FORTH-ICS / TR - 257 June 1999

IP Routing Table Organization and Management in the Wormhole IP Routing Filter

Giorgos Sapountzis

Abstract

Exisitng solutions to the IP over ATM (or ATM under IP) problem suffer from at least one of the following problems: (i) the first packets of a flow before the flow is identified and the packets that cross the boundaries of ATM subnetworks undergo long latency, (ii) the set of flow identifiers gets exhausted and the solution does not scale well. Performing fast IP address lookup is a key element to solving the above problems successfully as described in [8]. In this report we review the relevant bibliography and evaluate the proposed schemes for IP routing table (RT) organization; our metrics are set in compliance with the demands of the wormhole IP [8] FPGA-based routing filter prototype. We provide details on the route lookup operation and we analyze the complexity of the insert/delete prefix operations; a novel algorithm for extending the update engine to routing tables with more than 2 distinct prefix lengths is proposed. We, also, provide a thorough description of the datapath and control of the hardware blocks used in the system.

IP Routing Table Organization and Management in the Wormhole IP Routing Filter

Giorgos Sapountzis [1]

Institute of Computer Science,
Foundation for Research and Technology – Hellas,
P.O. Box 1385, 711-10, Greece.
E-mail: sapunjis@ics.forth.gr

Technical Report FORTH-ICS/TR-257 June 1999

Work performed as a B.Sc. Thesis at the Univ. of Crete

Abstract

Exisitng solutions to the IP over ATM (or ATM under IP) problem suffer from at least one of the following problems: (i) the first packets of a flow before the flow is identified and the packets that cross the boundaries of ATM subnetworks undergo long latency, (ii) the set of flow identifiers gets exhausted and the solution does not scale well. Performing fast IP address lookup is a key element to solving the above problems successfully as described in [8]. In this report we review the relevant bibliography and evaluate the proposed schemes for IP routing table (RT) organization; our metrics are set in compliance with the demands of the wormhole IP [8] FPGA-based routing filter prototype. We provide details on the route lookup operation and we analyze the complexity of the insert/delete prefix operations; a novel algorithm for extending the update engine to routing tables with more than 2 distinct prefix lengths is proposed. We, also, provide a thorough description of the datapath and control of the hardware blocks used in the system.

`This text is available in postscript form, via anonymous ftp, from`
`ftp.ics.forth.gr/tech-reports/1999/1999.TR257.RTOrgMng.ps.gz`

[1] The author is also with the Computer Science Department, University of Crete, Greece.

Contents

1 Introduction

The Internet was originally designed as a hierarchy of subnetworks; three classes of subnetworks were identified: A, with up to 16M nodes; B, with up to 64K nodes; and C, with up to 256 nodes. This approach, and the accompanying address allocation strategy, led to an inefficient use of the IP address space. As a result, a new allocation strategy was introduced: subnetworks were assigned bundles of class C addresses rather than a class B address; the side effect of this strategy was a considerable increase in the amount of routing information stored in IP routing tables and in the information exchanged between IP routers because a unique subnetwork outside the local domain required many route entries. So, CIDR (Classesless InterDomain Routing) [1] was deployed in order to reach a compromising solution between efficient use of IP address space, on one hand, and the amount of IP routing information, on the other hand.

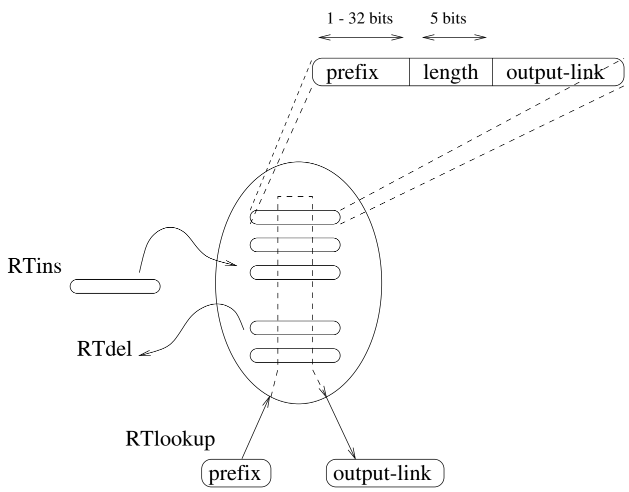

Figure 1: Abstract data type for a CIDR-compatible IP routing table

The key idea behind CIDR is to unify route entries for consecutive addresses allocated to the same subnetwork; this compresses the information for other domains stored in a domain's routing database but, also, modifies the address lookup operation since a set of consecutive addresses may contain "holes" allocated to other subnets. The routing table is a set of route entries whose format is shown in Fig. 1. When an IP address lookup is initiated, we first identify the set M of route entries, whose first *length* bits of *prefix* [2] match the first *length* bits of the address, we then choose from set M the *longest-matching* route entry LMP, the right output link is the one associated with LMP. Longest-matching is inherent in routing and it was deployed as soon as it was clear that the early simple schemes would not be able to cope with Internet growth.

Wormhole IP [8] is an innovative way for carrying IP packets over conventional ATM switches, by embedding routing filters at the periphery of existing ATM subnetworks; each filter manages a number of predetermined VCs in order to carry IP packets segmented to ATM cells according to AAL5. For each cell, a wormholeIP routing filter has to modify an entry in the connection table, perform an IP address lookup, and, probably, free or allocate a VC in the list of VCs maintained for each outgoing VP. The implementation of the filter's core is described thoroughly in [9]. High-speed

[2] *length, prefix* are the corresponding fields of the route entry

routing is key element for constituting the above solution to the *IP over ATM* problem efficient (minimization of the latency phased by IP packets, scalability), the main idea to achieve this is proposed in [5] and is a hardware-oriented application of controlled prefix expansion [4].

This report describes the IP routing table component of the wormholeIP routing filter and is organized as follows. In Section 2 we review and evaluate the proposed schemes for organizing the routing table, we also provide details on the route lookup operation of the selected scheme. Section 3 analyses the insert, delete operations; the software and hardware components of the routing table update engine are described thoroughly, as well. In Section 4 we propose a novel algorithm for extending the RT management scheme to deeper and smaller RTs. Section 5 presents auxiliary circuits of the update engine.

2 Routing Table Organization

First of all, we need to decide on a set of metrics for comparing algorithms; we, then, explore the design space of routing table lookup algorithms, list the feasible components of the design space and choose the suitable alternative for the FPGA-based filter prototype.

The set of chosen metrics quantifies wormohole IP objectives and the requirements of wormhole IP potential applications; chosen metrics are listed below:

- throughput of RT lookups: throughput should be at least 1 lookup(s)/cell-time; the higher the link bandwidth, the smaller the cell-time and the higher the minimum requirement on lookup throughput.
- latency of a RT lookup: this metric is critical because minimization of IP packet latency [3] through the filter is one of the main objectives of wormhole IP and latency of an RT lookup is a lower bound for the latency imposed by the filter.
- hardware cost: wormhole IP attempts to solve the IP over ATM problem at low cost, therefore, we should take into account the processing power (embedded microprocessor, FPGA) and memory requirements of each algorithm.
- implementation complexity: cell processing and per-output-list management are performed in hardware by the filter's core for minimization of latency; we would like to present to the filter's core a simple interface in order to minimize design effort.
- throughput of insert/delete prefix operations: most algorithm designers concentrate on the optimization of RT lookups and pose no limitations on the properties of insert/delete operations. The most important application of wormhole IP is expressed by the following equation: wormholeIP routing filter + ATM switching fabric = high performance IP router, that is, a backbone router. As stated in [4], approximately 10 msec are available, on average, for each update in a backbone router (the trace used in [5] is even more unstable: in the worst case 291 updates per second are generared), therefore, it is important to achieve high RT update throughput; however, it is likely that the upper time bound for wormhole IP is greater than 10 msec because the RT is distributed among incoming links.

[3] The source of IP packet latency is twofold: (1) latency imposed by the filter (2) segmentation-and-reassembly latency

High-speed routing lookups is a challenging problem, but, fortunately, a lot of research has been conducted in the field recently [2] [3] [4] [5] [6]. On the basis of the approach followed in [4], the parameters of the design space are:

- basic data structure: the most appropriate data structures for the organization of the RT seem to be (1) the *trie*, (2) a set of *hash trees*, one per prefix length. The latency of RT operations is dominated by memory accesses rather than computation; so, in terms of memory accesses, the cost of a RT lookup on both the trie data structure, and the one-hash-tree-per-prexif-length data structure is O(W), where W denotes the maximum prefix length, 32 for IPv4, 128 for IPv6 .
- "generic" techiques: (1) *controlled prefix expansion* [4] reduces the number of distinct prefix lengths from W to L; the upper bound for RT lookup latency becomes O(L) memory accesses for both multibit tries and hash-trees-per-prefix-length. (2) binary search: this technique has been applied only to hash-trees-per-prefix-length [2]: rather than search length levels serially, starting from the one with the longest prefixes, we perform a binary search on them in order to find the longest matching prefix; cost is reduced from O(L) to O($log_2 L$). (3) compression: both data structure specific techniques and dynamic programming can be used in order to reduce data structure size so that it fits in a smaller and *faster* memory.
- "local" optimizations: techiques applicable to specific implementations.

The feasible alternatives of the design space are:

WVTP [2]: software-based implementation of binary search on hash-trees-per-prefix-length with controlled prefix expansion. According to [4], optimized implementations (Pentium microprocessor with 1 MB of L2 cache) achieve an average time of $190 nsec$ for lookups in a backbone router environment. Since no incremental algorithms are available for insert/delete operations, updates are performed by accumulating a number of updates and rebuilding the database.

SrVa-MT [4]: software-based implementation of multibit tries [4]. According to [4], optimized implementations (Pentium microprocessor with 1 MB of L2 cache) achieve a worst-case time of $181 nsec$ for lookups and a worst-case time of $2.5 msec$ for insert/delete operations.

GuLK [5] hardware-based brute-force implementation of two-level multibit tries. IP lookups can be perfomed at DRAM access speed and, in the worst case, updates require 20208 [5] memory modifications per second, that is, 0.055 memory modifications per cell time at the rate of 155.52 Mbps.

MoSj [6] hardware-based implementation of multibit tries. We consider that this category consists of multibit tries with 3 to 5 distinct prefix lengths that take up to 2 MB and, thus, can fit in SRAM. RT lookups can be performed at SRAM access speed (if pipelining is deployed).

Hardware-based solutions have significant advantages over software-based solutions: very high lookup throughput can be sustained, in fact, if pipelining is deployed we can perform one lookup

[4] We do not consider scheme [3] which is a multibit trie with compression.. For a comparative study of multibit tries, see [4]

every memory (DRAM or SRAM) access time; they have a small and fixed latency which not only minimizes the delay imposed by the filter to cells but allows for a uniform design of the filter's core and eliminates the need for cell buffering; they do not require an embedded processor, at least for the lookup operation, and they meet the constraints for update operations. Thus, the GuLK and MoSj schemes seem to be the most appropriate ones for the routing filter. Both GuLK and MoSj schemes, in principle, constitute very good alternatives; in order to reach a final decision, comparison of the two schemes as independent blocks does not suffice, we need to further elaborate on the RT organization and management block as a subsystem of the wormholeIP routing filter. GuLK and MoSj schemes differ in the complexity of the update operations and their requirements on memory throughput and size.

The intented model for the update engine is the following: there is a software component that runs on a microprocessor and a harware component implemented in the filter's FPGA. The hardware component implements primitives that can be used by the software component to modify the filter's RT, the software component implements a routing protocol and *translates* updates generated by the routing protocol to the primitives offered by the hardware component; messages from the former to the latter component flow within ATM cells. The above model introduces the following cost-performance trade-off: the simpler the hardware component (no need for a microcontroller on the filter's board), the higher the overhead imposed on the network and the microprocessor. The network overhead is given by the following equation:

$$NO = \frac{0.0424 * M}{p * R} \tag{1}$$

where M is the number of messages per second exchanged between the two components, p is the number of messages packaged in the same cell and R is the link bandwidth. As measured in [5], for a backbone router, 291 update messages can be generated per second; thus, for an $155.52 Mbps$ link and a packaging factor of 2, the minimum value of network overhead achieved is 0.04 %.

The update engine for the GuLK scheme is rather simple and its complexity is analyzed in section 3.3, it requires that the software component sends 1 message to the hardware component for each update generated by the routing protocol and it places a neglible overhead to the microprocessor. The update engine for the MoSj scheme is a generalization of the GuLK update engine, as proposed in section 4 since the MoSj scheme employes 3 or 4 levels, each update generated by the routing protocol results into L messages to the hardware component (L denotes the number of RT levels), the overhead placed on the microprocessor is also increased but in a controlled fashion.

We now turn to the memory bandwidth requirements of each scheme. A bufferless routing filter accesses the following data structures:

- VCs per VP free lists: fit in SRAM within the FPGA
- Connection Table: 2^{17} entries, 32 bits each (512 KB).
- Routing Table - GuLK: 16M entries, 16 bits each and 0.5M entries, 15 bits each.
- Routing Table - MoSj: 653KB (4 levels), (1933KB) (3 levels)
- Routing Table free lists: are maintained within the free blocks of each level.

Thus, the structures that are kept in off-chip memories are the CT and the RT, and in the worst case, within one cell time the filter's core performs *serially* 1 RT lookup (which requires one memory access at each level of the RT), 1 access at internal SRAM to find a free VC and modifies 1 entry at the CT. To economize on I/O pins, we consider only not-pipelined memory organizations that use 32 data pins. The following configurations seem to be the most appropriate choices for the design parameters:

- GuLK with one EDO/FPM DRAM 16M x 32: The two levels of the RT and the CT are placed in DRAM. The GuLK scheme allows for a simple hardware component of the update engine. Bi-directional operation at $155.52 Mbps$ can be achieved so that *one* wormhole IP board connects each IP node to the ATM subnetwork. If we were to use SDRAM rather than EDO/FPM SRAM, uni-directional operation at $622.04 Mbps$ could be supported.

- GuLK with SDRAM 16M x 16 and SRAM 2 x 0.5M x 16: With this memory organization bi-directional operation at $622.04 Mbps$ can be achieved. The SDRAM chip is allocated to the first level of the RT, while the second level of the RT and the CTs for the two directions are placed in 2MB of SRAM. This configuration requires 20 more address pins than the cost-effective GuLK configuration. The above idea is demonstrated in Fig. 2

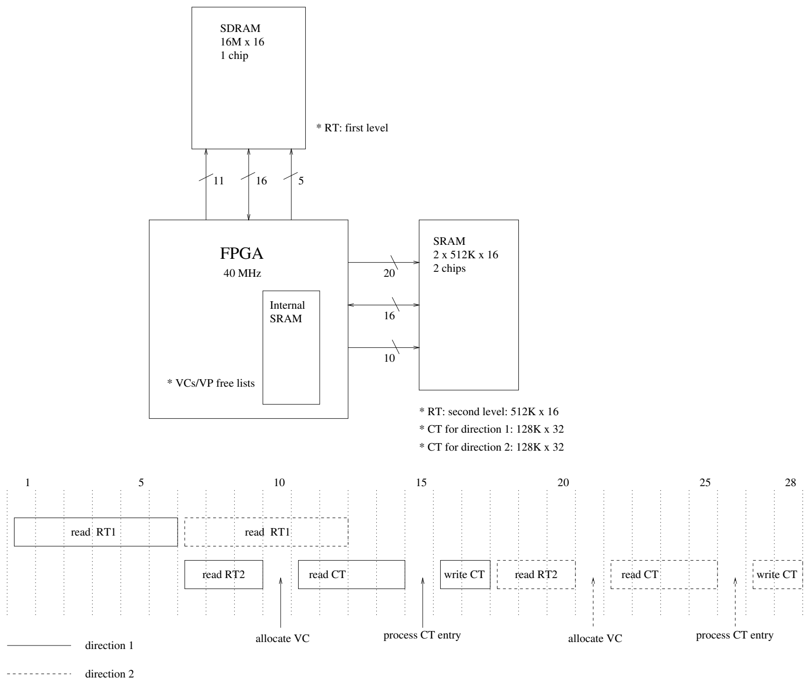

Figure 2: Memory organization and access timing within 1 cell time for bi-directional operation at $622.04 Mbps$. Each RT entry is 16 bits long, whereas each CT entry is 32 bits long. We suppose that the first access at SRAM takes 3 cycles and from then on we can perform 1 access per clock cycle. Thus reading a CT entry costs 4 cycles and writing it back costs 2 more cycles. Notice that SDRAM is occupied only 12 out of 28 cycles and SRAM 22 out of 28 cycles; the remaining cycles are used for RT management.

- MoSj with two SRAMs: 0.5M x 16 and 0.5M x 16: Bi-directional operation at $622.04 Mbps$. All of the 4 levels of the RT fit in 1 MB of SRAM. The second MB of SRAM keeps the CTs for the two directions. This organization requires 8 more pins than the high performance GuLK configuration and a more complex RT update engine. We see no reason for using this scheme in an FPGA-based implementation.

In conclusion, we think that a low-cost FPGA-based wormholeIP prototype should be organized as GuLK with EDO/FPM DRAM 16M x 32.

2.1 Futher Details on the GuLK Basic Scheme

We, now provide details on the IP address lookup operation of the GuLK scheme, and the list of free blocks of the second level of the RT; the complexity of the update operations is discussed in section 3.3.

