The Design of an Associative Processing system for Network Monitoring.

Gerald Tripp

Technical Report: 1–00
Computing Laboratory, University of Kent

19th January 2000

Abstract

This paper looks at the design of an embedded processor to be used for network traffic monitoring. This would operate on the stream of data between a network interface and a host computer. Associative processing techniques are used to implement multiple finite state machines that can be used to monitor various network streams or protocol layers. Network packets are selectively captured and passed along with other status information to a host computer for further processing. A first version of such a processor has been designed and modeled in VHDL. The processor design has been simulated and performance figures are presented in this paper along with some sample monitoring programs. This study shows that associative processing appears to be an efficient way to implement fast network monitoring tools.

1 Introduction

This report looks at the design of an associative processing system for use in network traffic monitoring. The aim here is to design an embedded processor that can be used between a network interface and a host computer. This system would then be programmed to monitor various streams of traffic that it receives from the network interface and would then make decisions as to which packets of data and other information should be passed on to the host computer. The idea here is to be able to make intelligent decisions concerning selective traffic capture close to the network interface. We then supply the host computer with a stream of data at a far lower data rate than that received from the network, but which has a high information content – i.e. the aim is to capture *useful* data. In this report, a prototype design is presented that could be implemented using a small number of components. This design is simulated and the model used to generate figures for performance. A critical evaluation identifies shortcomings and makes recommendations for improvements and alternative designs.

Ideas concerning this topic have been presented already in two previous reports. The initial report [1] gives a survey of various methods that have been used for data reduction in real time network monitoring. This makes a proposal to use a finite state machine (FSM) based system and recommends that this could be implemented using associative processing techniques to provide a parallel implementation of logic for next state and output generation. This subject is developed further in [2] which looks at possible associative processing architectures. Topics covered there are those of implementing multiple FSMs, managing access to network data and also possible styles of output event that could be generated to pass onto the host processor.

The next section gives an overview of the work presented in [1] and [2]. Section three gives the overall aims of the work presented in this paper. The forth section gives details of the types of components chosen for this design and the proposed board level implementation. Section five describes the design of the custom logic for the associative processor, and gives details of the individual modules along with the design choices that were made. The sixth section describes the methods used for simulation, gives performance figures for a few example monitoring programs, gives a critical evaluation of the overall performance and makes recommendations for improvements and alternative designs. The last section gives conclusions and ideas for further work.

2 The Associative processing system

Two previous reports have looked at this subject. The first report [1] starts with a review of work carried out in the field of network monitoring and makes a proposal that a finite state machine (FSM) approach should be used with associative processing used as the method of implementation. The proposal was that functional memory [3] should be used to implement look up tables for the FSM implementation. Functional memory is useful because this allows each bit in memory words to be specified as 0, 1 or don't care – this allows the memory to contain 'match patterns' rather than specifying specific binary values. We search the functional memory with a key consisting of the current state and the inputs, and retrieve a code word that gives a valid match. This code word also contains values for next state and outputs. To implement this, code words need to be generated for each state that together match all possible values of the input data. To cope with the possibility of multiple matches, the one with the highest priority (such as lowest memory address) is selected.

This work is developed further in [2] that looks at possible architectures that could be used to implement a practical system. The topics covered in [2] can be summarized as follows:

- The system described in [1] is extended to allow the implementation of multiple FSMs, by the use of a separate area of memory to hold state information for each instance of a FSM – referred to as a channel. A scheduling system is then used to swap between channels – one of which is the current active channel and able to receive input data from the network and the others being suspended. Channels communicate by sending event messages to each other, the passing of such a message causing the sender to be suspended and the recipient to be activated. This system is further extended by the provision of a procedural interface, allowing the return to a calling channel. Initial suggestions are given for the implementation of dynamic channel allocation.

- It is proposed that the output from the FSM should be able to specify a number of possible actions, including arithmetic operations, scheduling operations (as above), output event generation and control of the network data stream.

- Various styles of output event generation are proposed, from general-purpose message generation to simple packet capture.

- The access to the network data stream is discussed and sequential access to a data stream is compared to random access to a packet stored within the processor.

It is argued that it should be possible to build a practical associative processing system for network monitoring using existing components.

3 Aims

The aim of this work is to design a prototype associative processing system for use in a network-monitoring environment. At this stage there is no intention to build a piece of hardware, but to design and model such a system at a level of detail that will allow a prototype to be constructed at a later date if required. This low-level model would primarily be used to show that this type of processing is practical and also to show that it could be put into manufacture if required using a small number of existing components.

The work involved here has three distinct stages:

1 Develop a design for an associative processing system based on the principles described in [2]. The processor is then to be modeled in Register Transfer Level VHDL[4], the other components such as memory to be modeled in behavioral VHDL. The complete system then to be tested by simulation.

2 Synthesize logic for the associative processor, targeted at a Field Programmable Gate Array (FPGA). Use tools to generate an FPGA design from the synthesized logic. Constrain the design to operate at the required clock rate and with the setup and output delay times specified by the external components. At this stage the original design may require modification depending on the implementation size and speed of the synthesized logic. Modify design and iterate until required performance is met.

3 Use simulation to test the complete associative processing system with example network monitoring programs. Use this stage to determine the performance of the processor as a whole and to highlight areas for improvement.

The overall aim here is to be able to show whether this type of system is practical, to generate figures for the system performance and to evaluate the decisions made during the system design. Following experience in programming such a system and observing its performance, it should be possible to determine whether the mechanisms chosen are appropriate and if not the alternative approaches that could be taken in future designs.

4 Design Choices

A significant consideration in this work was to design a small prototype system that could be modeled in a reasonably short period of time. Also any system design should be practical; it should be possible to build a physical system if required using a small number of components on a single circuit board. With these self-imposed constraints in mind, the following subsections look at choice of components.

4.1 Associative memory components

This associative processor requires the use of ternary content addressable memory – often referred to as functional memory [3]. There are less sources of this than the standard content addressable memory, and the most suitable component at the time appeared to be the NL85721 [5] from NETLOGIC. There is a version of this component specified as operating at 66MHz, although this is heavily pipelined. This CAM has a 64 bit CBUS that is used to carry comparand values (keys) to be used in searches and can also be used for read and write access to the memory. It has a 16-bit instruction bus, to determine the function performed by the CAM and a Results bus (RBUS) that outputs the contents of the status register. In general, instructions can be started every clock tick – subject to various constraints. These may however take a few cycles to complete.

The NL85721 has a 128-bit word, with each memory word also having its own mask work. Internal global match registers are provided that allow the user to specify which bits of the CAM word are used in any search and hence allow the user to use part of the CAM word as an associated result if required. The fastest type of search can be performed by using a 64-bit key, as this can be transferred over the CBUS in a single clock cycle. Following a search instruction (*write comparand and compare*), match result flags are generated in 2 clock cycles and a status output including the highest priority match (HPM) address is generated on the RBUS in 3 clock cycles. If required, we can follow the CAM search instruction with a Read memory at HPM instruction. This will take 2 cycles before the value is presented on the CBUS – the same time as the status is available on the RBUS. Using this method of operation, we can use half of the CAM word to match against a 64-bit key and half to generate a 64-bit result. If we require a key of greater length than 64 bits, then we need to write part of the comparand first (*write to comparand*) and then follow this with a search (*write comparand and compare*) – this will now take 4 clock cycles and will also be at the expense of the size of the result. Alternatively, we could use all of the CAM word to match against a key of up to 128 bits and then use the HPM value on the RBUS to index into a separate area of RAM to generate an associated result from the search. This would take 4 clock cycles plus the time to retrieve data from the RAM.

For performance and simplicity, the first method was chosen, hence taking 3 clock cycles from presentation of a key of length $\leq$ 64 bits, to give an associated result of length $\leq$ 64 bits.

4.2 Other components

The content addressable memory uses a pipelined design, thus enabling a high clock rate. Given this style is used by the CAM, it was decided to use a similar style of operation for the other parts of the system. It is expected that future components may follow this trend of pipelining and high clock rates.

The random access memory – used for registers – uses synchronous random access memory. This is the MT55L64L32F [6] from MICRON technology, inc. This is 64K x 32 bits and a version of this part is specified as operating at 100 MHz. The part chosen here is the flow-through version of this memory that provides data in the cycle following a Read operation

Finally, it was proposed that the processor part of this system should be implemented using a Xilinx FPGA. The Xilinx X4000XLA series [7] was chosen as the most appropriate technology to target at the time, primarily because of the performance and availability of design tools. Using these components we are able to design a fully synchronous interface to the RAM and CAM that will operate at the maximum clock rate of the CAM. In practice, the overall maximum clock rate for the system will then depend on the performance of the logic within the FPGA and this requires us to avoid designs that will synthesize into multiple levels of logic.

4.3 Board level implementation

From the above it can be seen that the complete system can be implemented as 3 chips. The network data input and output busses are currently considered to be 32 bits wide. In any physical implementation, it is anticipated that these wide busses will be replaced by standard Utopia [8] style interfaces of 8 or 16 bits in width.

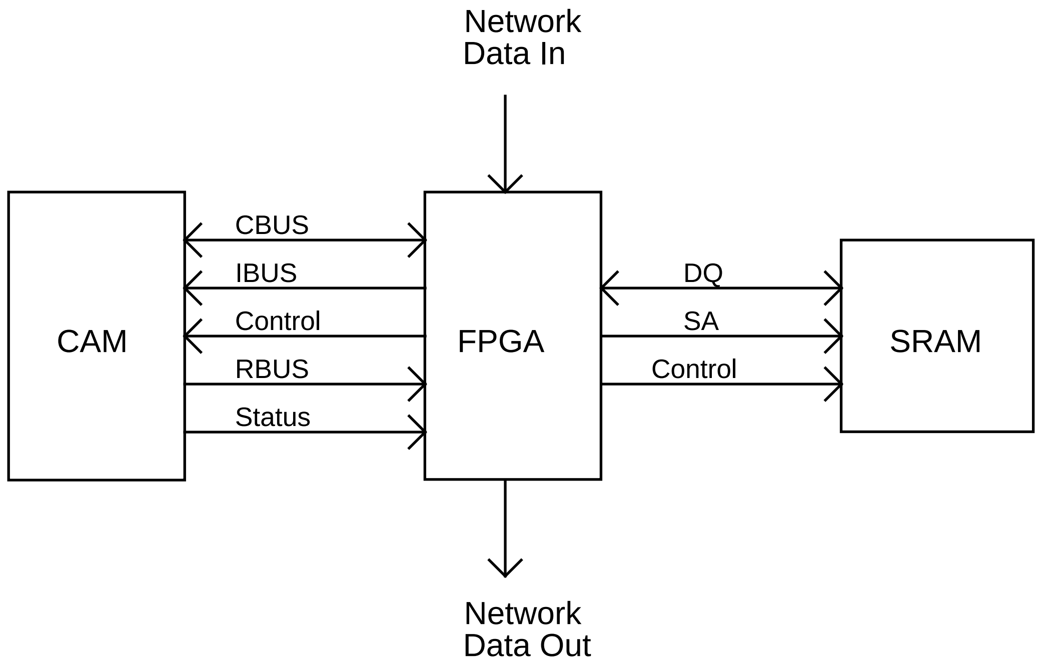

Figure 1 - Board level implementation of associative processor.

5 Processor Design

5.1 CAM search cycle

Much of the design for this processor is based around the timing of the access to the content addressable memory chip as described in the previous section. Using this CAM chip, we can obtain the fastest operation by performing a search using of a key of length 64 bits or below, and then follow this by a read of the memory location giving the highest priority match. This can give us an index and a 64-bit memory word result in 3 clock cycles.

To obtain the highest clock speed, we can use synchronous input to, and output from the FPGA. This gives a minimum delay of 2 cycles. If a search of the CAM uses a key that may depend on the results of the previous search, then generally we have an overall minimum search period of 5 clock cycles. Introducing combinatorial logic into the input or output to the CAM could optimize this, but this may increase the time required for a clock cycle. This could be investigated in later designs, particularly if faster FPGAs can be used.

A disadvantage of this system is that because we wait for the results of one search before starting the next, we are under-utilizing the CAM chip, which is capable of performing more frequent searches if required.

5.1.1 CAM code words and result

Given the method of searching described above, we are able to use a 64-bit key and 64-bit result. The key was subdivided as follows:

32 bits	Network data
12 bits	Current State
12 bits	Current Channel
4 bits	Current invocation event
2 bits	ALU status
1 bit	End of packet flag
1 bit	CAM word valid bit

The overall throughput depends on the number of bits used in the network data key, so it is advisable to allocate as much space as possible to this – although for simplicity this will need to be kept to a conventional word size such as: 8, 16, 32, 64 bits etc. For this design, half of the CAM key is used for network data. Of the remaining bits, 28 are used for the various processor states which is enough in this instance because only a small processor is being built. The ALU status bits allow searching to be conditioned upon the results of ALU operations; the end-of-packet flag allows us to test if we have reached the end of the current packet and the valid bit allows active CAM words to be marked to indicate whether they should be used as part of the current search.

A results width of 64 bits is less of a problem than for the key as this time we have no large data field to use up all the resources. This time however we have to include fields to set a new processor state, and also fields to specify any actions that need to be performed. These have been specified as follows:

12 bits	Next state
12 bits	Possible next channel / Immediate field
4 bits	Possible next event
4 bits	Network data control
29 bits	Action and parameters
3 bits	Reserved

The Next channel field can also be used for ALU operations as an immediate value.

5.2 Actions

For simplicity, all the possible actions except for control of network data have been grouped together into a single instruction to be executed following each search. This has the disadvantage that it is not possible to perform more than one of these operations in a single cycle, the advantage however is that it makes implementation a lot easier.

5.2.1 Channel Memory

The random access memory for this processor has been provided as a block of 64K x 32 bits. This is quite a small amount by current standards although this could be increased in later designs if required. It was specified previously that each channel should have its own local memory space. To avoid having a memory management system, the total memory is simply divided between the maximum number of channels – this is then referred to as the channel's register window. Each channel therefore has a fixed number of registers within its register window. It is very simple to implement this as we can form the memory address from the channel number and the register number used within the channel.

Memory address =	Channel number (12 bits)	Register number (4 bits)

In our system with a total of 64K words of memory and 4K channels, we have 16 registers per channel R0 → R15. If we reserve one of the channel numbers – such as 0 – we can use this register window for a block of global registers G0 → G15. This is only a small amount of memory per channel, however it does have the advantage that any address fields used within the actions can also be small.

5.2.2 ALU operations

In this particular environment, we have a long search time for each new CAM word, but have a reasonably large number of bits to specify any action. In this case it appears sensible to try to code most simple arithmetic operations in a single action. To do this, a three-address architecture has been used with instructions for ADD, SUB and MOVE. This is possible so long as we don't have large address fields to specify the operands. In this prototype, this has been done by using 6 bits for each operand address field – this specifies either: a global register, a local register, an immediate value from a separate field or other dedicated internal registers.

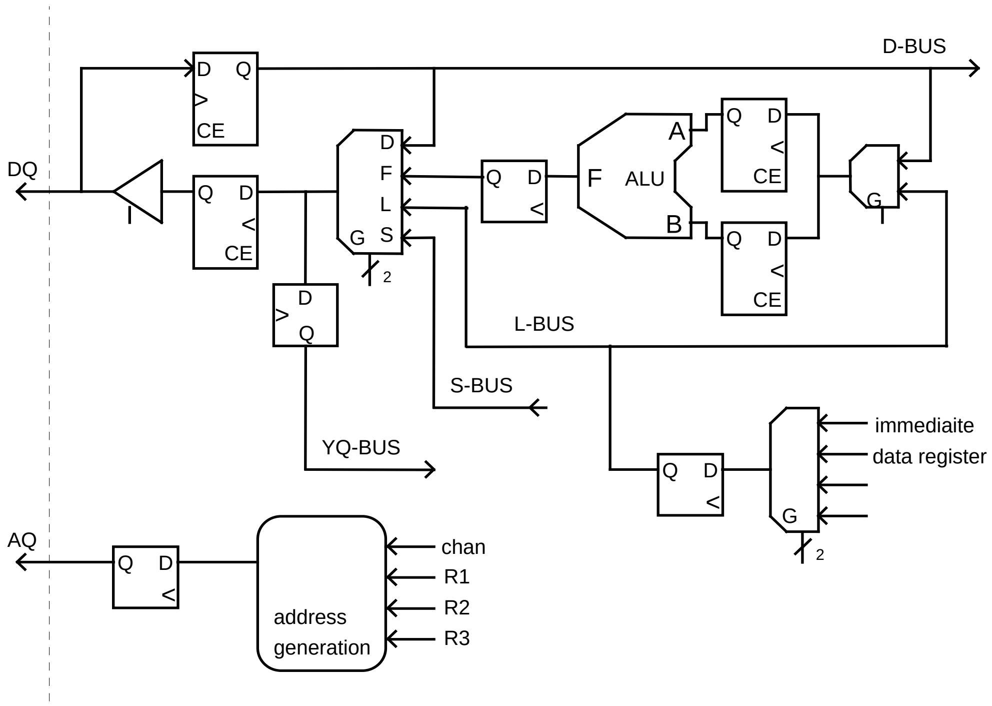

Figure 2 – ALU data paths.

A basic outline of the ALU data paths is shown in Figure 2. The D-BUS and S-BUS signals are connected to the scheduling sections to give access to the external memory. The YQ-BUS provides a copy of output data to internal registers.

5.2.3 Scheduling operations

As well as ALU operations, the scheduling operations also require access to memory and in this design are controlled using the same mechanism as the ALU. This unfortunately means that it will not be possible to have both types of these operations at the same time – it does however enable a single control system. To aid flexibility, the ALU operands can also be used in scheduling operations to specify the registers used to hold channel state etc.

The scheduling system provides ways of changing the current channel, event and state. The default action here is to leave the values of the channel and event as they are and to load a new value of state from the next state field from each new code word read from the CAM. There does however need to be alternative sources for each of these three key fields as we change from one channel to another using the various mechanisms described in [2]. The various alternative sources of these key values can be the scheduling stack, memory or from the CAM code word. A basic schematic of the scheduling system is shown below.

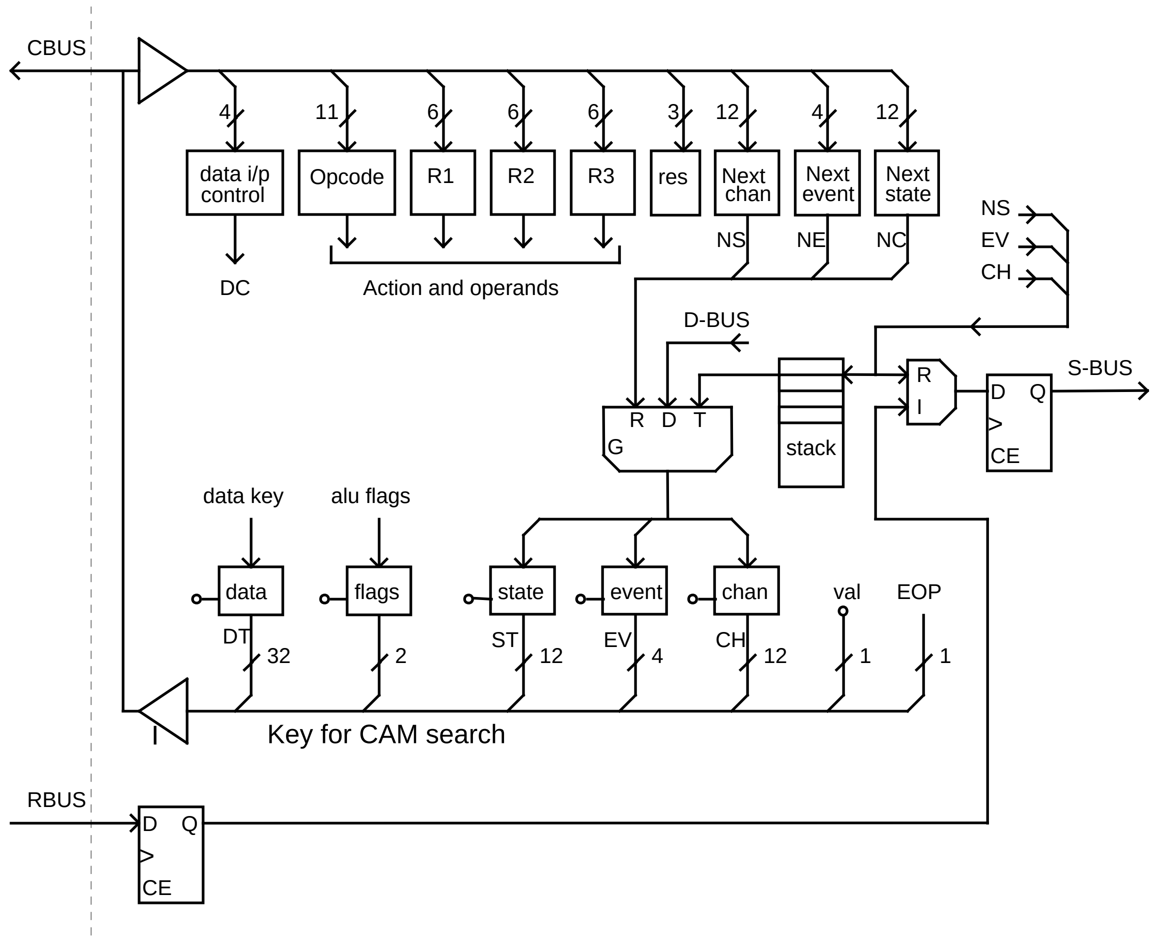

Figure 3 – Paths for keys and scheduling.

