APU user guide:

A new global trigger system will be installed in the next High-Luminosity LHC upgrade of the ATLAS experiment. Through this upgrade, the trigger hardware and algorithms will be improved. The new global trigger system contains many Global Event Processors (GEP) that receive the data from detectors, process the data through trigger algorithms, and send organized data to the CTP interface.

In the GEP, each trigger algorithm will be performed in an Algorithm Processing Unit (APU). The APU receives data from a BRAM that stores the data from a MGT, or an upstream APU. After performing an algorithm on the data, the APU will send the data to a downstream BRAM for the next APU to read. APUs that get data from multiple sources will have a separate BRAM for each incoming data stream. In the current floorplan, there are 16 different APUs in the GEP framework that handle different tasks. Some of them will process the raw data from the detector. One of them will combine data and send it to the CTP interface. The rest of them will perform algorithms in between APUs.

The data from different detectors to the GEP are not synchronized, therefore we use the Algorithm Processor Platform (APP) to synchronize the input to an APU. In the APP layer, there is a memory controller unit (MCU) to select the data input to the APU. The MCU will select memories that contain the data in the same BC. Therefore the data received by the APU will always be from one single BC.

Data Synchronization:

An APU is a unit inside the Algorithm Processor Platform (APP). The APP layer is used to synchronize data from other APUs or MCUs, and provide input to the APU. The input from the upstream APU(s) and/or MGT(s) to the BRAM has a different clock rate. The APP can handle the problem of unsynchronized clock rates. Within the APP layer, the clock rate is the same, which means APU is using the same clock as the associated APP. The APU developer can decide their own clock rate. The clock rate would be specified in the GEP layer.

More information can be found in ATL-FirmwareSpecification. Section 9.2.3.2

Memory bank:

All communication between APUs is handled via distributed memories (BlockRAMs, or BRAMs). That is, each communication between two APUs uses a private BRAM to send data between those APUs

Memory communication is banked, or double-buffered. That is, there will be space in the memory for at least two full transfers, so that the sender can send new data while the receiver is still processing the old. The APP system takes care of this banking behind the scenes, so that an APU knows that it will always be accessing only the current bank, and will change to the next bank via “done” signals sent to the APP.

The width of the data (i.e. the number of bits at each address) is 128 bits and this width is fixed, and will be the width needed to meet data capacity and transfer speed needs. The width can be 64-bits to 1024-bits. In this Guide, for simplicity we will assume all the data widths are 128-bits.

The data memory map is important for APU developers:
Memory location 0 is a header that holds control data. The format of this control data is:

- Bits [9:0] the top-pointer, giving the number of data items (addresses) in the transfer
- Bits [20:10] store the BCID. An identifier associated with the BC. Increases, but generally not consecutive.
- Bits [n-1:21] unused.

Memory locations 1..N are data words. The actual value of N for a transfer is communicated by the top-pointer (i.e. if the top-pointer is C, then addresses 0..C are used for a transfer).

.

Memory locations above N are unused, and should not be read nor written.

More information can be found in ATL-FirmwareSpecification. Section 9.2.1 and 9.2.2.5

APU I/O:

The I/O of APU can be divided into three parts: control, read and write.

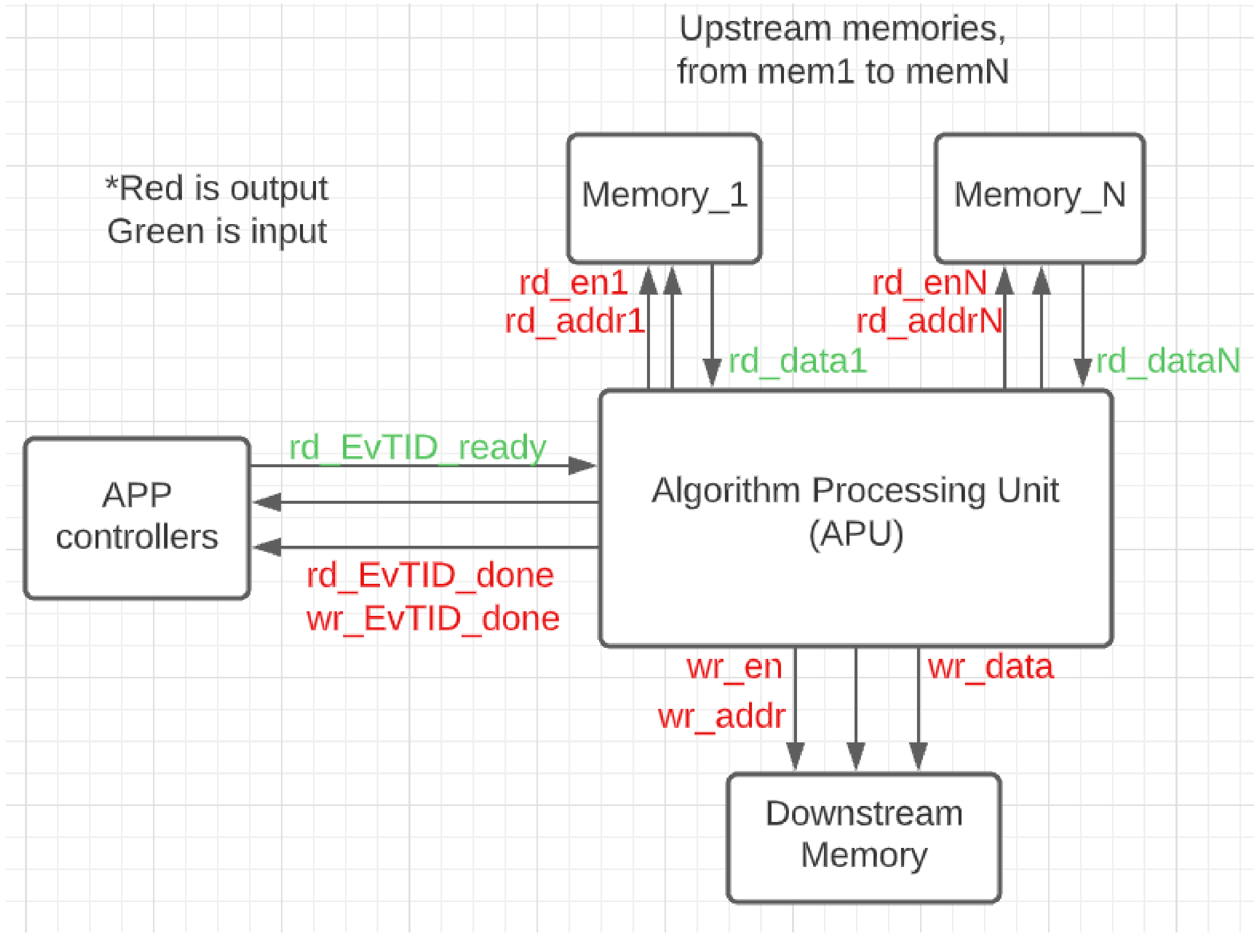
The control signals include one ready signal and two done signals, which are the communication signals to the APP controller. The ready signal (an APU input) is rd_EvTID_ready, and done signals (APU outputs) are rd_EvTID_done and wr_EvTID_done. The control signal also includes clk and reset. APU and APP are using the same clock, APU designer can decide the clock rate. Output control/data will be using the current APP/APU clock. Reset is synced to the clock.

An APU can have multiple inputs, and each input has a read interface. APUs have only a single output write interface, since this is believed to be sufficient for all anticipated APUs. Note that while an APU can only write one output, that output can be sent to multiple other APUs as inputs.

Each read interface is from a BRAM cluster and the APP layer uses the input MUX to select an appropriate BC/event that interacts between the APU and the BRAM. There are three signals in each APU read interface, which are rd_en (read enabler, APU output), rd_addr(read address, APU output), and rd_data(data from BRAM, APU input).

The Write interface is from APU to downstream, and we can only have one write interface. Write has three signals, which are wr_en, wr_data, and wr_addr. They are all APU outputs.

