AN03010: Incremental design extension

Publication Date: 2025/10/31
Document Number: XM-000000-AN v1.0.0

IN THIS DOCUMENT

1 Introduction

When using an XCORE, one approach involves obtaining an XCORE application and then extending it. This approach may include:

- Adding one or more new interface peripherals,
- Adding one or more custom port handlers,
- Placing one or more new threads of processing,
- Changing the tile location for existing threads of processing, and
- Combining threads of processing.

This application note discusses these topics and provides some guidance on achieving a suitable arrangement. It includes a concrete example which illustrates several of the design techniques it has discussed.

2 The design space

Every XCORE design involves a network of XCORE processors. This network may be small, two (or even one) XCORE tiles in a single XCORE device, or it may be larger, four or more XCORE tiles in one or more XCORE devices. Programming an XCORE involves:

- Placing interface peripherals, custom port handlers, and computational threads of processing on tiles in the system's XCORE network of processors,
- Connecting them together to pass data amongst them, and
- Designing each thread of processing, preferably operating in an event-driven manner, to accomplish the overall goals of the system.

3 Resource tracking

Each XCORE tile provides a set of resources used by the software running on it. An XCORE tile has a fixed maximum amount or number of each resource type. These resources include:

- Internal single-cycle SRAM,
- Channel ends,
- Clock blocks,
- Hardware locks,
- Hardware timers,
- Ports, and
- Processing threads.

Note

Some XCORE packages do not bring some ports or portions of ports out to external package pins.

Interface peripherals, custom port handlers, and computational threads of processing make use of these resources to implement their functionality. Typically, allocation of a resource for a particular use occurs during system initialisation, and re-allocation does not take place. Consequently, tracking available and used resources forms a key aspect of the design activity.

3.1 The thread diagram

A thread diagram visually describes the placement of threads of processing, ports, and the connections between them on a set of XCORE tiles. It consists of one large rectangle per tile with smaller rectangles representing ports, circles representing threads of processing, and lines representing connections. Each thread of processing may implement a purely computational piece of functionality, an interface peripheral, or a custom port handler. Fig. 1 shows an example thread diagram.

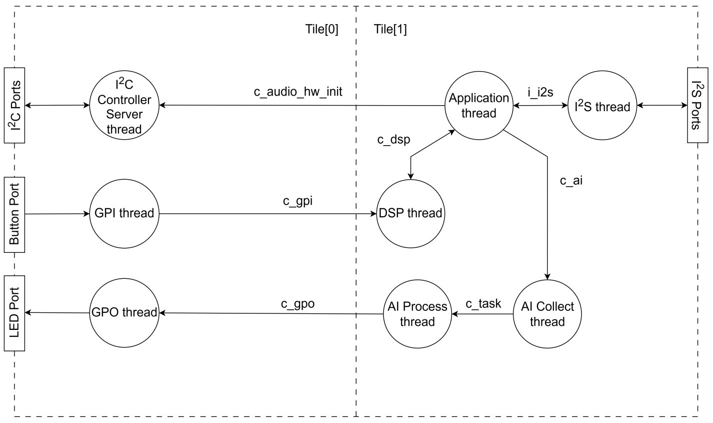

Fig. 1: Example thread diagram

The dashed line rectangles depict a processing network of two XCORE tiles. Each narrow rectangle located on a dashed line depicts a port used by the design. Each circle depicts a thread of processing. The placement of each thread within a dash-line rectangle indicates the tile hosting the thread. Arrows show the direction of data flow between threads and ports. Each arrow labelled with `c_<name>` indicates that the design uses a channel between the connected threads. Each arrow labelled with `i_<name>` indicates an interface to a thread that acts as an interface peripheral.

In this particular case:

- The I^{2}C Controller Server thread and I^{2}S thread operate as Interface Peripherals. The I^{2}S thread provides an internal interface while the I^{2}C thread uses a channel for communication.
- The GPI thread and GPO thread each act as custom port handlers.
- The AI Collect thread, AI Process thread, and DSP thread perform computational functions.
- The Application thread also operates computationally, directing or receiving data to or from the rest of the threads.

3.2 Resource tables

Keeping track of the number of allocated resources of each type can prove to be difficult. A table with one row per resource type and one column per tile can help.

The number in each Tile column generally shows the number of resources used on that tile. However, each clock block has a specific name (e.g., `XS1_CLKBLK_1`). Consequently, when allocating a clock block state its name in the Tile column. Other resources, except for ports, do not have specific names per instance, so stating the number allocated provides sufficient information.

A few special cases exist:

- By default, the build chain allocates clock block 0 for clocking all ports on a tile.
- Each clock block name identifies a clock block within a specific tile. For instance, `XS1_CLKBLK_1` on Tile 0 refers to a different clock block than `XS1_CLKBLK_1` on Tile 1.
- Threads of processing that use the XC programming language have one hardware timer allocated automatically by the build chain. All Timer instances defined within XC for that thread use the same thread-local hardware timer.

The datasheet for an XCORE device states the number of each resource per tile. The table below shows an example. The number in parentheses in the Resource column shows how many of each resource are available per tile. The values in the `Tile 0` and `Tile 1` columns come from the initial application example described later on in this application note.

Resource	Tile 0	Tile 1
Channel ends (32)	3	11
Clock Blocks (6)	XS1_CLKBLK_0	XS1_CLKBLK_0, XS1_CLKBLK_1
HW locks (4)	0	0
HW timers (10)	3	1
Internal Memory (512 kiB)	12,036	134,868
Threads (8)	3	5

Tracking port usage involves more complexity since some ports share package pins with other ports or may have an associated alternative function. The port map described in AN03009: *Placing interfaces on the XCORE* provides the best method to track port allocation.

4 Adding interface peripherals

Application note AN03009: *Placing interfaces on the XCORE* discusses how to configure an XCORE device with a set of interface peripherals. The guidance provided there also applies to adding a new interface peripheral to a set of existing ones. When placing a new interface peripheral, bear in mind that it will use some of the XCORE resources including:

- A thread of processing (although distribution may allow it to share that thread with an interface client),
- One or more ports,
- Some memory on the tile that hosts it,
- Possibly a channel end, and
- Possibly a clock block.

5 Composing processing threads

5.1 Performance categories

The performance requirements for embedded real-time systems often fall into two categories:

- Processing of periodic (or periodically-sampled) data whose result must be delivered at specific, often periodic, points in time (i.e., hard real-time delivery), and
- Processing of data received aperiodically where some leeway exists in the delivery schedule of the result (i.e., soft real-time delivery).

The XCORE works well for these performance categories. With careful design, it can support the processing of data from both categories at the same time.

5.2 Thread patterns

The XCORE supports a wide range of thread interconnection patterns. The two patterns described in this section serve as examples of the flexibility the XCORE makes available for parallel processing. Both of them assume an event-driven approach with no use of interrupts. The XCORE supports other patterns of parallel processing equally well.

Pipeline pattern

One common pattern involves passing data from thread to thread where each thread performs a particular transformation or sequence of transformations.

This pattern works well for processing periodic (or periodically-sampled) data against a hard delivery schedule. Separate threads allow parallel processing of data from different points in time (e.g., while one thread processes data taken at time *nT* through transform *A*, the thread immediately downstream of it processes data taken at time *(n-1)T* through transform *B*). The maximum end-to-end processing delay consists of the sum of the maximum delays for each thread plus a slight overhead for passing the data.

This pattern avoids the possibility of unexpected delays due to interrupts or data congestion at a common point (e.g., a hub). It provides deterministic scheduling of the operations used to transform the data with very low amounts of scheduling jitter.

This pattern may use the same or different interface peripherals as the end points for the pipeline. The data processing may operate on a sample-by-sample basis or samples may be collected into fixed-sized frames and processed on a frame-by-frame basis.

Hub and spoke pattern

Another common pattern uses a central hub in one thread which instructs satellite threads as necessary to accomplish specific operations.

This pattern works well for event-based data with a soft delivery schedule (e.g., for run-time system or component configuration due to input from an external control mechanism). An interface peripheral or custom port handler receives a configuration command aperiodically from an external controller, which it forwards on to a central local controller. The local controller:

- Decodes the command,
- Identifies the internal components that require reconfiguration,
- Sends each identified component the necessary sub-commands,
- Collects the results, and
- Possibly responds to the external controller through the interface peripheral or to another interface peripheral or custom port handler to display a status message or indication.

With this pattern, short delays may occur due to multiple internal components responding to the local controller at the same time. Alternatively, the local controller can sequence the operations for those commands that involve multiple internal components, which may result in different commands taking different amounts of time to complete. However, scheduling within the hub and spoke set of threads operates consistently. The same command will always take the same amount of time to complete within narrow limits.

Hub and spoke plus pipeline combination

Some designs separate the types of processing required into a data plane and a control or configuration plane. Often, the data plane operates with hard real-time scheduling with a soft real-time delivery schedule for the control/configuration plane operations. Such a design may use a pipeline in the data plane with a hub and spoke arrangement for the control/configuration plane.

5.3 Thread design

Each XCORE thread of processing has an entry point function. This function consists of a series of initialisation statements followed by an event-processing loop. Often, the loop has no exit condition, and it runs until something resets the processor or removes power from it. However, the loop may include an exit condition if the system design calls for a graceful shutdown.

Within the event-processing loop, the thread waits for the occurance of one or more events. The occurance of an event triggers the operation of a corresponding event handler. Only one event handler runs at a time, and it runs to completion. The event handler receives data associated with the event and processes it. Usually, it either stores the result for further processing after a subsequent event, possibly of another type, or forwards the result on to another thread. When the event handler completes, control returns to the `wait` operation, and the thread waits for the next event.

The XC `select` statement and the `SELECT_RES` macro in lib_xcore use a `wait` operation to wait for one or more events and to select the event handler to run. The `SELECT_RES` macro also includes the event-processing loop without an exit condition.

Warning

XC supports an `[[ordered]]` `select` statement and lib_xcore includes a `SELECT_RES_ORDERED` macro. Threads that use them cannot be combined together without modification to replace the ordered operation with an unordered one. Likewise, XC supports a `default` case in the `select` statement and lib_xcore includes the `DEFAULT_THEN`, `DEFAULT_GUARD_THEN`, and `DEFAULT_NGUARD_THEN` macros. Threads that use them cannot be combined together without removing the default case.

Typically, XCORE threads operate with interrupts disabled. Doing so allows each event handler to operate with highly deterministic timing.

Deciding how many events a thread should handle involves consideration of a few aspects.

Processing events in the same thread allows the sharing of variables between the event handlers without any risk of simultaneous access. Placing event handlers that closely co-operate with one another through shared variables into the same thread generally makes the design easier to understand, and it speeds up overall system performance.

Adding an event handler to a thread increases the worst-case time between the occurance of an event and its processing through its event handler. When an event occurs, some other event handler in the thread may be running, and it will have to complete before the `wait` operation can select the next event handler to start. Usually, however, adding an event handler has little effect on timing because event handlers run very quickly in comparison to the frequency at which events occur.

Especially when programming for hard real-time delivery, keep closely cooperating event handlers together in a thread but do not add in unrelated event handlers in the first instance. After verifying that the real-time functionality operates as needed, analysis or validation may show that sufficient time exists to add in further event handlers.

Application note AN02050 - Extend I2S loopback application with DSP and AI provides an example of the construction of XCORE threads.

6 Data communication options

Passing the data through a channel between threads gives the designer maximum flexibility in placing each thread. A thread can run on any tile in the XCORE processor network.

If using frame-based transformations, the design can pass a pointer to a frame through a channel instead of passing all of the data. Certain restrictions come with passing a pointer, however:

- It limits the options for thread placement since each tile only has direct access to its own internal SRAM. No direct access exists to the internal SRAM of another tile.
- Likewise, only one tile has direct access to external LPDDR memory, if the design includes such memory. In addition, accesses to external LPDDR memory takes more time that accesses to internal SRAM.
- It opens up the possibility of memory corruption due to mismanagement of the frames. If writing in C or C++, the designer has the responsibility to ensure that multiple threads do not attempt to access the same memory location at the same time.

7 Placing threads of processing

7.1 Constraints when placing threads

Few hard-and-fast rules exist for placing threads of processing on a network of XCORE tiles.

A thread that directly interacts with a port (i.e., an interface peripheral of custom port handler) must be placed on the tile that contains the port.

Likewise, a thread that directly interacts with an internal port option must be placed on the tile that hosts the internal port. The datasheet for the XCORE device describes its internal port options. The XMOS web site has a link to a port map for each processor generation, and these port maps also list some information about internal port options.

Threads that use shared memory to pass data between them must reside on the same tile. Each tile has direct access only to its own internal SRAM. If the hardware design includes external LPDDR memory, only one tile on a two-tile package has access to it.

Likewise only the even numbered tile in a two-tile package has access to external Flash memory.

7.2 Thread placement guidelines

In addition to the few rules given above, some guidelines can be stated.

Limit the number of channels that cross a tile boundary. When transmitting data across a tile boundary, a channel requires temporary use of a communication link to the inter-tile switch. The switch has a limited number of communication links to each tile. The xCONNECT Architecture provides further details on the number of links available.