The scheduling section connects to the ALU module via the D-BUS and the S-BUS to gain access to the random access memory. The ALU status flags and the End of Packet (EOP) flags are also used as part of the key as described in section 5.1.1. The data key is provided by the network data module. The next channel signal is also used to generate an IMMEDIATE value for both the ALU and the network data module. For clarity, control paths are not shown.

5.2.4 Output events

A very simple output event system has been implemented. One of the global registers doubles up as an input port to a token queue. Any of the channels can write a token to this queue to specify if the current packet should be saved or discarded. In addition to this, each packet passing through the processor has a block of global registers appended to the end of the packet. This is explained in more detail in section 5.3.3.

5.3 Network data

As described in [2], the network data could be handled in one of two different ways.

- As random access in the packet contents
- As sequential access

A disadvantage of random access is the potentially large amount of multi-port memory that is required to hold the packet within or closely coupled to the processor. The FPGAs chosen for the implementation of this system was the Xilinx XC4000XLA series. This can use look up tables (LUTs) for RAM implementation, however this only provides individual blocks of 16x1 bits when used as dual port. Although this would enable enough memory to be generated to handle ATM cells, it would be more difficult to handle the more standard length packets found on Ethernet etc. – as this will require large numbers of LUTs and will typically increase access times as the memory depth is increased. Due to this potential problem, it was decided to provide sequential access into the network data path.

5.3.1 Data register and network data selection

The program running on this system may not wish to access all of the data in sequence as it arrives from the network. For example it may only wish to access a few separate words within a packet. To allow us to determine the rate that data is accessed in software, we will need an input buffer to hold data as it arrives from the network until it can be processed. In writing the software for this system, the programmer will need to ensure that the average time taken processing a packet does not exceed the packet transmission time – thus ensuring that the input queue length does not increase over time. If the input queue is small compared to the size of the packet, then the programmer will also need to ensure that the input queue does not overflow during the processing of a packet.

To generate the data key to be used for CAM searches, a data register is used. The programmer can then insert instructions into the result section of each CAM code word to select new words of data to be written into this register. As this system is sequential access, the assumption is made that the programmer can keep the value that is currently in the data register or request that this is loaded with a new word of data that is further back up the input buffer. The programmer can not request a word of data that has already past, as this may no longer be available.

In some circumstances, the programmer may wish to save items of data and be able to load old values back into the data register. This could be the case for example if parts of a word were obtained from separate packets. Provision has therefore been made to read the data register and also to rotate values from RAM registers and selectively write individual bytes in the data register

5.3.2 Pipeline & cache

A potential problem with sequential access to the network data is the delay that may be caused by a reload of the data register. A program may ask for a new word of data that is several words back up the input queue. If we read words of data from the input queue until the required word is available, then this may take several clock cycles and hence degrade the performance of the system. To enhance performance, an input pipeline is used that holds up to five words at the head of the input queue. With the use of multiplexors, many possible words could be retrieved directly from the pipeline.

Unfortunately there is a separate problem, that of word alignment. Protocols are often carefully designed to allow 32-bit words to fall on word boundaries – for obvious reasons. This is not always the case however, and we may have the situation that the 32-bit word we are interested in spans 2 words from the input queue.

The current network data cache consists of a five word input pipeline and a four-word cache. The data is kept on its original word boundaries in the data pipeline and this pipeline moves forward as the data is accessed. The data may be required on arbitrary byte boundaries and it is the role of the data cache to enable this to take place. The pipeline is originally filled such that the first word of a packet is located in the first word of the pipeline. As the data is used, the data moves forward. The pipeline is maintained in a state such that the last word accessed is as close to the head of the pipeline head as possible. As the information may be accessed on any byte boundary, an offset register specifies the byte offset of the last accessed word from the head of the pipeline. This offset may be in the range 0 to 3 bytes and this therefore points to the current head word. It is likely that any forward movement will have a high probability of being a multiple of four bytes and this is assumed by the caching mechanism. When the pipeline is full, the data cache will save words that are at offsets of 0, 4, 8 and 12 bytes from the head word – each of these words can be selected from one of four positions in the pipeline, depending on the value of the offset register.

If we select one of the cached data words, or ask for no change, then we will get a cache HIT. This means that the data will be available immediately without any additional delay. Following this, the data cache will sometimes need to initiate an internal REFILL operation to move the latest word accessed up to the head of the pipeline and then refill the cache. The time for the REFILL operation for data offsets of 4, 8 and 12 is 5, 6 and 7 cycles respectively. Hence a move forward of 1 word takes the same time as the shortest action (by design) and will leave the cache ready for the next cache search.

If we ask for any other data offset, or if the cache isn't ready, then we will generate a cache MISS. This may require us to wait for a previous cache refill to complete, or it may require a word that isn't in the cache. In either case, the required word will be moved to the head of the pipeline, the offset register set appropriately and then the word accessed as being at offset 0. Searches for the next instruction are started in parallel with the action for a previous instruction assuming that a cache HIT will occur. If we have a cache MISS, then the search will be abandoned and restarted when the required data is available. It can be seen that large moves forward will often generate a cache MISS and hence cause a delay before processing can continue.

In addition to the data moves described above, we also have a NEXT instruction which flushes the remainder of the current packet to the output buffer and appends any status information to the packet end. A new packet will then be loaded into the pipeline. The NEXT instruction will give a cache HIT if the cache is ready. A move of 0 is required before accessing the first word of the next packet – this will give a cache MISS if the data transfer is not yet complete. A NOP instruction is also provided which will leave the data key unchanged. This will always generate a cache HIT, even if the cache is not ready. Careful spacing of changes to the data key may optimize the performance of programs.

The scheme above could easily be extended to use a larger pipeline if required. The caching system uses a lookup table to determine which words are in the cache when it is ready – hence this could easily be modified to cache more words or words with different offsets. This example design has been limited to a cache size of four words to limit the amount of resources used and also to enable the fast selection of the correct word from the cache.

5.3.3 Packet capture and discard

It is likely that any software processing a packet will not have made a decision as to whether the packet should be kept until most, or all, of the packet has been examined. Because of this, the packet will need to be written into the output queue before a decision has been made as to whether it is required. To circumvent this problem, a token queue is used in parallel with the output queue. When a decision has been made concerning the current packet, a single bit token is written to the token queue with the Boolean value specifying if the packet is to be kept. The process reading from the output queue will first need to wait for the value from the token queue to say if the packet is to be discarded or kept for further processing. This is hidden within the processor by the use of a secondary post-discard queue*. Apart from deciding whether or not to keep a packet, the programmer may also wish to return some status information with a saved packet. This is implemented by returning a copy of G0 and G1 at the end of the packet – this could be extended to provide copies of more registers if required albeit at the cost of having more data to transfer to any host system.

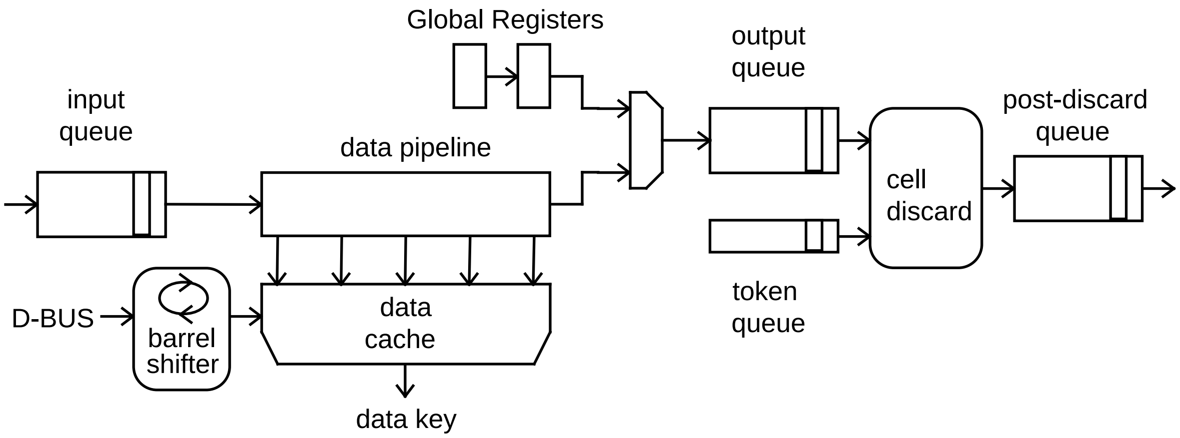

Figure 4 - Data cache and the packet discard mechanism.

---

* The cell discard mechanism limits the maximum size of output event packet that the processor can handle to the size of the output queue, which in this prototype is 64 bytes. The output queue could be expanded to a certain extent within the limits of the FPGA used. Alternatively, this queue could be supplemented with a larger external FIFO and the cell discard mechanism implemented either (back) within the FPGA or at the input to the next processing stage.

5.4 Control

The processor is controlled by the use of a conventional micro-program architecture [9] using a wide micro-code word. This uses 64 words of control store and a 16-entry mapping table. This controls the operations performed by the CAM, RAM, ALU and the various key generation and scheduling functions. A single instruction determines the action that should be performed.

Unlike a conventional processor, it is difficult to predict very far ahead where the software will be executing. A complete cycle consists of a search for a code word followed by the execution of any associated action. However, if the action for a code word does not affect the key values used for the next search, then we are able to start the search for the next code word in parallel with the execution of the action from the current one. Any action that changes the key values used for the next search will need to either perform a new search itself as soon as the new search values are valid, or it will need to initiate a pre-fetch after it has completed.

The only parts of the system not controlled directly by the micro-program are the network data paths, the pipeline and the caching system, which run autonomously. When a new code word is read from the CAM, the caching system will look at the instruction specifying changes to the data register and try to fulfill these if possible. If it cannot do so then it will set a flag to indicate a cache miss. The micro-program checks this flag during the execution of the current action. Following a cache miss, any pipelined search is abandoned and a piece of micro-code is invoked to wait for the cache to become ready and refill the execution pipeline.

5.4.1 Bootstrapping

Before the system can operate, the micro-code goes through a boot sequence during which it reads data from an external ROM and writes code words into the CAM or initializes registers in RAM.

6 Evaluation

This section looks at the testing of the design and the timing results for some example monitoring programs.

6.1 Modeling and simulation

The processor itself was designed in VHDL and tested by simulation. The type of VHDL code used was Register Transfer Level and the design was written in such a way as to be able to synthesize fast logic for the target FPGA. The other two components in the system were specified using behavioral simulation models. The model for the RAM component was obtained from the manufacturers and the model for the CAM was generated locally using details and timing information from the manufacturer's data sheet. These three parts were instantiated within a top-level design 'engine' which linked together the various signals.

The design 'engine' was then tested using a testbench design that includes other components such as a traffic source and a boot ROM. It also generates other signals such as reset and clock.