A diagram that shows the I/O of an APU:

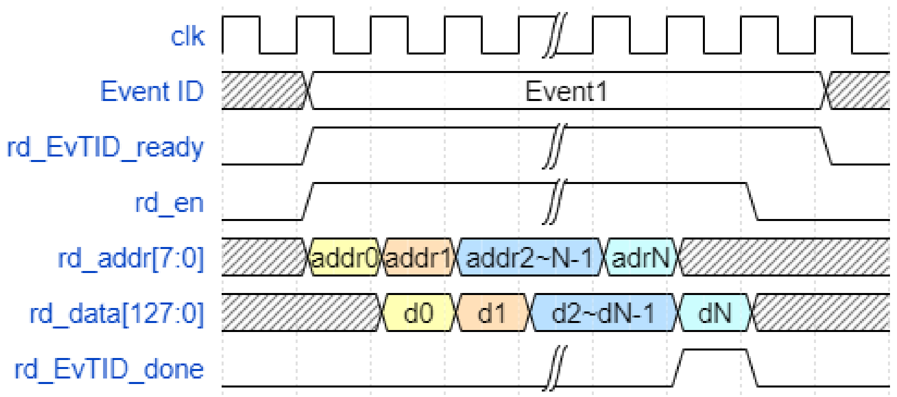


Control signals:	Memory control signals for mem1:	Memory control signals for memN:	Output signals for M_downstream:
input clk;	output rd_en1;	output rd_enN;	output wr_en;
input reset;	output [7:0] rd_addr1;	output [7:0] rd_addrN;	output [7:0] wr_addr;
input rd_EvTID_ready;	input [127:0] rd_data1;	input [127:0] rd_dataN;	output [127:0] wr_data;
output rd_EvTID_done;			
output wr_EvTID_done;			

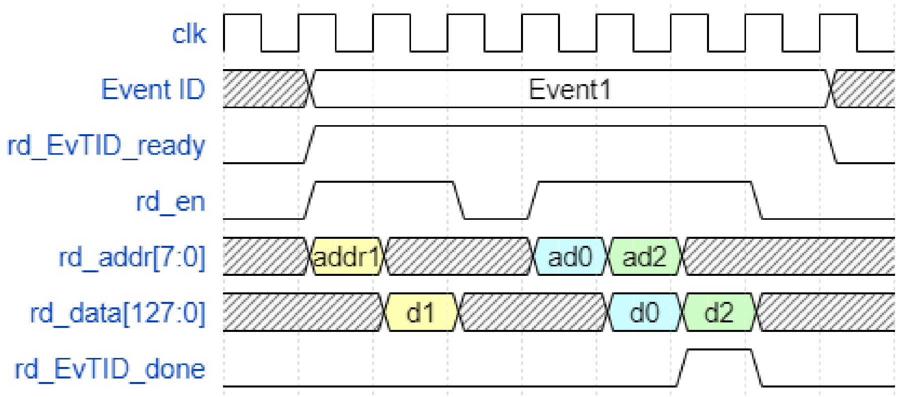
APU upstream signals in detail:

Name	Direction	Description
rd_EvTID_ready	input	An input to the APU. It tells the APU that data in all the input memories are ready to use. APU can only do I/O operations when rd_EvTID_ready is true, and rd_EvTID_ready stays true until APU sends one cycle true of rd_EvTID_done (indicating reading is done). rd_EvTID_ready can be interpreted as an enable to the read ports of APU. APU is allowed to do rd_en and rd_addr at the first cycle (the same cycle APU receives rd_EvTID_ready).
rd_EvTID_done	output	When the APU is done with all read operations, the done signal needs to be true for one cycle. This signals the APP to move to the next set of incoming data. rd_EvTID_done signal can be true in the same cycle of the final rd_en.
rd_en	output	If true, data at rd_addr will be delivered in the next clock cycle. The APU will receive data only when rd_en is true. Therefore rd_en will remain true in the last cycles for receiving the last data.
rd_addr	output	8 bits. The address of data in Memory. If rd_en is true, the data at this address will be delivered in the next cycle.
rd_data	input	128 bits. In cycle I, addr & rd_en pass to memory, and in cycle (I+1) the corresponding data is returned, if rd_en is true.

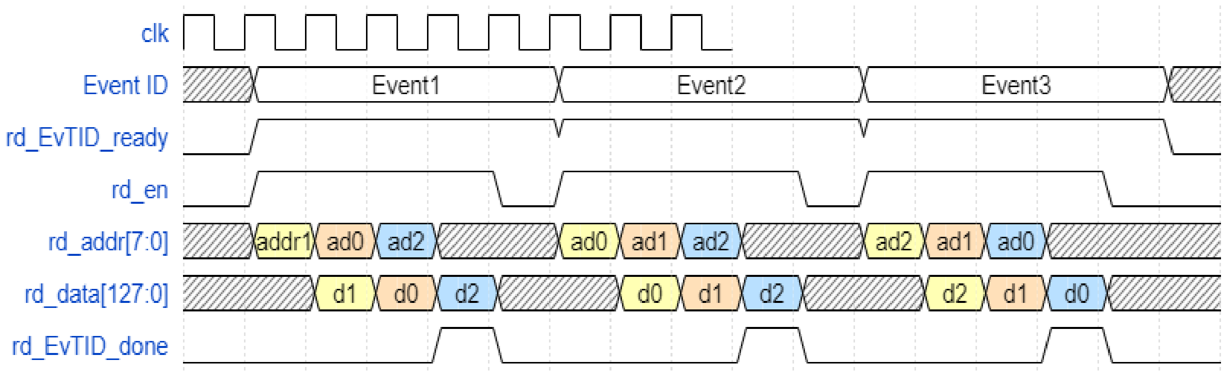
Below is a timing diagram for each upstream I/O:



*The signal EventID is not an I/O to the APU, and it is invisible to the APU, but only visible to the APP layer. Just for reference here.



*When rd_en is off, no data passes to the APU.
*The APU can read data in any order.

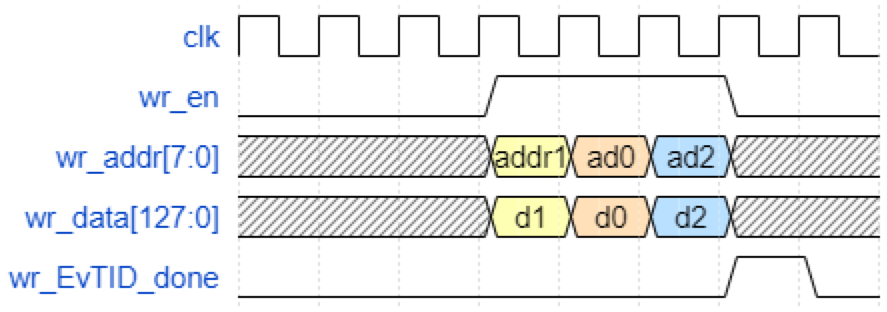


*The current event_ID won't be immediately updated after the rd_done signal.
*After the rd_done signal, the current event would stay for one more cycle.

APU downstream signals in detail:

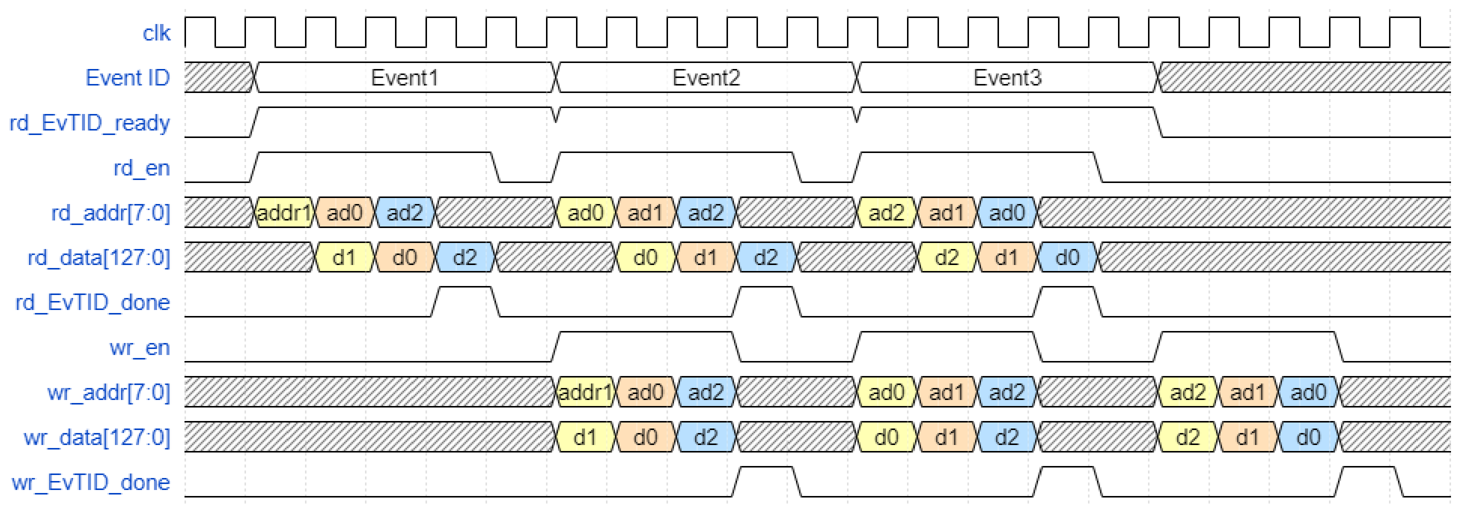
Name	Direction	Description
wr_EvTID_done	output	When done writing for an event, send one cycle true of wr_EvTID_done, telling the sync controller to enable a "new" memory to load data for next BC. wr_EvTID_done signal can be true in the same cycle of the final rd_en.
wr_en	output	Set to true to enable writing the processed data out of the APU. When it is false, data won't write into the memory. All write signals (wr_en, wr_addr, wr_data) would happen in the same clock cycle.
wr_addr	output	8 bits. The addr that you want to save the output data at. In this guide, the addr is 8 bits. There is no particular order that the developer needs to follow, and they can read or write in any order they intended to.
wr_data	output	128 bits, The data sent to the downstream memory synchronizes with wr_en and wr_addr, happening on the same clock cycle.

Below is a timing diagram for each upstream I/O:



*APU developers can decide the order of writing data. The wr_done signal is one cycle after the last write.

The overall timing diagram:



APU design example with test:

A sandbox for APU simulator testing and a example APU can be found at:
[atlas-tdaq-p2-firmware / Global-Trigger / GEP-FW · GitLab (cern.ch)]

Use Hog to create a project at app/GepAlgToy_x which is the sandbox for simulator testing.
Here is an tutorial of using HOG:[How to work with an existing Hog-handled repository]

If you do not know how to use HOG, you can check the project files at: [Sources/Libraries/Framework · zhixij/APUdesign · atlas-tdaq-p2-firmware / Global-Trigger / GEP-FW · GitLab (cern.ch)]
There would be two toy_apu examples in this project.

To use the simpleAPU.sv file, replace the toy_x_apu module with simpleAPU module in toy_x_app.v at line 138.

A test vehicle for on board testing can be found at:
[Files · master · atlas-tdaq-p2-firmware / Global-Trigger / GEP-FW · GitLab (cern.ch)]

Appendix:

GLOSSARY:

- LHC: The Large Hadron Collider
- GEP: Global Event Processor
  Receives and processes the data through trigger algorithms, builds a bit mask of trigger items representing the results of trigger hypotheses, and transmits the resulting 1024-b word (TIP) to the CTP interface; (ATL-FirmwareSpecification, pg3)
- CTP interface: Central Tigger Processor interface
  The layer after GEP, receives the data from GEP.
- APU: Algorithm Processing Unit
  read in data from multiple input sources, process data and output data to the lower stream.
- MGT: Multi-Gigabit Transceiver
  Send data from the detector to GEP.
- BRAM: Block Random Access Memory
  A memory that stores datas for one BC.
- APP: Algorithm Processing Platform
  A platform consists of APU, BRAM, MUX, MCU, that synchronizes the data input to APU.
- MCU: Memory Controller Unit
  Select a memory to read to write.
- BC: Bunch Crossing
  The protons within the two beams are grouped in bunches which are squeezed down in size to increase the chances of a collision. In the released data, the bunches crossed every 50 ns. There were about 30 collisions on average per bunch-crossing. (ATLAS events · GitBook)

Write by: Zhixing Ethan Jiang, Bowen Zuo
Reviewed by: Scott Hauck, Jeff Eastlack, Maayan Tamari