Without special configuration, if a design places *n* threads on a tile the xTIME Scheduler guarantees a minimum amount of processor bandwidth according to the formula:

$$ThreadBW = CoreFreq/n$$

where:

$$n = \begin{cases} 5 \text{ if } n \leq 5 \\ n \text{ if } 5 < n \end{cases}$$

The XCORE architecture supports two enhanced modes of operation: FAST mode and PRIORITY mode. These modes affect the scheduling of threads of processing. Application note AN02030: Improving IO response times using FAST or PRIORITY modes describes each mode and their effect on thread scheduling. Although this application note focuses on IO response time, the effects of using FAST mode or PRIORITY mode also apply to threads of processing that receive their input from other XCORE resources (e.g., a channel end).

7.3 Method of thread placement

In practice, we have not found a formulaic method of placing threads on the tiles of an XCORE processor network that works well in all cases. Three major factors affect thread placement:

- Thread memory usage,
- Thread bandwidth requirements, and
- Thread inter-connectivity.

Due to the constraints described above, it makes sense to place threads that directly interact with a port or with an alternative pin function or an internal port option before placing other threads. When making these decisions, bear in mind that the XCORE architecture provides fleibility on which ports perform which functions. It also provides some flexibility on the configuration of alternative pin functions and internal port options. If difficulties arise at this point, it may be possible to alter the placement of interface peripherals, custom port handlers, or threads that interact with an alternative pin function.

In placing the remaining threads, seek an arrangement that:

- Does not exhaust the memory on any tile,
- Does not place more than four threads using PRIORITY mode on any tile, and
- Limits the number of channels that cross a tile boundary.

8 Separating event handlers

In some cases, a thread of processing will not meet all of its delivery deadlines. When this situation happens, separating event handlers into multiple threads can add the necessary processing bandwidth to allow the design to work as intended.

8.1 Subsection 1

Sub section text

Sub-subsection 1a

Sub sub section text

Another subsubsection 1b

Sub sub section text

9 Combining threads of processing

At times, a design will have more threads of processing than the desired network of XCORE processors can support. When this situation occurs, it may become beneficial to combine two or more existing threads into a single one.

9.1 Opprotunities for thread combination

Combining threads has the benefit of reducing the number of threads in use. Often it may be possible to combine threads that handle aperiodic events with soft real-time delivery schedules without having a negative effect on system performance. In some cases, combining a thread with low performance requirements and a thread with higher performance requirements may also be possible. In this case the addition of the event handler for the low-performace thread must not extend the worse case response time for a high-performance event beyond an acceptable limit.

9.2 Manual thread combination

As described in *Thread design*, a thread of processing has an entry point function, and that function has the form of a series of initialisation statements followed by an event-processing loop. The event-processing loop generally includes a `wait` operation in the form of an XC `select` statement or the `SELECT_RES` macro for the C or C++ languages.

To manually combine two threads of processing:

1. Combine the initialisation statements from the old threads at the start of the entry point function for the combined thread; usually the order does not matter although it can aid clarity to include them as blocks one after the other.
2. Place each event handler from the old threads into the `select` statement (XC) or `SELECT_RES` macro (C or C++) for the combined thread; again the order doesn't matter.
3. If writing in XC, be sure that the `select` statement appears within a processing loop; if writing in C or C++, the `SELECT_RES` macro includes an infinite loop, but be sure to include a comma-separated list of `CASE_THEN`, `CASE_GUARD_THEN`, and `CASE_NGUARD_THEN` macros to associate each event with its handler.

10 Example application

10.1 Building the example

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `an03010` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd an03010
cd app_an03010
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software package. If any dependencies are missing, they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `app_an03010/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

10.2 Running the example

Prior to running the example for the first time, erase the QSPI Flash memory and then program it with the weights for the VNR model. From an XTC command prompt, run the following commands from the `an03010/app_an03010` directory:

```
xflash --target XK-EVK-XU316 --erase-all
xflash --target XK-EVK-XU316 --data ./src/vnr/model/weights.out
```

Flashing the weights only needs to occur once.

To run the application, from an XTC command prompt, use the following command in the `an03010/app_an03010` directory:

```
xrun ./bin/app_an03010.xe
```

Note

If the application reports `fast flash init err -1`, programming the QSPI Flash memory either didn't occur or did not complete successfully.

Alternatively, program the application into flash memory for standalone execution with:

```
xflash ./bin/app_an03010.xe
```

11 Further reading

- *XMOS* xCONNECT Architecture
  https://www.xmos.com/download/xCONNECT-Architecture%281.0%29.pdf
- *XMOS* XTC Tools Installation Guide
  https://www.xmos.com/xtc-install-guide
- *XMOS* XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation
- *XMOS* application build and dependency management system; *xcommon-cmake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest