XC Programming Guide

REV A

Publication Date: 2013/6/4

XMOS

SYNOPSIS

This document provides a comprehensive guide to XC programming. XC provides extensions to C that simplify the control over concurrency, I/O and time. These extensions map directly onto XCore hardware resources such as threads, channels and ports, avoiding the need to make extensive use of library calls. XC's constructs are efficient compiled into short instruction sequences, and free from many sources of deadlock, race conditions and memory violations. This makes programs easy to write, understand and debug.

Table of Contents

1 Computation

IN THIS CHAPTER

XC is an imperative programming language with a computational framework based on C. XC programs consist of functions that execute statements that act upon values stored in variables. Control-flow statements express decisions, and looping statements express iteration.

NEW XC In the following sections, constructs that are new to XC or that differ from C are noted in the margin.

1.1 Hello, World!

The first task often performed when learning a new programming language is to print the words "Hello, world!" A suitable XC program is shown below.

```
#include <stdio.h>

main(void) {
  printf("Hello, world!\n");
}
```

The first line of this program tells the compiler to include information from the header file `stdio.h`. This file contains a declaration of the function `printf`, which outputs a string to standard output, for example a terminal window on a development system.

Every program must contain a single `main` function, which is where the program begins executing. In this example, `main` is defined as a function that expects no arguments, indicated by the keyword `void`.

The body of a function is enclosed in braces `{}` and contains statements that specify operations to be performed. In this example, `main` contains a single statement that calls the function `printf` with a string literal as its argument. The escape sequence `\n` denotes a newline character.

1.2 Variables, Constants and Expressions

A variable represents a location in memory in which data is stored. All variables must be declared before use and given a type. The most common arithmetic types are `char` and `int`. A `char` is a byte that represents 8-bit integral numbers and an `int` represents 32-bit integral numbers. The declaration

```
char c;
```

declares `c` to be an 8-bit signed character that takes values between -128 and 127.

The qualifier `signed` or `unsigned` may be used to specify the signedness of a type.

The declaration

```
unsigned char c;
```

declares `c` to be an 8-bit unsigned character that takes values between 0 and 255.

A variable may be assigned an initial value. The declaration

```
int i=0, j=1;
```

declares `i` and `j` to be integers, initialised with the values 0 and 1.

The qualifier `const` may be applied to any variable declaration to prevent its value from being changed after initialisation. The declaration

```
const int MHz = 1000000;
```

declares `MHz` to represent an integer constant of value 1000000. Attempting to modify its value after initialisation is invalid.

One or more variables of the same type may be combined to form an *array*. The declaration

```
int data[3] = {1, 2, 3};
```

declares `data` to be an array of three integers and initialises it with values 1, 2 and 3. Array subscripts start at zero, so the elements of this array are `data[0]`, `data[1]` and `data[2]`. A subscript can be any integer expression that evaluates to a valid element of the array.

Arrays may be constructed from one another to form *multi-dimensional* arrays. The declaration

```
int matrix[2][3] = {{1 , 2, 3}, {4, 5, 6}};
```

declares `matrix` to be a two-dimensional array. The first dimension specifies a row, the second a column, producing the matrix below.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The subscripts are ordered by the largest dimension first so that, for example, the value of `matrix[0][1]` is 2.

1.2.1 Constants

A constant is a textual representation of a value, and has a data type. Entries in Figure 1 are all examples of constants.

Text	Type	Value
`123`	`int`	123
`123u`	`unsigned int`	123
`0b10000`	`int`	16
`020`	`int`	16
`0x10`	`int`	16
`0xAu`	`unsigned int`	10
`'x'`	`char`	120
`'0'`	`char`	48
`'\n'`	`char`	10 (newline)
`'\\'`	`char`	92 (backslash)
`'\0'`	`char`	0 (null terminator)
`"str"`	array of `char`	`'s', 't', 'r', '\0'`

Figure 1: Examples of Constants

A sequence of digits is by default an `int`. An unsigned constant is specified with the suffix `u`. An integer constant is specified in binary by using the prefix `0b`, in octal by using the prefix `0` and in hexadecimal by using the prefix `0x`.

A character constant is usually written as a character in single quotes. Its value is the numeric value of the character. Some characters that are not representable as characters are *escaped* using the backslash character.

A *string literal* is a sequence of zero or more characters enclosed in double quotes. The internal representation of a string literal includes a null character suffix `\0`, which allows programs to find the end of a string. This suffix also increases the string's storage requirements by a single byte. String literals are used to initialise arrays of characters, as in:

```
char msg[] = "Hello, world !\n";
```

This example declares msg to be an array of 15 characters, including the null terminator. If the size of the array is specified in the declaration, it must be at least as large as the string.

1.2.2 Expressions

An expression combines variables and constants with operators, producing a value. Entries in Figure 2 are all examples of expressions.

Algebraic Expression	XC Expression
a * b - c	`a * b - c`
(a + b)(c + d)	`(a + b) * (c + d)`
a/b + c	`(a/b) + c`

Figure 2: Examples of expressions

An expression without parenthesis is usually evaluated from left to right using the rules of precedence of operators. These rules state that the * operator has a higher precedence than the + operator, which means that the second expression in the table requires parenthesis around the two additions to force the required grouping.

Figure 3 summarises the expression operators supported in XC. Operators higher in the table have a higher precedence, and operators in the same section have the same precedence. The operators are defined to have the same meaning as in C; full details are given in *xc_spec_expressions*.

An expression becomes a statement when followed by a semicolon. Most statements are either assignments, as in:

```
x = a * b;
```

or function calls, as in:

```
printf("Hello, world!\n");
```

NEW XC

The value of an expression must be unambiguous. An ambiguity arises if the value of an expression depends on the order of evaluation of its operands, as in:

```
i = i++; /* invalid */
```

In this example, the value of `i` depends on the order in which the assignment and increment operators are performed.

In general, if one subexpression contains a modification of variable *V*, none of the other subexpressions are allowed to use *V*. This rule applies recursively to functions called in expressions that read or write global variables.

1.2.3 Type Conversions

If an operator has operands of different types, the operands are converted to a common type. In general, the "lower" type is *promoted* to the "higher" type before the operation proceeds; the result is of the higher type. For example, in the expression

```
'c' + 1
```

the binary operator + takes a `char` and an `int` operand. The `char` operand is promoted to an `int`, and the result of the expression is an `int`.

Operator	Description	Type	Associativity
`++ --`	Postfix increment/decrement	Unary	Left-to-right
`++ --`	Prefix increment/decrement	Unary	Right-to-left
`+ -`	Unary plus/minus		
`!`	Logical negation		
`~`	Bitwise complement		
(*type*)	Explicit cast		
`sizeof`	Determine size in bytes		
`isnull`	Determine whether null reference		
`* / %`	Multiplication, division, modulus	Binary	Left-to-right
`+ -`	Addition/subtraction	Binary	Left-to-right
`<< >>`	Bitwise shift left/right	Binary	Left-to-right
`<`	Relational less than	Binary	Left-to-right
`<=`	Relational less than or equal to		
`>`	Relational greater than		
`>=`	Relational greater than or equal to		
`== !=`	Relational equal to, not equal to	Binary	Left-to-right
`&`	Bitwise AND	Binary	Left-to-right
`^`	Bitwise exclusive OR	Binary	Left-to-right
`>`	Bitwise inclusive OR	Binary	Left-to-right
`&&`	Logical AND	Binary	Left-to-right
`||`	Logical OR	Binary	Left-to-right
c`?`*t*`:`*f*	Ternary conditional	Ternary	Right-to-left
`=`	Assignment	Binary	Right-to-left
`+= -= *= /=`	Arithmetic assignment		
`%= &= ^= |=`	Arithmetic assignment		
`<<= >>=`	Arithmetic assignment		

Figure 3: XC Expression Operators

The general rules of promotion and arithmetic conversion are stated in *xc_spec_arithmetic_conversions*, and for XS1 devices can be summarised as follows:

- Convert `char` and `short` to `int`, if an `int` can represent all the values of the original type, otherwise convert to `unsigned int`.
- If either operand is `unsigned int`, convert the other to `unsigned int`.

Explicit type conversions can be forced in an expression using the unary cast operator, as in:

```
(char)('a' + i); // cast 32-bit integer to an 8-bit char
```

Casts are often used with output statements to specify the amount of data to be communicated (see §3.3). The meaning of the cast is as if the expression were assigned to a variable of the specified type. A cast must not specify an array; neither must the expression.

1.3 Control Flow

Control-flow statements express decisions that determine the order in which statements are performed. A list of statements is executed in sequence by grouping them into a *block* using braces `{ }`, as in:

```
main(void) {
  int x = 2, y = 3;
  int z = x * y;
  z++;
}
```

In a block, all declarations must come at the top before the statements. A block is syntactically equivalent to a single statement and can be used wherever a statement is required.

1.3.1 If-Else

An `if-else` construct choses at most one statement to execute, as in:

```
if (n > 0)
  printf("Positive");
else if (n < 0) {
  printf("Negative");
  x = 0;
}
else
  printf("Zero");
```

Each parenthesised expression guards a statement or block that may be executed; the `else-if` and `else` statements are optional. The expressions are evaluated in the order they appear in the source code, and the first expression that produces a non-zero value causes the statement it guards to be executed. The entire construct then terminates.

An `else` statement is always associated with the previous `else`-less `if`, which is important to remember if multiple if statements are nested. Braces can be used to force a different association.

1.3.2 Switch

A `switch` statement tests whether an expression matches one of a number of constant integer values and branches to the body of code for the corresponding `case`, as in:

```
switch(state) {
  case READY :
    state = SET;
    if (x < 0)
      state = FAIL;
    break;
  case SET :
     state = GO;
     if (y > 0)
     state = FAIL;
     break;
  case GO:
    printf("Go!\n");
    break;
  case FAIL :
  default :
    /* error */
}
```

If a `default` label is present and none of the case constants equal the value of the expression, the code following `default` is executed instead.

NEW XC The body of each `case` must be terminated by either `break` or `return`, preventing control from flowing from one body to the next.

1.3.3 Loops

A `while` loop repeats a statement as long as the value of an expression remains non-zero, as in:

```
int i = 0;
while (i<n) {
  a[i] = b[i] * c[i];
  i++;
}
```

This example is typical of many programs that iterate over the first *n* elements of an array. An alternative form is to use a `for` loop, which provides a way to combine the initialisation, conditional test and increment together at the top of the loop. The example above may be alternatively written as:

```
for (int i=0; i<n; i++)
  a[i] = b[i] * c[i];
```

The first and third expressions are usually assignments, and the second a relational expression. The first assignment may form part of a variable declaration whose scope is local to the body of the loop. Any of these three parts may be omitted, but the semicolons must remain.

A `do-while` loop performs the test after executing its body, guaranteeing that its body is executed at least once. Its form is shown below.

```
do {
  body
} while (exp);
```

1.3.4 Break and Continue

A `break` statement exits from a loop immediately, as in:

```
while (1) {
  //...input and process data...
  if (error)
    break;
}
```

In this example, the `break` statement exits from the `while` loop upon encountering an error. In general, `break` causes the innermost enclosing loop or `switch` statement to exit.

A `continue` statement is similar to `break`, except that it causes the next iteration of the enclosing loop to begin, as in:

```
for (int i=0; i<n; i++) {
  if (a[i] == 0)
    continue;
  //...process non-zero elements...
}
```

In a `for` loop, the statement executed immediately after `continue` is the loop increment. In `while` and `do` loops, the next statement executed is the conditional test.

A `continue` statement is often used where the code that follows it is complicated, so that reversing the test and indenting another level would nest the program too deeply to be easily understood.

1.4 Functions

A function names a block of statements, providing a way to encapsulate and parameterise a computation. The program below defines and uses a function `fact`, which computes a factorial.

```
int fact(int);

/* test fact function */
int main(void) {
  for (int i=0; i<10; i++) {
    int f = fact(i);
    // ... print f ...
  }
}

int fact(int n) {
  for (int i=n-1; i>1; i--)
    n = n * i;
  return n;
}
```

The declaration of `fact` after `main`

```
int fact (int n)
```

declares `fact` to be a function that takes an `int` parameter named `n` and returns an `int` value. When `main` calls `fact`, the value of `i` is copied into a new variable `n`. The variable `n` is private to `fact`, and other functions can use this name without conflict.

The block of statements grouped in braces `{ }` following the declaration of `fact` makes it a *definition*. At most one definition of each function is permitted.

The `return` statement in the body of `fact` returns the computed factorial value to `main`. A function that does not return a value is specified with the return type `void`.

The first declaration of `fact` before `main`

```
int fact (int);
```

is a *prototype* that declares the type of `fact` without giving it a definition. Either a function prototype or its definition must appear in the source code before the function is used. The prototype must agree with its definition and all of its uses; the parameter names in the prototype are optional.

1.4.1 Function Arguments

Arguments to functions are usually passed by *value*, in which case the value is copied into a new variable that is private to the function.

NEW XC An argument may also be passed by *reference* so that any change made to the local XC variable also modifies the argument in the calling function. The program below swaps the values of two variables passed by reference.

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(void) {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

The declaration

```
void swap(int &x, int &y)
```

declares `swap` to be a function that accepts two variables by reference. A reference parameter is specified by prefixing its name with `&`.

NEW XC The creation of more than one reference to the same object is invalid. In the above example, calling the function `swap` using the same variable twice would be invalid.

Arrays are implicitly passed by reference. This means that an array cannot be passed to two parameters of a function but, for example, passing two different rows of a two-dimensional array to a function is permitted.

The largest dimension of an array parameter may be omitted from the function declaration, allowing the function to operate on arbitrary sized arrays, as in:

```
int strcount(char str[], int len);
```

If the size of an array parameter is specified, passing an array of larger size is permitted but the highest elements are not accessible; passing a smaller array is invalid.

1.4.2 Optional Arguments

NEW XC A pass-by-reference parameter can be specified *nullable*, meaning that it can contain either a valid reference or a special `null` reference. The program below determines how many corresponding elements of two arrays have the same value, assigning the value 0 or 1 to the element of a third array only if provided by the caller.

```
int compare(int x[], int y[], int ?matches[], unsigned size) {
  int n = 0;
  for (int i=0; i<size; i++) {
    int match = (x[i] == y[i]);
    n += match;
    if (!isnull(matches))
      matches[i] = match;
  }
  return n;
}

int main(void) {
  int x[5] = {0, 1, 2, 3, 4};
  int y[5] = {1, 1, 1, 3, 3};
  int z[5] = {1, 1, 1, 1, 1};
  int m[5] = {0};
  int n;

  (void)compare(x, y, m, 5);
  n = compare(y, z, null, 5);

  return 0;
 }
```

The declaration

```
int compare (int x[], int y[], int ?m[], unsigned size)
```

declares `compare` to be a function that accept three arrays and a `size` variable. The third parameter `m` is specified as nullable by prefixing its name with `?`.

The operator `isnull` produces a value 1 if its argument is a valid reference and 0 otherwise. Attempting to dereference or use a `null` object is invalid.

On the first call by `main` to `compare`, the array `m` is passed as the third argument; `compare` assigns the elements of this array. On the second call, `null` is passed as the third argument; `compare` does not attempt to assign to the array.

1.4.3 Multiple-Return Functions

NEW XC

A function may be declared as returning more than one value, as in:

```
{int, int} swap(int a, int b) {
  return {b, a};
}

void main(void) {
  int a  = 1;
  int b  = 2;
  {a, b} = swap(b, a);
}
```

The list of return types, the list of values following `return`, and the list of variables assigned are enclosed in braces. The number of elements in the assignment list must match the number of values returned by the function, but any of the returned values may be ignored using `void`, as in:

```
{a, void} = f();
```

1.5 Reinterpretation

NEW XC

A reinterpretation causes a variable to be treated as having a different type, but it undergoes no conversion. The function below uses a reinterpretation to transmit an array of bytes as 32-bit integers.

```
void transmitMsg(char msg[], int nwords) {
  for (int i=0; i<nwords; i++)
    transmitInt((msg, int[])[i]);
}
```

The construction

```
(msg , int [])
```

reinterprets the array `msg` as an array of integers, which is then indexed, as in:

```
(msg, int [])[i])
```

In this example, the size of the integer array is determined at run-time. If the function is called, for example, with an array of 10 bytes, the reinterpreted integer array has an upper bound of 2 and the topmost 2 characters are inaccessible in the reinterpretation. If size of the reinterpretation is given, it must not exceed the size of the original type. Attempting to reinterpret one object to another whose type requires greater storage alignment (as specified in XM-000968-PC) is invalid. The original declaration should specify the largest storage alignment required for all possible reinterpretations.

1.6 Comparison with C

XC provides many of the same capabilities as C, the main omission being support for pointers. Consequently, many programming errors that are undefined in C are known to be invalid in XC and can be caught either by the compiler or raised as run-time exceptions. All of XC's data types and operators have the same meaning as in C, and user-defined types including structures, unions, enumerations and typedefs are also supported. The extensions for pass-by-reference parameters and multiple-return functions provide support for operations usually performed using pointers in C. XC's scope and linkage rules are the same as with C, and both languages use the same preprocessor.

XC does not support floating point, `long long` arithmetic, structure bit-fields or `volatile` data types, and no `goto` statement is provided. These restrictions may be relaxed in future releases to improve compatibility between languages.

2 Input and Output

IN THIS CHAPTER

A port connects a processor to one or more physical pins and as such defines the interface between a processor and its environment. The port logic can drive its pins high or low, or it can sample the value on its pins, optionally waiting for a particular condition. Ports are not memory mapped; instead they are accessed using dedicated instructions. XC provides integrated input and output statements that make it easy to express operations on ports. Figure 4 illustrates these operations.

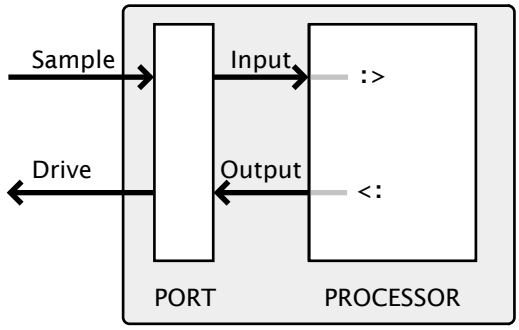


Figure 4: Input and Output Operations

Data rates can be controlled using hardware timers that delay the execution of the input and output instructions for a defined period. The processor can also be made to wait for an input from more than one port, enabling multiple I/O devices to be interfaced concurrently.

2.1 Outputting Data

A simple program that toggles a pin high and low is shown below.

```
#include <xs1.h>

out port p = XS1_PORT_1A;

int main(void) {
  p <: 1;
  p <: 0;
}
```

The declaration

```
out port p = XS1_PORT_1A;
```

declares an output port named p, which refers to the 1-bit port identifier 1A.[1]

The statement

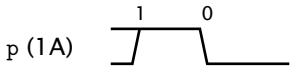
```
p <: 1;
```

outputs the value 1 to the port p, causing the port to drive its corresponding pin high. The port continues to drive its pin high until execution of the next statement

```
p <: 0;
```

which outputs the value 0 to the port, causing the port to drive its pin low. Figure 5 shows the output generated by this program.

Figure 5: Output waveform diagram



The pin is initially not driven; after the first output is executed it is driven high; and after the second output is executed it is driven low. In general, when outputting to an *n*-bit port, the least significant *n* bits of the output value are driven on the pins and the rest are ignored.

All ports must be declared as global variables, and no two ports may be initialised with the same port identifier. After initialisation, a port may not be assigned to. Passing a port to a function is allowed as long as the port does not appear in more than one of a function's arguments, which would create an illegal alias.

[1] The value XS1_PORT_1A is defined in the header file <xs1.h>. Most development boards are supplied with an XN file from which the header file <platform.h> is generated, and which defines more intuitive names for ports such as PORT_UART_TX and PORT_LED_A. These names are documented in the corresponding hardware manual.

2.2 Inputting Data

The program below continuously samples the 4 pins of an input port, driving an output port high whenever the sampled value exceeds 9.

```
#include <xs1.h>

in  port  inP = XS1_PORT_4A;
out port outP = XS1_PORT_1A;

int main(void) {
  int x;
  while (1) {
    inP :> x;
    if (x > 9)
      outP <: 1;
    else
      outP <: 0;
  }
}
```

The declaration

```
in port inP = XS1_PORT_4A;
```

declares an input port named `inP`, which refers to the 4-bit port identifier 4A.

The statement

```
inP :> x;
```

inputs the value sampled by the port `inP` into the variable `x`. Figure 6 shows example input stimuli and expected output for this program.

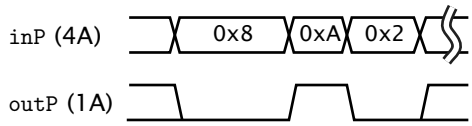


Figure 6: Input waveform diagram

The program continuously inputs from the port `inP`: when 0x8 is sampled the output is driven low, when 0xA is sampled the output is driven high and when 0x2 is sampled the output is again driven low. Each input value may be sampled many times.

2.3 Waiting for a Condition on an Input Pin

An input operation can be made to wait for one of two conditions on a pin: equal to or not equal to some value. The program below uses a *conditional input* to count the number of transitions on its input pin.

```
#include <xs1.h>

in  port oneBit  = XS1_PORT_1A;
out port counter = XS1_PORT_4A;

int main(void) {
  int x;
  int i = 0;

  oneBit :> x;
  while (1) {
    oneBit when pinsneq(x) :> x;
    counter <: ++i;
  }
}
```

The statement

```
oneBit when pinsneq(x) :> x;
```

instructs the port `oneBit` to wait until the value on its pins is not equal to `x` before sampling and providing it to the processor to store in `x`.

Figure 7 shows example input stimuli and expected output for this program.

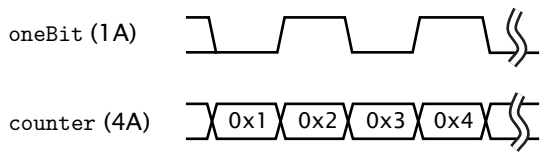


Figure 7: Conditional input waveform diagram

As another example, the only operation required to wait for an Ethernet preamble on a 4-bit port is

```
ethData when pinseq(0xD) :> void;
```

The processor must complete an input operation from the port once a condition is met, even if the input value is not required. This is expressed in XC as an input to `void`.

Using a conditional input is more power efficient than polling the port in software, because it allows the processor to idle, consuming less power, while the port remains active monitoring its pins.

2.4 Controlling I/O Data Rates with Timers

A timer is a special type of port used for measuring and controlling the time between events. A timer has a 32-bit counter that is continually incremented at a rate of 100MHz and whose value can be input at any time. An input on a timer

can also be delayed until a time in the future. The program below uses a timer to control the rate at which a 1-bit port is toggled.

```
#include <xs1.h>
#define DELAY 50000000

out port p = XS1_PORT_1A;

int main(void) {
  unsigned state = 1, time;
  timer t;
  t :> time;
  while (1) {
    p <: state;
    time += DELAY;
    t when timerafter(time) :> void;
    state = !state;
  }
}
```

The declaration

```
timer t;
```

declares a timer named `t`, obtaining a timer resource from the XCore's pool of available timers.

The statement

```
t :> time;
```

inputs the value of `t`'s counter into the variable `time`. This variable is then incremented by the value `DELAY`, which specifies a number of counter increments. The timer has a period of 10ns, giving a time in the future of 50,000,000 * 10ns = 0.5s.

The conditional input statement

```
t when timerafter (time) :> void;
```

waits until this time is reached, completing the input just afterwards.

Figure 8 shows the data driven for this program.

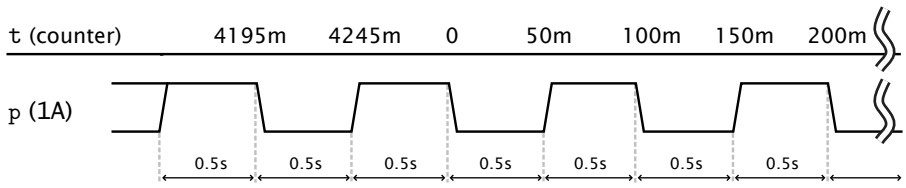


Figure 8: Timed output waveform diagram

The function `timerafter` treats the timer's counter as having two separate ranges, as illustrated in Figure 9.

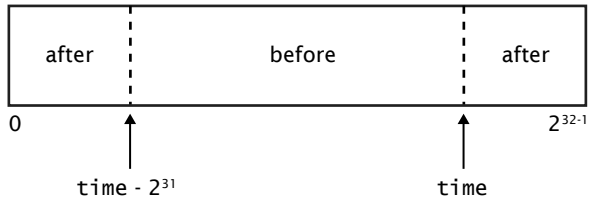


Figure 9: Range of `timerafter`

All values in the range (`time`-2^{31}..`time`-1) are considered to come before `time`, with values in the range (`time`+1..`time`+2^{32-1}, 0..`time`-2^{31}) considered to come afterwards. If the delay between the two input values fits in 31 bits, `timerafter` is guaranteed to behave correctly, otherwise it may behave incorrectly due to overflow or underflow. This means that a timer can be used to measure up to a total of 2^{31}/100,000,000 = 21s.

A programming error may be introduced by inputting the new time instead of ignoring it with a cast to `void`, as in

```
t when timerafter (time) :> time;
```

Because the processor completes the input shortly after the time specified is reached, this operation actually increments the value of `time` by a small additional amount. This amount may be compounded over multiple loop iterations, leading to signal drift and ultimately a loss of synchronisation with a receiver.

2.5 Case Study: UART (Part 1)

A universal asynchronous receiver/transmitter (UART) translates data between parallel and serial forms for communication over two 1-bit wires at fixed data rates. Each bit of data is driven for the time defined by the data rate, and the receiver must sample the data during this time. Figure 10 shows the transmission of a single byte of data at a rate of 115200 bits/s.

