# **ASP Design**

The ASP contains 2 vectors with N number of elements and 16bits in length for each element. Each vector is named: A and B. The variable N, is a template variable that is declared upon the declaration of the ASP module with the following line:

asp<N> my\_asp – Where by N is a positive number up to 512.

The ASP has the following in and out ports:

Input:

clk – The clock of the processor

data\_in – Contains OPCODE and other important data <26bits>

valid – Indicates valid packet for data\_in <1bit>

reset – Reset all vector and states <1bits>

Output:

data\_out – Contains response of the process <64bits>

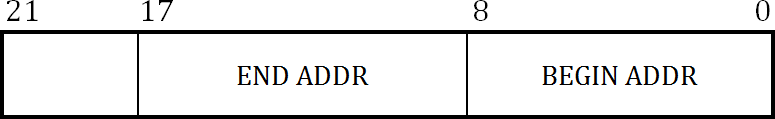
res\_ready – Indicate data out is valid <1bit>

busy – Indicate ASP is busy, cannot receive more instructions <1bit>

Currently the ASP will receive instructions upon the true state of the input, valid bit. Upon receiving a valid signal, the ASP will save the input data and assert the busy state – setting the busy bit to one. The data input will be of either the following two structures:

Command Packet:

* STORE: (bit 17: mel\_sel)

* INVOKE ASP:

Data Packet:

The ASP has three major states:

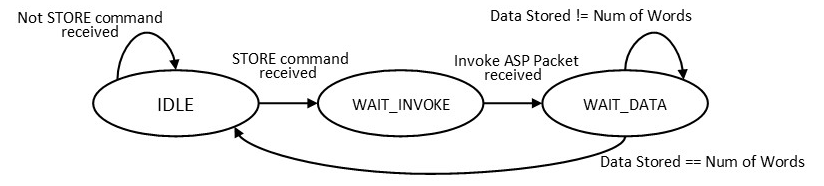
IDLE: This state is awaiting command packets for instructions to process

WAIT\_INVOKE & WAIT\_DATA: Both states are used in a specific operation (Store)

All operations are reactive upon logic state true of the valid bit, within the IDLE state, the main operation is decoding the command packet’s OPCODE. Each opcode has its own specific function as follows:

|  |  |  |
| --- | --- | --- |
| OPCODE | INSTRUCTION | DESCRIPTION |
| 0000 | STORE\_INIT | Store packet is sent. Sets all elements in vector A/B to 0.  If mem\_sel is 0 then vector A, else B. Value of 1 is sent out. |
| 0001 | STORE | Store packet is sent. Sets specific number of elements (num of words) in vector A/B to data given. If mem\_sel is 0 then vector A, else B. Upon receiving this packet the state is set to WAIT\_INVOKE. Invoke ASP packet is sent. Begin and End addresses are stored. State is then set to WAIT\_DATA. Data packets are sent until number of data packets is reached. State is set back to IDLE (detailed below). Each data value that is stored is sent out. |
| 0010 | XOR\_A | Invoke ASP packet is sent. XOR’s values from vector A from beginning address and end address and result is sent out. |
| 0011 | XOR\_B | Invoke ASP packet is sent. XOR’s values from vector B from beginning address and end address and result is sent out. |
| 0100 | MAC | Invoke ASP packet is sent. Multiply values from vector A with vector B and summed, from beginning address to end address and result is sent out. |
| 0101 | AVE4\_A | Invoke ASP packet is sent. Moving average filter with a window of 4 done to vector A. Result of 1 is sent out. |
| 0110 | AVE4\_B | Invoke ASP packet is sent. Moving average filter with a window of 4 done to vector B. Result of 1 is sent out. |
| 0111 | AVE8\_A | Invoke ASP packet is sent. Moving average filter with a window of 8 done to vector B. Result of 1 is sent out. |
| 1000 | AVE8\_B | Invoke ASP packet is sent. Moving average filter with a window of 8 done to vector B. Result of 1 is sent out. |

* *Opcodes 0111 and 1000 were added from the original brief, to differentiate (window size, L), L = 8 from L = 4.*

STORE Command state logic

Each state has a sub state: WAIT\_VALID and PROCESS\_INSTRUCTION. Full FSM/Detailed diagram at bottom of report.

# **ANI Design**

The ANI receives instructions/data from the NoC and sends to the ASP, as well as receives results from ASP and sends to the NoC.

The ASP has the following in and out ports:

Input (ASP side):

res\_ready – Indicates ASP’s result is ready to be read <1bit>

busy – Indicates ASP is current busy <1bit>

from\_asp – The result of the ASP process <64bits>

Output (ASP side):

to\_asp – Data packet to be sent to ASP <26bits>

valid – Indicates to ASP the data packet is valid <1bit>

reset – Resets both ASP vectors to 0 <1bit>

Input (NoC side):

pop – Indicate ANI to pop new result <1bit>

d\_from\_NoC – Instruction data from NoC to be sent to ASP <32bit>

Output (NoC side):

d\_to\_NoC – Response packet to be sent from ASP, to NoC <32bit>

Instructions from the NoC could possibly be sent faster than the ASP can process the current instruction, so a queueing mechanism is in place alleviate this issue. Instructions will be put into a queue will ASP is busy; this is ensuring the instruction requests will not be lost. Along with this, another queue is put in place to send instruction results back to the NoC.

*This queue functionality may change based on development of the NoC.*

* When instructions are queued, they are immediately written to the ASP, IF the ASP is currently not busy. If the ASP is busy, the instruction is then written after the falling edge of the Busy signal.
* Upon the rising edge of the busy bit, this signals that the ASP has accepted the instruction request and the ANI will de-assert the valid bit to make sure that the same instructions are not read multiple times.
* When an instruction result is produced, the answer is placed into a queue, the packet will only be displayed at the output (d\_to\_NoC) upon the rising edge of the pop signal.

Data packets (32 bits wide) to be sent to the NoC will only have results of 16bits wide, with the rest of the packet containing port information, legacy bit, valid bit and 2 unused bits. This poses an issue with results that are larger than 16bit wide, for instance values that are multiplied and summed will eventually become too big to fit within the 16bit wide space. With the maximum total number being: 2,198,956,147,200 a 41bit wide result ().

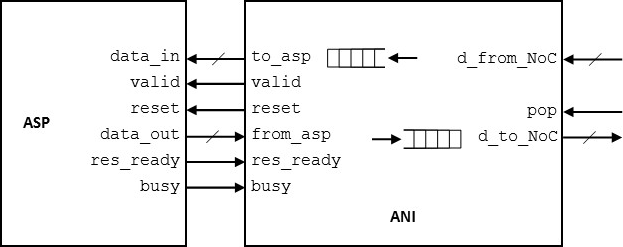
To alleviate this issue is to send results in parts, making the queuing functionality even more crucial. This is implemented using the 2 unused bits of the 32bit packet, whereby the result is split and numbered like the following example:

Packet Structure: (14bits of information) | (2 unused) | (16bits of data/access grant signal)

Decimal Value sent: 446812106 – Binary Value: 0001 1010 1010 0001 1100 1111 1100 1010

Result is split into 2 packets as follows:

* Packet 1: (valid, legacy, port info etc...) **01** 0001 1010 1010 0001
* Packet 2: (valid, legacy, port info etc...) **00** 1100 1111 1100 1010

The following diagram illustrates how the modules will be connected:

# **Simulation**

Simulation of the Model SIM ASP and ANI modules. Refer to the README.md file.

The following is a description of what is to be expected of the port values:

t\_data\_out: is the data value portion that is (or to be split) to be concatenated with the port information and unused bit that is sent to the NoC.

Simulation should run for .

1. The first 160 ns no data packet is sent to the ANI thus 0 is displayed from each pop signal.
2. At 160 ns, the STORE\_INIT, for memory vector A, instruction is send to the ASP and the access granted result is produced. It is only displayed to the d\_to\_NoC port once the pop signal is asserted at 171ns.
3. The following instruction send is STORE\_INIT, for memory vector B, instruction is send to the APS and the access granted result is produced.
4. Next instruction is store 100 words into Vector A, Invoke ASP packet is sent right after. Data packets numbered 0 to 99 is stored in index 0 to 99.
5. Next instruction is store 100 words into Vector B, Invoke ASP packet is sent right after. Data packets numbered 0 to 99 is stored in index 0 to 99.
6. XOR A instruction is sent from index 0 to 1, the result of 3 is sent at 4290ns and popped at 4291ns.
7. XOR B instruction is sent from index 10 to 30, result of 30 is sent at 4310 and popped at 4311ns.
8. Multiply and accumulate instruction from 1 to 10 with result of 385.
9. Average filter with a window of 4 for Vector A with result of 1.
10. Average filter with a window of 8 for Vector A with result of 1.
11. Average filter with a window of 4 for Vector B with result of 1.
12. Average filter with a window of 8 for Vector B with result of 1.
13. Addition pop requests with no queued results will not produce a new result
14. A multiply and accumulate instruction resulting in a value of 328419 should split into 2 separate packets, can be popped twice.

# **FSM**