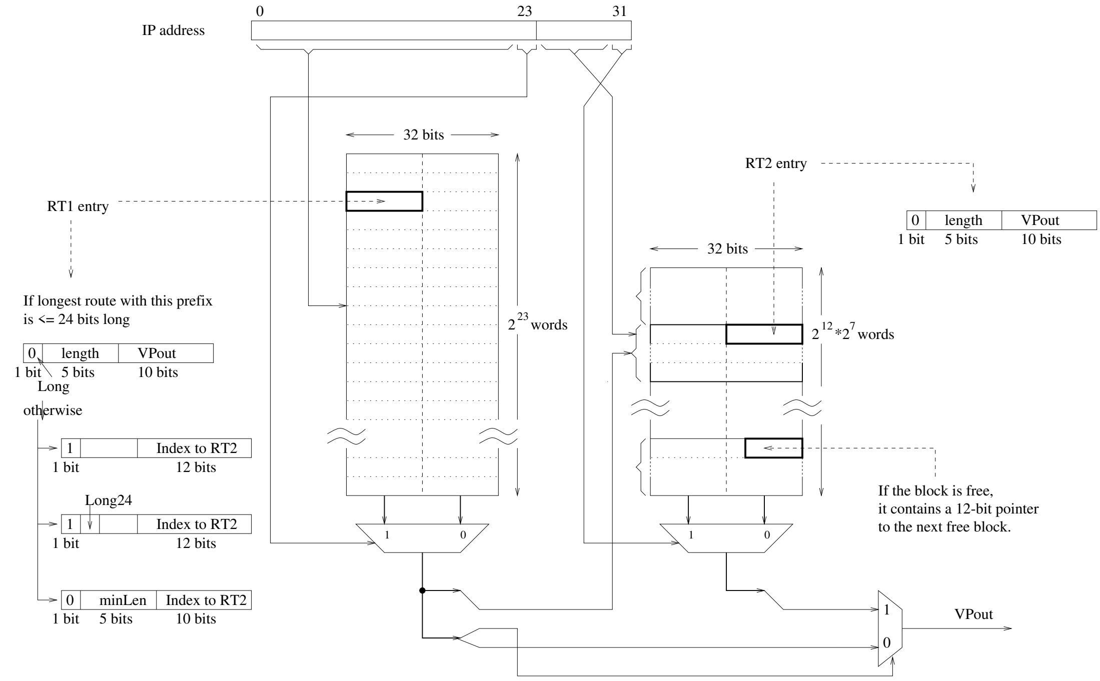

Figure 3: RT lookup operation for the GuLK scheme

Fig. 3 shows the organization of the RT when both the first (RT1) and the second level (RT2) are placed in a 16M x 32 DRAM. Each route that is less than or equal to 24 bits long is expanded to routes with $length^5 + 1 = 24$ and potentially determines the value of $2^{23-length}$ consecutive entries of RT1. To handle with the case that 2 or more routes have their first 24 bits in common, but designate different VPout as next-hop, a second level for the RT (RT2) is employed. RT2 is a set of 2^{12} blocks, each consisting of 2^8 entries and the free list is maintained within the set of free blocks. One RT2 block is allocated for each set of routes that have their first 24 bits in common, and a pointer to the block is placed at the *Index-to-RT2* field of the corresponding entry at RT1. Each route with $24 < length + 1 \leq 32$ is expanded to routes that are 32 bits long and potentially determines the value of $2^{31-length}$ consecutive entries within the RT2 block.

When an IP address lookup is performed, if there are no routes longer than 24 bits that match the address, then the next-hop associated with this address is found at RT1, otherwise, we use the first 24 bits of the address to obtain a pointer to the RT2 block associated with the set of routes that match the first 24 bits of the address, and the remaining 7+1 bits to locate the longest matching entry.

When the entry at RT1 points to a RT2 block, a possible optimization is to maintain either a flag that indicates whether each route associated with the RT2 block is longer than 24 bits (*Long24*)

[5] Each route has a 5-bit *length* field associated with it; for routes that are L bits long, it holds that $length = L - 1$

or the minimun length of the latter routes (*minLen*). This saves memory accesses but complicates RT management; since we expect that the savings are not significant, we do not implement either of the optimizations.

3 Routing Table Management Scheme

3.1 Analysis

The intented model for building the update engine was outlined in section 2 and the main implementation alternatives, proposed in [5], are depicted in Fig. 4.

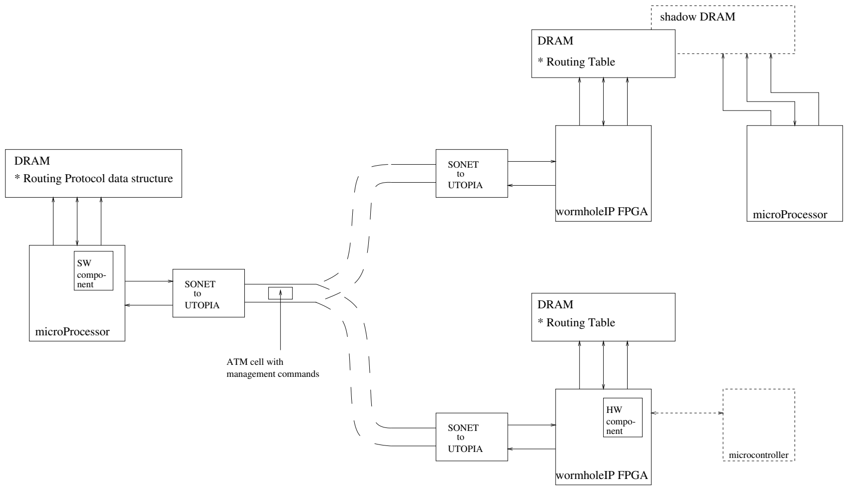

Figure 4: Possible implementations of the update engine

- microP-based: this organization, shown at the upper-right of Fig. 4, is simple but costly to implement: the updates are performed by software running on a microprocessor: it requires a microprocessor dedicated to the update task. In order to enable both the FPGA and the microprocessor to access the RT, we should either double the bandwidth of the DRAM that holds the RT, or use on-board TTL line drivers with 3-state-output that increase the capacitance of the address, data and control lines. Moreover, the latency between the generation of an update by the routing protocol and the time that the update becomes visible to wormholeIP is increased but the increase may not be crucial for the operation of the filter.
- FPGA-based: this organization, shown at the bottom-right of Fig. 4, is applicable only if the memory throughput *suffices* for performing all the basic operations within one cell time (lookup, CT entry modification) and there is still enough time for meeting the upper bound for worst-case update time. The evaluation of the alternatives can be found in [5] and is summarized below:

row update:

- significant microP overhead

- significant network overhead, unless we move the software component of the update engine on the wormholeIP board so that traffic is local

+ the throughput "stealed" from the DRAM that keeps the RT is the miniminum achieved with the GuLK scheme

+ very simple hardware component

subrange update:

- high microP overhead

+ low network overhead

+ the throughput "stealed" from the DRAM that keeps the RT is the miniminum achieved with the GuLK scheme

+ simple hardware component

one-instruction update:

+ low microP overhead

+ low network overhead; moreover the network traffic is *guaranteed* to be equal to the minimum one achieved with the GuLK scheme

+ the throughput "stealed" from the DRAM that keeps the RT is low

? the complexity of the hardware component needs investigation

Some additionial observations on the update schemes follow: there is no need to consider the optimized one-instruction scheme since (1) even with the additional *length* field, the RT entry fits in 16 bits (2) whenever an update is performed, the optimized one-instruction scheme starts accessing memory at the first RT entry affected by the update (opening parenthesis) until the last RT entry affected by the update (corresponding closing parenthesis); the simple one-instruction scheme can support these optimizations as long as the software component implements the necessary functionality. (3) the maintenance of parentheses is rather subtle and should be moved to the software component. In section 3.3, we analyse the complexity of the hardware component of the *one-instruction update* scheme; we show that it exhibits significant uniformity and, thus, can be realized with a subset of an FPGA without the use of a microcontroller. The *subrange update* scheme achieves the minimum requirement on DRAM throughput but places no upper bound on microP and network overhead, the *one-instruction update* scheme offers the minimum possible upper bound on network overhead and a low requirement on DRAM throughput but this requirement can potentially increase without constraint. The source of microP and network overhead in the subrange scheme is the number of "holes" in a RT, the source of memory accesses in the one-instruction scheme is the changes of routes with small length, that is routes to backbone networks; in future networks, it is more possible that there will be an increase in RT "holes" rather than in the number of route changes for backbone networks.

From the above analysis, we can conclude that the *one-instruction* scheme is the more attractive alternative for wormholeIP but we sould like to enhance the scheme with the "minimum DRAM throughput requirement" property. We now describe an idea for bounding the number of memory accesses in the one-instruction scheme: the idea is to build an auxialary data structure that will enable the one-instruction scheme to skip RT entries whose value is determined by longer routes. Our objective can be achieved with a list that contains two nodes (START, END) for each route. The following conditions hold on the list:

1. for every route A: START(A) is before END(A) and START(A) has a pointer to END(A)
2. for two routes A, B that do not overlap and prefix(A) < prefix(B): START(A) is before END(A) which is before START(B) which is before END(B)
3. for two routes A, B such that route B is nested in route A: START(A) is before START(B)

which is before END(B) which is before END(A)

4. for two routes A, B such that length(A) = length(B): we deem that the newest route is nested in the older route

The above list is maintained in a memory accesible by the hardware component of the update engine and is managed in software (through commands send within ATM cells), it requires 2 node insetions or deletions per routing table update and its size is O(#routes). The hardware component either updates RT entries between two consecutive nodes of the list or skips RT entries between the START and END nodes of a nested route.

3.2 Operations and their Format

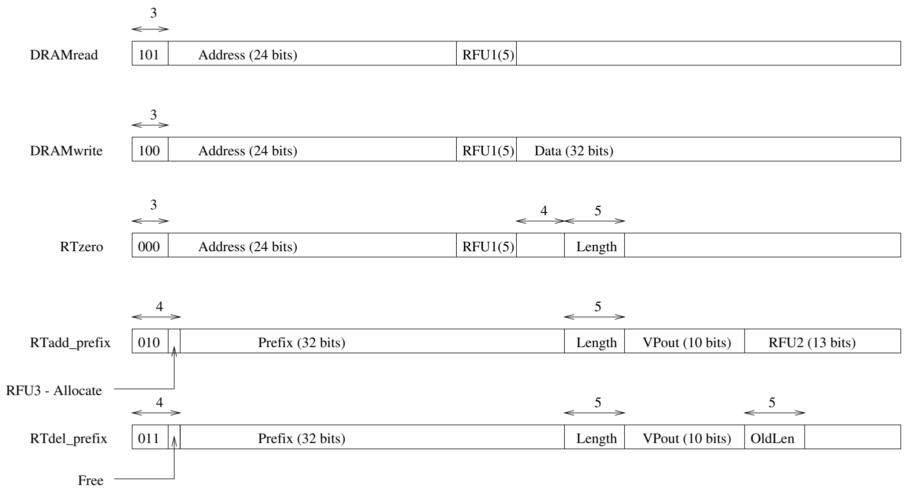

Figure 5: Format of supported operations

The protoype board contains two DRAM SIMMs, 8M x 32 bits each, thus, the address space is 2^{24} consecutive words; *DRAMread* and *DRAMwrite* are used to read and write any word within the address space of our FPGA-based filter prototype. The *RFU1* fields were Reserved for Future Use in case the address space of the routing filter increases.

RTzero nullifies the set of consecutive words starting from the address $a_{23}a_{22}a_{21}...a0$, specified in the *Address* field of the instruction, and ending to address $a_{23}a_{22}...a_{22-length}11...1$ ($0 \leq length < 23$). *RTzero* is used for initializing the CT and the RT; no other initialization operations are supported, non-zero values are set with the help of *DRAMread* and *DRAMwrite*; initialization of the list of blocks at RT2 falls in the latter category.

RTadd_prefix is issued by the software component when the routing protocol inserts a new route, suppose for subnet *sn1*, in its routing table. The *length* field depends on the number of consecutive "1" at the beginning of the subnet mask for *sn1*, for example, for a class B network $length+1 = 16$ [6] . *RTadd_prefix* supports an optimization which is demonstrated by the following example: suppose that we want to insert route A $a_0a_1a_2...a_{25}000000/25$ [7] and the software component computes that,

[6] routing protocols, also, allow for arbitrary patterns for the subnet mask but this capability was never used in implementations

[7] the notation $a_0a_1a_2...a_{31}$/L indicates a route with prefix $a_0a_1a_2...a_{31}$ and $length = L + 1$

because of nested routes, the first IP address affected by route A is $a_0a_1...a_{25}010000$, then, we can set the *prefix* field to the latter value and save memory accesses. Note that our instruction format does not support an analogous optimization for the last IP address affected by route A, therefore, we expect that our implementation of the one-instruction scheme performs more memory accesses than the optimized one-instruction scheme. The first 24 bits of the *Prefix* field are used to access the first level of the RT; since each memory word holds two RT entries, we shift the *Prefix* field one bit right with regard to the *Address* field of the DRAM operations in order to avoid using a multiplexor. *RTadd_prefix* is also used for route modifications.

Suppose that route A is removed by the routing protocol and route B is the route that now determines the next-hop for the set of IP addresses that were affected by route A, we call route B the *father* route of route A, in this case, the software component issues an *RTdel_prefix*. *OldLen* is set to the length of route A, *Length* is set to the length of route B (*Length* $\leq$ *OldLen*) and *Prefix* is set to the prefix of route A, or even to the first IP address affected by A. Note that the information for the length of the *father* prefix may have been dismissed from the wormholeIP RT, this is demonstrated by the following trace of RT updates: insert $a_0a_1...a_{28}000/28$, insert $a_0a_1...a_{28}000/29$, insert $a_0a_1...a_{28}100/29$, delete $a_0a_1...a_{28}000/29$.

The second level of the RT is a list of blocks, *RTadd_prefix* with $Length + 1 > 24$ potentially allocates a block at RT2 and *RTdel_prefix* with $OldLen + 1 > 24$ and $Length + 1 <= 24$ potentially causes a block to be freed. The flags *RFU3-Allocate*, *Free* indicate whether a block needs to be allocated or freed after a *RTadd_prefix* or a *RTdel_prefix* operation, respectively and RFU4 provides the address of a free block at the second level of the RT. Our implemention uses the *Free* field whereas reserves *RFU2, RFU3* fields for future use in case we decide to manage RT2 list entirely in software.

3.3 Complexity Analysis of the one-instruction Update Scheme

In this section we present the flow diagrams of the update operations; we show that each operation can be decomposed in a set of rather simple primitives and that the latter sets have many primitives in common. We also decide on the allocation of primitives to hardware or software components depending on their time critical character and complexity. Keep in mind that the flow diagrams are produced under the assumption that the RT is kept in a 16-bit wide memory, the subtles induced by the 2-entries-per-word RT orginization are discussed in section 3.4.

Two cases need to be considered for the *add_prefix* operation, we name these cases *ADD* (Fig. 6) and *ADD_LONG* (Fig. 7). The following terminology is used in the flow diagrams of the *add_prefix* operation:

- subcases: we call *ADD* and *ADD_LONG* the *subcases* of the *add_prefix* operation.
- primitives: each subcase can be assembled as a number of phases, which we call *primitives*; Iteration-over-RT1 and Iteration-over-RT2 are common primitives for the *ADD* and *ADD_LONG* subcases whereas *ADD_LONG* comprises also of Test-for-Block-Allocation, Allocate-RT2-Block and RT1-Update.
- *inst*Route: the route being added or deleted.

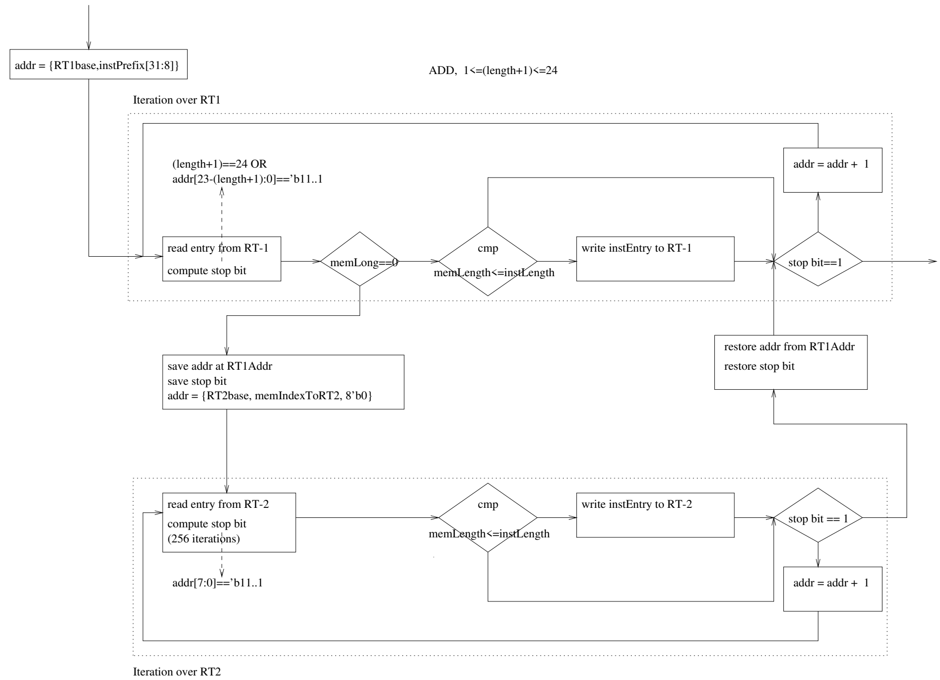

Figure 6: Flow diagram for *insert prefix* operation, $length + 1 \leq 24$

- *inst*Prefix, *inst*Length, *inst*VPout: the prefix, length and VP associated with *inst*Route; their values are determined by the *add_prefix* instruction fields.
- *inst*Entry: a 16-bit quantity equal to $\{0$, *inst*Length, *inst*VPout$\}$, obviously, RT entries whose value is determined by *inst*Route should be set equal to *inst*Entry (except from those pointing to RT2).
- *mem*Entry: a RT entry that was just read from the RT.
- *mem*Length, *mem*VPout, *mem*IndexToRT2: the fields of the *mem*Entry.
- length is used interchangeably with *inst*Length.
- RT1Addr, RT1Entry: registers necessary for manipulating the two levels of the RT, their usage is explained later in this section.
- headPtr, oldHeadPtr: registers used for RT2 list management.
- addr: the counter that controls the address lines of the DRAM that keeps the RT.
- Dout: the data lines of the DRAM that keeps the RT.

The *ADD* case is rather simple; it consists of an iteration over the first level of the RT interrupted by iterations over a block at the second level of the RT. Note that the iterations over the two RT levels are almost *identical*, the iterations require read-modify-write memory accesses and the stop bit can be computed in parallel with the memory access.

ADD_LONG requires an iteration over a single RT2 block but its flow diagram is more sublte. Each route longer than 24 bits is associated with one RT2 block; if the RT contains a route that is

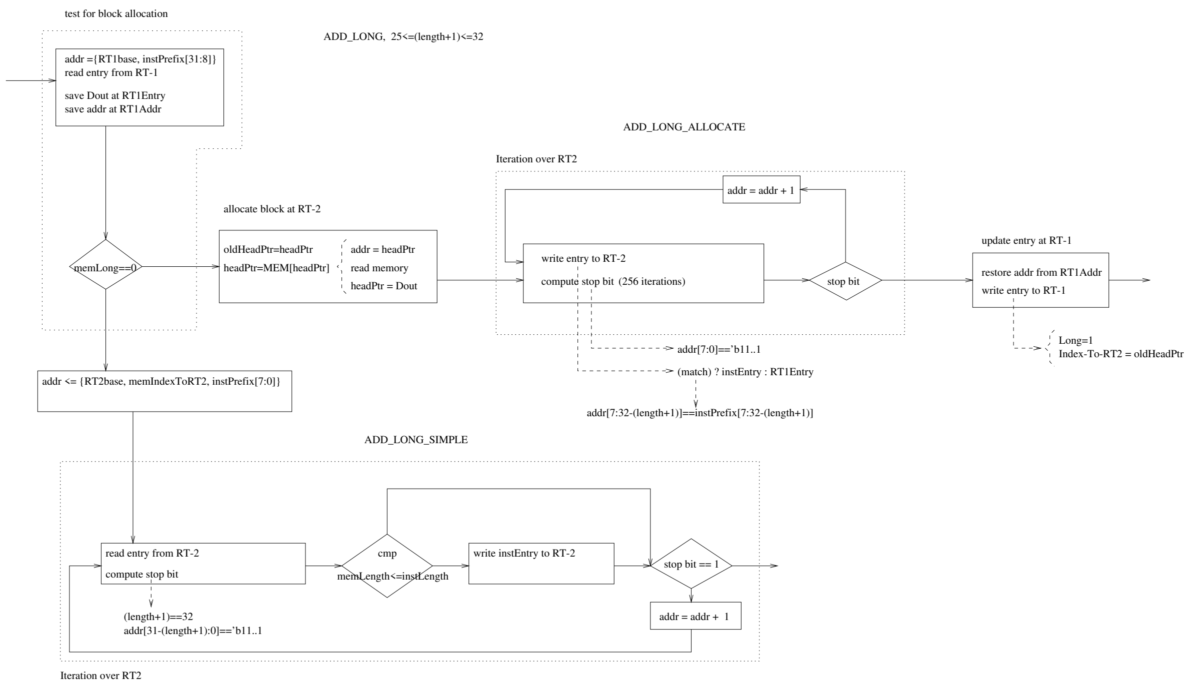

Figure 7: Flow diagram for *insert prefix* operation, $length + 1 > 24$

associated with the same RT2 block as the *inst*Route, then we simply obtain from RT1 the address of the corresponding block and iterate over it (*ADD_LONG_SIMPLE*), otherwise we need to allocate a RT2 block, fill it and, lastly update RT1 (*ADD_LONG_ALLOCATE*). *inst*Route determines the value of a subset of the entries within the newlly allocated block, the value for the rest of the entries is determined by the corresponding entry at RT1, which was saved at *RT1Entry*; the *match* bit distinguishes between the two cases. Note that we cannot unify the test for block allocation with RT1 update since that would cause an inconsistency for a short time and *ADD_LONG* iterations are very similar to *ADD* iterations. The flow diagram is *not* directly translated to a hardware implementation, in fact, the hardware implementation can be even more simple since we can move the test for block allocation, RT2 list management and RT1 update to the software component and thus, reduce the flow diagram to a *single-write* or *read-modify-write* iteration over a RT2 block. The software/hardware interface is discussed later in this section.

The *del_prefix* operation, similarly to the *add_prefix* operation, can be implemented as two distinct cases depending on the value of *OldLen*: *DEL* when $OldLen + 1 <= 24$ and *DEL_LONG* othewise. The flow diagram for *DEL* can be reduced from the *ADD* flow diagram by subsituting *length* with *OldLen* in the stop and cmp bit computations and the less-than-or-equal comparison with an is-equal test at the cmp bit computation. *DEL_LONG* requires an iteration over a single RT2 block and potentially a block deallocation; if all RT entries within the RT2 block are shorter than 25 bits [8] then *all* of them are identical and they are equal to the *father* entry of the deleted route, in this case the block needs to be freed and the corresponding RT1 entry updated. Note that we can move the test for block deallocation to the software component and thus, reduce the flow diagram for the hardware component to a *read-modify-write* iteration over the RT2 block.

[8] The condition should be tested just after *del_prefix*

The overhead on the software component is not significant since it must also compute the *father* route of the deleted route. The *DEL_LONG* case is called *DEL_LONG_FREE* if it causes a block deallocation or *DEL_LONG_SIMPLE* otherwise.

ADD, DEL	Iteration over RT1	$N*$ Iteration over RT2	
ADD_LONG	Test for block allocation	*ADD_LONG_SIMPLE* or *ADD_LONG_ALLOCATE*	
ADD_LONG_SIMPLE	Iteration over RT2		
ADD_LONG_ALLOCATE	Allocate RT2 block	Iteration over RT2	Update RT1
DEL_LONG_SIMPLE	Iteration over RT2		
DEL_LONG_FREE	Update RT1	Free RT2 block	

Table 1: Decomposition of operations to primitives

We now decide on the allocation of primitives to the software or hardware component, table 1 summarizes on the uniformity exhbited by the distinct subcases of *add_prefix* and *del_prefix* operations. Iterations over the RT levels are time critical and dominate the latency of the update operations, thus, they are allocated to the hardware component. RT2 blocks list management is not a time critical task and, thus it can be completely performed in software through dedicated operations or even general purpose *DRAMread*, *DRAMwrite*. However, since the hardware cost of a single list management does not seem to be significant, our implementation uses the following allocation: list completeness → software, test for block allocation → hardware, block allocation → hardware: requires two registers and pointers within the free blocks, RT1 update after a block allocation → hardware, test for block deallocation → software (note that this places little overhead to the software component since the software may have to compute the *father* prefix), block deallocation → hardware, RT1 update after a block deallocation → hardware. The above implementation is rather hardware-oriented; in case additional functionality needs to be implemented within the wormholeIP FPGA, RT2 list management is the *first* block to be moved *completely* to software. Table 2 demonstrates the decomposition of operations to primitives when RT2 list management, expect RT1 updates, are performed in software.

ADD, DEL	Iteration over RT1	$N*$ Iteration over RT2
ADD_LONG_SIMPLE	Iteration over RT2	
ADD_LONG_ALLOCATE	Iteration over RT2	Update RT1
DEL_LONG_SIMPLE	Iteration over RT2	
DEL_LONG_FREE	Update RT1	

Table 2: Decomposition of operations to primitives when software manages RT2 list

3.4 Datapath & Control

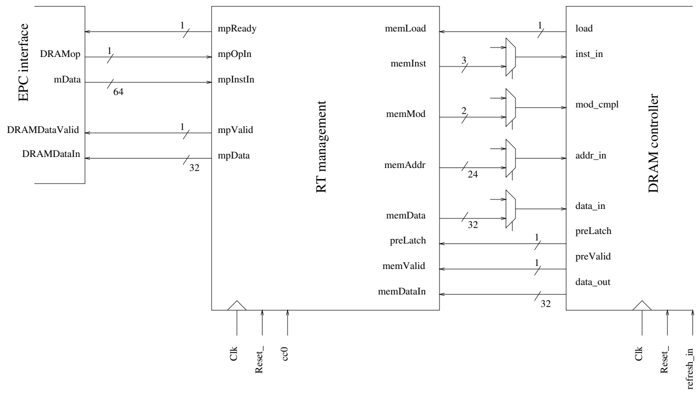

Figure 8: Block diagram of the RT management block

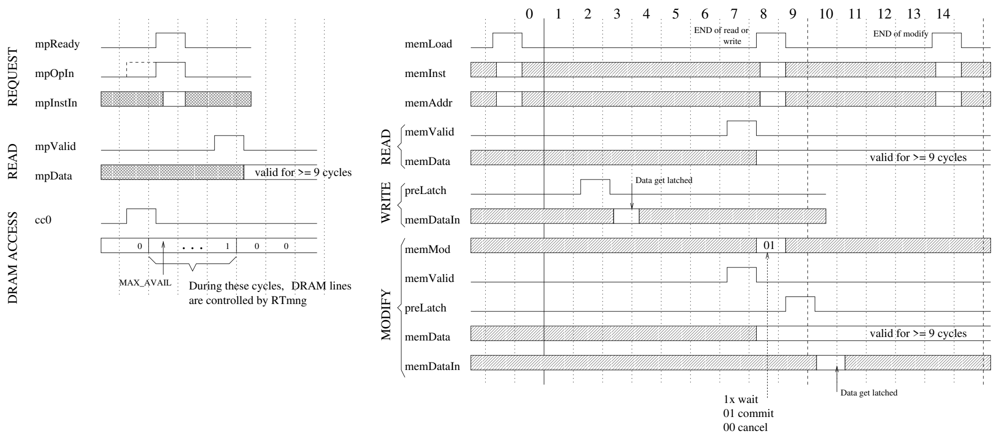

Figure 9: Interaction with the other blocks of wormholeIP

The block and timing diagrams that determine the interface of the RT management block are shown in Figs 8, 9 respectively; the above interface is used by the RT management block which implements the functionality discussed in section 3.3 and described thoroughly later in this section. Only note that DRAM is accessed by both RT management and the filter's core and one can either turn a *modify* operation into a simple read or cause the memory to wait until they prepare the word to be written back.

The initiation of a primitive is designated by the loading of the *addrCnt* counter. *DRAMread* and *DRAMwrite* simply load *addrCnt* with the *Address* field ({opcode[0], pref24}) of the *instruction* register and perform the corresponding memory access. *ADD*, *DEL* comprise of an iteration over RT1 interrupted by iterations within blocks of RT2, the strarting address of the RT1 iteration is found in the *instruction* register (*pref24*), whereas the computation of the address of each block at RT2 (*RT2blockAddr*) is based on the corresponding RT1 entry (*unqEntry*); register *RT1Addr* saves and restores the RT1 iteration. Updating RT1 after a block allocation/before a block is freed needs the address of the corresponding RT1 entry, the latter is available at the *pref24* field of the *instruction* register. Lastly, blocks are allocated from and freed to the head of the RT2 list, the

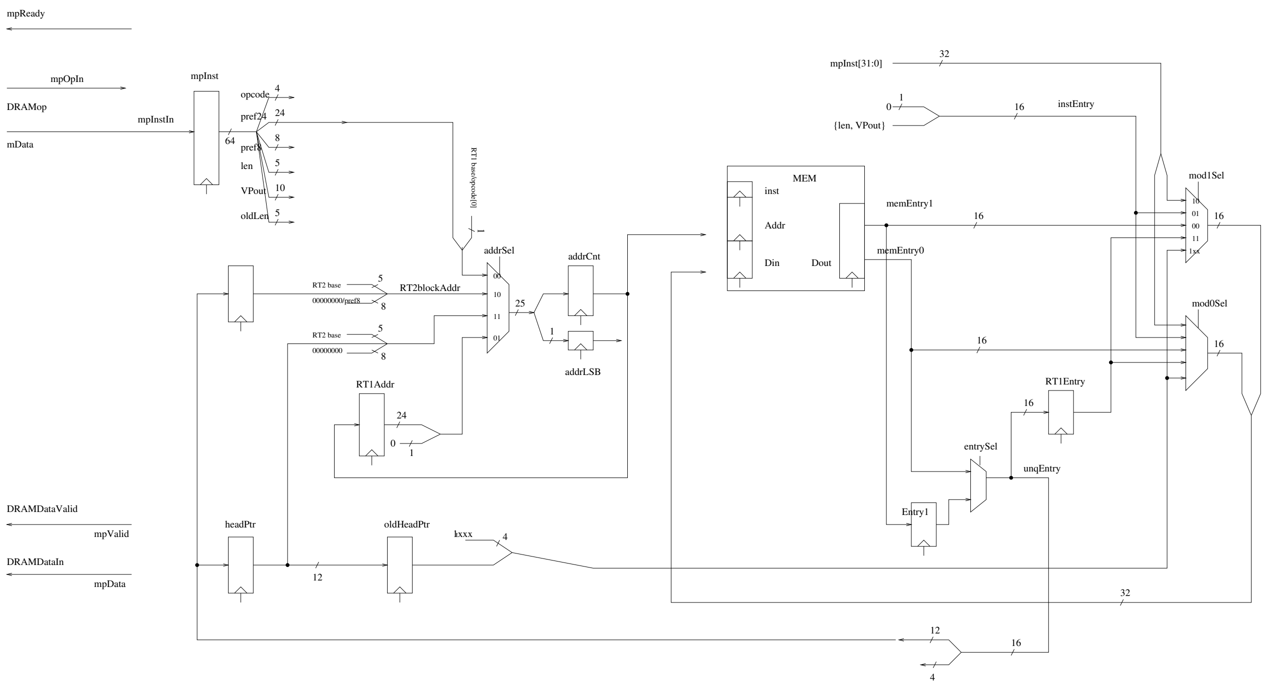

Figure 10: Datapath of the RT management block. The figure does not show the combinational logic that computes the *stop, match, cmp* bits and the state machine that generates the control signals.

headPtr register provides this address.

Our implementation exploits all the available memory throughput: RT entries are processed 2-by-2. *mod0Sel*, *mod1Sel* control the two multiplexors, for the two "sibling" RT entries, that choose the values to be written back to memory. *DRAMwrite* simply writes the corresponding field of the *instruction* register. Iterations over the RT levels either write back the RT entry without modification (*memEntry*), or an entry constructed from the fields of the *instruction* register (*instEntry*) depending on the result of a comparison. However, the iteration to fill a newly allocated block is special: the two alternatives are the *instEntry* and the *RT1Entry* (corresponds to the *father* route of the block) and the *match* bit choses between the two. RT1 update after a block allocation writes back the pointer to the newly allocated block which was saved at *oldHeadPtr* register, whereas RT1 update before a block is freed writes back *instEntry*.

Register *Entry1* stores the "right" entry of two "sibling" entries and is used handle the case that both entries point to RT2.

RT2 blocks are both allocated from and inserted at the head of the list. Register *oldHeadPtr* saves the value of *headPtr* before a list operation. The old value is used at the second step of block allocation (suppose that block *B1* must be allocated, then the first step is to set *headPtr* to MEM[*headPtr*] and the second step is to update RT1, that is set the *Index-to-RT2* field of the corresponding RT1 entry to address(*B1*)) and at the second step of block deallocation (suppose that block *B1* was just freed, then the first step is to set *headPtr* to address(*B1*) and the second step is to store in B1 a pointer to the rest of the list).

The abstract basis for the FSM that generates the control signals is table 1. The circuit's state is maintained in two registers *Iter* and *phase*. The *Iter* state register takes the following values:

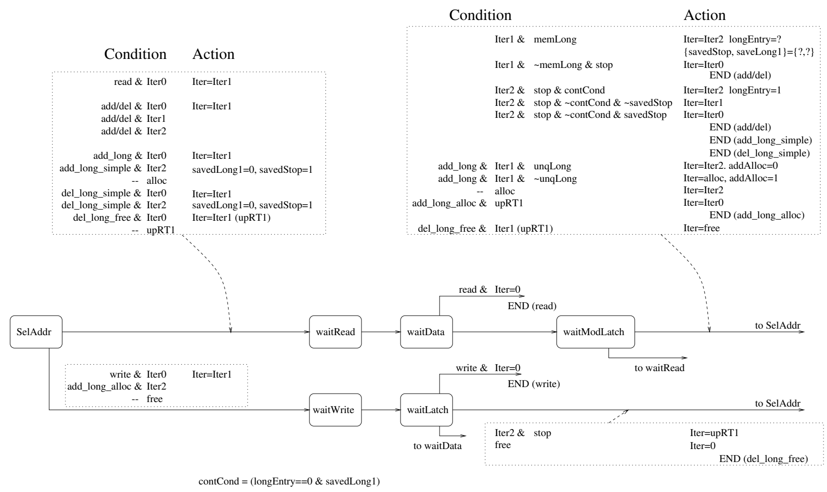

Figure 11: State transitions of the RT management FSM.

Iter0, Iter1, Iter2, alloc, free, upRT1 that correspond to the primitives described in section 3.3. The *phase* state register abstracts both the DRAM interface and the alternating access to DRAM by the RT manegement block and the filter's core. Moreover, we maintain the following bits that complement the circuit's state when it iterates over a RT2 block: *longEntry:* indicates which of the two sibling RT1 entries corresponds to the RT2 iteration, *savedStop:* saves the *stop* bit of the RT1 iteration, *savedLong1:* saves the *Long* field of the most significant (right) "sibling" RT1 entry. The above bits are saved when a RT1 iteration is interrupted by a RT2 iteration and they are used to restore the RT1 iteration; moreover, they help face the subtles introduced when both "sibling" entries point to RT2 blocks, without reaccessing memory.

State transitions are performed as shown in Fig. 11. Fig. 11 can easily be "inversed" and, thus, produce the functions for *Iter*, *phase* and the auxiliary bits. *memLong* is the OR function of the *Long* fields of the RT entries being processed in parallel and indicates whether RT1 iteration should be interrupted; *unqLong* is the *Long* flag of the RT1 entry that corresponds to the *Prefix* of the current operation and participates in the test for block allocation when $Length + 1 >= 24$. Note that some conditions (e.g. $Iter1\&memLong$, $add\_long\&Iter1\&unqLong$) may overlap, in this case we take the action of the more specific condition ($add\_long\&Iter1\&unqLong$). Lastly, for simplicity, we use Iter1 in liue of upRT1 at the *DEL_LONG_FREE* case.

Fig. 12: provides details on the relation of *phases* of the RT management block and the DRAM controller interface, shows the exact timing of read-modify-write and single-write iterations over the levels of the RT, especially the computation of the *stop, match, cmp* bits. Note that we let two cycles for the computation of the *stop, match* bits since it requires a 22-bit wide mask. We also wait two cycles[9] before latching the modified entry since it has to pass through a 5→1 16-bit multiplexor. The timing diagram shows that we perform a distinct access for each RT entry; a possible optimization is to exploit *Fast Page Mode* since consecutive RT entries are kept in the same DRAM line. The latter optimization would significantly reduce update operations' latency but it is not applicable since the DRAM throughput is shared between the RT management block and the filter's core, therefore, the RT manegemnt block accesses memory in a distributed rather

[9]We don't have to use the *memMod* signal, since this is the minimum DRAM latency.

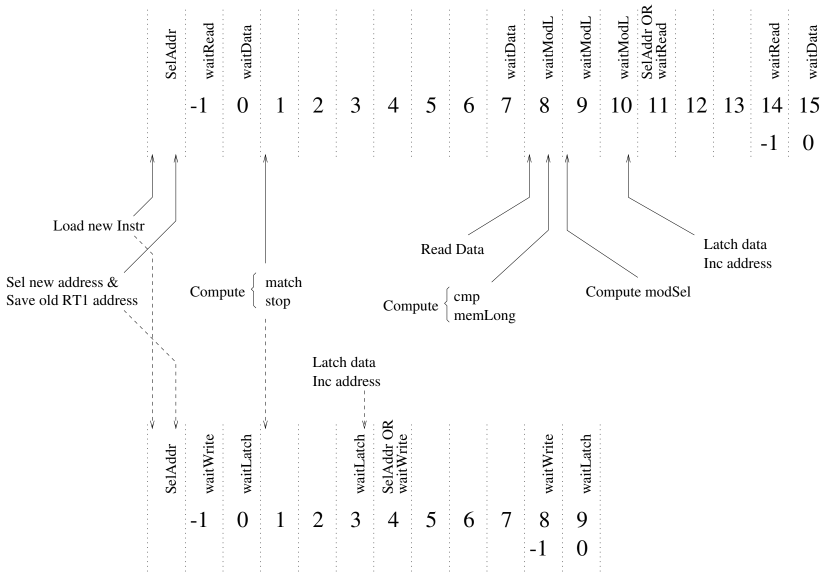

Figure 12: Timing of iteration over the two levels of the RT. Iterations over the first level require only read-modify-write acceses, whereas iterations over the second level require both modifications and simple writes. The numbering of cycles is the same as in Fig. 9.

than in a burst fashion.

4 Managing Deeper and Smaller Routing Tables

The analysis in section 2 showed that the brute force two-level GuLK approach is the best choice for the FPGA-based wormholeIP routing filter. However, there are very interesting potential applications for deeper tables, too. For example, *high performance* piplelined implementations: deeper RT are smaller and, thus, can fit in SRAM, *IPv6*: deeper RTs may be a solution to the 128-bit wide IPv6 address space. For the above reason we now describe an idea for managing deeper RTs that can be considered along with the other models described in section 3.1.

The scheme is outlined below:

- One list is maintained for the blocks of each RT level[10]. Block B1 is before B2 in the list iff $prefix(B1) < prefix(B2)$.
- For each block, we keep the length of the shorter route that affects an entry within the block.
- List management and minimum length update are essentially performed by the software component.
- When the routing protocol generates an update, the software component issues one message per RT level, which either indicates the route update and the range of blocks affected by the route (RTupd message) or it is a list management message (ListMng message). Note that,

[10] The first level is not a block list but an index

if we order the messages by level depth, RTupd and ListMng messages do not interleave; moreover, RTupd messages preceed and ListMng messages follow.

- The hardware component uses the block list and the minimum length information to access only the affected blocks.
- The scheme generates L messages (L: # RT levels) which can be packed in the same ATM cell.

5 FPM DRAM Controller

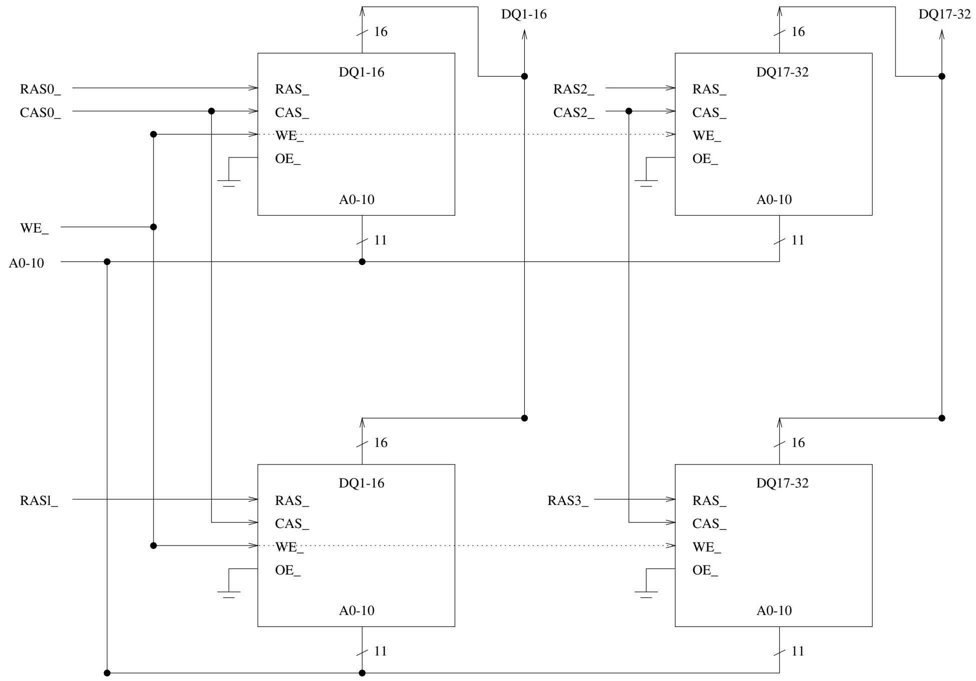

Figure 13: Block diagram of a $2^{23}x32$ FPM DRAM module.

The wormholeIP routing filter maintains its data structures in two $8Mx32$ DRAM SIMMs. We first make an overview of DRAM operation and then provide the timing diagrams for our implementation. The block diagram of a DRAM module is shown in Fig. 13: the inteface of such a module consists of 11 address lines, 32 data lines and 7 control lines and each building block is made up of four $2^{22}x4$ DRAM chips [11]. Read and write DRAM operations generally consist of the following phases: i. assert RAS - latch row address ii. assert CAS - latch column address iii. perform operation depending on WE iv. deassert CAS, RAS - precharge time.

[11]Each $2^{22}x4$ chip is organized as a matrix of 2^{11} columns and 2^{11} rows.

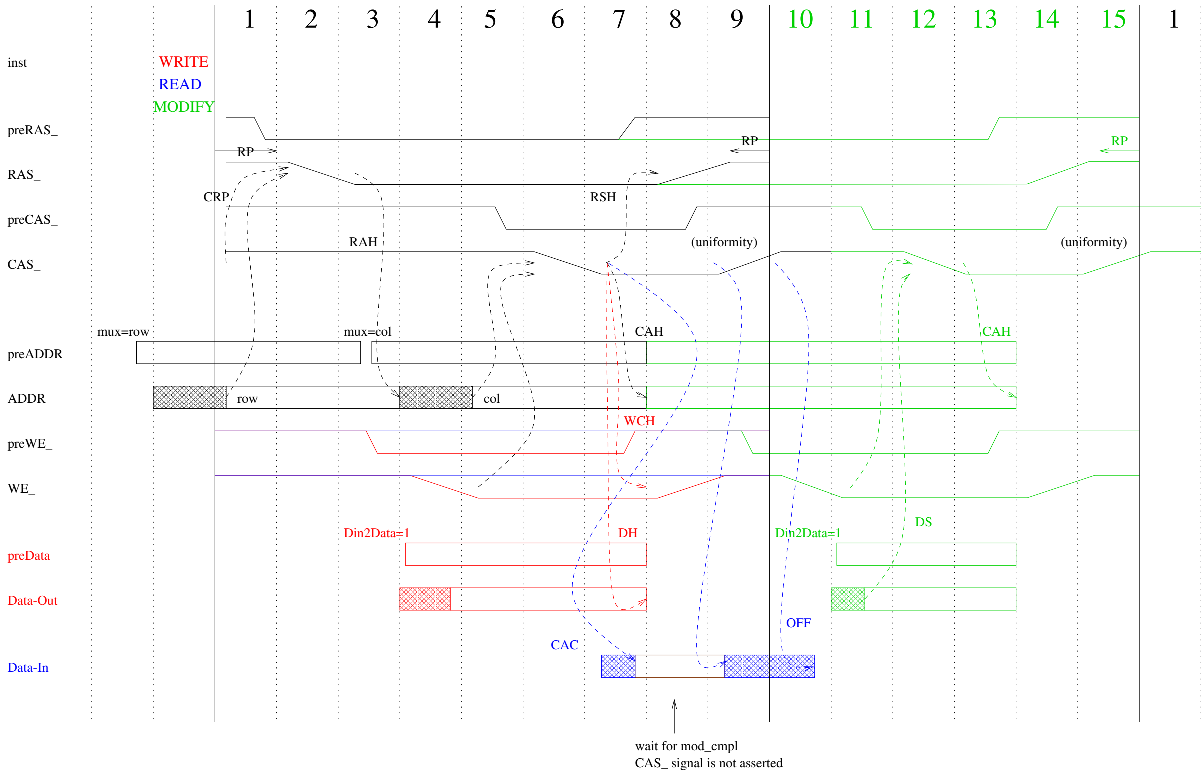

Figure 14: DRAM read, write & read-modify-write operations. The on-chip signals are preceded with the *pre-* prefix. The black color is common for read & write. The read-modify-write operation is an extension of the read operation, the first 9 cycles of the two operations are identical with the exception of the RAS_ signal.