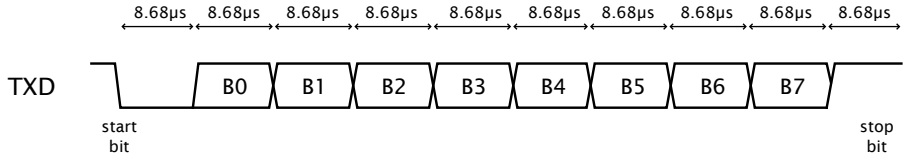


Figure 10: UART timing diagram

The quiescent state of the wire is high. A byte is sent by first driving a *start bit* (0), followed by the eight data bits and finally a *stop bit* (1). A rate of 115200 bits/s means that each bit is driven for 1/115200 = 8.68µs.

UARTs are often implemented with microcontrollers, using interrupts to schedule memory-mapped input and output operations. Implementing a UART with an XMOS device is easy due to its dedicated I/O instructions. The program below defines a UART transmitter.

```
#include <xs1.h>

#define BIT_RATE 115200
#define BIT_TIME 100000000 / BIT_RATE

out port TXD = XS1_PORT_1A;
in  port RXD = XS1_PORT_1B;

void transmitter(out port TXD) {
  unsigned byte, time;
  timer t;

  while (1) {
    /* get next byte to transmit */
    byte = getByte();
    t :> time;

    /* output start bit */
    TXD <: 0;
    time += BIT_TIME;
    t when timerafter(time) :> void;

    /* output data bits */
    for (int i=0; i<8; i++) {
      TXD <: >> byte;
      time += BIT_TIME;
      t when timerafter(time) :> void;
    }

    /* output stop bit */
    TXD <: 1;
    time += BIT_TIME;
    t when timerafter(time) :> void;
  }
}
```

The transmitter outputs a byte by first outputting a start bit, followed by a conditional input on a timer that waits for the bit time to elapse; the data bits and stop bit are output in the same way.

The output statement in the `for` loop

```
TXD <: >> byte;
```

includes the modifier >>, which right-shifts the value of `byte` by the port width (1 bit) after outputting the least significant port-width bits. This operation is performed in the same instruction as the output, making it more efficient than performing the shift as a separate operation afterwards.

The function below receives a stream of bytes over a 1-bit wire.

```
void receiver(in port RXD) {
  unsigned byte, time;
  timer t;

  while (1) {
    /* wait for start bit */
    RXD when pinseq(0) :> void;
    t :> time;
    time += BIT_TIME/2;

    /* input data bits */
    for (int i=0; i<8; i++) {
      time += BIT_TIME;
      t when timerafter(time) :> void;
      RXD :> >> byte;
    }

    /* input stop bit */
    time += BIT_TIME;
    t when timerafter(time) :> void;
    RXD :> void;

    putByte(byte >> 24);
  }
}
```

The receiver samples the incoming signal, waiting for a start bit. After receiving this bit, it waits for 1.5 times the bit time and then samples the wire at the midpoint of the the first byte transmission, with subsequent bits being sampled at 8.68µs increments. The input statement in the `for` loop

```
RXD :> >> byte;
```

includes the modifier >>, which first right-shifts the value of byte by the port width (1 bit) and then inputs the next sample into its most significant port-width bits. The expression in the final statement

```
putByte(byte >> 24);
```

right-shifts the bits in the integer byte by 24 bits so that the input value ends up in its least significant bits.

2.6 Responding to Multiple Inputs

The program below inputs two streams of data from two separate ports using only a single thread. The availability of data on one of these ports is signalled by the toggling of a pin, with data on another other port being received at a fixed rate.

```
#include <xs1.h>

#define DELAY_Q 10000000

in port toggleP = XS1_PORT_1A;
in port dataP   = XS1_PORT_4A;
in port dataQ   = XS1_PORT_4B;

int main(void) {
  timer t;
  unsigned time, x = 0;

  t :> time;
  time += DELAY_Q;
  while (1)
    select {
      case toggleP when pinsneq(x) :> x :
        readData(dataP);
        break;
      case t when timerafter(time) :> void :
        readData(dataQ);
        time += DELAY_Q;
        break;
    }
}
```

The `select` statement performs an input on either the port `toggleP` or the timer `t`, depending on which of these resources becomes ready to input first. If both inputs become ready at the same time, only one is selected, the other remaining ready on the next iteration of the loop. After performing an input, the body of code below it is executed. Each body must be terminated by either a `break` or `return` statement.

Case statements are not permitted to contain output operations as the XMOS architecture requires an output operation to complete but allows an input operation to wait until it sees a matching output before committing to its completion.

Each port and timer may appear in only one of the `case` statements. This is because the XMOS architecture restricts each port and timer resource to waiting for just one condition at a time.

In this example, the processor effectively multi-tasks the running of two independent tasks, and it must be fast enough to process both streams of data in real-time. If this is not possible, two separate threads may be used to process the data instead (see §3).

2.7 Case Study: UART (Part 2)

The program on the following page uses a `select` statement to implement both the transmit and receive sides of a UART in a single thread.

```
void UART(port RX, int rxPeriod, port TX, int txPeriod) {
  int txByte, rxByte;
  int txI, rxI;
  int rxTime, txTime;
  int isTX = 0;
  int isRX = 0;
  timer tmrTX, tmrRX;
  while (1) {
    if (!isTX && isData()) {
      isTX = 1;
      txI = 0;
      txByte = getByte();
      TX  <: 0;             // transmit start bit
      tmrTX :> txTime;      // set timeout for data bit
      txTime += txPeriod;
    }
    select {
      case !isRX => RX when pinseq(0) :> void :
        isRX = 1;
        tmrRX :> rxTime;
        rxI = 0;
        rxTime += rxPeriod;
        break;
      case isRX => tmrRX when timerafter(rxTime) :> void :
        if (rxI < 8)
          RX :> >> rxByte;
        else { // receive stop bit
          RX :> void;
          putByte(rxByte >> 24);
          isRX = 0;
        }
        rxI++;
        rxTime += rxPeriod;
        break;
      case isTX => tmrTX when timerafter(txTime) :> void :
        if (txI < 8)
          TX <: >> txByte;
        else if (txI == 8)
          TX <: 1; // stop bit
        else
          isTX = 0;
        txI++;
        txTime += txPeriod;
        break;
} }  }
```

The variables `isTX`, `txI`, `isRX` and `rxI` determine which parts of the UART are active and how many bits of data have been transmitted and received.

The `while` loop first checks whether the transmitter is inactive with data available to transmit, in which case it outputs a start bit and sets the timeout for outputting the first data bit.

In the `select` statement, the guard

```
case !isRX => RX when pinseq(0) :> void :
```

checks whether `isRX` equals zero, indicating that the receiver is inactive, and if so it enables an input on the port `RX` when the value on its pins equals 0. The expression on the left of the operator `=>` is said to *enable* the input. The body of this case sets a timeout for inputting the first data bit.

The second guard

```
case isRX => tmrRX when timerafter(rxTime) :> void :
```

checks whether `isRX` is non-zero, indicating that the receiver is active, and if so enables an input on the timer `tmrRX`. The body of this case inputs the next bit of data and, once all bits are input, it stores the data and sets `isRX` back to zero.

The third guard

```
case isTX => tmrTX when timerafter(txTime) :> void :
```

checks whether `isTX` is non-zero, indicating that the transmitter is active, and if so enables an input on the timer `tmrTX`. The body of this case outputs the next bit of data and, once all bits are output, it sets `isTX` to zero.

If this UART controller is to be used in noisy environments, its reliability may be improved by sampling each input bit multiple times and averaging the result. A more robust implementation would also check that the stop bit received has an expected value of 1.

2.8 Parameterised Selection

Select statements can be implemented as functions, allowing their reuse in different contexts. One such example use is to parameterise the code used to sample data on a two-bit encoded line. As shown in Figure 11, a quiet state on the line is represented by the value (0, 0), the value 0 is signified by a transition to (0, 1) and the value 1 is signified by a transition to (1, 0). Either of these transitions is followed by another transition back to (0, 0).

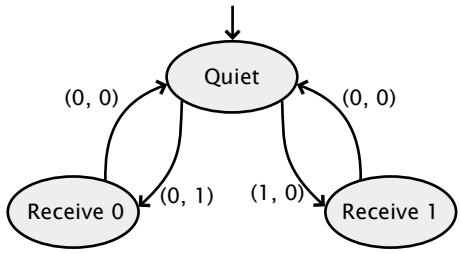


Figure 11: State transition diagram

The program below makes use of a *select function* to input a single byte of data from two pins using this scheme.

```
#include <xs1.h>

in port r0 = XS1_PORT_1A;
in port r1 = XS1_PORT_1B;

select inBit(in port r0, in port r1, int &x0, int &x1, char &byte) {
  case r0 when pinsneq(x0) :> x0 :
    if (x0 == 1)  /* transition to (1, 0) */
      byte = (byte << 1) | 1;
    break;
  case r1 when pinsneq(x1) :> x1 :
    if (x1 == 1)  /* transition to (0, 1) */
      byte = (byte << 1) | 0;
    break;
}

int main(void) {
  int x0 = 0, x1 = 0;
  char byte;
  for (int i=0; i<16; i++)
    inBit(r0, r1, x0, x1, byte);
}
```

The declaration

```
select inBit(in port r0, in port r1, int &x0, int &x1, char &byte)
```

declares `inBit` to be a select function that takes five arguments and has an implicit return type of `void`; its body contains two case statements.

Figure 12 shows example input stimuli for this program. The bit values received are 0, 1, 0, 0, 0, 0, 0 and 1 ('A').

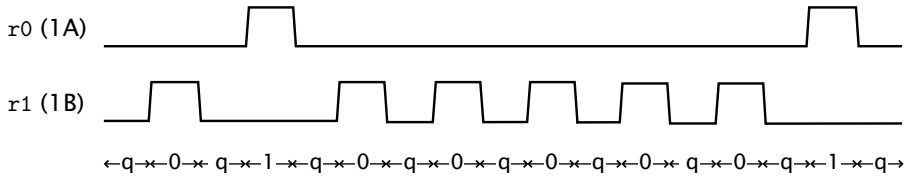


Figure 12: Select function waveform diagram

In contrast to a UART, which transmits data at a fixed rate, this scheme allows for the fastest possible transmission supported by an XMOS device and the component to which it is connected.

A benefit of defining `inBit` as a select function is that its individual cases can be used to form part of a larger select statement, as in the program below which decodes a two-byte value sampled on four pins.

```
#include <xs1.h>
#define NBYTES 2

in port r[NBYTES*2] = { XS1_PORT_1A, XS1_PORT_1B,
                        XS1_PORT_1C, XS1_PORT_1D };
int main(void) {
  int state[NBYTES*2] = {0, 0, 0, 0};
  char byte[NBYTES];
  for (int i=0; i<8*NBYTES; i++)
    select {
      case inBit(r[0], r[1], state[0], state[1], byte[0]);
      case inBit(r[2], r[3], state[2], state[3], byte[1]);
    }
}
```

The `select` statement calls the function inBit in two of its case statements, causing the processor to enable events on the ports passed to it as arguments.

A more concise way to specify the top-level select statement is to use a *replicator*, as in:

```
select {
  case (int i=0; i<NBYTES; i++)
    inBit(r[i*2], r[i*2+1], state[i*2], state[i*2+1], byte[i]);
}
```

The replicator

```
(int i=0; i <NBYTES; i++)
```

iterates twice, each time calling the select function `inBit`, which enables the ports indexed by different values of `i`. The number of iterations need not be constant, but the iterator must not be modified outside of the replicator.

3 Concurrency

IN THIS CHAPTER

Many designs require a collection of tasks to be performed at the same time. Some of these tasks may perform independent activities, while others engage with one another to complete shared objectives. XC provides simple mechanisms for creating *concurrent threads* that can run independently and interact with one another on demand. Data is communicated between threads using *channels*, which provide point-to-point connections between pairs of threads. Channels can be used to communicate data either synchronously or asynchronously.

3.1 Creating Concurrent Threads

The program below creates four concurrent threads, all of which run separate tasks independently of one another. Two of these threads are executed on XCore 0, one on XCore 1 and one on XCore 2.

```
#include <platform.h>

on stdcore[0] : out port tx      = XS1_PORT_1A;
on stdcore[0] : in  port rx      = XS1_PORT_1B;
on stdcore[1] : out port lcdData = XS1_PORT_32A;
on stdcore[2] : in  port keys    = XS1_PORT_8B;

int main(void) {
  par {
    on stdcore[0] : uartTX(tx);
    on stdcore[0] : uartRX(rx);
    on stdcore[1] : lcdDrive(lcdData);
    on stdcore[2] : kbListen(keys);
  }
}
```

The header file `platform.h` provides a declaration of the global variable `stdcore`, which is used to specify the locations of ports and threads.[2]

The declaration

```
on stdcore [0] : out port p = XS1_PORT_1A;
```

declares a 1-bit output port named `p` that refers to the port identifier 1A on standard core number 0.

The four statements inside the braces of the `par` are run concurrently as four separate threads using *fork-join parallelism*: at the opening brace `{` the parent creates three more threads; each of these threads then executes a function; and at the closing brace `}` the parent waits for all functions to return before continuing.

`par` statements may be used anywhere in a program. Each XS1 device has a limit of eight threads available on each of its processors, and a program that attempts to exceed this limit is invalid.

The `on` statement is used to specify the physical location of components connected to ports and to partition a collection of threads between the available XCores.

For single-core programs, none of the port declarations need be prefixed with `on`, in which case all ports and threads are placed on XCore 0. For multicore programs, all ports and threads must be explicitly prefixed with `on`.

A multicore `main` function may contain only channel declarations, a single `par` statement and an optional `return` statement. The `on` statement may be used to specify the location of threads only within this function.

3.2 Thread Disjointness Rules

All variables are subject to usage rules that prevent them from being shared by threads in potentially dangerous ways. In general, each thread has full access to its own private variables, but limited access to variables that are shared with other threads. The rules for disjointness on a set of threads $T_0...T_i$ and a set of variables $V_0...V_j$ are as follows:

- If thread T_x contains any modification to variable V_y then none of the other threads (T_t, $t \neq x$) are allowed to use V_y.
- If thread T_x contains a reference to variable V_y then none of the other threads (T_t, $t \neq x$) are allowed to modify V_y.
- If thread T_x contains a reference to port V_p then none of the other threads are allowed to use V_p.

[2]The target platform is described using the XMOS network specification language XN. Most board support packages provide a corresponding XN file, which describes the available devices and their connectivity. This data is used during the mapping stage of compilation to product a multi-node executable file that can boot and configure the entire system.

In other words, a group of threads can have *shared* read-only access to a variable, but only a single thread can have *exclusive* read-write access to a variable. These rules guarantee that each thread has a well-defined meaning that is independent of the order in which instructions in other threads are scheduled. Interaction between threads takes place explicitly using inputs and outputs on channels (see §3.3).

3.2.1 Examples

The example program below is legal, since `k` is shared read-only in threads X and Y, `i` is modified in X and not used in Y, and `j` is modified in Y and not used in X.

```
int main(void) {
  int i = 1, j = 2, k = 3;
  par {
    i = k + 1;  // Thread X
    j = k - 1;  // Thread Y
  }
}
```

If either `i` or `j` is also read in another thread, the example becomes illegal, as shown in the program below.

```
int main(void) {
  int i = 1, j = 2, k;
  par {
    i = j + 1;  // Thread X: illegal sharing of i
    k = i - 1;  // Thread Y: illegal sharing of i
  }
}
```

This program is ambiguous since the value of `i` read in thread Y depends upon whether the assignment to `i` in thread X has already happened or not.

The program below is legal, since `a[0]` is modified in thread X and not used in thread Y, and `a[1]` is modified in Y and not used in X.

```
int main(void) {
  int a[2];
  par {
    a[0] = f(0); // Thread X
    a[1] = f(1); // Thread Y
  }
}
```

The program below is illegal since `a[1]` is modified in thread X and an unknown element of `a` is modified in thread Y.

```
int x;

int main(void) {
  int a[10];
  par {
    a[1] = f(1); // Thread X: illegal sharing of x[1]
    a[x] = f(x); // Thread Y: illegal sharing of x[1]
  }
}
```

In general, indexing an array by anything other than a constant value is treated as if all elements in the array are accessed.

The program below is illegal since the array `a` is passed by reference to the function `f` in both threads X and Y, which may possibly modify its value.

```
void f(int[]);

int main(void) {
  int a[10];
  par {
    f(a); // Thread X: illegal sharing of a
    f(a); // Thread Y: illegal sharing of a
  }
}
```

If `f` does not modify the array then its parameter should be declared with `const`, which would make the above program legal.

The disjointness rules apply individually to parallel statements in sequence, and recursively to nested parallel statements. The example program below is legal.

```
int main(void) {
  int i = 1, j = 2, k = 3;
  par {
    i = k + 1;  // Thread X
    j = k - 1;  // Thread Y
  }
  i = i + 1;
  par {
    j = i - 1;  // Thread U
    k = i + 1;  // Thread V
  }
}
```

In this example, `i` is first declared and initialised in the main thread; it is then used exclusively in thread X (thread Y is not allowed access). Once X and Y have joined, `i` it is used again by the main thread; finally it is shared between threads U and V.

3.3 Channel Communication

A *channel* provides a synchronous, point-to-point connection between two threads over which data may be communicated. The program below uses a channel to communicate data from a producer thread on one processor to a consumer thread on another.

```
#include <platform.h>

on stdcore[0] : out port tx   = XS1_PORT_1A;
on stdcore[1] : in  port keys = XS1_PORT_8B;

void uartTX(chanend dataIn, port tx) {
  char data;
  while (1) {
    dataIn :> data;
    transmitByte(tx, data);
  }
}

void kbListen(chanend c, port keys) {
  char data;
  while (1) {
    data = waitForKeyStroke(keys);
    c <: data;
  }
}

int main(void) {
  chan c;
  par {
    on stdcore[0] : uartTX(c, tx);     // Thread X
    on stdcore[1] : kbListen(c, keys); // Thread Y
  }
}
```

The declaration

```
void uartTX(chanend dataIn, port tx)
```

declares `uartTX` to be a function that takes a channel end and a port as its arguments.

The declaration

```
void kbListen(chanend c, port keys);
```

declares `kbListen` to be a function that takes a channel end and a port as its arguments.

In the function `main`, the declaration

```
chan c;
```

declares a channel. The channel is used in two threads of a `par` and each use implicitly refers to one of its two channel ends. This usage establishes a link between thread X on XCore 0 and thread Y on XCore 1.

Thread X calls the function `uartTX`, which receives data over a channel and outputs it to a port. Thread Y calls `kbListen`, which waits for keyboard strokes from a port and outputs the data on a channel to the UART transmitter on thread X. As the channel is synchronous, when `kbListen` outputs data, it waits until `uartTX` is ready to receive the data before continuing.

Channels are lossless, which means that data output in one thread is guaranteed to be delivered for input by another thread. Each output in one thread must therefore be matched by an input in another, and the amount of data output must equal the amount input or else the program is invalid.

3.3.1 Channel Disjointness Rules

The rules for disjointness on a set of threads $T_0 ... T_i$ and a set of channels $C_0 ... C_j$ are as follows:

- If threads T_x and T_y (where $x \neq y$) contain a use of channel C_y then none of the other threads (T_t, $t \neq x,y$) are allowed use C_y.
- If thread T_x contains a use of channel end C_y then none of the other threads (T_t, $t \neq x$) are allowed to use C_y.

In other words, each channel can be used in at most two threads. If a channel is used in only one thread then attempting to input or output on the channel will block forever.

The disjointness rules for variables and channels together guarantee that any two threads can be run concurrently on any two processors, subject to a physical route existing between the processors. As a general rule, threads that interact with one another frequently should usually be located close together.

3.4 Transactions

Input and output statements on channels usually synchronise the communication of data. This is not always desirable, however, as it disrupts the flow of the program, causing threads to block. The time taken to synchronise, including the time spent idle while blocking, can reduce overall performance.

In XC it is possible for two threads to engage in a *transaction*, in which a sequence of matching outputs and inputs are communicated over a channel asynchronously, with the entire transaction being synchronised at its beginning and end. As with individual channel communications, the total amount of data output must equal the total amount input.

The program below uses a transaction to communicate a packet of data between two threads efficiently.

```
#include <platform.h>

int snd[10], rcv[10];

int main(void) {
  chan c;
  par {
    on stdcore[0] : master {   // Thread X
      for (int i=0; i<10; i++)
        c <: snd[i];
    }
    on stdcore[1] : slave {    // Thread Y
      for (int i=0; i<10; i++)
        c :> rcv[i];
    }
  }
}
```

A transaction consists of a *master* thread and a *slave* thread running concurrently. The threads first synchronise upon entry to the master and slave blocks. Ten integer values are then communicated asynchronously: thread X blocks only if data can no longer be dispatched (due to the channel buffering being full), and thread Y blocks only if there is no data available. Finally, the threads synchronise upon exiting the master and slave blocks.

Each transaction is permitted to communicate on precisely one channel. This ensures that deadlocks do not arise due to an output on one channel blocking as a result of a switch being full with incoming data that is not yet ready to be received.

The program below defines the body of a transaction as a function, which is called as the master component of a communication.

```
transaction inArray(chanend c, int data[], int size) {
  for (int i=0; i<size; i++)
    c :> data[i];
}

int main(void) {
  chan c;
  int snd[10], rcv[10];
  par {
    master inArray(c, rcv, 10);
    slave {
      for (int i=0; i<10; i++)
        c <: snd[i];
    }
  }
}
```

The declaration

```
transaction inArray(chanend c, char data[], int size)
```

declares `inArray` to be a transaction function that takes one end of a channel, an array of integers and the size of the array. A transaction function must declare precisely one channel end parameter.

In `main`, the call to `inArray` is prefixed with `master`, which specifies that the function is called as a master that communicates with a slave.

A slave statement may be used in the guard of a `select` statement, as in:

```
select {
  case slave { inArray(c1, packet, P_SIZE); } :
    process(packet);
    break;
  case slave { inArray(c2, packet, P_SIZE); } :
    process(packet);
    break;
}
```

A master operation by definition commits to completing and is therefore disallowed from appearing in a guard.

3.5 Streams

A *streaming channel* establishes a permanent route between two threads over which data can be efficiently communicated without synchronisation. The program below consists of three threads that together input a stream of data from a port, filter the data and output it to another port.

```
#include <platform.h>

on stdcore[0] : port lineIn = XS1_PORT_8A;
on stdcore[1] : port spkOut = XS1_PORT_8A;

int main(void) {

  streaming chan s1, s2;

  par {
    on stdcore[0] : audioRcv(lineIn, s1);
    on stdcore[0] : BiQuadFilter(s1,s2);
    on stdcore[1] : audioSnd(spkOut, s2);
  }
}
```

The declaration

```
streaming chan s1 , s2;
```

declares `s1` and `s2` as channels that transport data without performing any synchronisation. A route is established for the stream at its declaration and is closed down when the declaration goes out of scope.

Streaming channels provide the fastest possible data rates. An output statement takes just a single instruction to complete and is dispatched immediately as long as there is space in the channel buffer. An input statement takes a single instruction to complete and blocks only if the channel buffer is empty. In contrast to transactions, multiple streams can be processed concurrently, but there is a limit to how many streaming channels can be declared together as streams established between XCores require capacity to be reserved in switches. This limit does not apply to channels and transactions.

3.6 Parallel Replication

A *replicator* provides a concise and simple way to implement concurrent programs in which a collection of nodes perform the same operation on different datasets. The program on the following page constructs a communications network between four nodes running on four different threads.

```
#include <platform.h>

port p[4] = {
  on stdcore[0] : XS1_PORT_1A,
  on stdcore[1] : XS1_PORT_1A,
  on stdcore[2] : XS1_PORT_1A,
  on stdcore[3] : XS1_PORT_1A
};

void node(chanend, chanend, port, int n);

int main(void) {
  chan c[4];
  par (int i=0; i<4; i++)
    on stdcore[i] : node(c[i], c[(i+1)%4], p[i], i);
  return 0;
}
```

The replicator

```
(int i=0; i<4; i++)
```

executes four bodies of code, each containing an instance of the function `node` on a different thread. The number of iterations must be constant, and the iterator must not be modified outside of the replicator. The communication network established by this program is illustrated in Figure 13.

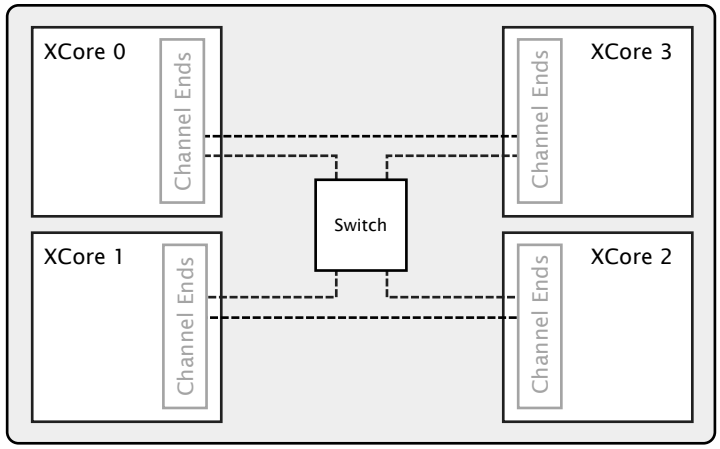


Figure 13: XCore Network

The structure of this program is similar to a token ring network, in which each thread inputs a token from one of its neighbours, performs an action and then outputs the token to its other neighbour.

3.7 Services

An XMOS network can interface with any device that implements the XMOS Link protocol. The program below communicates with an FPGA service connected to the network.

```
#include <platform.h>

port p = XS1_PORT_1A;

void inData(chanend c, port p) {
  // ... input data from p and output to c
}

service fpgaIF(chanend);

int main(void) {
  chan c;
  par {
    inData(c, p);
    fpgaIF(c);
} }
```

The declaration

```
service fpgaIF (chanend);
```

declares `fpgaIF` to be a service available on the XMOS network. A function declared as a service may contain only channel-end parameters and must not be given a definition. The characteristics of the links used to implement the channel ends are defined in the XN file.