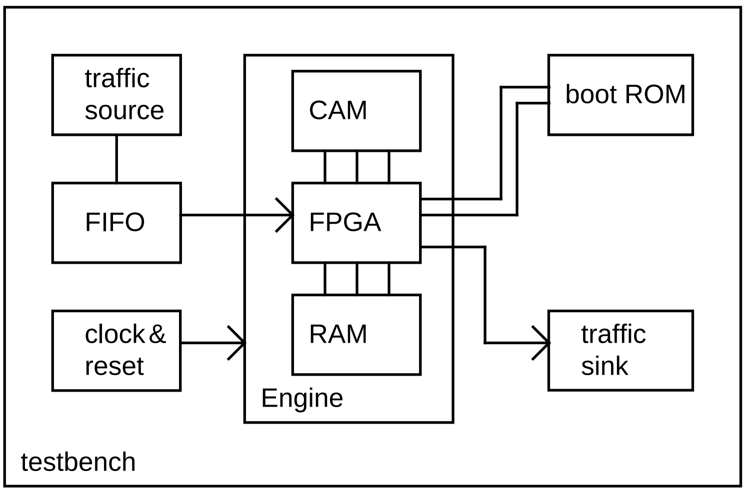

Figure 5 – Testbench for Associative Processor.

The traffic source simulates a stream of cells from an ATM physical layer interface. The ATM cells in this cell stream are 52 bytes long – the HEC byte being assumed to have been removed in the physical layer, as is common practice. A small FIFO was included to emulate the cell FIFO such as may be found in these physical layer interfaces. This external FIFO is monitored throughout the simulation to ensure it never becomes full and discards incoming data. The traffic source runs at a high data rate (266.7 Mbps) and also starts before the associative processor to ensure that the associative processor always has data available on its input for processing. The level of the input FIFO within the FPGA was monitored throughout the simulation to ensure that this never became empty. This was done to ensure consistent timing results and also to ensure that we were measuring the performance of the processor and not the speed of the traffic source. The traffic sink reads data from the processor output FIFO and discards it.

The boot ROM contains the code words for the content addressable memory and also details of any variables in RAM which need initializing at load time. It also contains a number of other constants required to initialize registers within the CAM.

This structure was used for debugging the associative processor design model and was also used subsequently to evaluate performance.

6.2 Evaluation of synthesized logic

As well as building a model of the associative processor, it was also an objective to be able to build a design for an FPGA. The process involved here is to synthesize logic for the design model and then to use the FPGA tools to build the code for the FPGA. To be able to do this, we need to ensure that it is possible to synthesize logic from our model, that the logic synthesized is of a sensible size and also that it runs fast enough. It is very easy in VHDL to model systems that will synthesize into large amounts of slow logic.

Once the code for the FPGA has been built we can obtain figures for the worst case timing for the FPGA and also the amount of FPGA resources that are used. The former point is important as we now have some real timing figures for our design. To push the design tools in the right direction we can give constraints that can specify for example the target system clock speed and also the setup and delays figures for attached components.

For this design, the target clock rate was 50 MHz – i.e. requiring a 20ns clock period. Initially the logic synthesized had a minimum clock period of over 40ns; this however was quickly traced to the ALU carry logic. The ALU had been designed by instantiating four 8 bit Add/subtract units and there were large carry delays between the units. The design was modified by specifying the ALU operation in VHDL and allowing the compiler to synthesize logic for this itself. This generated a far better carry chain than the original and the clock period of 20ns was achieved without further problems.

6.3 Evaluation of system performance

Following the completion of the logic synthesis, it was possible to perform test simulations with an accurate figure for the system clock speed. This section looks at some example programs running on the simulation and gives figures for the time taken to process each cell. As previously noted, these tests were performed using a heavy continuous traffic stream, thus enabling consistent figures to be obtained for cell processing time. If we look at the timing of cells passing into the associative processor, then this gives rather variable results due to the effect of the small input FIFO. To remove this effect, the cells are observed at the point at which they leave the input FIFO. The reference point used is the time instance when the first word of a cell is read from the network data cache and the time for each cell is measured from this point to the time when the first word for the next cell is read from the cache. This form of measurement is valid in this context as we have a continuous source of cells arriving from the network. The time measured gives the time for processing of the cell plus the time taken to flush out the remainder of the cell and to refill the cache – this accurately reflects the time the processor was tied up with processing a particular cell, it also gives consistent results.

In the following examples, references are made to pseudo-code in the appendices. For the simulations, this pseudo-code was converted by hand into a form of wide micro-code and then assembled to give the data used by the boot ROM.

6.3.1 Cell count program

The first program to be tested was the cell count program described in [2] and referred to here as cellcount-1. This does a simple decode on AAL5 frames arriving on a given VCI/VPI and counts the number of cells received on a small number of IP address pairs. The psuedo-code for this is given in Appendix A. There are three separate paths through the code depending on whether: (a) the cell is on a different VCI and hence ignored, (b) it is the first cell of an AAL5 frame and the IP addresses need to be checked, (c) it is part of the body/end of an AAL5 frame and the IP addresses are already known.

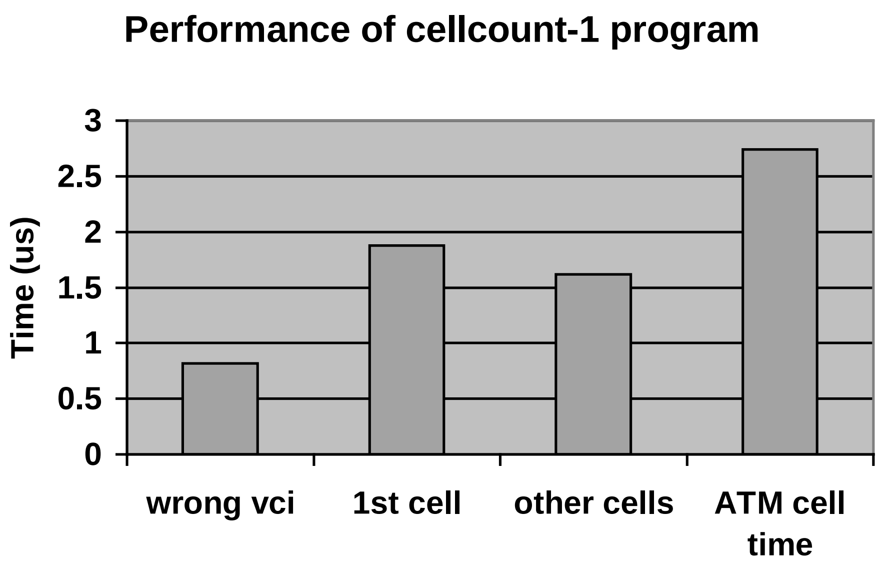

Figure 6 - Cellcount-1 program.

The results of the timing for cellcount are shown in Figure 6, along with the time taken to transmit one ATM cell at 155 Mbps for comparison. The time for cases (b) and (c) were slower than expected. By checking the simulation, it was noted that there was a lot of time spent flushing out the cell after it had been processed. By inspecting the code, this becomes obvious – the request for the next cell does not take place until we return to the bottom level. A problem here is that we have gone through a series of return and other statements without moving forward though the cell stream. If we modify the software (cellcount-2) to request the next cell as soon as it knows it no longer requires the previous cell, then we can improve the performance as shown below. This happens because the flushing of the old cell and loading of the new cell happens in parallel with the remaining software to be executed for that cell.

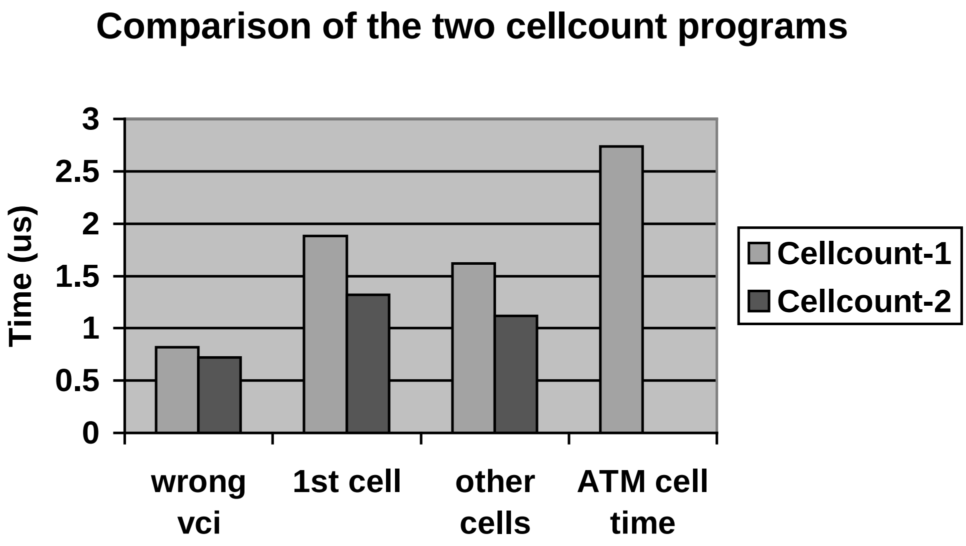

Figure 7 - Comparison of cellcount-1 and cellcount-2.

As can be seen, modifying the software can give quite large improvements to performance, albeit at the cost of making the software rather untidy.

6.3.2 String searching

For the second test, a rather different example is used. Here we use a program to look for a text string in any arbitrary position within a cell. This makes extensive use of the parallel matching facilities and shows that some quite complex matching can be done using this system.

As an example we could search for the string "abcd", although any string could be used as long as it is shorter than the cell length. For this particular string, matching is easy, as the string does not contain the initial letter anywhere within its body. To start with we can build a simple 'one character at a time' finite state machine that searches for this string in a series of bytes until it hits the end of the packer (which is flagged as EOP). This is shown as a simple FSM in figure 8.

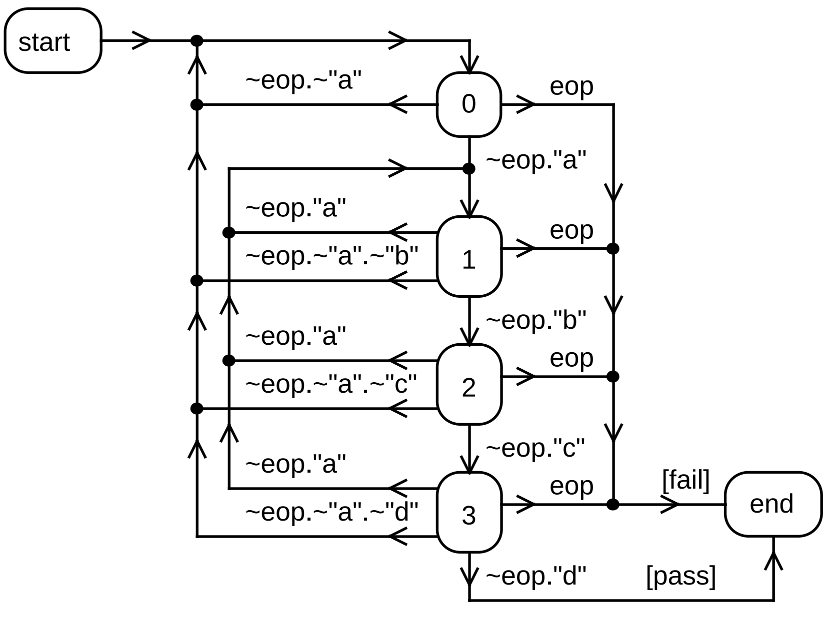

Figure 8 - FSM for a four character pattern match.

