In this exercise, you will be working with **NetASM**, nascent work on a network assembly language for programmable network devices (e.g., FPGAs, RMT, Intel's FlexPipe, NPUs). NetASM provides assembly instructions that directly reflect the capabilities of the underlying device, thus providing either a human programmer or compiler precise, fine-grained control over the device's resources. It exposes the details in the language such as creating tables and defining layouts of the processing pipeline.

The purpose of this exercise is to give you a glimpse into the future of hardware support for software-defined networking (SDN), where the data plane is no longer a fixed-function device, but rather a fully programmable device whose behavior is dictated by the programmer, with the ability to reconfigure it on-demand.

First, we will provide a brief overview of the NetASM language and its instruction set. We will then take you through the process of installing the NetASM assembler on your VM and testing it by running Hub and stateful MAC learner examples. After the overview, you will be asked to augment the stateful MAC learner example with an Access Control List (ACL) table to implement a Layer 2 firewall. We will provide more details on creating and submitting the code later in the instructions. As always, make sure that you follow each step carefully.

Overview

In this exercise, you will learn how to define new data-plane layouts, add custom state elements like tables and registers, and control how each packet is processed in the data plane. To do so, you will be learning and using **NetASM**, a new domain-specific language for configuring programmable data planes on a variety of targets.

NetASM is analogous to an x86 or MIPS-like instructions set. However, unlike updating main memory and registers, it defines the layout (i.e., topology and states) of the data plane. Topology defines how the packet is traversed through the data plane, and state refers to the type of memory element (i.e., register or table). These state elements have a well-defined data structure and type declaration in NetASM, which makes it easy to identify bugs early in the compilation process.

The NetASM instruction set has three types of instructions:

- **Initialization**: to create state elements (like tables and registers)
- **Topology**: to define how the packet is traversed and processed in the data plane
- **Control**: to provide an external control to populate the states (i.e., over OpenFlow or other interfaces)

The syntax of the NetASM language is defined in the `Core/Language.hs` file.

Setting up NetASM on your VM

NetASM is based on Haskell, so before continuing with the installation process make sure that the Haskell platform is setup on your VM.

Installing Haskell Platform:

Perform the following steps to install Haskell platform on your machine:

- Change to root directory

```
$ cd ~
```

- Install the haskell-platform package from the ubuntu install-base. This process may take some time (several minutes to tens of minutes) depending on the speed of your connection.

```
$ sudo apt-get install haskell-platform
```

Important: It's been reported that the above command is failing on most of the VMs. If this is happening with you, install haskell using the following command instead.

```
$ sudo apt-get install ghc6
```

- To test if the haskell platform is installed correctly, run ghci (a haskell interpreter)

```
$ ghci
```

- If installation is successful, you should see the following output:

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

- On the gchi terminal, type `:quit` to exit the ghci interpreter

```
Prelude> :quit
```

Once the Haskell platform is setup, you are now ready to install NetASM.

Installing NetASM:

Perform the following steps to install NetASM on your machine:

- Change to root directory

```
$ cd ~
```

- Clone the NetASM-haskell public repository

```
$ git clone https://github.com/NetASM/NetASM-haskell.git
```

- To test if NetASM is working on your VM, run the Hub example
    - Change to NetASM-haskell root directory

```
$ cd ~/NetASM-haskell
```

    - Run the Hub example

```
$ runhaskell Apps/Hub/Run.hs
```

    - If this test run is successful, you should see the following output:

```
[[("PID",(1,True)),("DRP",(0,True)),("inport",(1,True)),("outport",
(2147483646,True))]
[("PID",(1,True)),("DRP",(0,True)),("inport",(1,True)),("outport",(
2147483645,True))]]

"Instructions executed: 2"
```

Understanding NetASM with the Hub example

Now let's take a look at the Hub example in detail.

The table below shows the program for the Hub example written in NetASM.

```
module Apps.Hub.Code where


import Utils.Map
import Core.Language
import Core.PacketParser


----------
-- Hub ---
----------


-- Topology code for hub
c = [ OPF("outport", "inport", Xor, _1s)
    , HLT]
```

Table 1: Program for Hub example

Almost every program in NetASM, starts by importing three modules:

- **`Utils.Map:`** provides utility functions for operating on maps (i.e., [key, value] pairs)
- **`Core.Language:`** contains definitions of the syntax and semantics of the NetASM language
- **`Core.PacketParser:`** not used in this code, but provides helping functions for generating sample headers. *Note: in actual system, based on FPGA or any other network device, this header information will be passed on to NetASM from the packet parser.*

A code `c`, shown in the latter half of the program above, is an ordered list of NetASM topology instructions. The code `c` contains two instructions: `OPF` and `HLT`. The `OPF` instruction performs an `Xor` operation on the `inport` field with a vector of all `1s` and stores its result in the `outport` field of the header. The `Xor` operation sets all bits of the `outport` field (which is a bitmap) to `1s` except the bit for the incoming port, effectively performing a flood operation. A brief description and usage of `OPF` and `HLT` instructions are provided in the Appendix-A.

Running the Hub program:

To run the Hub program, we have written another program, `Run.hs`, which generates a series of input packet headers and executes the Hub code on them in the order in which they arrived. The following table shows the body of the `Run.hs` program.

```
module Apps.Hub.Run where


import Utils.Map
import Core.Language
import Core.PacketParser
import Apps.Hub.Code
```

```
----------
-- Hub ---
----------

-- Test header (a.k.a packet) stream
h0 = genHdr([("inport", 1), ("outport", 0)])
h1 = genHdr([("inport", 2), ("outport", 0)])

-- Input sequence
is = [HDR(h0),
      HDR(h1)]

-- Emulate the code
emulateEx :: [Hdr]
emulateEx = emulate([], is, c)

-- Profile the code
profileEx :: String
profileEx = profile([], is, c)

-- Main
main = do
        putStrLn $ prettyPrint $ emulateEx
        putStrLn $               profileEx
```

Table 2: Program for testing Hub example

- As in the Hub program, the first thing we do in the `Run.hs` program is to import the three primary modules, this time also importing the Hub code.
- We use the `genHdr` function (defined in the `Core.PacketParser` module) to generate two headers `h0` and `h1`. The `genHdr` function takes a list of key value pairs (i.e., header description with default values) as input and returns a header object `Hdr` (as defined in the `Core.Language` module).
- We create a test input sequence `is`, containing the two headers, with `h0` being the first header to be executed.
- We then create two wrapper functions, `emulateEx` and `profileEX`, for emulating and profiling the Hub code, respectively. These functions take *initialization code* (an empty list in this case), input sequence, and the program code as arguments. The `emulate` and `profile` functions are defined in the `Core.Language` module.

- Finally, `main` executes these two functions.

To test the Hub program:

- Change to NetASM-haskell root directory

```
$ cd ~/NetASM-haskell
```

- Run the program

```
$ runhaskell Apps/Hub/Run.hs
```

Analysing the output of the Hub program:

```
[[("PID",(1,True)),("DRP",(0,True)),("inport",(1,True)),("outport",(
2147483646,True))]
[("PID",(1,True)),("DRP",(0,True)),("inport",(1,True)),("outport",(2
147483645,True))]]

"Instructions executed: 2"
```

Table 3: Output of the Hub program

The output is an updated list of headers, h0 and h1, shown in red and green. The Hub code sets the `outport` field (bitmap) to all ones except the incoming port[1], effectively, flooding the packet on all ports except the incoming one. Also, notice that there are two extra fields in the header: `PID` and `DRP`. These are special fields and can't be modified by the programmer, directly. The `PID` field contains the ID of the current instruction that is processing the header. The `DRP` field indicates whether the packet is to be dropped or not. The last line shows the number of assembly instructions executed during this run.

Understanding NetASM with the stateful MAC learner example

Now let's take a look at a relatively more complex example, a stateful MAC learning switch. ("Stateful" indicates that the state elements (like table and registers) are updated locally by the switch without the intervention of the remote controller.) The table below shows the program for the stateful MAC learning example written in NetASM.

---

[1] The `outport` field is a 31-bit wide bitmap, where each bit corresponds to an egress interface. If all bits are set to 1 then, in hex, it's shown as: `7FFFFFFF`, meaning send packets to all egress interfaces. Now, in the case of flood, when the packet comes in from the first input port then the flood value for `outport` field will be: `7FFFFFFE` i.e., last bit set to 0 which in decimal notation is shown as `2147483646`.

```
module Apps.StatefulMACLearner.Code where

import Utils.Map
import Core.Language
import Core.PacketParser

---------------------------
-- Stateful MAC Learner ---
---------------------------

-- Table size
t_s = 10

-- Match table specs with default values
mt_s = t_s
mt_t = Dynamic("mt0", (mt_s, ["dstmac"]))
mt_v = Static([[("dstmac", 0)] | x <- [1..mt_s]])
mt_p = [("dstmac", "srcmac")]

-- Modify table specs with default values
md_s = t_s
md_t = Dynamic("md0", (md_s, ["outport"]))
md_v = Static([[("outport", 0)] | x <- [1..md_s]])
md_p = [("outport", "inport")]

-- Initialisation code for MAC learning
ic = [MKT(mt_t, mt_v)
    , MKT(md_t, md_v)
    , MKR("r", 0)]

-- Topology code for MAC learning
tc = [IBRTF(mt_t, "i", "l_miss")
    , LDFTF(md_t, "i")
    , JMP("l_end")
    , LBL("l_miss")
    , LDTFR(mt_t, mt_p, "r")
    , LDTFR(md_t, md_p, "r")
    , OPF("outport", "inport", Xor, _1s)
    , OPR("r", "r", Add, 1)
    , BRR("r", Lt, t_s, "l_end")
    , LDR("r", 0)
    , LBL("l_end")
    , HLT]
```

Table 4: Stateful MAC learning example

Recall that in a MAC learning switch, the destination MAC address of the incoming packet header is matched against a `MAC` table that is populated with destination MAC addresses corresponding to any source MAC address that the switch has already seen. If the table contains the destination MAC address for the packet, the switch updates the packet header with the corresponding output port from the `Output Port` table, otherwise, the `MAC` and `Output Port` tables are updated with the source MAC address and input port of the incoming header, respectively, and the packet is flood on all output ports except the incoming port.

Table 4 describes a switch layout that implements such a stateful MAC learner.

- First, we define two dynamic tables, each of size `10` with default values of `0`.
    - The first table, `mt_t`, maintains a history of source MAC addresses.
    - The second table, `md_t`, contains the corresponding output port for each MAC address in the `mt_t` table.
- We use the initialization instructions (e.g., `MKT` and `MKR`) to create the two tables and a register `r`. The code is defined as a sequence of these instructions in the list, `ic`. *Note: the order of these initialization instructions does not affect the final layout.*
- The MAC learning behavior (or layout) is defined by the code in list, `tc`, using the topology instructions. *Note: here the order instruction will affect the behavior and the final layout.*

<u>Step-by-step explanation of the topology code in `tc`:</u>

Here's a step-by-step explanation for each instruction in `tc`: (A brief description of these instructions is provided in Appendix-A)

- **`IBRTF(mt_t, "i", "l_miss")`**
    - We start the layout by doing a match on the `mt_t` table using `IBRTF`, branch instruction. If the destination MAC address is found in the table, set the index field `i` in the header with the matched index and move to the next instruction, else, jump to the label `l_miss`, indicating that the packet is not found in the table.
- **`LDFTF(md_t, "i")`**

    - The `LDFTF`, load instruction, is used to modify the output port field of the header with the one in the table, `md_t`, at index `i`.
- `JMP("l_end")`
    - The `JMP`, jump instruction, jumps to the label `l_end` signifying the end of the code.
- `LBL("l_miss")`
    - We create a new label `l_miss` using the `LBL` instruction.
- `LDTFR(mt_t, mt_p, "r")`
    - The `LDTFR`, load instruction, loads the `mt_t` table with the source MAC address of the incoming header at the index value stored in register `r`.
- `LDTFR(md_t, md_p, "r")`
    - This `LDTFR`, load instruction, loads the `md_t` table with the input port of the incoming header at the index value stored in register `r`.
- `OPF("outport", "inport", Xor, _1s)`
    - The `OPF`, operate instruction, sets the output port field in the header (which is a bitmap) to all 1s except the bit for the port specified in the input port field (which is a one-hot bit vector). Each bit in the output port field specifies a given output interface. The bitmap format is used to enable actions like flood and multicast.
- `OPR("r", "r", Add, 1)`
    - This increments the register `r`.
- `BRR("r", Lt, t_s, "l_end")`
    - The `BRR`, branch instruction, checks if the index value stored in register `r` is less than the table size (`t_s`). If yes, then jump to the label `l_end`, otherwise, move on to the next instruction.
- `LDR("r", 0)`
    - This load the register `r` with value `0`.
- `LBL("l_end")`
    - This creates a label `l_end`.
- `HLT`
    - The HLT, halt instructions, indicates the end of code.

<u>Running the MAC learner program:</u>

To run the MAC learner program, like in the case of Hub example, we have written another program, `Run.hs`, that generates a series of input packet headers and executes the MAC learner code on them in the sequence of their arrival. The following table shows the body of the `Run.hs` program.

```
module Apps.StatefulMACLearner.Run where


import Utils.Map
import Core.Language
import Core.PacketParser
import Apps.StatefulMACLearner.Code

---------------------------
-- Stateful MAC Learner ---
---------------------------

-- Test header (a.k.a. packet) stream
h0 = genHdr([("inport", 1),    ("outport", 0)
            ,("srcmac", 1234), ("dstmac",  4321), ("i", 0)])

h1 = genHdr([("inport", 3),    ("outport", 0)
            ,("srcmac", 6543), ("dstmac",  5432), ("i", 0)])

h2 = genHdr([("inport", 4),    ("outport", 0)
            ,("srcmac", 4321), ("dstmac",  1234), ("i", 0)])

h3 = genHdr([("inport", 2),    ("outport", 0)
            ,("srcmac", 5432), ("dstmac",  6543), ("i", 0)])

-- Input sequence
is = [HDR(h0), HDR(h1), HDR(h2), HDR(h3)]

-- Emulate the code
emulateEx :: [Hdr]
emulateEx = emulate(ic, is, tc)

-- Profile the code
profileEx :: String
profileEx = profile(ic, is, tc)

-- main
main = do
        putStrLn $ prettyPrint $ emulateEx
        putStrLn $               profileEx
```

Table 5: Program for testing MAC learner example

- The `genHdr` function generates the four headers `h0`, `h1`, `h2`, and `h3`.
- We create a test input sequence, `is`, containing the four headers with `h0` being the first header to be executed.
- The wrapper functions, `emulateEx` and `profileEx`, emulate and profile the code.
- In the end, the `main` executes these two functions.

To test the MAC learner program:

- Change to the NetASM-haskell root directory

```
$ cd ~/NetASM-haskell
```

- Run the program

```
$ runhaskell Apps/StatefulMACLearner/Run.hs
```

Analyzing the output of the MAC learner program:

```
[[("PID",(11,True)),("DRP",(0,True)),("inport",(1,True)),("outport"
,(2147483646,True)),("srcmac",(1234,True)),("dstmac",(4321,True)),(
"i",(0,True))]
[("PID",(11,True)),("DRP",(0,True)),("inport",(3,True)),("outport",
(2147483644,True)),("srcmac",(6543,True)),("dstmac",(5432,True)),("
i",(0,True))]
[("PID",(11,True)),("DRP",(0,True)),("inport",(4,True)),("outport",
(1,True)),("srcmac",(4321,True)),("dstmac",(1234,True)),("i",(0,Tru
e))]
[("PID",(11,True)),("DRP",(0,True)),("inport",(2,True)),("outport",
(3,True)),("srcmac",(5432,True)),("dstmac",(6543,True)),("i",(1,Tru
e))]]

"Instructions executed: 18"
```

Table 6: Output of the MAC learner program

The output consists of a list of four updated headers, shown in red (`h0`), yellow (`h1`), green (`h2`), and blue (`h3`).

- When `h0` arrives, it's the first time the packet is seen by the switch, so, it updates the `MAC` and `Output Port` tables with the source MAC (`1234`) and input port (`1`), respectively, and floods the packet.
- Similarly for `h1`, it sees the packet for the first time and updates the tables with the source MAC (`6543`) and input port (`3`), and floods the packet.

- Now for `h2`, it's a reply packet for `h0` and thus finds a match in the `MAC` table and is forwarded to output port (`1`).
- Similarly for `h3`, it's a reply packet for `h1` and thus finds a match in the `MAC` table and is forwarded to output port (`3`).

The last line shows the number of assembly instructions executed during this run.

Assignment

What's different from the previous assignments

In previous assignments on POX and Pyretic, you wrote a layer-2 firewall program for the controller only, which was then compiled to OpenFlow rules and installed on the Open vSwitches in Mininet. In those assignments, you had a fixed-function switch (i.e., Open vSwitch) with match+action stages, and you could not change the internals of that switch.

By contrast, in this assignment, using NetASM, you will change the layout of an existing switch i.e., stateful MAC learner, and will augment the layout by adding a *stateless* layer 2 firewall stage that sits after the MAC learner and enables communication only for the source and destination MAC pair for which there is an entry in the Access Control List (ACL). *By stateless, we mean that the state elements (like table and registers) are updated by the remote controller and not the switch itself. This is a perfect example showcasing a combination of both stateful and stateless operations.*

To implement this stateless firewall, you will need to perform the following tasks:

- Create a new ACL table using the `MKT` instruction and update the initialization code.
- Update the topology code using the topology instructions `BRTF` and `DRP`, such that only communication between the MAC pair entries listed in the ACL table are allowed.
- Write firewall rules in the ACL table with allowed MAC pairs, using the control instruction `WRT`.

Understanding the Code

To start this exercise, download netasm-assignment.zip and save it under the NetASM-haskell/Apps folder:

- Change to ~/NetASM-haskell/Apps folder

```
$ cd ~/NetASM-haskell/Apps
```

- Download the assignment

```
$ wget -O ACL_Assignment_.zip
https://d396qusza40orc.cloudfront.net/sdn/srcs/netasm-assignment.zip
```

- Unzip the assignment

```
$ unzip ACL_Assignment_.zip
```

Note: if the `unzip` utility is not installed on your VM, run the following to install it:

```
$ sudo apt-get install unzip
```

The assignment, by default implements a stateful MAC learner. To test it:

- Change to NetASM-haskell root directory

```
$ cd ~/NetASM-haskell
```

- Run the assignment

```
$ runhaskell Apps/ACL_Assignment_/Run.hs
```

The assignment comes with two files:

- `Code.hs`: containing the actual program
- `Run.hs`: for testing the program

In `Code.hs`, you have to complete the following TODO sections:

File (Line no.)	**Comment**
Code.hs (69)	Specify a dynamic table for implementing access control. The table should have the following specs: ● Two columns for holding source and destination MAC pairs i.e., `srcmac` and `dstmac` ● A table of size `5` (Hint: see the **“Specify a table”** example under the section on *Useful NetASM Instructions/Types*, below.)
Code.hs (86)	Create the ACL table, as defined above, using the `MKT` instruction with default values of zero (`0`). (Hint: see the **“Create a table”** example under the section on *Useful NetASM Instructions/Types*, below.)
Code.hs (108)	Add assembly code for implementing access control using `BRTF` and `DRP`. ● Use the `BRTF` instruction to compare the header with the ACL table ● Use the `DRP` instruction to tag the header as dropped

	(Hint: see the section on *Useful NetASM Instructions/Types* for their usage.)

In `Run.hs`, complete the following TODO sections:

File (Line no.)	**Comment**
Run.hs (80)	Write rules in the ACL table using the `WRT` instruction. Create `WRT` instructions for the following rules: ● Write `srcmac=6543` and `dstmac=5432` at index 0 ● Write `srcmac=4321` and `dstmac=1234` at index 1 (Hint: see the **"Write to table"** example under the section on *Useful NetASM Instructions/Types*, below.)

Run.hs (90)	Uncomment line 90 and 91 by removing `--`. In other words, replace the following:
	
	`--   CTRL(c0)`
	`-- , CTRL(c1),`
	
	with:
	
	`  CTRL(c0)`
	`, CTRL(c1),`

Useful NetASM Instructions/Types for Completing the Assignment

- Drop:
    - Syntax: `DRP`
    - Type: Topology instruction
    - Description: Marks the packet for drop.
    - Example:

```
tc = [ ...
       DRP
       ... ]
```

- Specify a table:
    - Syntax: `tbl = Dynamic(table_name, (table_size, [Fields]))`

- Description: declares a new dynamic table with the give name, size and list filed a.k.a., columns of the table
- Example: Define a table `t` with following attribtues:
  - A column for holding source IP and give it a name `srcip`
  - Table size for holding `10` rules

```
t = Dynamic("t", (10, ["srcip"]))
```

- Create a table:
  - Syntax: `MKT (Tbl, Tbl)`
  - Type: Initialization instruction
  - Description: It takes two arguments (a dynamic table specification and static table with default values) and creates a new table.
  - Example: Create the table t, defined above, with default values of zero (`0`).

```
v = Static([[("srcip", 0)] | x <- [1..10]])
ic = [ ...
       MKT(t, v)
       ... ]
```

- Branch on table:
  - Syntax: `BRTF (Tbl, Fld, Lbl)`
  - Type: Topology instruction
  - Description: Branch to a label if the header matches with the contents of the table and set the given header field with the matched index, else, move to next instruction. Note, this instruction is the inverse of `IBRTF` instruction used in the stateful MAC learner example, above. In `IBRTF`, the jump to label happens when the header doesn't match the contents of the table.
  - Example: Branch on the table t, created above, and jump to label ("end") setting the header field `"i"` with the matched index.

```
tc = [ ...
       BRTF(t, "i", "end")
       ...
       LBL("end")
       ... ]
```

- Write to table:
  - Syntax: `WRT (Tbl, Ptrn, Val)`
  - Type: Control Instruction
  - Description: Write the table with pattern at index value.

- Example: In table t, created above, write a source IP value of 1234 at index 5.

```
WRT(t, [("srcip", 1234)], 5)
```

Testing your Code

Once you have updated the `Code.hs` and `Run.hs` file:

- Change to NetASM-haskell root directory

```
$ cd ~/NetASM-haskell
```

- Run the assignment

```
$ runhaskell Apps/ACL_Assignment_/Run.hs
```

Submitting your Code

Save the output of the above run in a text file and upload it on `coursera` using the `submit` button for Module x under the Week 5 section on the Programming Assignment page.

Once uploaded, it will prompt the results on the new page (either as passed or failed).

APPENDIX-A - List of NetASM Instruction Set

1. Initialization Instructions

Instruction	Description	Usage
`MKR (Reg, Val)`	Make Register: make a new register (r) with default value (v)	MKR("r0", 0): make a register r0 with default value of 0
`MKT (Tbl`[2]`, Tbl`[3]`)`	Make Table: Make a new dynamic table (t) and load it with content from static table (t0).	MKT(t0, v0): make a new dynamic table t0 and load it with defaults values v0. t0 and v0 are defined using Dynamic and Static table types. See the MAC learner code, above, for an example usage.

[2] `Tbl = Dynamic (String, (Int, [Fld]))`
[3] `Tbl = Static ([Ptrn])`

2. Topology Instructions

Instruction	Description	Usage
`HLT`	Halt: Indicates the end of topology code	see Hub example above
`OPF (Fld, Fld, Op[4], Val)`	f0 = f1 op v: apply operation (op) on field (f1) and value (v) and store it in field (f0)	OPF("outport", "inport", Add, 1): Add 1 to inport and store it in outport field
`OPR (Reg, Reg, Op, Val)`	r0 = r1 op v: apply operation (op) on register (r1) and value (v) and store it in register (r0)	OPF("r0", "r1", Add, 1): Add 1 to r1 and store it in r0 register
`LDR (Reg, Val)`	Load Register: load register (r) with value (v)	LDR("r0", 10): Loads registers r0 with value 10
`LBL (Lbl)`	Label: label (l) for jump and branch instructions	LBL("l0"): Creates a label l0
`JMP (Lbl)`	Jump: jump control to label (l)	JMP("l0"): Skips the instructions till the label
`BRR (Reg, CmpOp[5], Val, Lbl)`	if (r op v) -> l: branch control to label (l) if the result of the comparison on register (r) and value (v) is true	BRR("r0", Gt, 10, "l0"): Jump to label l0, if r0 is greater than 10
`IBRTF (Tbl, Fld, Lbl)`	Branch control to label (l) if any pattern in table (t) is not present in the header, else, move to the next instruction and set the field (f) to matched index	IBRTF(t0, "i", "l0"): Jump to label l0 if any pattern in table t0 does not match the header, else, move to the next instruction and set the field "i" to matched index
`LDFTF (Tbl, Fld)`	Load header with table at field: load header with the table (t) at index field (f)	LDFTF(t0, "i"): Load header with contents of table t0 at index value in field "i".
`LDFTR (Tbl, Reg)`	Load header with table at register: load header with the table (t) at index register (r)	LDFTF(t0, "r0"): Load header with contents of table t0 at index value in register "r0".

3. Control Instructions

[4] `Op = Add | Sub | And | Or | Xor`

[5] `CmpOp = Eq | Neq | Lt | Gt | Le | Ge`

Instruction	Description	Usage
WRR (Reg, Val)	Write Register: write register (r) with value (v)	WRR(“r0”, 10): Write register r0 with default value of 10
WRT (Tbl, Ptrn, Val)	Write table with pattern [(f,v)] at index value: write table (t) with pattern (p) at index value (v).	WRT(t0, [("inport", 1)], 1): Write the pattern, inport=1, in table t0 at index value 1

Learning Haskell: http://learnyouahaskell.com/