

XMOS programmable chips bring together the capabilities of processors, DSPs, ASICs and FPGAs. XMOS technology allows development times to be reduced whilst maintaining product flexibility and keeping system costs down.

	Low Level		High Level
Software	Assembler	→	C
Hardware	Verilog	→	XC

XMOS technology allows the hardware design process to be described in a high-level language (XC) rather than at a low-level in Verilog, reducing development times while maintaining product flexibility and keeping system costs down.

The XMOS XCore® processor is specifically designed for implementing hardware functions in software. Its event driven execution provides an instant response to external signals without the processing overhead that interrupts incur. Intelligent I/O ports allow precision timing of interface signals to specific clock edges or times.

The XC programming language is key to writing hardware designs for the XCore processor. XC is based on the widely used C language with extensions to support the XCore hardware features.

Benefits of XC over Verilog

- More intuitive and quicker to write
- Easier to understand and maintain
- More concise
- Low level constructs are automatically generated

Hardware written in Verilog maps code sections to the specific registers and low-level logic that it describes. This is equivalent to low-level software written in assembler mapping to specific processor instructions. Software written in high-level languages is quicker and easier to write because it is abstracted away from the low-level machine instructions. It is also concise, readable and easier to maintain. Describing hardware in XC gives a designer the benefits of using a high-level language, whilst maintaining tight control over the signal protocol on the chip's pins.

The rest of this whitepaper compares Verilog and XC implementations for several simple applications to demonstrate how Verilog structures are implemented in XC, and to highlight the basic XC structures. It assumes a working knowledge of C and Verilog. Note that some of the Verilog example code is abridged to make the code comparison clearer.

Simple I/O Application

Loop back an input signal to drive an output.

In XC, signal I/O is achieved through ports. A port is declared by naming it (RXD and TXD below) and binding it to a hardware port (e.g. XS1_PORT_1H) and hence to a pin on the chip.

The signal on a chip's pin is sampled by a port and loaded into a variable using the `>` operator. Similarly a signal is driven out of a chip's pin by outputting a variable to a port using the `<` operator.

XC	Verilog
<pre> in port RXD = XS1_PORT_1I; out port TXD = XS1_PORT_1H; int main(void) { int RXD_sample; while (1) { RXD >: RXD_sample; TXD <: RXD_sample; } } </pre>	<pre> module uart_loopback (RXD, TXD); input RXD ; output TXD ; reg TXD ; always @ (RXD) TXD <= RXD; endmodule </pre>

Clocked I/O Application

Sample a port on a rising clock edge, and output on the next falling edge.

By default, ports are clocked by a 100MHz reference clock (see *Simple I/O* example above). However an external signal can be used to clock the port. In this example a clock is declared (`clkblk`) which selects one of the XCore hardware clock blocks (`XS1_CLKBLK_1`). `configure_clock_src()` is then used to select the external signal `ext_clk` as the clock source for `clkblk`. The `data_in` and `data_out` ports are then configured to use `clkblk` to as their clock source.

XC fundamental

Code execution pauses to wait for a hardware resource to become ready to provide or accept data using the `>` or `<` resource operators

Within the `while(1)` loop, code execution will pause on the line where the `data_in` port is sampled using `>` to wait for the positive edge of `ext_clk` to occur. When it does, `data_in` is sampled and code execution restarts. The sampled data is then sent to the `data_out` port, which accepts it ready to drive the data out on the falling edge of `ext_clk`. The code does not pause here because the port accepts the output data.

XC samples data on a clock's rising edge but outputs it on the falling edge. Doing this means that output data can be sampled by the reading device on the clock's rising edge without having to worry about the skew between the clock and data across the PCB. Clock inputs can be inverted to provide fully flexible operation.

XC	Verilog
<pre> in port data_in = XS1_PORT_8A; out port data_out = XS1_PORT_8B; in port ext_clk = XS1_PORT_1A; clock clkblk = XS1_CLKBLK_1; int main(void) { int sample_data; configure_clock_src (clkblk,ext_clk); configure_in_port (data_in ,clkblk); configure_out_port (data_out,clkblk,0xaa); while (1) { data_in > sample_data; // Pause here data_out <: sample_data; } } </pre>	<pre> ... reg [7:0] data_out; reg [7:0] sample_data; always @ (posedge ext_clk or negedge reset_n) if (reset_n == 0) sample_data <= 8'haa; else sample_data <= data_in; always @ (negedge ext_clk // NOTE negedge or negedge reset_n) if (reset_n == 0) data_out <= 8'haa; else data_out <= sample_data; ... </pre>

Timing Control Application

Increment an output counter 10kHz from a clock derived from a 100MHz clock.