In the processor described in this paper, we have a word size of 32 bits or 4 bytes. To maintain the efficiency of this system, we will need to perform matching on up to 4 bytes at a time. To do this, we can re-write our FSM to operate on more than one symbol at a time – this method of improving network monitoring performance is described by Hershey in [10]. This gives us an increase in performance, although we can end up with a rather more complex FSM.

With the above example, in state 0 we will need to search for the string "abcd" as starting at any of the 4 byte positions in a word, this could give us an immediate match for this string if it starts in byte 0, otherwise we can move to states 1,2,3 after matching this number of bytes. In states 1,2,3 we will need to look for the remainder of the string in the next word. However if we do not find the remainder of the string in this new word, we also need to look for the start of a new word as we did in state 0. This system can of course expand to strings of any length, with a corresponding increase in the number of possible states.

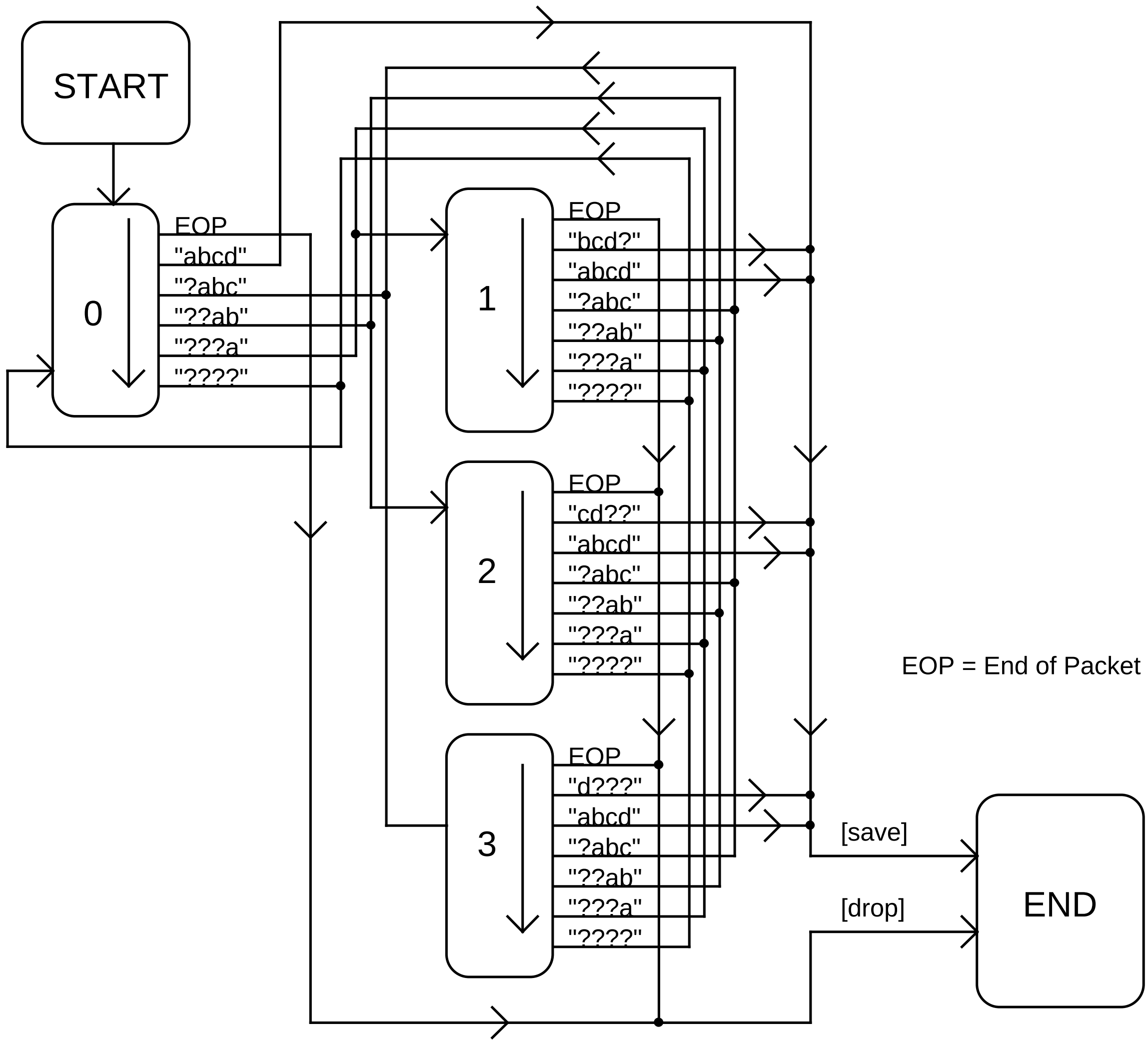

Figure 9 - String pattern matching, 4 bytes at a time.

Here the conditions attached to each transition become more complex. Whilst some of the tests are mutually exclusive, this is not always the case. For state 0 we always move to state END and drop the cell if EOP is set. If EOP is false, then we have four possible string matches that in this case are mutually exclusive, but in the case of repeated letters in the pattern this might not be the case – here we would probably give preference to the longest possible match. Finally if EOP is false and none of the string patterns match, we go back to state 0. For the sake of clarity, I have shown this in Figure 9 with an arrow that indicates the order in which the conditions should be tested – rather than expanding out the logic for each condition.

An interesting point to note is that although we have quite a complex FSM, the FSM will make a maximum of $n+2$ transitions for a packet of length n words. If we increase the length of the string we are searching for, then all that happens is that we increase the number of states in the FSM and increase its complexity, we still operate the pattern matching at the same speed.

For our example program, we search for the string "abcdefgh" in all possible positions within the cell body. The FSM for this is quite large, so is not included here as a schematic. The pseudo-code program to implement this is given in Appendix C.

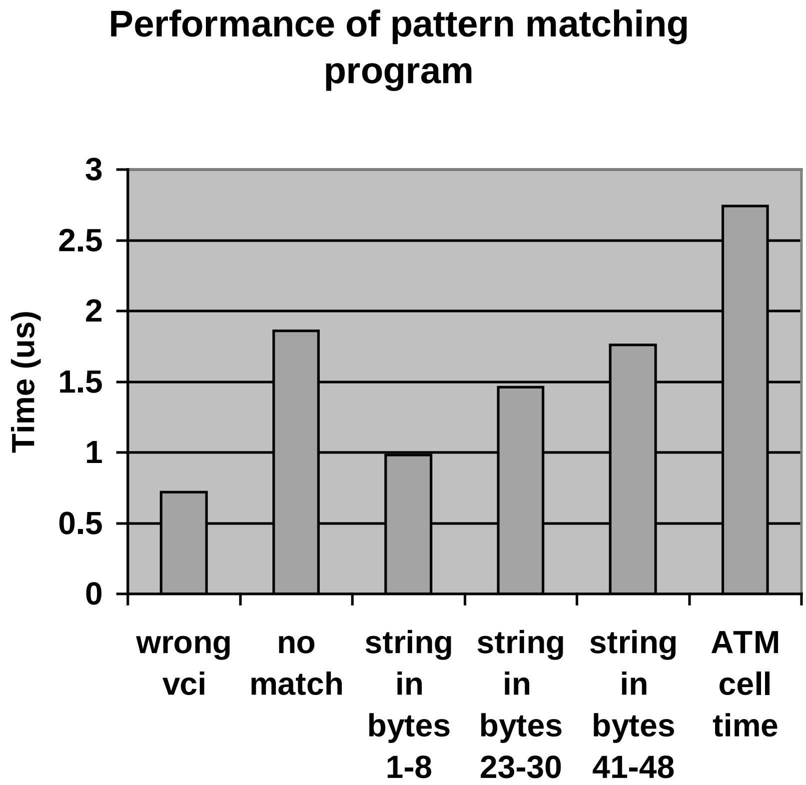

Figure 10 – Performance of pattern matching program (pattern-1).

The worst case performance is when the pattern is not found in the cell as all the network data needs to be examined and end of packet discovered. Assuming that the cell is on the correct VCI, then the best case is when the string appears at the start of the cell, as the remainder of the cell does not require any examination. As can be seen, this simple pattern matching operates faster than the ATM cell time, so should work successfully on an ATM network operating at 155 Mbps.

6.3.3 Per VCI continuous pattern matching

In this final example, the pattern-matching program discussed in the previous sub-section is modified to operate on multiple VCI streams. In addition, rather than performing a pattern match on each cell in isolation, this is extended to treat the bodies of cells on the same VCI as a stream of data on which to perform pattern matching. This is a relatively simple addition to the previous example. All that needs to be done is to have a separate channel for each VCI that is being monitored and then to save the current state of the “pattern match” when the end of the cell is found. Thus we may enter the process for a particular channel with a different initial state depending on what happened during the last cell on that VCI.

As before, we are saving a cell on a successful pattern match, however this time it will be the cell in which the pattern match completed. In addition, the saved cell will be labeled with the channel number on which the pattern match occurred. The pseudo code for this final program is given in Appendix D and the performance figures are shown below.

