# **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:

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>

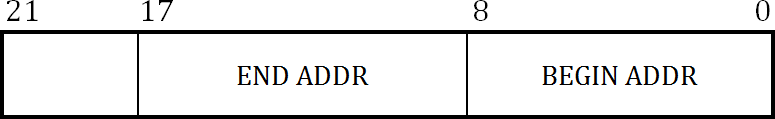
res\_ready – Indicate data out is valid <1bit>

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

Currently the ASP will receive instructions upon the rising edge of the valid bit input. Upon receiving a valid signal, the ASP will save the input data, stall for 2 ns before asserting 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:

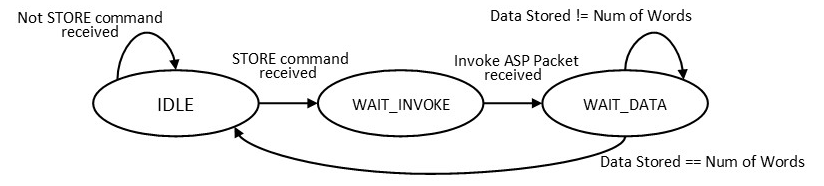
IDLE: This state is awaiting command packets for instructions to process

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

All operations are reactive upon the rising 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. Average of 4 values from vector A starting from begin address. Result is stored at being address and is also sent out. |
| 0110 | AVE4\_B | Invoke ASP packet is sent. Average of 4 values from vector B starting from begin address. Result is stored at being address and is also sent out. |
| 0111 | AVE8\_A | Invoke ASP packet is sent. Average of 8 values from vector A starting from begin address. Result is stored at being address and is also sent out. |
| 1000 | AVE8\_B | Invoke ASP packet is sent. Average of 8 values from vector B starting from begin address. Result is stored at being address and is also 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

# **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 written after the falling edge of the Busy signal.
* When an instruction result is produced, the answer is placed into a queue, the packet will only pop 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 ().

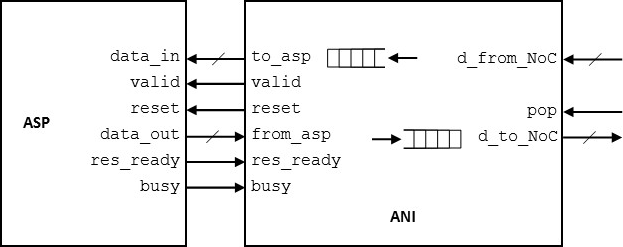
To alleviate this issue is to send results in parts, making the queuing functionality even more prominent. This is implemented using the 2 unused bits, whereby the result is split and numbered like the following example:

Decimal Value: 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: