统一流状态管理方案设计

# 引言

有状态的网络功能（如有状态防火墙、有状态NAT），需要记录流的状态信息，并根据该信息处理报文。但是，传统的网络功能往往各自维护流的状态信息，增大了网络功能开发的难度。

我们的目标是设计可重用的统一流状态管理（UniMon，unified state monitor）为有状态网络功能提供统一的流状态维护功能，以简化网络功能开发和避免网络功能间的冗余开发。因此，基于UniMon开发的网络功能只需要调用UinMon提供的编程接口（UniMon API）即可获取流的状态信息。另外，为了满足网络（如数据中心）对报文处理性能的需求（高吞吐率、低处理延时），我们将部分流状态的管理（如TCP连接状态）卸载到FPGA上，以加速流状态的管理。如图1所示，我们提出基于FPGA+CPU架构实现的统一流状态管理模型，UniMon，利用FPGA维护流的基本状态，而将完整流状态管理放在软件（CPU）实现。UniMon既能保证流状态类型的多样性，以支持复杂的有状态处理需求（如Web注入攻击检测）；同时能够支持只需基本状态即可完成报文处理的网络功能（如有状态防火墙）在FPGA上的卸载，从而获得加速。



图1 UniMon整体视图

本文档描述了基于FAST开源平台实现UniMon的设计方案。目前，我们还仅支持TCP的流状态管理，其他协议的流状态管理功能计划在下一版本中得到支持。获取更多的相关文档和源码，请访问<https://github.com/JunnanLi/UniMon>。

# UniMon模型

本节首先介绍基于FAST开源平台实现UniMon模型（2.1节），然后定义UniMon为网络功能提供的编程接口（2.2节），接着介绍UniMon所提供的事件类型（2.3节），最后介绍UniMon内部部件之间的接口定义（2.4节）。

## 基于FAST开源平台实现UniMon模型

统一流状态模型（UniMon）的基本功能是提供TCP流的状态管理。如图1所示，为了加速流状态的管理，我们将流的基本状态维护功能卸载到FPGA上实现，而流的建立、删除功能，以及流的完整状态维护保留在软件中实现。另外，为了给上层网络功能提供良好定义的开发接口，我们采用事件—处理函数（event-handler）编程模型，即开发者只需基于UniMon提供的多种类型的事件（例如TCP连接建立请求）编写报文处理函数。而当事件被触发时，UniMon会调用开发者所编写的报文处理函数，以实现相应的处理功能。为此，我们在UinMon的流状态管理部件之上增加事件管理部件，用于实现event与handler映射关系的维护，和调用handler函数处理报文的功能。

我们基于FAST开源平台实现UniMon模型（获取FAST开源平台的相关内容，请访问www.fastswitch.org）。如图2所示，我们将UniMon的功能部件和基于UniMon开发的网络功能部署在FAST的UDA和UA中。其中，在UDA中部署的功能模块包括TCP流的基本状态管理部件（Essential State Manager, ESM）、报文处理模块（网络功能卸载）；在UA中部署的功能模块包括TCP流的完整状态管理（Completed State Manager, CSM）、事件管理（Event Manager，EM），和网络功能（如有状态防火墙、有状态NAT）。值得注意的是，FPGA中实现的网络功能从属于软件实现的网络功能，是软件网络功能的卸载，因为我们发现部分网络功能可以将部分网络功能放在软件，而将另一部分功能放在FPGA中实现，以获得加速。例如，有状态NAT，可以将port的动态分配放在软件实现，而将ip到port的地址转化功能放在硬件实现。为实现软硬件功能模块的信息交互，我们为软件网络功能提供基于FAST的寄存器读写函数，以实现FPGA网络功能中规则表的配置（如配置ip到port的映射）。



图2 基于FAST开发平台实现UniMon原型

## UniMon为网络功能提供的开发接口

UniMon为网络功能提供的编程接口分为两类，即UniMon软件编程接口和硬件编程接口，如图1所示。其中，硬件编程接口相对简单，采用硬件UDA间通信的信号定义（Metadata），将TCP流的状态信息，以及触发的事件类型（采用BitVector表示）填写在自定义Metadata字段中（即metadata[1]），详细FAST UDA开发模式请参考《FAST开发手册》。软件编程接口所包含的接口函数较多，包括事件—处理函数注册接口函数，流状态读取接口函数。

* + 1. 硬件编程接口：

TCP连接的基本状态管理（ESM）与网络功能均基于UDA实现，之间采用metadata实现通信，我们将ESM产生的中间信息，包括标准协议类型PST，流唯一标识FlowID，TCP流状态state，触发的事件类型event bit，上报的事件类型pkt\_in\_cnd，最后活跃报文的时间戳last tmp，填写在metadata[1]中，其格式如图3所示。



图3 Metadata[1]定义

* + 1. 软件编程接口：

UniMon为软件实现的网络功能提供两类接口函数，即事件—处理函数注册接口函数，流状态读取、配置接口函数。前者用于注册事件对应的处理函数，从而能够在事件被触发是调用相应的处理函数。后者实现流状态的访问功能，目前仅支持访问当前所触发事件的流状态信息。两种接口函数如下表所示，值得注意的是，在注册事件—处理函数映射关系之前，需要初始化该网络功能的序号（即网络功能服务链中位置），以确认网络功能调用的顺序。

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 类型 | 函数名称 | 含义 |
| 1 | 事件注册 | int unimon\_initial\_networkFunction\_order(int networkFunction\_id, int networkFunction\_order) | 定义（注册）网络功能的在服务链中的序号；  id唯一标识网络功能，但网络功能的order可以相同；  成功返回1，失败返回0； |
| 2 | int unimon\_register\_callback(event\_t ev, int hook\_side, callback\_t cb) | 注册事件处理函数；  event\_t是指event id，唯一标识unimon提供的事件类型，参考2.3节；  hook\_side指处理函数所绑定的位置（  即客户端为HOOK\_CLI，服务器端为SER，  两端都绑定为BOTH），callback\_t是处理函数名（指针）；  成功返回1，失败返回0； |
| 3 | 获得流状态 | connection\_t unimon\_get\_currentFlowState() | 获取当前所触发事件的流状态信息；  connection\_t定义见3.2.2节 |

函数的使用方式，请参考附录1中的有状态防火墙编程示例。

## UniMon为上层网络功能提供的事件类型

我们采用事件—处理函数（event-handler）编程模型，即开发者只需基于UniMon提供的多种类型的事件（例如TCP连接建立请求）编写报文处理函数。而当事件被触发时，UniMon会调用开发者所编写的报文处理函数，以实现相应的处理功能。

UniMon提供的事件类型如下：

|  |  |  |
| --- | --- | --- |
| 事件类型 | | 含义 |
| 1 | UINMON\_ON\_PKT\_IN | 接收到报文 |
| 2 | UINMON\_ON\_CONN\_START | 连接建立请求（syn） |
| 3 | UINMON\_ON\_CONN\_SETUP | 完成三次握手 |
| 4 | UINMON\_ON\_TIMEOUT | 流超时（长时间未到达新报文） |
| 5 | UINMON\_ON\_STATE\_CHANGE | 流状态发生转化 |
| 6 | UINMON\_ON\_CONN\_END | 完成4次结束握手，或流超时，或接收RST报文 |
|  |  |  |

值得注意的是，事件本身不区分触发实现的报文方向（客户端的报文或者服务器端报文），用户需要结合事件注册函数（unimon\_register\_callback(event\_t ev, int hook\_side, callback\_t cb)）中的hook\_side来绑定所需处理的报文方向。例如，有状态NAT，对客户端的报文需要将其源IP、Port替换，而对于服务器端的报文需要将其目的IP、Port替换。

## UniMon内部部件之间的接口定义

如图2所示，UniMon由三部分组成，即硬件的流状态管理（ESM）、软件的流状态管理（CSM）、软件的事件管理（EM）。本小节主要介绍各部件之间的接口定义。

* + 1. ESM与CSM之间的接口定义

在UniMon中，FPGA维护的TCP连接与软件所维护的相同，只是各自维护的连接状态不同，即FPGA负责流的基本状态维护；软件负责维护复杂的连接状态（具体维护的状态类型见各自模块设计中的关键数据结构部分）。另外，为保证软硬件中的连接数量一致，连接的创建、删除仅由UniMon的软件负责，而FPGA负责上报流状态变化（如上报超时连接）。

**目前，FPGA支持配置的上报条件与UinMon事件类型相对应**：即未命中的报文（连接请求）、每个报文、连接建立、连接超时、连接状态发生变化、连接关闭6种。CSM通过调用FAST提供的接收报文函数（即fast\_ua\_recv），获取FPGA上报的消息。消息采用metadata+packet的格式，位宽为128b，其中metadata[0]为平台定义的相关内容，metadata[1]为用户自定义字段，我们将上报的内容填写在metadata[1]中，其格式与UniMon的硬件编程接口相同，如图3所示。

此外，CSM负责TCP连接的增加删除功能。CSM还需调用FAST提供的寄存器读写接口（fast\_reg\_rd/wr）实现ESM中TCP连接表的配置。（其中的虚拟地址空间范围，请参考3.1节）

* + 1. CSM与EM之间的接口定义

CSM会根据接收的报文以及该报文所属流的状态，产生6种事件，并通告事件管理部件，同时携带流和报文信息。

CSM与EM的接口函数定义如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 交互对象 | 交互内容 | 函数名称 | 含义 |
| 与EM交互 | 触发基本事件 | int unimon\_raise\_event(event\_bitmap\_t evb, connection\_t conn) | 调用事件处理函数；event\_bitmap\_t为触发的事件类型，采用bitmap表示；  connnection\_t表示所命中的流状态信息（指针），定义见3.2.2节 |

# UniMon设计

本节主要介绍UniMon的设计，分三层介绍，即硬件的流状态管理层（3.1节）、软件的流状态管理层（3.2节）、软件的事件管理层（3.3节）。

## FPGA中的TCP连接管理部件（ESM）

本节主要介绍在FPGA中实现的TCP连接管理部件的整体框架（3.1.1节），负责接收数据报文，并根据接收的报文维护基本的TCP连接状态（3.1.2小节），产生6类基本事件，供后续UDA使用。然后，介绍子模块间的连接关系和接口设计（3.1.3小节）。最后，介绍TCP连接表所采用的2-left hash和超时判断逻辑（3.1.4小节）。

### TCP连接管理部件（FPGA部分）整体设计

TCP连接管理部件（FPGA）的整体设计如图4所示，主要包含5部分：

|  |  |
| --- | --- |
| 模块名称 | 功能 |
| Connection table（双端口RAM） | 用于保存流连接信息 |
| Connection searcher | 实现连接表的查找、更新功能（包含hash查找功能），产生packet\_in消息（报文触发） |
| Build-in event generator | 产生基本事件类型（采用bitmap表示），需要汇聚报文触发的基本事件以及超时模块触发的超时事件 |
| Connection table configuration | 实现连接表的配置、读取、删除操作，供软件使用 |
| Connection out-time inspector | 判断连接是否超时，产生packet\_in消息（超时触发） |



图4 ESM部件设计

值得注意的是，ESM部件的输入为metadata+pkt（134b），输出为metadata+pkt（134b），如果metadata中无法提供4-tuple以及tcp flag信息，则ESM解析获得。

（这个图没有改成中文的原因是，代码中的模块名采用英文，方便读者一一对应）。

* + 1. 数据结构

ESM部件的数据结构包括连接表（connection table，在代码中缩写为connTb）、事件戳表（或者老化表，aging table，在代码中缩写为agingTb）。如下表所示，连接表包含8个成员，即flowID, flow\_key, state, packet\_in\_cnd, action, Next\_index, counter\_info, aging\_info。其中，flowID是表项在RAM中的索引值，因此无需显示存储。另外，aging\_info仅供硬件自身使用，不支持配置（软件不可见）。

FPGA目前支持的规则数量为512条（预留最大规则数64K），对应的虚拟地址空间为（0x3000\_0000-0x3000\_01ff）。其中[15:4]表示规则ID（flowID/Addr），[3:0]用于拼接完整表项（6个32b组成）。

|  |  |  |  |
| --- | --- | --- | --- |
| 连接表成员 | 子成员 | 在表项中的位置 | 含义 |
| Connection table（entry） | | | |
| Flow\_key | - | [19  9:104] | 流关键字信息（4元组，即src\_ip, dst\_ip, src\_port, dst\_port）具体存储顺序为bigger\_ip (ip\_1), smaller\_ip (ip\_2), port,\_1 prot\_2 |
| state | Reserved | [103:101] | 保留字段 |
| Dirction | [100] | 方向，0表示bigger\_ip为src\_ip，1表示bigger\_ip为dst\_ip，方便client/server state维护 |
| State\_cli | [99:98] | 端状态，三种：0表示closed，1表示requested，2表示established； |
| State\_ser | [97:96] | 端状态，三种：0表示closed，1表示requested，2表示established； |
| Packet\_in\_cnd | - | [95:88] | 采用bitmap表示报文上报的条件，其中前7b（从后往前）分别表示为every packet，new\_flow，conn\_setup，time\_out，state\_change, conn\_end，conn\_start，前1b目前保留 |
| Acton | Action\_cli | [87:84] | 采用bitmap表示处理动作，4’h1表示硬件转发，4’h2表示根据上报条件送软件；4’h4与4’h8保留。 |
| Action\_ser | [83:80] |
| Next\_index | - | [79:64] | 用于构造hash list； |
| Counter\_info | Pkt\_counter | [63:32] | 报文计数 |
| Byte\_counter | [31:0] | 字节计数 |
| Aging table（entry） | | | |
| Aging\_info | last packet timestamp | [7:0] | 最近一个报文的时间戳，加上超时时长即得超时时刻（单位10ms） |
| Aging\_tag | [8] | 超时标识（硬件专用，不支持配置） |

* + 1. 模块连接关系和接口信号设计

我们在3.1.1节中介绍了5个子模块功能，其连接关系和接口信号定义如图5所示。接口信号定义如下表。接口信号定义（列表）在附录2中展示。



图5 内部子模块的连接关系和接口信号定义

值得注意的是连接表在物理上有Connection RAM和Aging RAM组合，Connection RAM负责维护连接状态信息，Aging RAM由于保存最近报文命中事件，以及Aging tag。因此Connection table configuration和Connection out-time inspector可以同时读取两个RAM，但配置时，仍然以Connection table configuration优先。

* + 1. 子模块详细设计

5个模块中，Connection table采用双端口RAM实现，其中包含两个ram，一个用于保存connectino信息，一个用于保存aging信息，aging ram仅供硬件自身使用；Connection table configuration模块则实现Connection table（RAM）的读写功能；基本事件生成模块，用于汇聚报文触发的基本事件和超时触发的超时事件，起到了多路选择器的功能，设计相对简单。这里着重介绍以下两个模块的设计。

1. **查找模块**

**Hash查找算法设计**

采用2-left hash算法，如图8所示，首先将关键字输入两个hash函数获得两个hash值。然后分别查找两个hash表，并比较hash表中的简化的4元组信息（16b），能够获得目标表项所在的索引值（16b）。最后，搜索目索引值所指的hash链，获得匹配的连接表项。

虚拟地址空间为（0x3001\_0000-0x3002\_01ff）。采用的hash算法分别为[79:64]^ [31:16] 和[47:32]^[15:0]。其中hash表深度为9b，简化的4元组信息压缩算法为[79:64]^ [47:32]^[31:16] ^[15:0]。



图8 2-left hash示意

**TCP状态维护**

查找模块需要实现的另一个功能是，维护TCP连接状态，目前支持的状态类型为3种，即closed，requested，established，并分为client/server三部分。其状态切换关系，如下表所示。

|  |  |  |
| --- | --- | --- |
| 当前状态 | 接收到的报文 | 下一个状态 |
| Client\_state | | |
| closed | Syn报文 | requested |
| requested | 接收server回复的syn-ack，并收到client发送的ack报文 | Established |
| Established | 接收client发送的FIN报文和server回复的ACK报文，或者RST报文 | Closed |
| Server\_state | | |
| closed | 接收到server发送的Syn-ack报文 | requested |
| requested | 接收client回复的ack报文 | Established |
| Established | 接收server发送的FIN报文和client回复的ACK报文，或者RST报文 | Closed |

1. **超时模块设计**

双端口Aging RAM中的一个端口用于超时维护。超时维护主要轮询Aging table以发现超时连接。当发现表项超时，则标记aging\_tag位，后续匹配的报文都不再修改该连接状态，而是上报给软件。

## CPU中的事件管理模块（EM）

本节主要介绍事件管理部件的整体框架（3.3.1节），负责接收连接管理部件通告的基本事件类型（bitmap表示），并根据该信息查询event-handler树表（3.3.2小节），生成自定义事件，调用callback函数。然后，介绍子模块的详细介绍（3.3.3小节）。最后，编写了实现事件管理部件的伪代码（3.3.4小节）。

### 事件管理部件整体设计

事件管理部件的设计如图11所示，主要包含三部分：

|  |  |
| --- | --- |
| **模块名称** | **功能** |
| 事件解析模块 | 根据完整流状态管理CSM上传的信息解析出事件类型，根据事件类型查询event-handler表，调用处理函数执行模块来完成事件的处理。 |
| 处理函数执行模块 | 根据回调函数名和函数参数来调用处理回调函数执行。 |
| 事件/处理函数注册模块 | 将用户定义的事件处理函数注册到event-handler table，以便事件解析模块根据事件类型来查询事件处理函数。 |



图11 事件管理模块设计

### 主要数据结构

Event-handler table，数据结构如下:

|  |
| --- |
| /\*回调函数entry定义\*/  typedef struct \_callback\_entry{  int priority; //优先级  callback\_t callback ; //回调函数  callback\_entry \* next\_callback; //下一个回调函数entry  }callback\_entry ;  callback\_entry event\_list[64]; //声明一个64大小的数组，表示可以支持64种事件。 |

### 事件管理流程

1. 事件处理函数注册流程

注册事件的处理回调函数时，用户使用接口函数int unimon\_register\_callback( event\_t ev, callback\_t cb)，根据事件ID ev去查询event\_list数组，在事件ID对应的数组项的链表中根据优先级将该事件回调函数插入链表。处理流程如图12所示。



图12 事件处理回调函数注册流程

1. 事件处理流程

事件处理流程如图13所示。当收到事件信息后，分析事件bitmap，判断产生的事件类型，依次对事件进行处理。在处理事件时，首先根据事件ID找到对应的数组项，然后遍历事件链表，依次调用事件处理函数来执行。当所有事件都处理完毕后，接收下一个事件信息来进行处理。



图13 事件处理流程图

### 子模块详细设计

**1）事件解析模块:**

根据完整流状态管理模块CSM上传的事件bitmap信息，解析出该报文产生的事件类型，根据bitmap比特位事件的定义解析出事件。按照事件ID去查询事件/处理函数表，找到该事件对应的事件处理函数链表，依次获取事件列表中的事件处理函数，调用处理函数执行模块来完成事件的处理函数的处理。

1. **处理函数执行模块:**

根据处理函数的功能定义来执行。

1. **事件/处理函数注册模块:**

用户根据回调函数注册接口函数来完成事件/处理函数的注册。将事件的处理回调函数写入event-handler表中，供事件解析模块查询。

模块主要包含以下处理函数：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 头文件 | 函数名称 | 功能 |
| 用户接口 | unimon\_api.h | int unimon\_register\_callback( event\_t ev, callback\_t cb) | 注册事件回调函数。ev是事件ID，cb是回调函数。  返回1表示回调函数注册成功，返回0表示回调函数注册失败。 |
| 对TCP流管理软件端接口 | event\_raise.h | int unimon\_raise\_event(event\_bitmap\_t eb, connection\_t conn) | 调用事件处理函数。Eb是事件的bitmap，conn是连接entry。 |
| 内部函数 | event.h | callback\_t find\_handler(event\_t eventid) | 根据事件ID获取处理函数 |
| int add\_register\_callback(event\_t ev, callback\_entry \* cb\_entry) | cb\_entry是事件ev对应的一个回调函数entry |
| callback\_entry \* get\_register\_callback (event\_t ev) | 获取事件ev对应的回调函数列表. |

### 事件管理模块整体实现伪代码

|  |
| --- |
| /\*创建事件/处理映射表（由用户在main函数中调用）\*/  /\*在事件处理链表中添加事件的处理函数\*/  int fast\_register\_callback( event\_t *eventid*, callback\_t *cb*)；  /\*事件的执行\*/  /\*为流管理模块CSM提供的接口函数\*/  fast\_raise\_event(event\_bitmap\_t *evb*, connection\_t conn);{  /\* 获得底层传上来的事件 \*/  event\_t eventid = 0;  **for** *each\_bit* **in** *evb*:  eventid++;  **if** (*each\_bit*)  {  callback\_entry \* callback\_list = event\_list[eventid];  for each\_entry in callback\_list:  run callback function;  *}*  } |

# 附录1：基于UniMon编程示例

1. 有状态防火墙开发

* **有状态防火墙功能需求**

有状态的防火墙主要实现以下三个功能：

（1）防止tcp半连接攻击。

（2）内网可以访问外网，外网不能访问内网。

（3）窗口外的报文丢弃。

* **功能实现**

针对功能需求，分三个方面实现功能：

（1）分析半连接事件TCP\_HALF\_CONNECTION。在该事件的处理函数中对该事件的发生次数进行计数，在单位时间内发生的次数超过了阈值，则将该地址设置于黑名单，阻止其通过。

（2）分析OUT\_TO\_IN事件。在该事件的处理函数中设置标记，阻止报文通过。

（3）分析OUT\_WIN\_PKT事件。当产生OUT\_WIN\_PKT事件后，丢弃该报文。

* **事件类型**

实现有状态的防火墙需要产生以下3种事件：

|  |  |
| --- | --- |
| **事件类型** | **含义** |
| TCP\_HALF\_CONNECTION | 接收到syn报文并回复syn-ack，但未收到ack报文，同时连接超时 |
| OUT\_TO\_IN | 外网向内网发送连接请求 |
| OUT\_WIN\_PKT | 报文序列号超过接收窗口 |

* **应用整体处理流程**

首先，用户根据应用需求，对事件回调函数进行注册，详细逻辑见图12。接着初始化CSM，主要是FAST接口的初始化，初始化内容见CSM模块设计。初始化完毕后通过FAST接口函数来获取ESM模块上传的信息，信息是metadata和pkt。若有接收到信息，则通过CSM的连接管理模块，进行数据报文的连接管理，包括连接表的维护。然后，调用EM模块的事件信息注册接口函数注册事件信息。最后逐个处理事件，事件的处理详细逻辑如图13所示。处理完毕后接收下一个ESM模块上传的信息进行处理。



图13 用户应用处理流程

* **有状态防火墙伪代码**

|  |
| --- |
| ##main.c  /\*三个事件；  TCP\_HALF\_CONNECTION表示半连接超时事件；  OUT\_TO\_IN表示外网向内网发送请求事件；  OUT\_WIN\_PKT表示序列号超出接收窗口的事件\*/  #define TCP\_HALF\_CONNECTION 7  #define OUT\_TO\_IN 8  #define OUT\_WIN\_PKT 9  /\*事件处理函数定义\*/  /\*半连接处理函数\*/  int half\_conn\_handle()  {  drop packet;  Write log;  }  /\*外网访问内网事件处理函数\*/  int out\_to\_in\_handle()  {  drop packet;  Write log;  }  /\*超接收窗口事件处理函数\*/  int out\_win\_handle()  {  drop packet;  Write log;  }  main(){  /\*事件处理函数注册\*/  /\*半连接事件处理函数注册\*/  fast\_register\_callback(TCP\_HALF\_CONNECTION*, half\_conn\_handle*)；  /\*外网访问内网事件处理函数注册\*/  fast\_register\_callback( OUT\_TO\_IN*, out\_to\_in\_handle*)；  /\*超接收窗口事件处理函数注册\*/  fast\_register\_callback(OUT\_WIN\_PKT*, out\_win\_handle*)；  /\*调用软件流管理入口函数\*/  tcp\_manager();  }  ##tcpmanager.c  void tcp\_manager(){  fast\_init\_hw(0,0);  ua\_init();  fast\_ua\_recv();  }  int callback(struct fast\_packet \**pkt*, int *pkt\_len*)  {  对报文进行解析，产生事件bitmap和流的连接信息conn；  /\*调用事件管理的接口函数，处理事件\*/  unimon\_raise\_event(event\_bitmap\_t *evb*, connection\_t *conn*);  } |

1. 有状态NAT开发
2. …

# 附录2：ESM模块内部接口信号定义

（详细内容参考代码注释，<https://github.com/JunnanLi/UniMon>）

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 位宽 | 方向 | 注释 |
| Unimon\_top（顶层） | | | |
| clk | 1 | In | 时钟 |
| Reset | 1 | In | 复位，低有效 |
| Pkt\_in\_valid | 1 | In | 报文有效位，高有效 |
| Pkt\_in | 134 | In | Pkt\_in[133:132] 表示报文头、尾表示：  2’b01 is header;  2’b11 is body  2’b10 is tail  ***采用Metadata+pkt格式，即前两拍为metadata***  Metadata[0]（假设，还需修改）  [127:32] 为src&dstIP, src&dstPort；  [31:26] 为tcp flag； |
| Pkt\_out\_valid | 1 | Out |  |
| Pkt\_out | 134 | out | ***采用Metadata+pkt格式，即前两拍为metadata***  Metadata[1]  [127:120] 为pst;  [119:104] 为flowID  [103:96] 为state  [95:88] 为FPGA所触发的事件类型（BitVector）  [87:80] 为pkt\_in\_cnd/event bitmap;  [79:72] 为last timestamp  [71:0] 为reserved |
| Ctrl\_in\_valid | 1 | In | 控制信号有效位，高有效 |
| Ctrl\_opt | 2 | In | 操作类型：  2’d0为读；  2’d1为配置/添加  2’d2为删 |
| Ctrl\_addr | 32 | In | 虚拟地址 |
| Ctrl\_data\_in | 32 | In | 控制输入 |
| Ctrl\_out\_valid | 1 | Out | 控制输出有效位，高有效 |
| Ctrl\_data\_out | 32 | Out | 控制输出 |
| Connection\_searcher | | | |
| 信号名 | 位宽 | 方向 | 注释 |
| Reset | 1 | In | 时钟 |
| Clk | 1 | In | 复位，低有效 |
| Metadata\_in\_valid | 1 | In | 有效位，高有效 |
| Metadata\_in | 134 | In | Metadata信号，（pkt\_in别名） |
| Event\_bitmap\_valid | 1 | Out | 事件有效位，高有效 |
| Event\_bitmap | 10 | Out | 低位→高位 含义：  Every packet: 1  New\_flow: 2  Conn\_setup: 4  Time\_out: 8  State\_change: 16  Conn\_end: 32  Conn\_start: 64  Reverserd: 128  Pkt\_in: 256  Forward: 512 |
| pktIn\_info | 32 | Out | flowID(16b) + state(8b) + original event bitmap (8b) |
| hashV\_1 | 9 | Out | Hash表1的输入地址 |
| hashV\_2 | 9 | Out | Hash表2的输入地址 |
| hashV\_valid | 1 | Out | 读取有效位，高有效 |
| Ctx\_hashTb\_1 | 33 | In | Hash表内容，三部分：  Valid\_bit+simplified flow key (16b)+index(flowID) |
| Ctx\_hashTb\_2 | 33 | In | Hash表内容，三部分：  Valid\_bit+simplified flow key (16b)+index(flowID) |
| Idx\_connTb | 9 | Out | 连接表的输入地址 |
| Data\_connTb | 200 | Out | [199:104] 为4-tuple info  [103:101] 为reversed  [100] 为direction  [99:98] 为 state\_cli  [97:96] 为 state\_ser  [95:88] 为packet\_in\_cnd  [87:84] 为 action\_cli  [83:80] 为action\_ser  [79:64] 为 next\_index  [63:32] 为packet count  [31:0] 为byte count |
| rdValid\_connTb | 1 | Out | 连接表读取信号 |
| wrValid\_connTb | 1 | Out | 连接表写信号 |
| wrValid\_agingTb | 1 | Out | 老化表写信号 |
| Ctx\_connTb | 201 | Out | [200] is the aging tag in agingTb |
| Connection\_outTime\_inspector | | | |
| 信号名 | 位宽 | 方向 | 注释 |
| reset | 1 | In | 时钟 |
| Clk | 1 | In | 复位，低有效 |
| Aging\_enable | 1 | In | Mux with connTb configuration |
| Idx\_agingTb | 9 | Out | 老化表输入地址 |
| Data\_agingTb | 9 | Out | Aging tag + timestamp of last packet; |
| rdValid\_agingTb | 1 | Out | 读老化表信号 |
| wrValid\_agingTb | 1 | Out | 写老化表信号 |
| Ctx\_agingTb | 9 | In | 老化表内容，1b agingTag + 8b timestamp |
| agingInfo\_valid | 1 | Out | 老化信息有效位，高有效 |
| agingInfo | 40 | Out | flowID + state(0) + event bitmap (TIME\_OUT) + Timestamp of last packet |
| Cur\_timestamp | 8 | in | Interval is 10ms |
| Buildin\_event\_generator | | | |
| 信号名 | 位宽 | 方向 | 注释 |
| Reset | 1 | In | 时钟 |
| Clk | 1 | In | 复位，低有效 |
| Metadata\_in\_valid | 1 | In | Metadata输入有效位，高有效 |
| Metadata\_in | 134 | In | Metadata信号（pkt\_in别名） |
| Metadata\_out\_valid | 1 | Out | Metadata输出有效 |
| Metadata\_out | 134 | Out | ***采用Metadata+pkt格式，即前两拍为metadata***  Metadata[1]  [127:120] pst;  [119:104] flowID  [103:96] state  [95:88] pkt\_in\_cnd/event bitmap;  [87:80] last timestamp  [79:0] reserved |
| Event\_bitmap\_valid | 1 | In | 事件有效位，高有效 |
| Event\_bitmap | 10 | In | 低位→高位 定义为：  Every packet: 1  New\_flow: 2  Conn\_setup: 4  Time\_out: 8  State\_change: 16  Conn\_end: 32  Conn\_start: 64  Reverserd: 128  Pkt\_in: 256  Forward: 512 |
| pktIn\_info | 32 | In | flowID(16b) + state(8b) + original event bitmap (8b) |
| agingInfo\_valid | 1 | In | flowID + state |
| agingInfo | 40 | In | flowID + state(0) + event bitmap (TIME\_OUT) + Timestamp of last packet |
| Cur\_timestamp | 8 | in | Interval is 10ms |
| Connection\_table\_configuration | | | |
| 信号名 | 位宽 | 方向 | 注释 |
| reset, | 1 | 1 | 时钟 |
| clk, | 1 | 1 | 复位，低有效 |
| aging\_enable, | 1 | 2 | “1” enable aging logic |
| cur\_timestamp, | 8 | 1 | Interval is 10ms |
| idx\_connTb, | 9 | 2 | 连接表输入地址 |
| data\_connTb, | 200 | 2 | [199:104] 4-tuple info  [103:101] reversed  [100] direction  [99:98] state\_cli  [97:96] state\_ser  [95:88] packet\_in\_cnd  [87:84] action\_cli  [83:80] action\_ser  [79:64] next\_index  [63:32] packet count  [31:0] byte count |
| wrValid\_connTb, | 1 | 2 | 写连接表信号 |
| rdValid\_connTb, | 1 | 2 | 读连接表信号 |
| ctx\_connTb, | 200 | 1 | 连接表返回内容 |
| idx\_hashTb, | 9 | 2 | Hash表输入地址 |
| data\_hashTb, | 33 | 2 | Valid\_bit + simplified flow key + flowID（index） |
| wrValid\_hashTb\_1, | 1 | 2 | 写hash表1 |
| wrValid\_hashTb\_2, | 1 | 2 | 写hash表2 |
| rdValid\_hashTb\_1, | 1 | 2 | 读hash表1 |
| rdValid\_hashTb\_2, | 1 | 2 | 读hash表2 |
| ctx\_hashTb\_1, | 33 | 1 | Hash表1返回内容 |
| ctx\_hashTb\_2, | 33 | 1 | Hash表2返回内容 |
| idx\_agingTb, | 9 | 2 | 老化表输入地址 |
| data\_agingTb, | 9 | 2 | Aging tag + last timestamp |
| rdValid\_agingTb, | 1 | 2 | 读老化表信号 |
| wrValid\_agingTb, | 1 | 2 | 写老化表信号 |
| ctx\_agingTb, | 9 | 1 | 老化表返回内容 |
| ctrl\_in\_valid, | 1 | 1 | 控制信号有效位，高有效 |
| ctrl\_opt, | 2 | 1 | 操作类型：  2’d0为读；  2’d1为配置/添加  2’d2为删 |
| ctrl\_addr, | 32 | 1 | 虚拟地址 |
| ctrl\_data\_in, | 32 | 1 | 控制输入 |
| Ctrl\_out\_valid | 1 | 2 | 控制输出有效位，高有效 |
| Ctrl\_data\_out | 32 | 2 | 控制输出 |

# 附录3：UniMon整体设计概述



图2 基于UniMon实现有状态NAT功能

1. FPGA中实现的基本流状态管理（ESM）设计

FPGA（UDAs）实现的功能包括两方面，一是基本流状态管理，二是有状态网络功能加速（功能卸载）。

1. ESM设计

ESM主要包含五个模块，即基于hash的连接表查询、基于双端口RAM的连接表、连接超时管理、连接表配置、事件生成模块。

1. **基于hash的连接表查询模块**的功能是接收数据报文，根据pst提取五元组信息，并查询连接表，获得该流对应的连接状态，并根据流的状态以及接收报文类型，更新并写会连接状态，若未命中，则将flowID置0（后续上报软件，以增加新的连接表项）；另外，该模块还需要基于连接状态以及连接状态变化生成相应的事件，交给事件生成模块，以判断是否需要通过packet\_in方式送给cpu处理；
2. **连接表**的功能是缓存连接的状态，基于双端口RAM实现，一个端口用于查找，另一个用于配置；
3. **连接配置模块**的功能是实现软件增加（新流）、删除（连接超时、关闭）连接表项的功能；
4. **连接超时管理模块**的功能是检测connection是否发生超时，并将超时连接的信息通过agingInfo交给事件生成模块，以构造packet-in消息，上报CPU；
5. **事件生成模块**的功能将事件（bitmap表示）、连接状态、flowID写入metadata中，传递给下一个模块；
6. 有状态网络功能加速

有状态网络功能加速（功能卸载）的推荐实现模型如上图所示，由四部分组成，即flowID对应的action table（如NAT转化表），查表模块，逻辑处理模块（如NAT地址转化），以及action table配置模块。

1. 完整流状态管理（CSM）设计

CSM的主要功能是接收并解析来自FPGA的上报报文（metadata+packet），可以分为三种类型的处理。1）针对新流，分配一个空闲flowID（即增加一条新的连接表），初始化连接状态，并在硬件中增加相应的连接表；2）针对以建立连接的流（未超时或连接关闭），需要将该流的状态更新至软件的连接表；3）针对超时或连接关闭的流，则需要删除该连接（包括软硬件的相应表项），并回收flowID。通过解析上报报文，获得事件类型，将该事件类型（bitmap）以及连接的指针（包含报文）上报给EM部件。

CSM包含六个模块，即报文解析、插入连接、更新连接、删除连接、FPGA连接表配置/更新、触发事件模块组成。

值得注意的是，我们的状态管理方案是，软件管理连接的数量（即增加、删除连接），而硬件负责流状态的维护。而不采用硬件负责连接的数量，软件负责流状态维护的原因是，我们分析有状态网络功能处理发现，如SFW，SNAT，SLB，首报文都需要送软件处理，以判断连接是否可以建立，或者获得NAT映射关系，获得目标服务器。因此，采用硬件负责连接数量管理，动态分配flowID，并不能获得性能的提升，相反，如果只有首报文上报，而后续报文经FPGA处理即可，则能够获得加速效果（网络流量的长尾分布，长流占主要的流量）。

1. 事件管理（EM）设计

事件管理部件的功能是接收并解析CSM通告的事件，查询事件—处理函数映射表，获得事件对应的处理函数。通过调用用户编写的处理函数实现相应的处理功能。

EM包含4个模块，即事件解析模块、处理函数执行模块、事件—处理函数映射表、事件—处理函数映射注册模块。

1. **事件解析模块**的功能是接收CSM通告的事件，根据事件类型分别查询事件—处理函数映射表（事件优先级按照bitmap的顺序排列），从而获得处理函数名称；
2. **处理函数执行模块**的功能是根据事件解析模块查询获得的处理函数名，调用用户编写的该函数，实现报文/流处理。（注意，用户可以在处理函数中调用fast提供的寄存器读写接口，以配置FPGA中功能模块的表项，如NAT转化表）。另外，针对报文的发送，用户可以修改connection指针中的drop\_tap（‘1’表示丢弃，‘0’表示转发），而具体丢弃转发由unimon系统调用fast提供的fast\_ua\_send()实现；
3. **事件—处理函数注册模块**的功能将用户注册的event-handler映射关系写入event-handler table中（采用数组+链表结构）；
4. **事件—处理函数映射表**，采用数组+链表的结构组织，每一种基本事件对应一个数组，该数组又采用链表结构，根据处理函数优先级构造相应的功能函数链；