X-ISA: Basic Cross-Connects

Diving into the Nitty-Gritty: Implementing a Basic Network Cross-Connect

A Peek Under the Hood: A Basic Network Cross-Connect using the X-Switch ISA

Ever wonder about the fundamental mechanisms that drive network switches? While configuring high-level networking might seem abstract, the core of packet forwarding often relies on low-level microcode (uCode) instructions. Today, we're providing a simplified look at this with an example based on our company's exposed X-Switch Instruction Set Architecture (X^{ISA}). We will explore a very basic use-case: a simple cross-connect between two ports.

First, let's define a simple cross-connect: forwarding incoming packets from an ingress port to a corresponding egress port without modifying the packet. The control plane program populates an **Ingress Port** lookup table that maps an ingress port to a specific egress queue (on the desired egress port), making the association configurable at run-time, rather than pre-determined.

To send a packet out of a given egress port, we need to direct it to one of the egress queues associated with that port. The X-Switch ports typically have multiple egress queues to facilitate packet prioritization. To determine the appropriate egress queue number for a packet received on a specific ingress port, a lookup mechanism is necessary. While we could build a custom table for this purpose (which we will explore later), we can leverage the special Ingress Port table that is inherently available on the device. In this table, the ingress port acts as the key, and the corresponding egress queue number will be the value. The control plane is responsible for populating this Ingress Port table with the desired egress queue number for each ingress port. Once a packet reaches the Match-Action Processor (MAP), the result of the lookup in the Ingress Port table will automatically be available in the MAP register **R1**.

The MAP microcode retrieves the egress queue number ID (QID) from the Ingress Port table result available in register R1. The QID indicates to the hardware where to send the packet.

We'll continue to explore the microcode in detail here.
Also, see the recently opened X^{ISA} for further Instruction details.

Examining the Microcode: A Simplified X^{ISA} Flow

Packet switching is processed via the X^{ISA}, first via the Parser, and then via the MAP.

Parser uCode

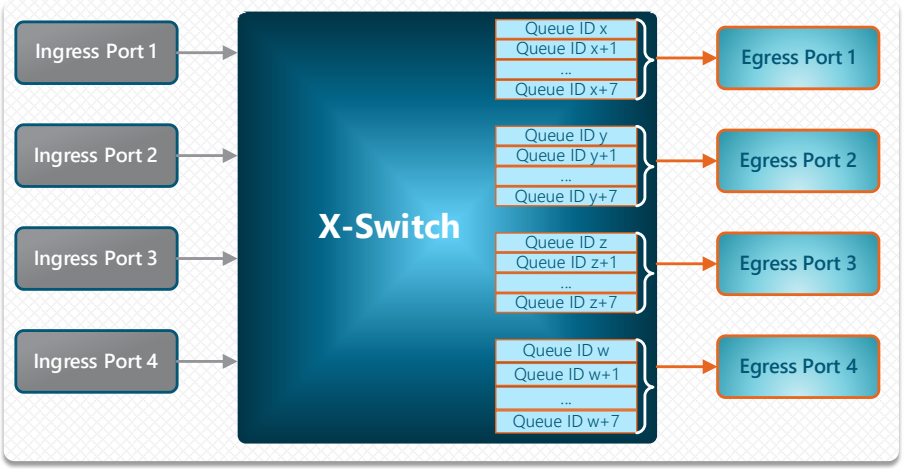
For a simple cross-connect example, the packet Parser processing is simply to HALT.

```
{
    special_entry_points
    ingress: entry_point_ethernet
}
entry_point_ethernet:
HALT
```

Here's a breakdown of these steps:

- **special_entry_points**: This section defines the parser's initial transition table, determining the starting point for packet processing based on the packet's arrival path (Ethernet, Host CPU or other). An ethernet entry point is specified in our example, and this directs all standard packets arriving through the ingress' path to begin processing at the `entry_point_ethernet` label.
- **entry_point_ethernet**: This label marks the beginning of the processing logic arriving at ingress.
- **HALT**: This Parser instruction immediately stops any further parsing of the packet header at this stage. No packet inspection is done at this point. And now the packet is ready for MAP processing.

Simple Uni-directional Cross-Connect Example



Ingress Port Table

Ingress Port Number	Egress Queue ID (QID)
1	w
2	x
3	y+1
4	z

MAP uCode

Now, let's look at the MAP uCode, which prepares information needed for sending out, using MAP Registers. Once the registers are prepped with the info, a send instruction ferries the packet descriptor to an egress queue.

Note: There are 14 MAP registers (`R0` to `R13`), each holding four 32-bit words (0-3), supporting big-endian architecture.
For example, `R1.3` specifies the fourth word of the `R1` MAP register, bits 0 to 31.

In the microcode example below, Lines 2 to 4 prepare the registers for the egress SENDOUT instruction in Line 5.

As discussed previously, the 16-bit QID field (i.e., part of the Ingress Port Entry result value) will have been preloaded into MAP Register R1 Word 0 (R1.0), at offset 16, by the HW accelerator.

The SENDOUT instruction uses two registers as its operands.

- The first operand must include the **QID** and **FrameDelta** fields - to be loaded into a MAP register at these locations:
  - **QID**: at offset 0, of the lowest Word (Word 3).
  - **FrameDelta**: at offset 0, of the second lowest Word (Word 2).
- The second operand includes additional header editing instructions (zero-ed out for our example purposes).

Lines 2 and 3 are used to prepare the SENDOUT's first operand, while Line 4 is used to prepare its second operand.

```
0   ingress:
1
2   MOVI R1.2, 0
3   CONCAT.CD R1.3, 0, R1.0, 16, 16
4   MOVI.CD RO.3, 0
5   SENDOUT.H R1, R0, 0
```

Here's a breakdown of these steps:

- **ingress**: This pre-defined label name determines the starting point for packet processing in the MAP, upon ingress (there may be other entry points).
- **MOVI R1.2, 0**: The program will expect to find the **FrameDelta** field info in Word 2 of the register, (in this case, register R1), later in the SENDOUT instruction. This is the number of bytes added to or removed from the header. Since, in our example, the program is not meant

to modify the header, we'll load 0 into that word by employing the Move Immediate (MOVI) instruction, indicating that no header has been added or removed.

- **CONCAT.CD R1.3, 0, R1.0, 16, 16**: This step is crucial for forwarding. It prepares packet fields for sending as follows:
  - The QID is currently located in `R1.0`, offset 16 (placed there by the HW accelerator).
  - The subsequent SENDOUT command will expect to find the QID in `R1.3` at offset 0.
  - As such, we're using the Concatenation instruction (`CONCAT`) with a clear destination extension (`.CD`) to clear R1.3 and then place the 16-bit QID field from R1.0 in R1.3.

  Further breaking down this step – using the instruction syntax:

`CONCAT`	`.CD`	`R1.3`	`0`	`R1.0`	`16`	`16`
Bitwise concatenation – being used here as "COPY".	Option set to **c**lear **d**estination register before placing data there.	Destination Register, Word 3.	Destination Register Offset.	Source Register, Word 0.	Source Register Offset.	Number of bits to copy from source register to destination register.

- **MOVI.CD R0.3, 0**: This instruction disables any header adjustments on the egress port clearing the entire register R0 (i.e. not just Word 3 of R0).
- **SENDOUT.H R1, R0, 0**: Finally, this instruction sends the packet to an egress queue and halts the processing for this packet. Note that here, the packet header is an implicit operand of the instruction.

  Further breaking down this step – using the instruction syntax:

`SENDOUT`	`.H`	`R1`	`R0`	`0`
Instruction to Send packet to egress queue associated with required egress port.	Option set to **h**alt MAP processing.	This register includes all the fields that describe the packet sending information. For this example only these fields are relevant: • ParamsReg[15:0]: **TargetQID** • ParamsReg[40:32]: **FrameDelta**	Additional header editing instructions.	Specifies the number of additional buffers needed.

Beyond the Simplicity: X^{ISA}'s Potential

While this cross-connect example demonstrates a fundamental operation in a clear and concise manner, X^{ISA} is capable of handling much more sophisticated networking tasks. Any standard routing and/or bridging feature, and more importantly, any other unique or customer-driven feature (just ask us 😊) can be implemented due to the X-Switch's extreme flexibility and programmability.

We, at Xsight Labs, believe this transparency will drive innovation and facilitate the development of future networking technologies.

Stay tuned for further insights into the capabilities of the X-Switch ISA and the exciting possibilities it unlocks.