Another use of services is for interfacing with functions pre-programmed into the non-volatile memory of an XCore by a third-party manufacturer. Typically, the manufacturer provides an XN file that contains all service declarations, which are available in the file `<platform.h>`.

3.8 Thread Performance

The XMOS architecture is designed to perform multiple real-time tasks concurrently, each of which is guaranteed predictable thread performance. Each processor uses a round-robin thread scheduler, which guarantees that if up to four threads are active, each thread is allocated a quarter of the processing cycles. If more than four threads are active, each thread is allocated at least $1/n$ cycles (for n threads). The minimum performance of a thread can therefore be calculated by counting the number of concurrent threads at a specific point in the program.

The graph below shows the guaranteed performance obtainable from each thread on a 400MHz XCore, depending on the total number of threads in use.

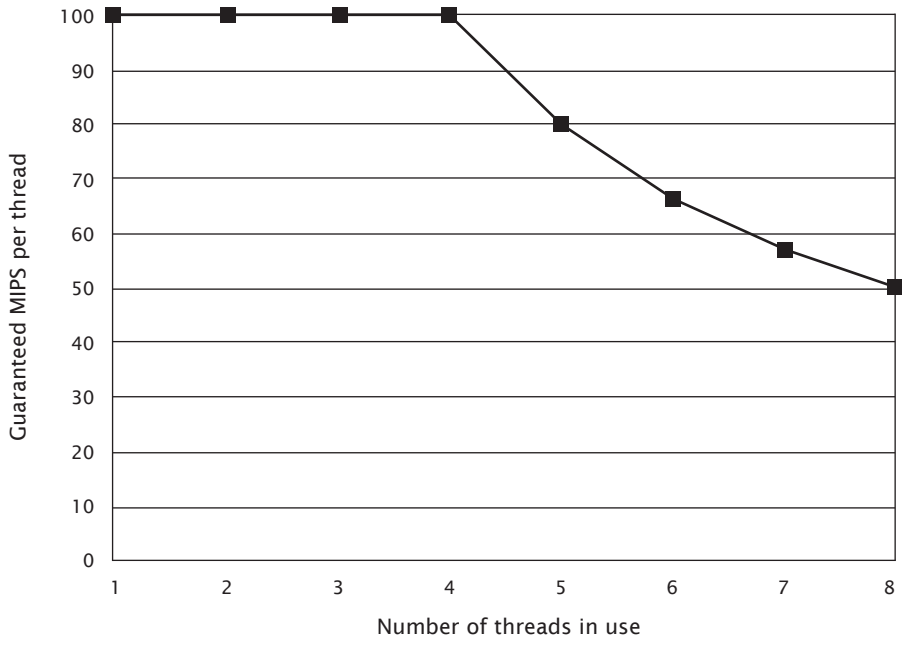


Figure 14: Thread performance graph

Because individual threads may be delayed on I/O, their unused processor cycles can be taken by other threads. Thus, for more than four threads, the performance of each thread is often higher than the minimum shown above.

4 Clocked Input and Output

IN THIS CHAPTER

Many protocols require data to be sampled and driven on specific edges of a clock. Ports can be configured to use either an internally generated clock or an externally sourced clock, and the processor can record and control on which edges each input and output operation occurs. In XC, these operations can be directly expressed in the input and output statements using the *timestamped* and *timed* operators.

4.1 Generating a Clock Signal

The program below configures a port to be clocked at a rate of 12.5MHz, outputting the corresponding clock signal with its output data.

```
#include <xs1.h>

out port outP     = XS1_PORT_8A;
out port outClock = XS1_PORT_1A;
clock    clk      = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk, 100, 8);
  configure_out_port(outP, clk, 0);
  configure_port_clock_output(outClock, clk);
  start_clock(clk);

  for (int i=0; i<5; i++)
    outP <: i;
}
```

The program configures the ports `outP` and `outClock` as illustrated in Figure 15.

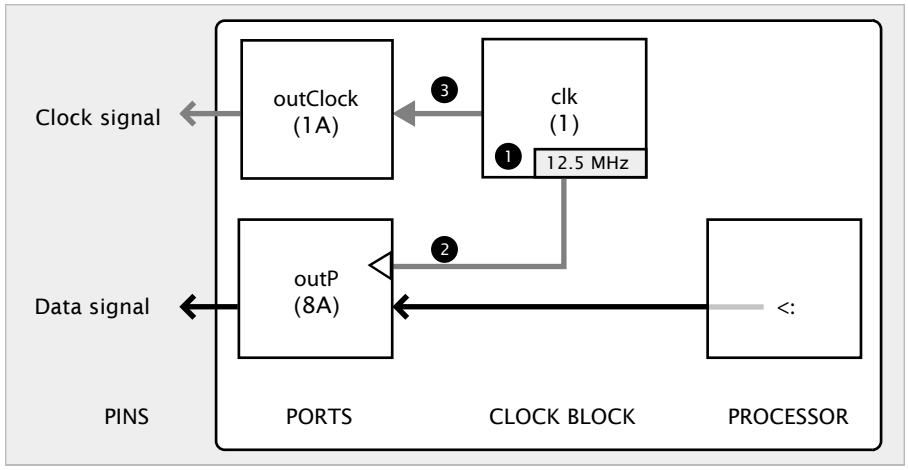


Figure 15: Port configuration diagram

The declaration

```
clock clk = XS1_CLKBLK_1;
```

declares a clock named `clk`, which refers to the clock block identifier `XS1_CLKBLK_1`. Clocks are declared as global variables, with each declaration initialised with a unique resource identifier.

1. The statement

```
configure_clock_rate(clk, 100, 8);
```

configures the clock `clk` to have a rate of 12.5MHz. The rate is specified as a fraction (100/8) because XC only supports integer arithmetic types.

2. The statement

```
configure_out_port(outP, clk, 0);
```

configures the output port `outP` to be clocked by the clock `clk`, with an initial value of 0 driven on its pins.

3. The statement

```
configure_port_clock_output(outClock, clk)
```

causes the clock signal `clk` to be driven on the pin connected to the port `outClock`, which a receiver can use to sample the data driven by the port `outP`.

The statement

```
start_clock(clk);
```

causes the clock block to start producing edges.

A port has an internal 16-bit counter, which is incremented on each falling edge of its clock. Figure 16 shows the port counter, clock signal and data driven by the port.

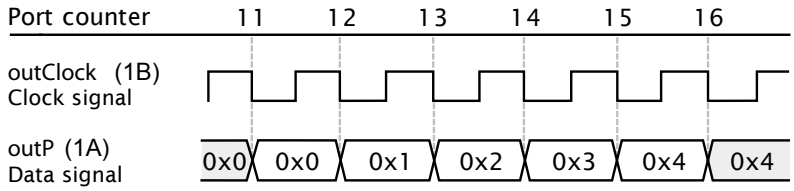


Figure 16: Waveform diagram

An output by the processor causes the port to drive output data on the next falling edge of its clock; the data is held by the port until another output is performed.

4.2 Using an External Clock

The following program configures a port to synchronise the sampling of data to an external clock.

```
#include <xs1.h>

in port inP     = XS1_PORT_8A;
in port inClock = XS1_PORT_1A;
clock   clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port(inP, clk);

  start_clock(clk);
  for (int i=0; i<5; i++)
    inP :> int x;
}
```

The program configures the ports `inP` and `inClock` as illustrated in Figure 17.

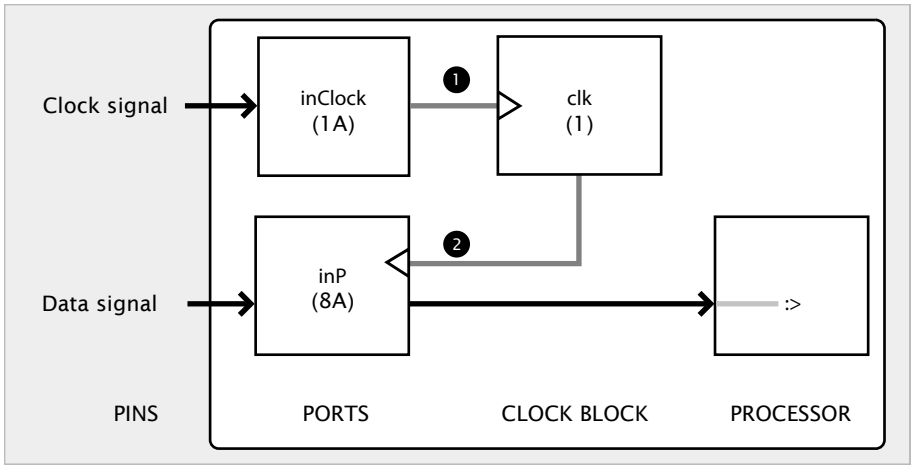


Figure 17: Port configuration diagram

1. The statement

```
configure_clock_src(clk, inClock);
```

configures the 1-bit input port `inClock` to provide edges for the clock `clk`. An edge occurs every time the value sampled by the port changes.

2. The statement

```
configure_in_port(inP, clk);
```

configures the input port `inP` to be clocked by the clock `clk`.

Figure 18 shows the port counter, clock signal, and example input stimuli.

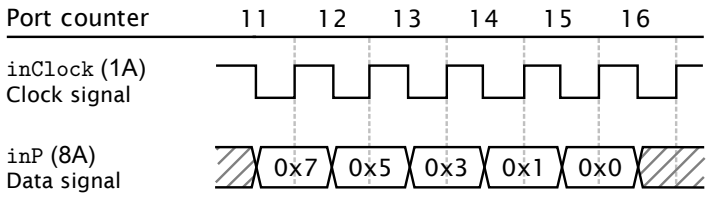


Figure 18: Waveform diagram

An input by the processor causes the port to sample data on the next rising edge of its clock. The values input are 0x7, 0x5, 0x3, 0x1 and 0x0.

4.3 Performing I/O on Specific Clock Edges

It is often necessary to perform an I/O operation on a port at a specific time with respect to its clock. The program below drives a pin high on the third clock period and low on the fifth.

```
void doToggle(out port toggle) {
  int count;
  toggle <: 0 @ count;    // timestamped output
  while (1) {
    count += 3;
    toggle @ count <: 1;  // timed output
    count += 2;
    toggle @ count <: 0;  // timed output
  }
}
```

The statement

```
toggle <: 0 @ count;
```

performs a *timestamped output*, outputting the value 0 to the port `toggle` and reading into the variable `count` the value of the port counter when the output data is driven on the pins. The program then increments count by a value of 3 and performs a *timed output* statement

```
toggle @ count <: 1;
```

This statement causes the port to wait until its counter equals the value `count+3` (advancing three clock periods) and to then drive its pin high. The last two statements delay the next output by two clock periods. Figure 19 shows the port counter, clock signal and data driven by the port.

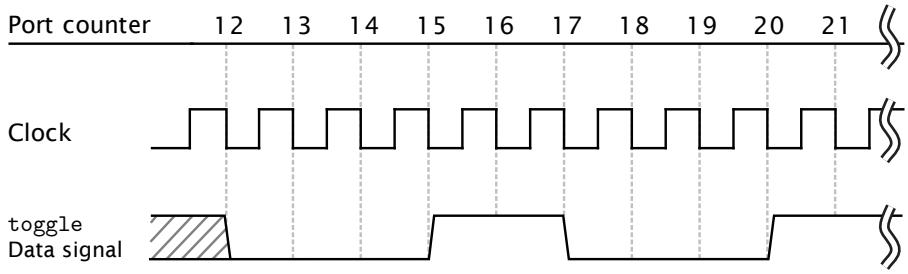


Figure 19: Waveform diagram

The port counter is incremented on the falling edge of the clock. On intermediate edges for which no value is provided, the port continues to drive its pins with the data previously output.

4.4 Case Study: LCD Screen Driver

LCD screens are found in many embedded systems. The principal method of driving most screens is the same, although the specific details vary from screen to screen. Figure 20 illustrates the operation of a Hitachi TX14 series screen[3], including the waveform requirements for transmitting a single frame of video.