In traditional hardware design, timing control is performed by clock manipulation. Ensuring that the frequencies required by the design are available can mean implementing clock divider circuits, counters and adding additional crystal oscillators to the design.

Timing control in XC is achieved by deriving delays from the 100MHz timers. Before the `while(1)` loop in this example, the 32 bit time is read from timer `t` using the `>` operator. Within the loop the counter is output to a port and incremented ready for the next iteration. `update_time` is increased so that it corresponds to a time in the future when the next update should occur. `timerafter()` is then used to pause code execution until this time.

Note that the time is also read by this code line, but is ignored by reading it to `void`. The loop completes and the next counter value is output.

Use of timers makes the creation of designs with precision timing simple and intuitive in XC and allows the designer to work at a level above that of 'clock manipulation'.

XC	Verilog
<pre>#include <platform.h> #define UPDATE_RATE 10000 // 10kHz #define U_DELAY (XS1_TIMER_HZ/UPDATE_RATE) out port count_port = XS1_PORT_8A; int main(void) { int count = 0; int update_time; timer t; t :> update_time; // Get the time while (1) { count_port <: count; count++; update_time += U_DELAY; t when timerafter(update_time) :> void; } }</pre>	<pre>... `define U_CLK_DIVIDER 10000 // 10kHz reg u_clk ; reg [15:0] u_clk_cnt ; reg [7:0] count_port; always @ (posedge clk or negedge reset_n) if (reset_n == 0) begin u_clk <= 0; u_clk_cnt <= 0; end else begin if (u_clk_cnt == 0) begin u_clk <= ~u_clk; u_clk_cnt <= `U_CLK_DIVIDER; end else begin u_clk <= u_clk; u_clk_cnt <= u_clk_cnt -1; end end always @ (posedge u_clk or negedge reset_n) if (reset_n == 0) count_port <= 0; else count_port <= count_port +1; ... </pre>

XC Resource Operators

- `>` Input From a port, timer or channel end
- `<` Output To a port or channel end

Design Partitioning and Communication Example Application

Use a design block to generate an index to a look up table in another block.

Partitioning a design into optimally sized blocks is a skill that requires engineering judgment whether the implementation is in Verilog or XC. Design blocks must be cohesive and transfer data between each other over carefully designed interfaces.

XC allows threads of code to be spawned to run design blocks in parallel. These parallel threads may execute on the same processor or a remote processor. Either way, the threads communicate with each other by sending data over high-speed data channels that avoid excessive latency between threads.

In this example, parallel code threads for the `index_gen()` and `lut()` design blocks are spawned by function calls within the `par{}` block. The functions are declared and called as in C, but because the calls are within the `par{}` block they execute in separate threads.

Instead of declaring and connecting wires at the top level to allow sub modules to exchange data, XC uses a communications channel (`chan`), called `index_ch`, which is created by declaring it. `index_ch`'s two channel ends (`chanend`) are connected to the two parallel design blocks called within the `par{}` so they can exchange data through the channel.

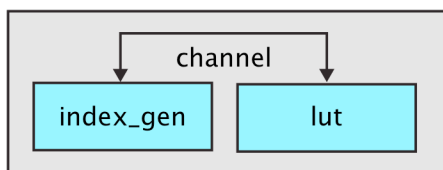
Within the `index_gen` code thread, a 32 bit counter value (`count`) is passed down `index_ch` to the `lut()` thread. The `<:` operator is used to write data to the resource (in this case the channel), whilst the `>` operator is used to read the data from the channel. The least significant bits are then driven onto the 8-bit port in this example.

XC fundamental

Two threads synchronize when one writes data into a channel and the other reads it out. Usually one thread will pause, waiting for the other to reach that point in the code.

The lut () thread will pause every iteration until it synchronizes with index_gen () when that thread issues the next index value down index_ch, delayed using a timer.

XC	Verilog
<pre>#include <platform.h> port data_out = XS1_PORT_8A; void index_gen(chanend index_ch); void lut(chanend index_ch, port data_out); int main(void) { chan index_ch; par { index_gen(index_ch); lut(index_ch, data_out); } }</pre>	<pre>module lut_top (input reset_n , input clk , output [7:0] data_out); wire [7:0] index; wire index_valid; index_gen index_gen_inst (.clk(clk), .reset_n (reset_n), .index (index), .index_valid (index_valid)); lut lut_inst (.clk(clk), .reset_n (reset_n), .index (index), .index_valid (index_valid), .data_out (data_out)); endmodule</pre>
<pre>void index_gen(chanend index_ch){ int count = 0; int u_time; timer t; t := u_time; while(1){ index_ch <: count++; u_time += U_DELAY; t when timerafter(u_time) :=> void; } };</pre>	<pre>... always @ (posedge clk or negedge reset_n) if (reset_n == 0) begin index <= 0; index_valid <= 0; count <= 0; end else begin if (u_time == 0) begin index <= count; index_valid <= 1; count <= count +1; end else begin index <= index; index_valid <= 0; count <= count; end end end</pre>
<pre>void lut(chanend index_ch, port data_out){ int index; while(1){ index_ch :=> index; data_out <: lut_array[index]; } }</pre>	<pre>... always @ (posedge clk or negedge reset_n) ... if (index_valid == 1) data_out <= lut_array[index]; else data_out <= data_out;</pre>

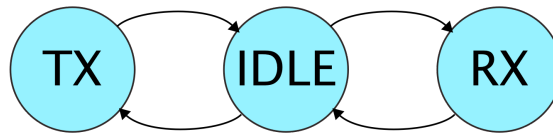


XC Fundamental: 'instantiate' parallel design block threads with par{} and connect with channels not wires.

```
chan index_ch;
par {
    index_gen(index_ch);
    lut(index_ch, data_out);
}
```

State Machine

A state machine that can move from IDLE to either TX or RX mode and back.



The state machine is explicitly declared in the Verilog version below, with the state changing to TX_MODE or RX_MODE at times before returning to IDLE. By contrast, there is no need to explicitly encode the state machine in XC. Instead the state of the design is determined by the thread's position within the code.

XC fundamental

Events cause a thread to execute specific code when a specific hardware condition occurs. Many events may be setup by a `select` statement, but only the first event to occur is taken.

In the code below, a thread may pause at the `select` statement, but restarts and executes the case code for whichever event occurs first. This is equivalent to moving from the IDLE state to either TX_MODE or RX_MODE in the Verilog version. When that case code completes it is equivalent of returning to IDLE.

Only one of the cases within a `select` statement will execute. However in this example the select is within a `while(1)` loop so the `select` statement will be rerun and the events will be setup again.

The first case uses `pinseq()` to enable an event which triggers when the `tx_mode_p` input port is equal to 1, causing `tx()` to run. The second case enables an event to be triggered when an `rx` command is received down a channel from another thread, causing `rx()` to execute.

XC	Verilog
<pre> while (1) { select { case tx_mode_p when pinseq(1):> void : tx (tx_data, tx_port); break ; case rx_mode_chan :> rx_cmd : rx (rx_cmd, rx_port) break ; } } </pre>	<pre> always @ (posedge clk or negedge reset_n) ... case (state_machine) `IDLE: if (tx_mode == 1) state_machine <= `TX_MODE; else if (rx_mode == 1) state_machine <= `RX_MODE; else state_machine <= `IDLE; `TX_MODE: begin ... `RX_MODE: begin ... endcase </pre>

UART: Real World Application

A simple UART with asynchronous transmit and receive.

Although simple examples and short code fragments give a useful insight into how hardware applications can be coded in XC rather than Verilog, it is often easier to appreciate how they work together when brought together in a real world application.

The files attached to this document (select View>Navigational Panels>Attachments in Acrobat Reader or go to <http://xmos.com/kbase/index.php?View=entry&EntryID=32>) provide real-world UART applications in both XC and Verilog for comparison.

Note that the transmit and receive design blocks are asynchronous, so they must be executed on separate threads.

Other Useful XC Features

This document covers the main structures in XC. Additional XC features that Verilog designers might want to investigate include:

- **Buffered Ports:** Allow port data to be serialized, so that a single 32 bit output to an 8 bit port can provide the next 4 output values on successive internal or external clock cycles. Similarly input data can be deserialized, so 32 samples from a 1 bit input port can be obtained with a single input.
- **Strobing:** Enables input ports to only input when a data valid signal on another port is also present. Similarly, a data valid output signal can be driven to a port when valid data is output on a separate output port.
- **Timestamping:** Input ports can be sampled after a specific number of port clock cycles. Similarly, outputs can be timed to occur on specific clocks too.
- **Pad Delays:** Signals on input ports can be delayed by up to 5 system clock cycles. This can counter signal skew through external buffer chips and on the PCB with an accuracy of 2ns.

Summary

Using XMOS devices, designers can now describe hardware using a high-level language, XC, rather than at a low-level in Verilog. Like writing software in C rather than assembler, the benefits of using XC are that code is:

- More intuitive and quicker to write
- Easier to understand and maintain
- More concise
- Low-level constructs are automatically generated
- Software compiles and links much faster than RTL synthesizes and routes, so using XC also means shorter design and debug cycle.

Further Reading

The full details of the XC language are described in *Programming XC on XMOS Devices* - www.xmos.com/published/xc1_en

Additional documentation is available on the XMOS website - www.xmos.com/documentation

Code Example

The XC and Verilog example UART code can be downloaded from <http://xmos.com/kbase/index.php?View=entry&EntryID=32>