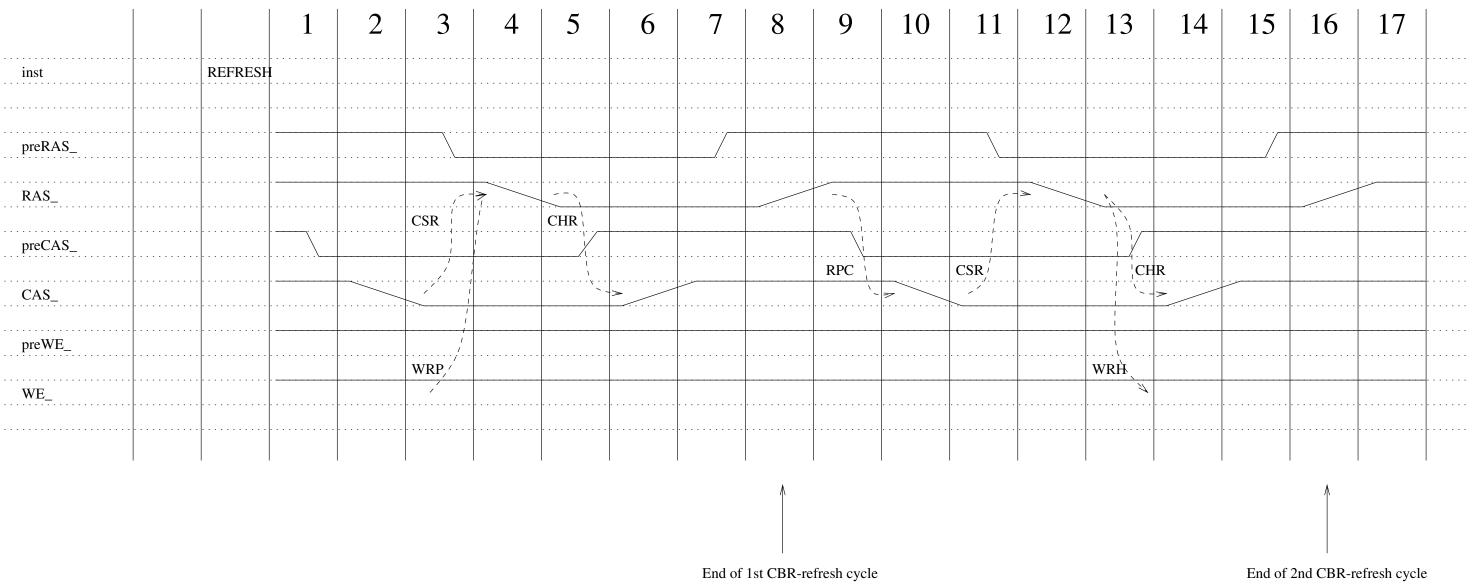

Figure 15: DRAM CBR refresh operation

The DRAM controller for the FPM DRAM SIMMs is based on the timing diagrams of Fig. 14 and Fig. 15. The synchronization diagrams are the sequential derivatives of the the FPM DRAM module interface [10], when the lines driving latency (FPGA clock-to-output delay + TTL buffers latency [12]) is about 22 to 25 nsec. The diagrams are rather simple and uniform, we need to elaborate only on few things: the modify operation can be delayed (multiple 8th cycle) if the "caller" needs more than 3 cycles, after read data are available, to prepare the modified data; the modify operation can be turned into a simple read, in this case we "jump" from the 8th cycle of modify to the 7th cycle of read; in order for the DRAM to stop driving the data lines, the CAS_ signal needs to be asserted and deasserted (cycles 9 to 11); if DRAM afforded an OE signal, then we could turn-off data lines only in 1 cycle. The RAS_ *precharge* time (RP) is shared between the first cycle of an operation and the last cycle of the previous operation, this trick saved 1 cycle for all operations. The block diagram and the timing interface of the DRAM controller were shown in Fig. 8.

The DRAM controller implements a *CBRefresh* operation which refreshes two DRAM lines when it is issued. For a survey on various methods of DRAM refresh see [11]. The DRAM SIMMs of the filter prototype board contain 2048 lines that need to be refreshed once every 32 msec. Thus, the *CBRefresh* should be issued every 11th cell time at 155.52Mbps.

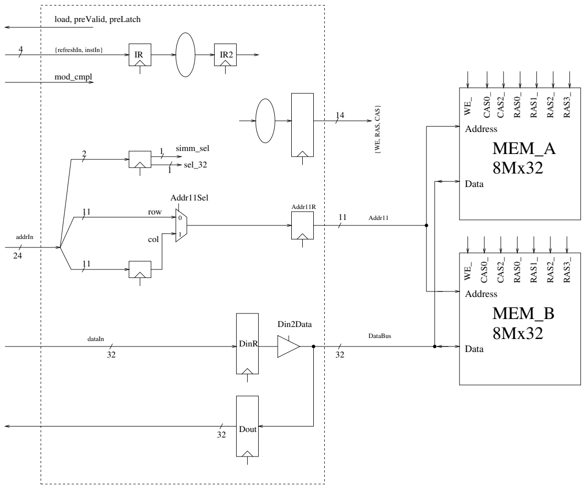

Figure 16: DRAM controller datapath

The datapath of the DRAM controller is shown in Fig. 16. Registers *IR* and *IR2* are used to implement the ovellaping between the last cycles of an operation and the loading of the next operation. *refresh_in* has a higher priority than *inst_in*, that is, when *refresh_in* is asserted, DRAM performs a CBRefresh regardless of *inst_in*.

[12] The design of the DRAM controller started before PCB design finished. The constructed PCB does not contain TTL buffers for the data lines, thus the data driving latency is about 11 nsec and one can save the 9th cycle of the modify operation (14 cycles).

6 Acknowledgements

I would like to thank ICS-FORTH for supporting my work with a trainee fellowship, professor Manolis Katevenis for his overall guidance, and my parents and sister for their support.

References

[1] Y. Rekhter, T. Li, "An Architecture for IP Address Allocation with CIDR," *Request for Comments: 1518,* September 1993;
http://www.faqs.org/rfcs/rfc1518.html

[2] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, "Scalable High-Speed IP Routing Lookups," *Proceedings of ACM SIGCOMM 1997,* pp 25-36, Cannes, France;
http://www.ccrc.wustl.edu/~varghese/publications.html

[3] M. Degermark, A. Brobnik, S. Carlsson, S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proceedings of ACM SIGCOMM 1997,* October 1997

[4] V. Srinivasan, G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion," *Proceedings of ACM SIGMETRICS Sep 98 and ACM TOCS 99,*
http://www.ccrc.wustl.edu/~varghese/publications.html

[5] P. Gupta, S. Lin, N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proceedings of IEEE Infocom,* San Francisco, CA USA, April 1998;
http://tiny-tera.stanford.edu/~nickm/papers/Infocom98_lookup.ps

[6] A. Moestedt, P. Sjdin, "IP Address Lookup in Hardware for High-Speed Routing," *IEEE Hot Interconnects VI,* pp. 31-39, Stanford, CA USA, August 1998;
http://www.sics.se/cna/publications.html

[7] C. Hedrick, "Routing Information Protocol," *Request for Comments: 1058,* June 1988;
http://www.faqs.org/rfcs/rfc1058.html

[8] M. Katevenis, Iak. Mavroidis, Ioan. Mavroidis, G. Glykopoulos, "Wormhole IP over (Connectionless) ATM,"
http://archvlsi.csi.forth.gr/wormholeIP/

[9] Iak. Mavroidis, "Hardware Implementation of a Routing Filter to support Wormhole IP over ATM," *Technical Report TR-258, ICS-FORTH,* June 1999;
ftp://ftp.csi.forth.gr/tech-reports/1999/1999.TR258.RoutingFilterCore.ps.gz

[10] Micron Technology, Inc. "MT16D832(X) DRAM Module datasheet";
http://www.micron.com/mti/msp/pdf/obsolete/dm44.pdf

[11] Micron Technology, Inc. "Various Methods of DRAM Refresh," Technical Note TN-04-30;
http://www.micron.com/mti/msp/pdf/datasheets/tn0430.pdf