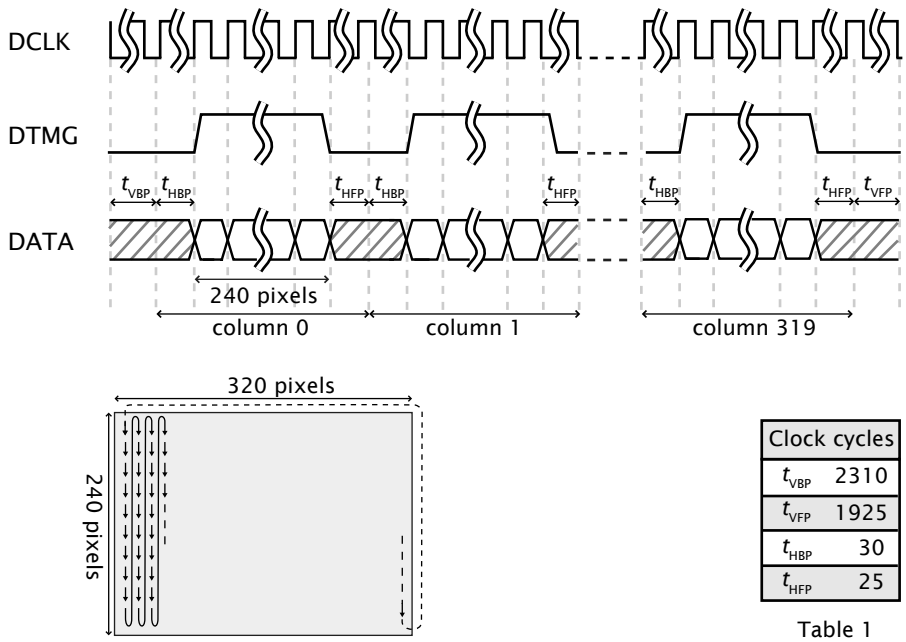


Figure 20: LCD Screen Driver Example

Clock cycles	
t_{VBP}	2310
t_{VFP}	1925
t_{HBP}	30
t_{HFP}	25

Table 1

The screen has a resolution of 320x240 pixels. It requires pixel data to be provided in column order with each value driven on a specific edge of a clock. The signals are as follows:

- DCLK is a clock signal generated by the driver, which must be configured within the range of 4.85MHz to 7.00MHz. The value chosen determines the screen refresh rate.
- DTMG is a data valid signal which must be driven high whenever data is transmitted.
- DATA carries 18-bit RGB pixel data to the screen.

The specification requires that pixel values for each column are driven on consecutive cycles with a 55 cycle delay between each column and a 4235 cycle delay between each frame (see Table 1).

[3] http://www.xmos.com/references/hitatchi-tx14-ds

LCD screens are usually driven by dedicated hardware components due to their clocking requirements. Implementing an LCD screen driver in XC is easy due to the clock synchronisation supported by the XMOS architecture. The required port configuration is illustrated in Figure 21.

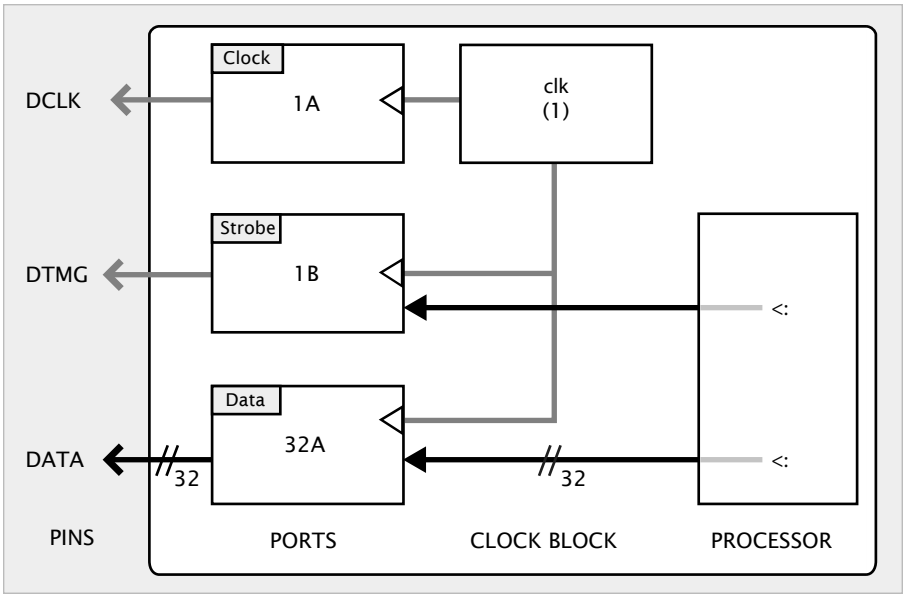


Figure 21: Port configuration diagram

The ports `DATA` and `DTMG` are both clocked by an internally generated clock, which is made visible on the port `DCLK`. The program below defines a function that configures the ports in this way.

```
#include <xs1.h>

out port DCLK = XS1_PORT_1A;
out port DTMG = XS1_PORT_1B;
out port DATA = XS1_PORT_32A;
clock    clk  = XS1_CLKBLK_1;

void lcdInit(void) {
  configure_clock_rate(clk, 100, 17); // 100/17 = 5.9MHz
  configure_out_port(DATA,  clk, 0);
  configure_out_port(DTMG, clk, 0);
  configure_port_clock_output(DCLK, clk);
  start_clock(clk);
}
```

The clock rate specified is 5.9MHz. The time required to transmit a frame is 320 * 240 + 240 * 55 + 4235 = 94235 clock ticks, giving a frame rate of 5.9/94235 =

62Hz. The function below outputs a sequence of pixel values to the LCD screen on the clock edges required by the specification.

```
void lcdDrive(streaming chanend c,
              out port DATA,
              out port DTMG) {

  unsigned x, time;

  DTMG <:0 @ time;
  while (1) {
    time += 4235;
    for (int cols=0; cols<320; cols++) {
      time +=30;
      c :> x;
      DTMG @ time <: 1;      // strobe high
      DATA @ time <: x;      // pixel 0
      for (int rows=1; rows<240; rows++) {
        c :> x;
        DATA <: x;           // pixels 1..239
      }
      DTMG @ time+240 <: 0; // strobe low
      time += 265;
    }
  }
}
```

A stream of data is input from a channel end. The body of the `while` loop transmits a single frame and the body of the outer `for` transmits each column. The program instructs the port `DTMG` to start driving its pin high when it starts outputting a column of data and to stop driving afterwards.

An alternate solution is to configure the port `DATA` to generate a ready-out strobe signal on `DTMG` (see §6.4) and to remove the two outputs to `DTMG` by the processor in the source code.

4.5 Summary of Clocking Behaviour

The semantics for inputs and outputs on clocked (unbuffered) ports are summarised as follows.

Output Statements

- An *output* causes data to be driven on the next falling edge of the clock. The output blocks until the subsequent rising edge.
- A *timed output* causes data to be driven by the port when its counter equals the specified time. The output blocks until the next rising edge after this time.
- The data driven on one edge continues to be driven on subsequent edges for which no new output data is provided.

Input Statements

- An *input* causes data to be sampled by the port on the next rising edge of its clock. The input blocks until this time.
- A *timed input* causes data to be sampled by the port when its counter equals the specified time. The input blocks until this time.
- A *conditional input* causes data to be sampled by the port on each rising edge until the sampled data satisfies the condition. The input blocks until this time, taking the most recent data sampled.

Select Statements

A select statement waits for any one of the ports in its cases to become ready and completes the corresponding input operation, where:

- For an *input*, the port is ready at most once per period of its clock.
- For a *timed input*, the port is ready only when its counter equals the specified time.
- For a *conditional input*, the port is ready only when the data sampled satisfies the condition.
- For a *timed conditional input*, the port is ready only when its counter is equal or greater than the specified time and the value sampled satisfies the condition.

For a timestamped operation that records the value t, the next possible time that the thread can input or output is $t + 1$.

On XS1 devices, all ports are buffered (see XM-000971-PC). The resulting semantics, which extend those given above, are discussed in §5.

5 Port Buffering

IN THIS CHAPTER

The XMOS architecture provides buffers that can improve the performance of programs that perform I/O on clocked ports. A buffer can hold data output by the processor until the next falling edge of the port's clock, allowing the processor to execute other instructions during this time. It can also store data sampled by a port until the processor is ready to input it. Using buffers, a single thread can perform I/O on multiple ports in parallel.

5.1 Using a Buffered Port

The following program uses a buffered port to decouple the sampling and driving of data on ports from a computation.

```
#include <xs1.h>

in  buffered port:8 inP     = XS1_PORT_8A;
out buffered port:8 outP    = XS1_PORT_8B;
in port             inClock = XS1_PORT_1A;
clock               clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port(inP, clk);
  configure_out_port(outP, clk, 0);
  start_clock(clk);
  while (1) {
    int x;
    inP  :> x;
    outP <: x + 1;
    f();
  }
}
```

The program configures the ports `inP`, `outP` and `inClock` as illustrated in Figure 22.

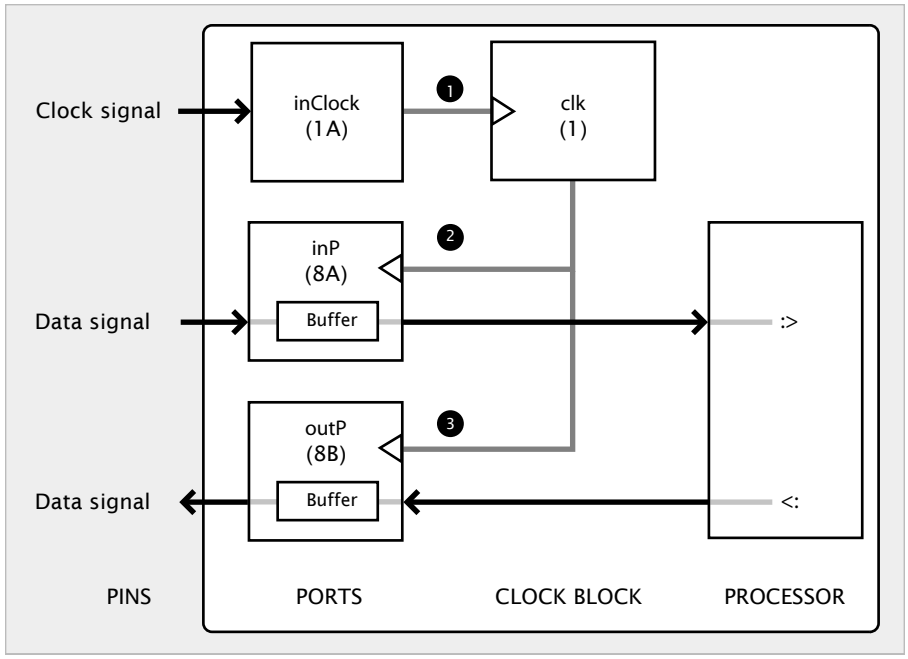


Figure 22: Port configuration diagram

The declaration

```
in buffered port:8 inP = XS1_PORT_8A;
```

declares a buffered input port named `inP`, which refers to the 8-bit port identifier 8A.

1. The statement

```
configure_clock_src(clk, inClock);
```

configures the 1-bit input port `inClock` to provide edges for the clock `clk`.

2. The statement

```
configure_in_port(inP, clk);
```

configures the input port `inP` to be clocked by the clock `clk`.

3. The statement

```
configure_out_port(outP, clk, 0);
```

configures the output port `outP` to be clocked by the clock `clk`, with an initial value of 0 driven on its pins.

Figure 23 shows example input stimuli and expected output for this program. It also shows the relative waveform of the statements executed in the `while` loop by the processor.

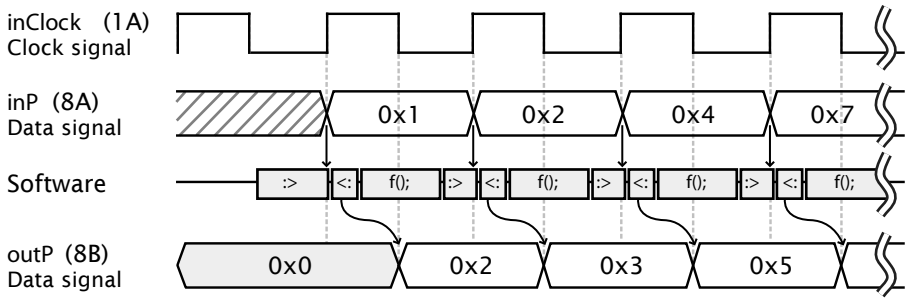


Figure 23: Waveform diagram relative to processor execution

The first three values input are 0x1, 0x2 and 0x4, and in response the values output are 0x2, 0x3 and 0x5.

Figure 24 illustrates the buffering operation in the hardware. It shows the processor executing the `while` loop that outputs data to the port. The port buffers this data so that the processor can continue executing subsequent instructions while the port drives the data previously output for a complete period. On each falling edge of the clock, the port takes the next byte of data from its buffer and drives it on its pins. As long as the instructions in the loop execute in less time than the port's clock period, a new value is driven on the pins on every clock period.

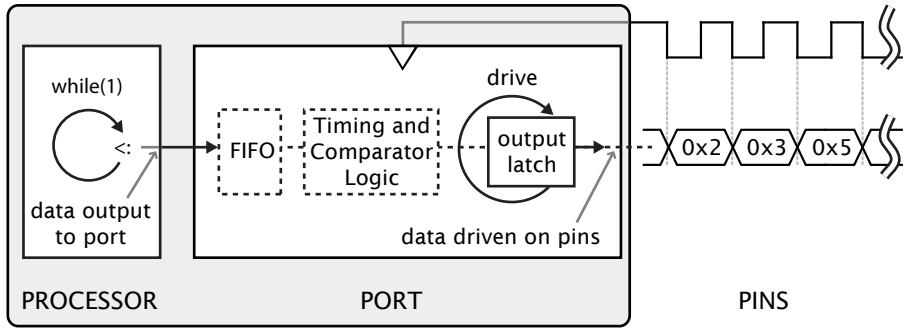


Figure 24: Port hardware logic

The fact that the first input statement is executed before a rising edge means that the input buffer is not used. The processor is always ready to input the next data before it is sampled, which causes the processor to block, effectively slowing itself down to the rate of the port. If the first input occurs after the first value is sampled, however, the input buffer holds the data until the processor is ready to accept it and each output blocks until the previously output value is driven.

