p;
NBPOP_InQ(p, v);
if (p) {
  // pop successful...
}

Example q32booltest.c

File q32booltest.c contains example code that demonstrates the use of all the 32-bit queue operations that use a BR register to indicate success or failure of a non-blocking queue operation. This example code requires the TIE contained in q32bool.tie.

User-Defined Registers

As shown in Section 5, FIFO values can be pushed from a user-defined register and popped to a user-defined register. When a user-defined register is used to hold the FIFO value, a non-blocking push or non-blocking pop operation can be efficiently implemented by using an AR register (or BR register as described above) to signal the success or failure of the operation. The following example TIE shows an implementation of a non-blocking pop for the XR24 regfile and In24Q queue defined in Section 5.
Implementing FIFO Operations Using TIE Queues

User-Defined State

As shown in Section 5, FIFO values can be pushed from a user-defined state and popped to a user-defined state. When a user-defined state is used to hold the FIFO value, a non-blocking push or non-blocking pop operation can be efficiently implemented by using an AR register (or BR register as described above) to signal the success or failure of the operation. The following example TIE shows an implementation of a non-blocking push for the \texttt{S24} state and \texttt{Out24Q} queue defined in Section 5.

7 Example Code

The example TIE and C code referenced in this application note are contained in the associated archive. The archive contains \texttt{q32.tie}, \texttt{q32mem.tie}, \texttt{q32bool.tie}, \texttt{q16.tie}, \texttt{q32test.c}, \texttt{q32memtest.c}, \texttt{q32booltest.c} and \texttt{q16test.c}.

For each example first use the TIE Compiler to compile the TIE (see the \textit{TIE User's Guide} for more details). Some of the TIE files contain implementation of non-blocking operations that will not compile on LX cores. If you are using an LX core, first edit the TIE files to remove the non-
blocking operations. To compile `q32bool.tie` your configuration must have the Boolean Registers option. To compile `q32mem.tie` your configuration must have the Arbitrary Byte Enables option. For example, use the following command to compile `q32.tie`.

```
tc -d q32_tdk q32.tie
```

Next compile the corresponding C code using XCC and the `--xtensa-params` flag to point to the TDK created by `tc`. If you are using an LX core, compile like this to exclude the code that uses the non-blocking FIFO operations (the non-blocking operations are not supported for LX cores).

```
xt-xcc --xtensa-params=q32_tdk -O2 -DNO_NB q32test.c -o q32test
```

If you are using a LX2 core, compile like this to include the code that uses the non-blocking FIFO operations.

```
xt-xcc --xtensa-params=q32_tdk -O2 q32test.c -o q32test
```

If you use the `-S` flag when invoking `xt-xcc`, the compiler will output the assembly file for the source instead of generating object code. For example, the following command will produce `q32test.s`. You can examine `q32test.s` to see how XCC generated code for each of the FIFO operations.

```
xt-xcc --xtensa-params=q32_tdk -O2 -S q32test.c
```

The examples assume a single processor system with a loop-back queue connecting the `OutQ` interface to the `InQ` interface, as shown below. When compiling for this single processor system, XCC has no notion of the dependence between `InQ` and `OutQ`, and so accesses to these queues may be reordered by XCC, potentially leading to system deadlock. To prevent XCC from reordering dependent queue accesses, the examples use pragmas described in the `Xtensa C and C++ Compiler User’s Guide` and in the TIE queue “Synchronization and Ordering” section of the `TIE Reference Manual`.

```
With Xtensa Tools version 7 and later, a SystemC model of this system can be created with the following `xtsc-run` command (for more information about `xtsc-run`, see the `Xtensa XTSC User’s Guide`).

```
xWSC-run --xtensa_params=<tdk> --core_program=<program> -create_core=core
 -set_queue_parm=depth=3 -set_queue_parm=bit_width=<width> -create_queue=Q
 -connect_core_queue=core,OutQ,Q -connect_queue_core=Q,InQ,core
```

Where:

- `<tdk>` is the TDK directory generated from the example TIE file (for example, the TDK generated from `q32.tie`).
- `<program>` is the program to run on the core (for example, the program compiled from `q32test.c`).
- `<width>` is the bit-width of the queue. This should be 16 for `q16.tie` and 32 for all other TIE files.