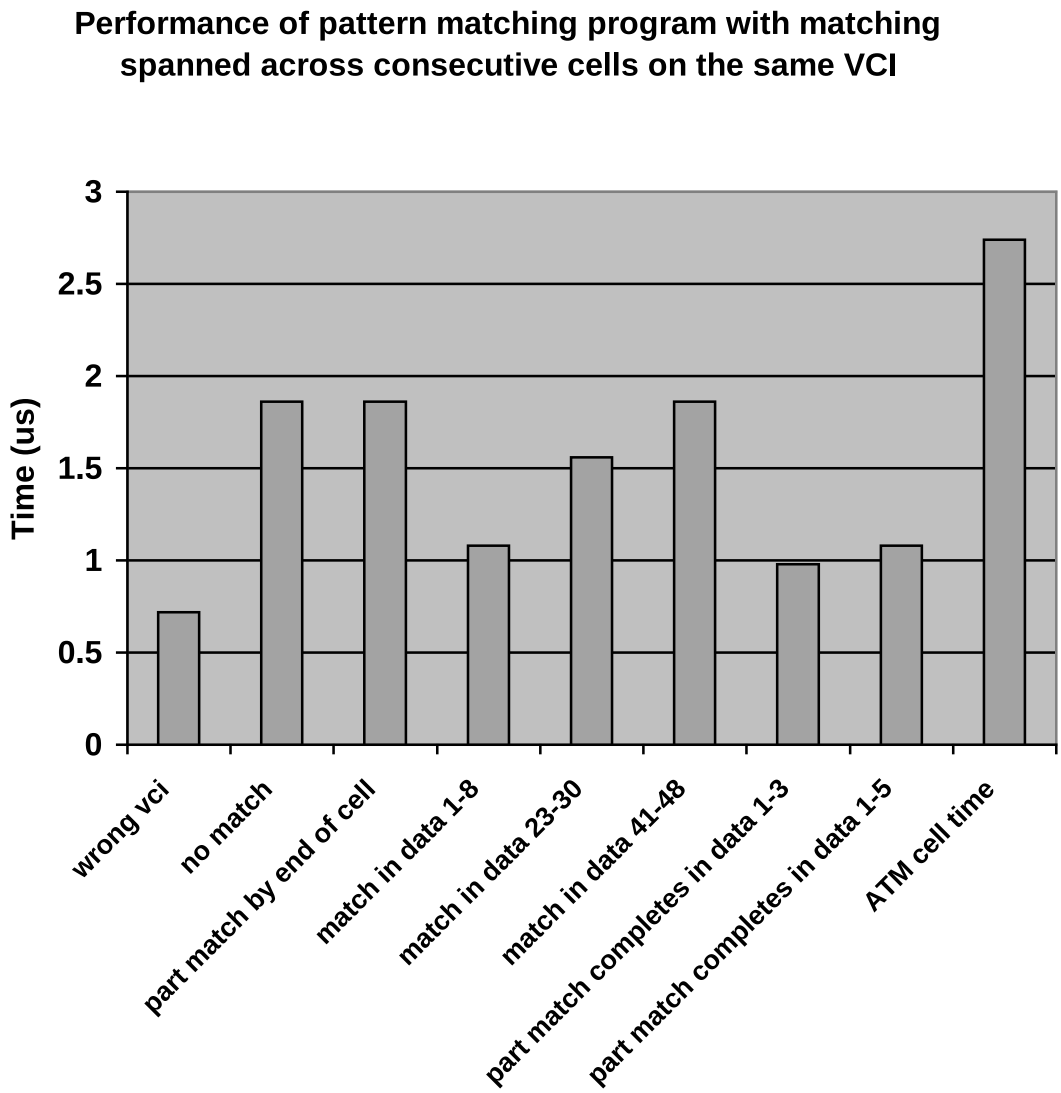

Figure 11 - Per VCI stream pattern matching (pattern-2).

The time for no match to take place is the same as that for the program pattern-1. This time we also have various cases where a part match at the end of a cell has occurred – this is where we hit the end of the cell body part way through a pattern match. The time for all cases of a part match is the same as for the no match as all data in the cell needs to be examined until we hit the end of packet flag. The times for a match to take place within a single cell are higher than for pattern-1, as this time we need to save the value of the current channel number in global memory for return with the cell. The time for part matches to complete are related to the number of words of data examined.

This program scales well to monitor larger numbers of VCI's, as only one extra CAM code word is required per VCI. For longer strings, this scales reasonably well, however for strings where the initial character is repeated with the string body we will require more complex FSMs for the matching.

6.4 Observations on performance

It is instructive to look at how the time spent processing a cell is distributed between the various operations involved. In an ideal world this should consist of or at least be dominated by the time performing searches for data patterns in the CAM. Other operations such as changing process and waiting for caches to fill are just overheads and hence valid targets for optimization. The time taken searching for data is a separate issue that needs to be dealt with.

The following figure shows how the processing time is divided. Note that due to these functions being carried out in parallel there is overlap between the various categories. The time for each of the four categories is accounted for in the following priority.

1 Data search.
2 (a) Actions, (b) Scheduling.
3 Network data control.
4 Waiting for data.

Hence a call instruction that matches network data will be accounted for partly in category 1 and the remainder in category 2(b). If a cache miss occurs, then this will also contribute to category 4.

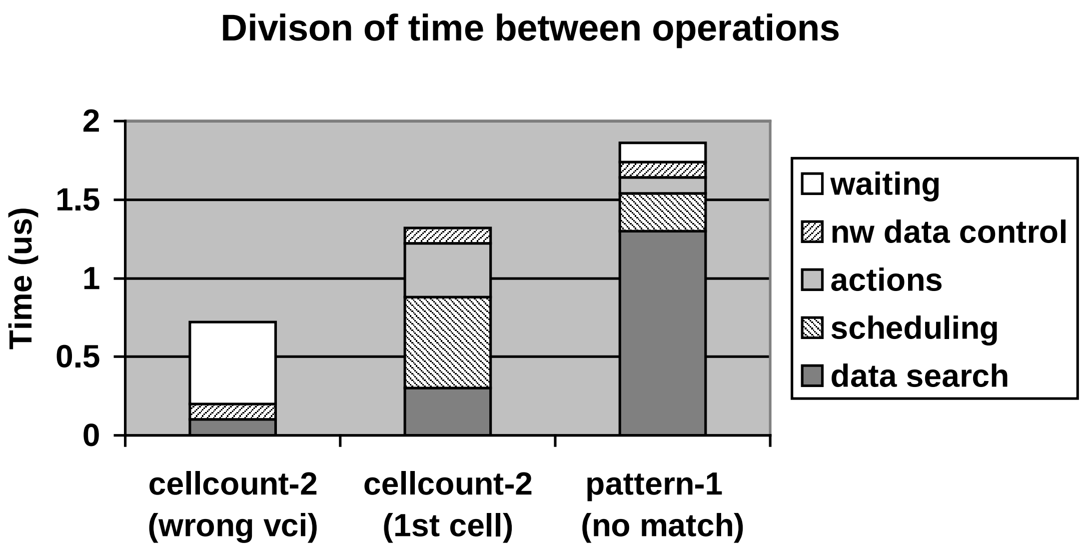

Figure 12 - Break down of execution time between the various operations.

For the pattern-matching example, we can see that the execution time is made up primarily of data searches as would be expected. The cellcount-2 program for the 1st cell of an AAL5 frame spends a lot of time on scheduling as it changes between various processes for each protocol layer. There is also quite a lot of time spent here on various actions, these include operating the counters and also generating output events. The case with cells arriving on the wrong VCI is interesting because it does not really do very much, just check the cell header and drop the cell. This however spends a lot of time waiting as the unwanted cell needs to be flushed through the system and the next cell loaded in its place. The operations for this generally happen in parallel with the last few instructions executed for the cell. When the cellcount-2 program has a cell to process, then the cell flush operations happen in parallel with the scheduling operations. With the pattern matching case we are already at the end of the cell, so very little work needs to be done.

The performance of the system in general is determined by the maximum clock speed that the system can be operated at. The current CAM component used restricts this to 66 MHz (15ns clock period). Using the FPGA noted in section 4.1, and operating with synchronous input and output, we are limited to a minimum clock period of 14ns to allow for the setup times to be met on component inputs. This is less than the minimum clock period required for the CAM, so this presents no problems. The system is however limited by the speed of the logic within the FPGA itself. For the current design, this was constrained to be 20ns, although it is likely that this could be made slightly faster. Two particular pieces of logic that may restrict the speed of this device are the ALU and also the ROM used for holding the micro-code. The ALU has a limit in operation speed due to the long carry chain, if this posed a problem in faster designs then this could be split and the ALU operation performed in two clock cycles. The micro-code ROM may cause problems in performance because the depth of memory (64 locations) requires each bit to be implemented as multiple CLBs and the width (128 bits) requires a high fan-out for the ROM address lines. The design currently uses two sources for the ROM address lines – the number of sources could be increased if required, although this makes the design rather untidy. In general, performance could be improved by fine tuning the design and using more intervention in the FPGA design process – such as using specific library parts and floor planning the design.

6.5 Random vs. Sequential access

From the previous section, it can be seen that sequential access is not always the best method of operation. This can often lead to times where the program is waiting for the correct data to arrive in the cache. As the caching system runs in parallel with the program, this is not always a problem as the cache fill may occur whilst other parts of the program are executing. As we have seen in section 6.3.1, the problem of delays occurring when requesting the next cell can be reduced or even eliminated by modifying the program to request the next cell at the earliest possible time. Delays during the examination of a cell can be reduced by careful arrangement of the changes to the current data item so as to avoid cache misses occurring.

However, an advantage of the sequential access is that it is possible to step through the contents of a packet of data without explicitly specifying an address. This was used in the pattern-matching program that used the same set of states for testing all 12 words of the cell body and only stopped on end of packet. There is also the problem of requiring a process to start operations on a packet in different places – such as might be required if a lower protocol layer had facilities for variable length headers. This is easy with sequential access, but becomes more difficult with random access.

Some of the problems noted in the previous paragraph were covered in [2], where it was suggested that we could use an index register as a base pointer to point at the start of the data for the current process. Here I would recommend that where possible, the implementation method for handling network data should be to read the entire packet into internal memory before processing begins. This enables far more parallelism to be used in the implementation, as we can store multiple packets in memory and be processing a packet whilst the next packet is being read into memory. By using dual port memory, we could use one port for access from the processor and the other port to share between network data in and packet output events. The program access method could be left to the choice of the programmer and combinations of sequential and random access methods could be provided. Caching could still be used to improve performance and to cope with random word alignment if this was required. A disadvantage would be that the overall throughput on the input and output ports would be halved due to the two needing to share access to one port of the dual port memory. The present implementation operates at a maximum input/output rate of 1600 Mbps, so this is probably not a problem. The requirement for large amounts of dual port memory is not easy on the FPGAs currently being used, although this problem would be removed if the design was targeted at one of the newer FPGAs such as the Xilinx Virtex [11] series, which have larger amounts of on chip memory.

6.6 Future versions

A number of ideas come to mind for a revised version of this design. Many of these will depend on availability of tools and also on the advance information provided by manufacturers on new CAM designs. However, a number of general issues are detailed below:

- Network data access. The cache control system used for this processor was actually quite complex to design and it only provides fair performance at best. A move to a random access system as discussed in the previous section is probably advisable.

- Scheduling. Causes quite a lot of overhead in switching tasks. Where possible, this should be improved. May be beneficial to have separate memory space for channel state information, to avoid clashes with other actions. Separate control over scheduling may also be useful.

- Output event generation. A more sophisticated scheme would be useful, such as suggested in [2]. Should be able to generate short events from any process without sending the whole packet.

- Data width. May be worth investigating the possibility of increasing the word size used for data. This will however be affected by the CAMs used in the design – using a 128 bit key with the current CAMs and a separate RAM for results would increase our basic cycle time from 5 to approx. 7 clock cycles.

- Clock rate. Likely to be possible to use a higher clock rate with the next generation of CAM chips. Moving to the Xilinx Virtex [11] series of FPGAs provides a number of features that will help support this.

- Dynamic channel allocation. The architecture of the current design supports this to a limited extent, although at the time of writing more work is needed on the micro-code. Any new version will need to give more consideration to this issue and also the topic of garbage collection of old processes. It is proposed that this topic should be investigated in more detail before the next generation processor is designed.

6.7 Summary

Section 6 has taken a critical look at the performance of this processor and ways in which the performance could be improved. On a positive note, it is interesting to look at the performance figures obtained for the program examples. The most difficult in terms of conventional processing is the program used for pattern matching. This allows us to check the cell header to select the appropriate channel for the VCI and then provides a pattern match within the body of an ATM cell in 1.86us. This corresponds to a data rate of 223 Mbps – i.e. easily fast enough to process a 155 Mbps ATM channel. If we ignore the various overheads, we have a per-word search time of 100ns or 320 Mbps.

If we look at the maximum theoretical throughput of the CAM based system, this would be to use a 64-bit data key and have the results in external memory. If we run the CAM at its top speed, then the time for one 64 bit word is 15 x 7 = 105 ns – i.e. 609 Mbps. The next generation of CAMs looks to be faster and offer a wider memory word – if the current system was updated to use these new devices then it may well have a higher overall throughput.

7 Conclusions

This paper presents the design of an associative processor for use in network monitoring. The design uses three main components: an FPGA, a CAM and a RAM. The design for custom parts of this system have been modeled in register transfer level VHDL and this can be synthesized into logic for a Xilinx FPGA which should operate at 50 MHz. The system has been simulated, both to enable debugging and also to provide timing figures for some example network monitoring programs.

7.1 Results of Evaluation and Comments

The processor described in this paper is a first prototype, however it allows us to deduce that this type of processor is a practical method of implementation for use in the design of network monitoring tools. At present, the usual role of content addressable memory in network interfaces is generally for routing and address translation. The work in this paper shows that it is also more generally useful in network systems for unlocking the inherent parallelism that can be coded in designs defined as finite state machines.

7.2 Further Study

A number of issues have been highlighted in section 6. It appears that if possible the packet being processed should be stored in memory within or closely coupled to the processor and for the processor to have random access to the data. This mechanism can be used irrespective of the access method used by any software running on the processor and should allow programs to execute without delays in fetching the network data. The disadvantage is that memory needs to be made available that can hold the maximum sized packet that could be received from the network. Secondly, the output event generation system could be improved to allow the generation of short output events from any process – it would probably be wise to look at using similar semantics for generation of events to internal and external processes. Thirdly, the issue of dynamic channels needs further investigation. The work required is primarily in the area of garbage collection of idle channels and also highlights the need for the provision of a timeout mechanism. Finally, any future design should aim for yet higher performance – ideas for achieving this have been detailed in section 6.

To enable further tests to take place with real networks it would be interesting to build a prototype physical design of such a system. This could be built as a general-purpose test harness, using a large state-of-the-art FPGA. As the current system is specified in VHDL, it should be easy to retarget this at a newer FPGA technology. This should enable tests to be undertaken with the existing designs, whilst providing the opportunity to upgrade to newer systems as these are developed. Testing designs and software in a real environment should identify shortcomings of current designs and hence help to target effort into useful improvements.

Finally, at present the system relies on specifying monitoring programs in a form of micro-code. Whilst this is acceptable during development, it would be useful to have proper tools to allow code tables to be compiled or synthesized from a high level language.

Acknowledgements

I would like to give acknowledgement to Robert Cole and his colleagues at Hewlett Packard Research Laboratories Bristol, as our source for the idea of using state machine based filtering in network-monitoring systems. The work presented in this paper is part of a separate spin-off project, which has looked at fast methods of implementation for such a system.

References

[1] Gerald Tripp, Real Time Network Traffic Monitoring, Technical Report 5-99, Computing Laboratory, University of Kent, May 1999.

[2] Gerald Tripp, Network Traffic Monitoring - an architecture using associative processing, Technical Report 7-99, Computing Laboratory, University of Kent, September 1999.

[3] K.E.Grosspietsch, Associative Processors and Memories: A Survey. IEEE Micro Vol. 12, No. 3, June 1992, pp. 12-19.

[4] Douglas L. Perry, VHDL, McGraw-Hill, Inc. 1991.

[5] IPCAM-2, NL85721, Ternary Content Addressable Memory (IPCAM™) 8K × 128 Advanced Information, Revision 2.1. Netlogic Microsystems.

[6] 2.25Mb ZBT™ SRAM 128K x 18, 64K x 32/36 LVTTL, FLOW-THROUGH ZBT SRAM. MT55L128L18F.pm6 – Rev. 2/98. Micron Technology, Inc.

[7] Xilinx® XC4000XLA/XV Field Programmable Gate Arrays. Product Specification. May 14, 1999 (Version 1.2), Xilinx Inc.

[8] The ATM Forum Technical Committee, UTOPIA Specification Level 1, Version 2.01, af-phy-0017.000, March 21, 1994.

[9] J.Mick and J.Brick. Bit sliced Microprocessor design, McGraw Hill, 1980.

[10] P.C.Hershey. Information collection architecture for performance measurement of computer networks. Ph.D. Dissertation. University of Maryland College Park, 1994.

[11] Xilinx Virtex™ 2.5V Field Programmable Gate Arrays. Advance Product Specification. February 16, 1999 (Version 1.3). Xilinx, Inc.

Appendix A – Original 'Cell count' program – cellcount-1

```
-- process for cell level, channel 1
cell: process( cs, data )
registers
    cs(chan = 1) <= FETCH;
begin
    case cs is
        FETCH =>
            ns <= ATMCELL;
            DataAction(Go to 1st word of packet);

        ATMCELL =>
            if (data.vpi = 1) and (data.vci = 15) and
               (data.pti /= Tail) then
                -- Cell
                ns <= ATMEND;
                DataAction(Forward 4 bytes);
                call(chan <= 2, event <= stream);
            elsif (data.vpi = 1) and (data.vci = 15) then
                -- Last cell in an AAL5 frame
                ns <= ATMEND;
                DataAction(Forward 4 bytes);
                call(chan <= 2, event <= streamf);
            else
                -- cell not on correct VPI/VCI
                ns <= ATMEND;
            end if;

        -- cell processing complete
        ATMEND =>
            DataAction(Get next cell);
            ns <= FETCH;
     end case;
end process;


-- process to decode AAL5 framing, channel 2
-- all cells on appropriate VPI/VCI
AAL: process( cs, data )
registers
    cs(chan = 2) <= AAL5IDLE;
begin
    case cs is
        AAL5IDLE =>
            if (event = stream) then
                -- first cell of AAL5 frame
                ns <= AAL5IPEND;
                DataAction(Forward 12 bytes);
                uplink := call(chan <= 3, event <= first);
            else
                -- first and last cell of AAL5 frame
                ns <= AAL5END;
                DataAction(Forward 12 bytes);
                uplink := call(chan <= 3, event <= first);
            end if;
        AAL5IPEND =>
            ns <= AAL5BODY;
            return;

        AAL5BODY =>
            if event = stream then
                ns <= AAL5IPEND;
                upcall(chan <= uplink, event <= body);
            else
                ns <= AAL5END;
                upcall(chan <= uplink, event <= body);
            end if;

        AAL5END =>
            ns <= AAL5IDLE;
            return;

    end case;
end process;


-- process to look up IP addresses and count cells
IP: process( cs, data )
registers
    cs(chan = 3) <= IPSRC;
    cs(chan = 4) <= IPNONE;
    cs(chan = 10,20,30);
    cs(chan = 11,12,13,21,22,23,31,32,33);
    cellcount(chan = 11,12,13,21,22,23,31,32,33) <= 0;
globals
    status;
constants
    IP1 <= 0810C1508H;
    IP2 <= 0810C1509H;
    IP3 <= 0810C150AH;
    IP4 <= 0810C1501H;
    IP5 <= 0810C1502H;
    IP6 <= 0810C1503H;
begin
    case cs is
        -- first cell of an AAL5 frame, channel 3
        -- (check IP source)
        IPSRC =>
            if event = first then
                if data.ipsrc = IP1 then
                    DataAction(Forward 4 bytes);
                    ns <= IPDST; goto(chan <= 10);
                elsif data.ipsrc = IP2 then
                    DataAction(Forward 4 bytes);
                    ns <= IPDST;  goto(chan <= 20);
                elsif data.ipsrc = IP3 then
                    DataAction(Forward 4 bytes);
                    ns <= IPDST;  goto(chan <= 30);
                else
                    ns <= IPNONE; goto(chan <= 4);
                end if;
            else
                ns <= IPSRC;
                return;
        end if;

        -- not one of the IP address pairs we were
        -- looking for, channel 4
        IPNONE =>
            ns <= IPNONE;
            return;

        -- first cell of an AAL5 frame, channels 10,
        -- 20, 30 (Check IP destination)
        IPDST =>
            if (chan = 10) and (data.ipdst = IP4) then
                ns <= IPCELL; goto(chan <= 11);
            elsif (chan = 10) and (data.ipdst = IP5) then
                ns <= IPCELL; goto(chan <= 12);
            elsif (chan = 10) and (data.ipdst = IP6) then
                ns <= IPCELL; goto(chan <= 13);
            elsif (chan = 20) and (data.ipdst = IP4) then
                ns <= IPCELL; goto(chan <= 21);
            elsif (chan = 20) and (data.ipdst = IP5) then
                ns <= IPCELL; goto(chan <= 22);
            elsif (chan = 20) and (data.ipdst = IP6) then
                ns <= IPCELL; goto(chan <= 23);
            elsif (chan = 30) and (data.ipdst = IP4) then
                ns <= IPCELL; goto(chan <= 31);
            elsif (chan = 30) and (data.ipdst = IP5) then
                ns <= IPCELL; goto(chan <= 32);
            elsif (chan = 30) and (data.ipdst = IP6) then
                ns <= IPCELL; goto(chan <= 33);
            else
                ns <= IPNONE; goto(chan <= 4);
            end if;

        -- cell on a known IP address pair,
        -- chans 11, 12, 13,  21, 22, 23, 31, 32, 33
        IPCELL =>
            -- First save a copy of old value of register
            -- cellcount for this channel
            status <= cellcount;
            ns <= IPC1;
        IPC1 =>
            -- Write a token to keep current cell
            cellsavetoken <= Keep;
            ns <= IPC3;
        IPC2 =>
            -- Drop subsequent cells
            cellsavetoken <= Drop;
            ns <= IPC3;
        IPC3 =>
            cellcount <= cellcount + 1;
            ns <= IPC4;
        IPC4 =>
            ns <= IPC2;
            return;

    end case;
end process;
```

Appendix B – Modified 'Cell count' program – cellcount-2

```
-- process for cell level, channel 1
cell: process( cs, data )
registers
    cs(chan = 1) <= FETCH;
begin
    case cs is
        FETCH =>
            ns <= ATMCELL;
            DataAction(Go to 1st word of packet);

        ATMCELL =>
            if (data.vpi = 1) and (data.vci = 15) and
               (data.pti /= Tail) then
                -- Cell
                ns <= FETCH;
                DataAction(Forward 4 bytes);
                call(chan <= 2, event <= stream);
            elsif (data.vpi = 1) and (data.vci = 15) then
                -- Last cell in an AAL5 frame
                ns <= FETCH;
                DataAction(Forward 4 bytes);
                call(chan <= 2, event <= streamf);
            else
                -- cell not on correct VPI/VCI
                ns <= FETCH;
                DataAction(Get next cell);
            end if;

        -- No ATMEND state

    end case;
end process;


-- process to decode AAL5 framing, channel 2
-- all cells on appropriate VPI/VCI
AAL: process( cs, data )
registers
    cs(chan = 2) <= AAL5IDLE;
begin
    case cs is
        AAL5IDLE =>
            if (event = stream) then
                -- first cell of AAL5 frame
                ns <= AAL5IPEND;
                DataAction(Forward 12 bytes);
                uplink := call(chan <= 3, event <= first);
            else
                -- first and last cell of AAL5 frame
                ns <= AAL5END
                DataAction(Forward 12 bytes);
                uplink := call(chan <= 3, event <= first);
            end if;
        AAL5IPEND =>
            ns <= AAL5BODY;
            return;

        AAL5BODY =>
            if event = stream then
                ns <= AAL5IPEND;
                upcall(chan <= uplink, event <= body);
            else
                ns <= AAL5END;
                upcall(chan <= uplink, event <= body);
            end if;

        AAL5END =>
            ns <= AAL5IDLE;
            return;

    end case;
end process;

-- process to look up IP addresses and count cells
IP: process( cs, data )
registers
    cs(chan = 3) <= IPSRC;
    cs(chan = 4) <= IPNONE;
    cs(chan = 10,20,30);
    cs(chan = 11,12,13,21,22,23,31,32,33);
    cellcount(chan = 11,12,13,21,22,23,31,32,33) <= 0;
globals
    status;
constants
    IP1 <= 0810C1508H;
    IP2 <= 0810C1509H;
    IP3 <= 0810C150AH;
    IP4 <= 0810C1501H;
    IP5 <= 0810C1502H;
    IP6 <= 0810C1503H;
begin
    case cs is
        -- first cell of an AAL5 frame, channel 3
        -- (check IP source)
        IPSRC =>
            if event = first then
                if data.ipsrc = IP1 then
                    DataAction(Forward 4 bytes);
                    ns <= IPDST;  goto(chan <= 10);
                elsif data.ipsrc = IP2 then
                    DataAction(Forward 4 bytes);
                    ns <= IPDST;  goto(chan <= 20);
                elsif data.ipsrc = IP3 then
                    DataAction(Forward 4 bytes);
                    ns <= IPDST;  goto(chan <= 30);
                else
                    ns <= IPNONE; goto(chan <= 4);
                end if;
            else
                ns <= IPSRC;
                return;
        end if;

        -- not one of the IP address pairs we were
        -- looking for, channel 4
        IPNONE =>
            ns <= IPNONE;
            return;

        -- first cell of an AAL5 frame, channels 10,
        -- 20, 30 (Check IP destination)
        IPDST =>
            if (chan = 10) and (data.ipdst = IP4) then
                ns <= IPCELL; goto(chan <= 11);
            elsif (chan = 10) and (data.ipdst = IP5) then
                ns <= IPCELL; goto(chan <= 12);
            elsif (chan = 10) and (data.ipdst = IP6) then
                ns <= IPCELL; goto(chan <= 13);
            elsif (chan = 20) and (data.ipdst = IP4) then
                ns <= IPCELL; goto(chan <= 21);
            elsif (chan = 20) and (data.ipdst = IP5) then
                ns <= IPCELL; goto(chan <= 22);
            elsif (chan = 20) and (data.ipdst = IP6) then
                ns <= IPCELL; goto(chan <= 23);
            elsif (chan = 30) and (data.ipdst = IP4) then
                ns <= IPCELL; goto(chan <= 31);
            elsif (chan = 30) and (data.ipdst = IP5) then
                ns <= IPCELL; goto(chan <= 32);
            elsif (chan = 30) and (data.ipdst = IP6) then
                ns <= IPCELL; goto(chan <= 33);
            else
                ns <= IPNONE; goto(chan <= 4);
            end if;

        -- cell on a known IP address pair,
        -- chans 11, 12, 13,  21, 22, 23, 31, 32, 33
        IPCELL =>
            -- First save a copy of old value of register
            -- cellcount for this channel
            status <= cellcount;
            -- Finished with current cell, so get the next.
            DataAction(Get next cell);
            ns <= IPC1;
        IPC1 =>
            -- Write a token to keep current cell
            cellsavetoken <= Keep;
            ns <= IPC3;
        IPC2 =>
            -- Drop subsequent cells
            cellsavetoken <= Drop;
            -- Finished with current cell, so get the next.
            DataAction(Get next cell);
            ns <= IPC3;
        IPC3 =>
            cellcount <= cellcount + 1;
            ns <= IPC4;
        IPC4 =>
            ns <= IPC2;
            return;

    end case;
end process;
```

Appendix C - Pattern matching program – pattern-1

```
-- process for cell level, channel 1
cell: process( cs, data )
registers
    cs(chan = 1) <= FETCH;
begin
    case cs is
        FETCH =>
            DataAction(go to 1st word of packet);
            ns <= ATMCELL;

        ATMCELL =>
            if (data.vpi = 1) and (data.vci = 15) then
                -- Cell
                DataAction(forward 4 bytes);
                ns <= FETCH;
                call(chan <= 2, event <= stream);
            else
                -- cell not on correct VPI/VCI
                ns <= FETCH;
                cellsavetoken <= Drop;
                DataAction(get next cell);
            end if;

    end case;
end process;


-- process to search for the text string "abcdefgh" in
-- the cell and to save the cell if the string is found
-- Channel 2

PATTERN: process( cs, data )
registers
    cs(chan = 2) <= PAT0;
begin
    case cs is
        PAT0 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT1 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="bcde" then
                ns <= PAT5; DataAction(Forward 4 bytes);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT2 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="cdef" then
                ns <= PAT6; DataAction(Forward 4 bytes);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT3 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="defg" then
                ns <= PAT7; DataAction(Forward 4 bytes);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT4 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="efgh" then
                ns <= PATEND; cellsavetoken <= Keep;
                DataAction(Get next cell);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT5 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="fgh?" then
                ns <= PATEND; cellsavetoken <= Keep;
                DataAction(Get next cell);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT6 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="gh??" then
                ns <= PATEND; cellsavetoken <= Keep;
                DataAction(Get next cell);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT7 =>
            if (end_of_packet) then
                ns <= PATEND; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="h???" then
                ns <= PATEND; cellsavetoken <= Keep;
                DataAction(Get next cell);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PATEND =>
            -- Start pattern matching from scratch in the
            -- next cell.
            ns <= PAT0;
            return;

    end case;
end process;
```

Appendix D - Per VCI pattern matching program – pattern-2

```
-- process for cell level, channel 1
-- check for cells on various VCIs and make a call to the
-- appropriate channel for that VCI.
cell: process( cs, data )
registers
    cs(chan = 1) <= FETCH;
begin
    case cs is
        FETCH =>
            DataAction(go to 1st word of packet);
            ns <= ATMCELL;

        ATMCELL =>
            if (data.vpi = 1) and (data.vci = 1) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 2, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 2) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 3, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 3) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 4, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 4) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 5, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 5) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 6, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 6) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 7, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 7) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 8, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 8) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 9, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 9) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 10, event <= stream);
            elseif (data.vpi = 1) and (data.vci = 10) then
                -- Cell on a valid VCI/VPI
                DataAction(forward 4 bytes); ns <= FETCH;
                call(chan <= 11, event <= stream);
            else
                -- cell not on a correct VPI/VCI
                ns <= FETCH;
                cellsavetoken <= Drop;
                DataAction(get next cell);
            end if;

    end case;
end process;


-- Process to search for the text string "abcdefgh".
-- Current state is saved between invocations to allow
-- matching across multiple cells on the same VCI.
-- Used for channels 2 to 11.
-- If a match succeeds, then the current cell is saved
-- along with the channel number.

PATTERN: process( cs, data )
registers
    cs(chan = 2,3,4,5,6,7,8,9,10,11) <= PAT0;
    ch(chan = 2) <= 2;
    ch(chan = 3) <= 3;
    ch(chan = 4) <= 4;
    ch(chan = 5) <= 5;
    ch(chan = 6) <= 6;
    ch(chan = 7) <= 7;
    ch(chan = 8) <= 8;
    ch(chan = 9) <= 9;
    ch(chan = 10) <= 10;
    ch(chan = 11) <= 11;

begin
    case cs is
        PAT0 =>
            if (end_of_packet) then
                ns <= PATEND0; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT1 =>
            if (end_of_packet) then
                ns <= PATEND1; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="bcde" then
                ns <= PAT5; DataAction(Forward 4 bytes);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT2 =>
            if (end_of_packet) then
                ns <= PATEND2; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="cdef" then
                ns <= PAT6; DataAction(Forward 4 bytes);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT3 =>
            if (end_of_packet) then
                ns <= PATEND3; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="defg" then
                ns <= PAT7; DataAction(Forward 4 bytes);
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT4 =>
            if (end_of_packet) then
                ns <= PATEND4; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="efgh" then
                ns <= MATCH;   status <= ch;
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT5 =>
            if (end_of_packet) then
                ns <= PATEND5; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="fgh?" then
                ns <= MATCH;   status <= ch;
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;
```

```
        PAT6 =>
            if (end_of_packet) then
                ns <= PATEND6; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="gh??" then
                ns <= MATCH;   status <= ch;
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        PAT7 =>
            if (end_of_packet) then
                ns <= PATEND7; cellsavetoken <= Drop;
                DataAction(Get next cell);
            elsif data="h???" then
                ns <= MATCH;   status <= ch;
            elsif data="abcd" then
                ns <= PAT4; DataAction(Forward 4 bytes);
            elsif data="?abc" then
                ns <= PAT3; DataAction(Forward 4 bytes);
            elsif data="??ab" then
                ns <= PAT2; DataAction(Forward 4 bytes);
            elsif data="???a" then
                ns <= PAT1; DataAction(Forward 4 bytes);
            else
                ns <= PAT0; DataAction(Forward 4 bytes);
            end if;

        MATCH =>
            -- Keep current cell and request next
            cellsavetoken <= Keep;
            DataAction(Get next cell);
            ns <= PATEND0;

        -- Save current state, so we can continue the
        -- search in the next cell on this VCI.

        PATEND0 =>
            ns <= PAT0; return;
        PATEND1 =>
            ns <= PAT1; return;
        PATEND2 =>
            ns <= PAT2; return;
        PATEND3 =>
            ns <= PAT3; return;
        PATEND4 =>
            ns <= PAT4; return;
        PATEND5 =>
            ns <= PAT5; return;
        PATEND6 =>
            ns <= PAT6; return;
        PATEND7 =>
            ns <= PAT7; return;

    end case;
end process
```