Timed operations represent time in the future. The waveform and comparator logic allows timed outputs to be buffered, but for timed and conditional inputs the buffer is emptied before the input is performed.

5.2 Synchronising Clocked I/O on Multiple Ports

By configuring more than one buffered port to be clocked from the same source, a single thread can cause data to be sampled and driven in parallel on these ports. The program below first synchronises itself to the start of a clock period, ensuring the maximum amount of time before the next falling edge, and then outputs a sequence of 8-bit character values to two 4-bit ports that are driven in parallel.

```
#include <xs1.h>

out buffered port p:4     = XS1_PORT_4A;
out buffered port q:4     = XS1_PORT_4B;
in           port inClock = XS1_PORT_1A;
clock        clk          = XS1_CLKBLK_1;

int main(void) {

  configure_clock_src(clk, inClock);
  configure_out_port(p, clk, 0);
  configure_out_port(q, clk, 0);
  start_clock(clk);

  p <: 0;  // start an output
  sync(p); // synchronise to falling edge

  for (char c='A'; c<='Z'; c++) {
    p <: (c & 0xF0) >> 4;
    q <: (c & 0x0F);
  }
}
```

The statement

```
sync(p);
```

causes the processor to wait until the next falling edge on which the last data in the buffer has been driven for a full period, ensuring that the next instruction is executed just after a falling edge. This ensures that the subsequent two output statements in the loop are both executed in the same clock period. Figure 25 shows the data output by the processor and driven by the two ports.

The recommended way to synchronise to a rising edge is to clear the buffer using the standard library function `clearbuf` and then perform an input.

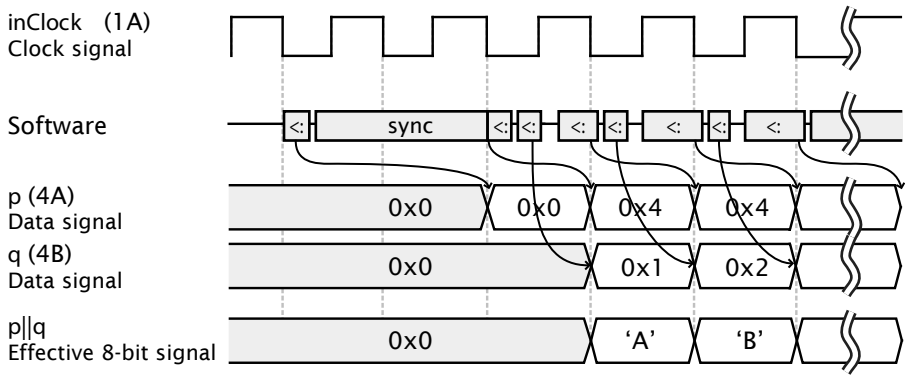


Figure 25: Processor synchronizing data on two output ports

5.3 Summary of Buffered Behaviour

The semantics for I/O on clocked buffered ports are summarised as follows.

Output Statements

- An output inserts data into the port's FIFO. The processor waits only if the FIFO is full.
- At most one data value is removed from the FIFO and driven by the port per period of its clock.
- A timed output inserts data into the port's FIFO for driving when the port counter equals the specified time. The processor waits only if the FIFO is full.
- A timestamped output causes the processor to wait until the output is driven (required to determine the timestamp value).
- The data driven on one edge continues to be driven on subsequent edges.

Input Statements

- At most one value is sampled by the port and inserted into its FIFO per period of its clock. If the FIFO is full, its oldest value is dropped to make room for the most recently sampled value.
- An input removes data from a port's FIFO. The processor waits only if the FIFO is empty.
- Timed and conditional inputs cause any data in the FIFO to be discarded and then behave as in the unbuffered case.

6 Serialisation and Strobing

IN THIS CHAPTER

The XMOS architecture provides hardware support for operations that frequently arise in communication protocols. A port can be configured to perform *serialisation*, useful if data must be communicated over ports that are only a few bits wide (such as in §2.5), and *strobing*, useful if data is accompanied by a separate data valid signal (such as in §4.4). Offloading these tasks to the ports frees up more processor time for executing computations.

6.1 Serialising Output Data using a Port

A clocked port can serialise data, reducing the number of instructions required to perform an output. The program below outputs a 32-bit value onto 8 pins, using a clock to determine for how long each 8-bit value is driven.

```
#include <xs1.h>

out buffered port:32 outP    = XS1_PORT_8A;
in           port    inClock = XS1_PORT_1A;
clock                clk     = XS1_CLKBLK_1;

int main(void) {
  int x = 0xAA00FFFF;
  configure_clock_src(clk, inClock);
  configure_out_port(outP, clk, 0);
  start_clock(clk);

  while (1) {
    outP <: x;
    x = f(x);
  }
}
```

The declaration

```
out buffered port:32 outP = XS1_PORT_8A;
```

declares the port `outP` to drive 8 pins from a 32-bit *shift register*. The type `port:32` specifies the number of bits that are transferred in each output operation (the *transfer width*). The initialisation `XS1_PORT_8A` specifies the number of physical pins connected to the port (the *port width*). Figure 26 shows the data driven by this program.

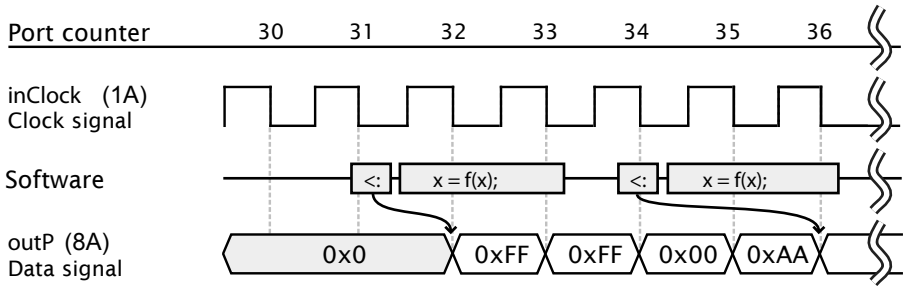


Figure 26: Serialised output waveform diagram

By offloading the serialisation to the port, the processor has only to output once every 4 clock periods. On each falling edge of the clock, the least significant 8 bits of the shift register are driven on the pins; the shift register is then right-shifted by 8 bits.

On XS1 devices, ports used for serialisation must be qualified with the keyword `buffered`; see XM-000971-PC for further explanation.

6.2 Deserialising Input Data using a Port

A port can deserialise data, reducing the number of instructions required to input data. The program below performs a 4-to-8 bit conversion on an input port, controlled by a 25MHz clock.

```
#include <xs1.h>
in  buffered port:8 inP      = XS1_PORT_4A;
out          port   outClock = XS1_PORT_1A;
clock               clk25    = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk25, 100, 4);
  configure_in_port(inP, clk25);
  configure_port_clock_output(outClock, clk25);
  start_clock(clk25);
  while (1) {
    int x;
    inP :> x;
    f(x);
} }
```

The program declares `inP` to be a 4-bit wide port with an 8-bit transfer width, meaning that two 4-bit values can be sampled by the port before they must be input by the processor. As with output, the deserialiser reduces the number of instructions required to obtain the data. Figure 27 shows example input stimuli and the period during which the data is available in the port's buffer for input.

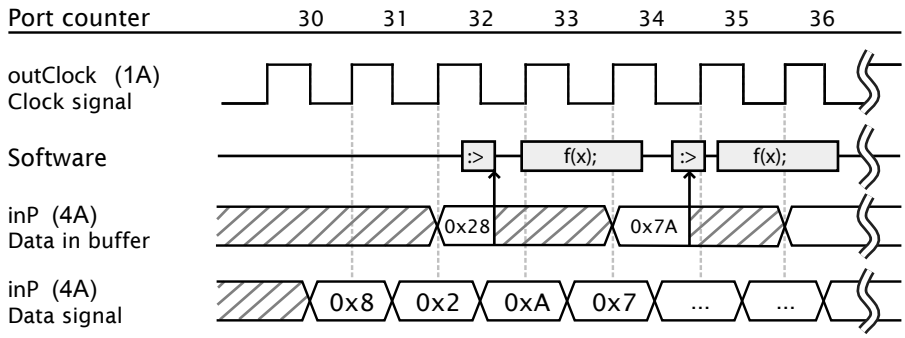


Figure 27: Deserialised input waveform diagram

Data is sampled on the rising edges of the clock and, when shifting, the least significant nibble is read first. The sampled data is available in the port's buffer for input for two clock periods. The first two values input are 0x28 and 0x7A.

6.3 Inputting Data Accompanied by a Data Valid Signal

A clocked port can interpret a *ready-in* strobe signal that determines the validity of the accompanying data. The program below inputs data from a clocked port only when a ready-in signal is high.

```
#include <xs1.h>

in buffered port:8 inP     = XS1_PORT_4A;
in          port   inReady = XS1_PORT_1A;
in          port   inClock = XS1_PORT_1B;
clock              clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port_strobed_slave(inP, inReady, clk);
  start_clock(clk);

  inP :> void;
}
```

The statement

```
configure_in_port_strobed_slave(inP, inReady, clk);
```

configures the input port `inP` to be sampled only when the value sampled on the port `inReady` equals 1. The ready-in port must be 1-bit wide. Figure 28 shows example input stimuli and the data input by this program.

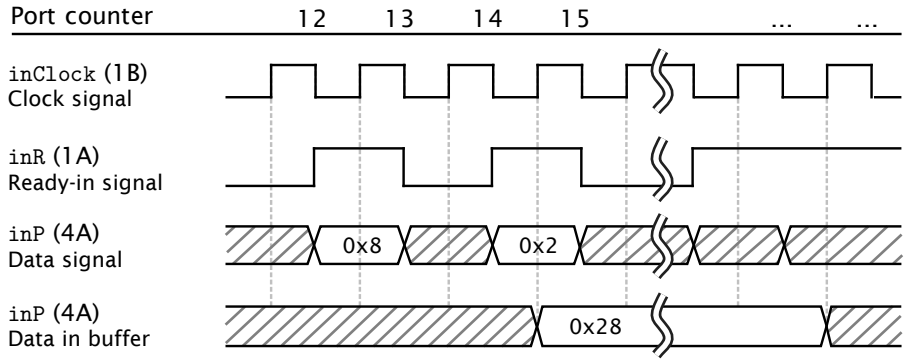


Figure 28: Input data with data valid signal

Data is sampled on the rising edge of the clock whenever the ready-in signal is high. The port samples two 4-bit values and combines them to produce a single 8-bit value for input by the processor; the data input is 0x28. XS1 devices have a single-entry buffer, which means that data is available for input until the ready-in signal is high for the next two rising edges of the clock. Note that the port counter is incremented on every clock period, regardless of whether the strobe signal is high.

6.4 Outputting Data and a Data Valid Signal

A clocked port can generate a *ready-out* strobe signal whenever data is output. The program below causes an output port to drive a data valid signal whenever data is driven on a 4-bit port.

```
#include <xs1.h>

out buffered port:8 outP    = XS1_PORT_4B;
out           port  outR    = XS1_PORT_1A;
in            port  inClock = XS1_PORT_1B;
clock               clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_out_port_strobed_master(outP, outR, clk, 0);
  start_clock(clk);

  outP <: 0x85;
}
```

The statement

```
configure_out_port_strobed_master(outP, outR, clk, 0);
```

configures the output port `outP` to drive the port `outR` high whenever data is output. The ready-out port must be 1-bit wide. Figure 29 shows the data and strobe signals driven by this program.

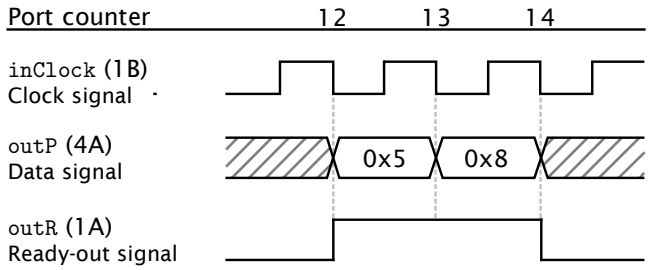


Figure 29: Output data with data valid signal

The port drives two 4-bit values over two clock periods, raising the ready-out signal during this time.

It is also possible to implement control flow algorithms that output data using a ready-in strobe signal and that input data using a ready-out strobe signal; when both signals are configured, the port implements a symmetric strobe protocol that uses a clock to handshake the communication of the data (see XM-000969-PC).

On XS1 devices, ports used for strobing must be qualified with the keyword `buffered`; see XM-000971-PC for further explanation.

6.5 Case Study: Ethernet MII

A single thread on an XS1 device can be used to implement a full duplex 100Mbps Ethernet Media Independent Interface (MII) protocol[4]. This protocol implements the data transfer signals between the link layer and physical device (PHY). The signals are shown in Figure 30.

[4] http://www.xmos.com/references/ethernet-mii-spec

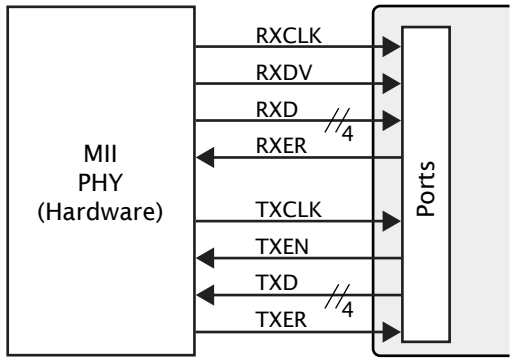


Figure 30: MII signal diagram

6.5.1 MII Transmit

Figure 31 shows the transmission of a single frame of data to the PHY. The error signal TXER is omitted for simplicity.

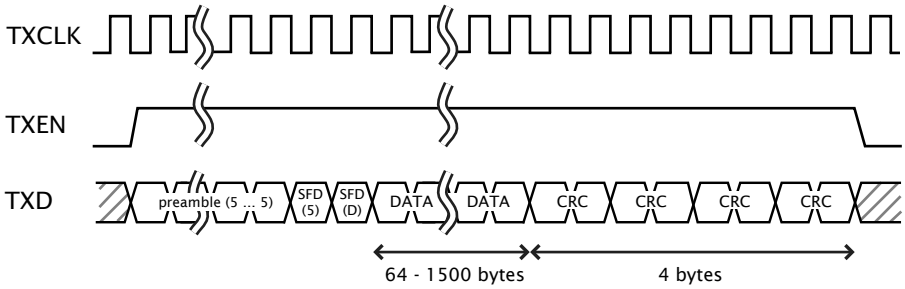


Figure 31: MII transmit waveform diagram

The signals are as follows:

- TXCLK is a free running 25MHz clock generated by the PHY.
- TXEN is a data valid signal driven high by the transmitter during frame transmission.
- TXD carries a nibble of data per clock period from the transmitter to the PHY. The transmitter starts by sending a preamble of nibbles of value 0x5, followed by two nibbles of values 0x5 and 0xD. The data, which must be in the range of 64 to 1500 bytes, is then transmitted, least significant bit first, followed by four bytes containing a CRC.

Figure 32 illustrates the port configuration required to serialise the output data and produce a data valid signal.

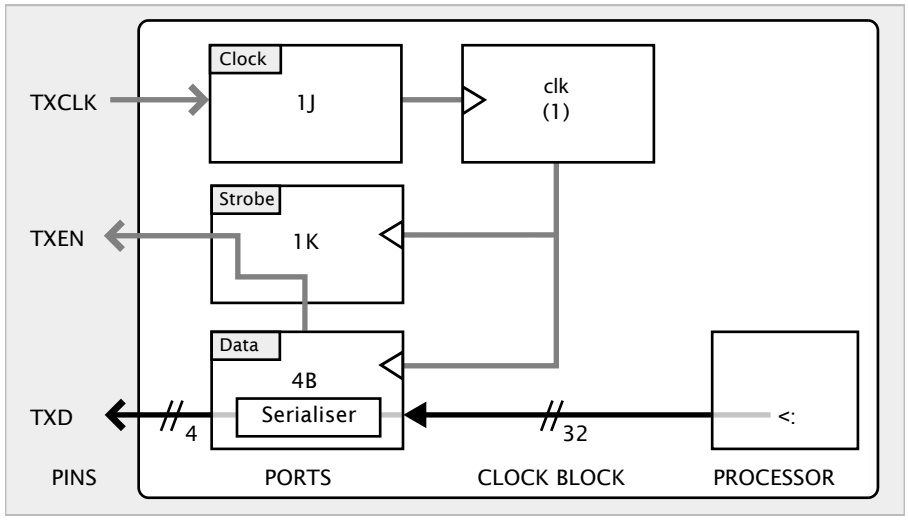


Figure 32: MII transmit port configuration

The port `TXD` performs a 32-to-4 bit serialisation of data onto its pins. It is synchronised to the 1-bit port `TXCLK` and uses the 1-bit port `TXEN` as a ready-out strobe signal that is driven high whenever data is driven. In this configuration, the processor has only to output data once every eight clock periods and does not need to explicitly output the data valid signal. The program below defines and configures the ports in this way.

```
#include <xs1.h>
out buffered port:32 TXD   = XS1_PORT_4B;
out          port    TXEN  = XS1_PORT_1K;
in           port    TXCLK = XS1_PORT_1J;
clock                clk   = XS1_CLKBLK_1;

void miiConfigTransmit(clock clk,
  buffered out port:32 TXD, out port TXEN) {

  configure_clock_src(clk, TXCLK);
  configure_out_port(TXD, clk);
  configure_out_port(TXEN, clk);
  configure_out_port_strobed_master(TXD, TXEN, clk, 0);
  start_clock(clk);
}
```

The function below inputs frame data from another thread and outputs it to the MII ports. For simplicity, the error signals and CRC are ignored.

```
void miiTransmitFrame(out buffered port:32 TXD,
                      streaming chanend c) {

      int numBytes, tailBytes, tailBits, data;

  /* Input size of next packet */
  c :> numBytes;
  tailBytes = numBytes / 4;
  tailBits = tailBytes * 8;

  /* Output row of 0x5s followed by 0xD */
  TXD <: 0xD5555555;

  /* Output 32-bit words for serialisation */
  for (int i=0; i<numBytes-tailBytes; i+=4) {
    c   :> data;
    TXD <: data;
  }

  /* Output remaining bits of data for serialisation */
  if (tailBits != 0) {
    c :> data;
    partout(TXD, tailBits, data);
  }
}
```

The program first inputs from the channel c the size of the frame in bytes. It then outputs a 32-bit preamble to `TXD`, which is driven on the pins as nibbles over eight clock periods. On each iteration of the for loop, the next 32 bits of data are then output to `TXD` for serialising onto the pins. This gives the processor enough time to get around the loop before the next block of data must be driven. The final statement

```
partout(TXD, tailBits, data);
```

performs a *partial output* of the remaining bits of data that represent valid frame data.

6.5.2 MII Receive

Figure 33 shows the reception of a single frame from the PHY. The error signal RXER is omitted for simplicity.

The signals are as follows:

- RXCLK is a free running clock generated by the PHY.
- RXDV is a data valid signal driven high by the PHY during frame transmission.

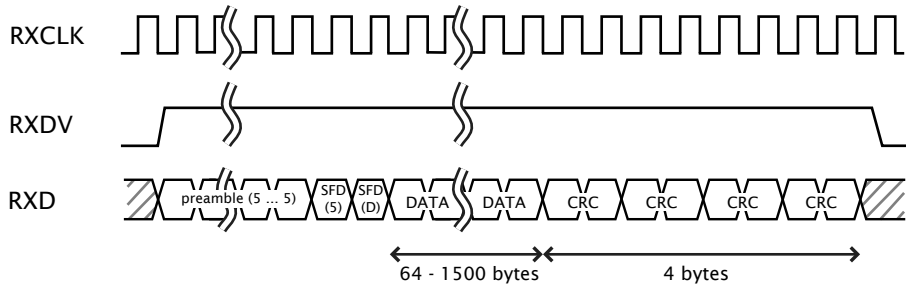


Figure 33: MII receive waveform diagram

- RXD carries a nibble of data per clock period from the PHY to the receiver. The receiver waits for a preamble of nibbles of values 0x5, followed by two nibbles with values 0x5 and 0xD. The actual data is then received, which is in the range of 64 to 1500 bytes, least significant nibble first, followed by four bytes containing a CRC.

Figure 34 illustrates the port configuration required to deserialise the input data when a data valid signal is present.

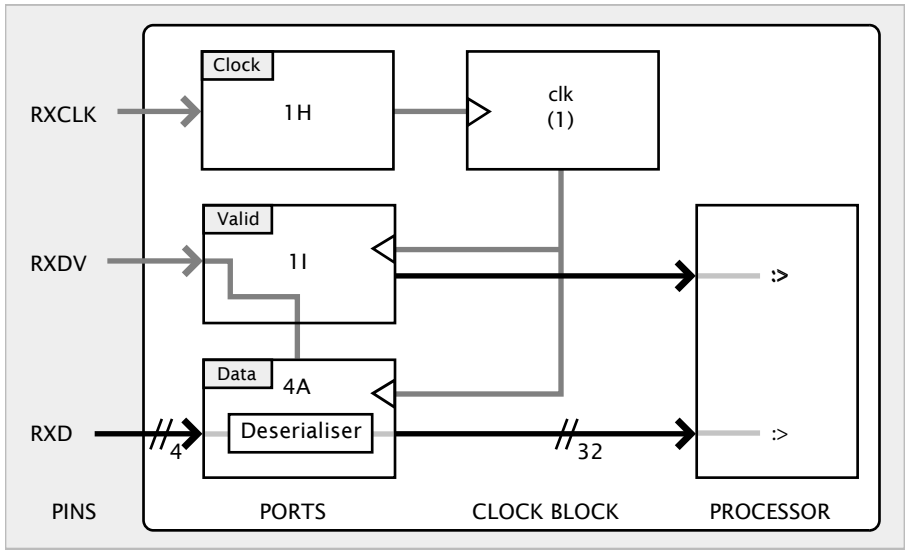


Figure 34: MII receive port configuration

The port `RXD` performs a 4-to-32-bit deserialisation of data from its pins. It is synchronised to the 1-bit port `RXCLK` and uses the 1-bit port `RXDV` as a ready-in strobe signal that causes data to be sampled only when the strobe is high. In this configuration, the port can sample eight values before the data must be input by the processor, and the processor does not need to explicitly wait for the data valid signal. The program below defines and configures the ports in this way.

```
#include <xs1.h>
in buffered port:32 RXD   = XS1_PORT_4A;
in          port    RXDV  = XS1_PORT_1I;
in          port    RXCLK = XS1_PORT_1H;
clock               clk   = XS1_CLKBLK_1;

void miiConfigReceive(clock clk, in port RXCLK,
  buffered in port:32 RXD, in port RXDV, in port RXER) {

  configure_clock_src(clk, RXCLK);
  configure_in_port(RXD, clk);
  configure_in_port(RXDV, clk);
  configure_in_port_strobed_slave(RXD, RXDV, clk);
  start_clock(clk);
}
```

The function below receives a single error-free frame and outputs it to another thread. For simplicity, the error signal and CRC are ignored.

```
#define MORE 0
#define DONE 1

void miiReceiveFrame(in buffered port:32 RXD, in port RXDV,
                     streaming chanend c) {
  int notDone = 1;
  int data, tail;

  /* Wait for start of frame */
  RXD when pinseq(0xD) :> void;

  /* Receive frame data/crc */
  do {
    select {
      case RXD :> data :
        /* input next 32 bits of data */
        c <: MORE;
        c <: data;
        break;
      case RXDV when pinseq(0) :> notDone :
        /* Input any bits remaining in port */
        tail = endin(RXD);
        for (int byte=tail>>3; byte > 0; byte-=4) {
          RXD :> data;
          c <: MORE;
          c <: data;
        }
        c <: DONE;
        c <: tail >> 3;
        break;
    }
  } while (notDone);
}
```

The processor waits for the last nibble of the preamble (0xD) to be sampled by the port RXD. Then on each iteration of the loop, it waits for either next eight nibbles of data to be sampled for input by RXD or for the data valid signal RXDV to go low.

An effect of using a port's serialisation and strobing capabilities together is that the ready-in signal may go low before a full transfer width's worth of data is received.

The statement

```
tail = endin(RXD);
```

causes the port RXD to respond with the remaining number of bits not yet input. It also causes the port to provide this data on the subsequent inputs, even though the data valid signal is low and the shift register is not yet full.

XS1 devices provide a single-entry buffer up to 32-bits wide and a 32-bit shift register, requiring up to 64 bits of data being input over two input statements once the data valid signal goes low.

6.6 Summary

The semantics for I/O on a serialised port are as follows (where *p* refers to the port width and *w* refers to the transfer width of a port):

- An output of a *w*-bit value is driven over *w*/*p* consecutive clock periods, least significant bits first. The ready-out signal is driven high on each of these periods.
- For a timed output, the port waits until its counter equals the specified time before *starting* to serialise the data. The ready-out signal is not driven while waiting to serialise.
- An input of a *w*-bit value is sampled over *w*/*p* clock periods, with earlier bits received ending up in the least significant bits of *w*. (If a ready-in signal is used, the clock periods may not be consecutive.)
- For a timed input, the port provides the *last p* bits of data sampled when its counter equals the specified time.

If a port is configured with a ready-in signal:

- Data is sampled only on rising edges of the port's clock when the ready-in signal is high.

If a port is configured with a ready-out signal:

- The ready-out signal is driven high along with the data and is held for a single period of the clock.