Design and Analysis of Matching Circuit Architectures for a Closest Match Lookup

Kieran McLaughlin[1], Friederich Kupzog[2], Holger Blume[2], Sakir Sezer[1], Tobias Noll[2], John McCanny[1]

The Institute of Electronics, Communications and Information Technology at QUB[1]
The Institute of Electrical Engineering and Computer Systems at RWTH Aachen University[2]

Abstract— *This paper investigates the implementation of a number of circuits used to perform a high speed closest value match lookup. The design is targeted particularly for use in a search trie, as used in various networking lookup applications, but can be applied to many other areas where such a match is required. A range of different designs have been considered and implemented on FPGA. A detailed description of the architectures investigated is followed by an analysis of the synthesis results.*

1. Introduction

The Internet is changing and moving towards interactive multimedia communications, with existing discrete services integrated onto a single platform. To enable this requires greater bandwidth, lower end-to-end propagation delays and improved Quality of Service (QoS) guarantees. Applications such as VoIP (Voice over Internet Protocol), streaming audio, video and other specialised applications have specific bandwidth and propagation delay requirements. Such demands create a bottleneck in the routers that form the infrastructure of the Internet as they must process ever increasing amounts of data. Greater bandwidth requires faster transmission of packets which in turn requires faster search and lookup techniques for data associated with packets, paths and packet flows. In fact interactive services are usually based on small packets to reduce end to end delays. As a result the packet classification, lookup and scheduling speeds required increase even more than the bandwidth required. It is difficult for the traditional software solutions currently used in routers to perform high-speed data retrieval as required for next generation QoS enabled networking. Future designs require hardware architectures that can deliver greater control over memory management and the number of accesses per lookup to slow off-chip memory.

This paper investigates the design and implementation of a hardware based closest match lookup circuit. A novel design based on a trie is proposed, which is composed of distributed memory blocks for parallel and pipelined sort and lookup. The latest FPGA technology has been chosen due to the embedded memory features, which is useful in particular for implementing the pipelined search trie.

2. Related Work

Associative memory has been widely investigated for network processing and pattern, speech and image recognition. Most of these architectures were designed under application related constraints, such as the number of entries, cost and lookup performance. A number of existing associative memory implementations are available.

Content Addressable Memories (CAMs) are "hit or miss" components. An entry is either present or not and they therefore have limited suitability for closest or non-exact match lookups, although a number of implementations have been examined. One common approach makes use of standard CAMs, which give either an exact match or no match. Different bits of the desired match are masked during a series of requests. At first, no masking bits are set. If there is no match then one bit of the word is masked and requested again. The masking pattern is then altered, masking all combinations until a match is found. Obviously this is time consuming, especially for wide data words. This approach is used in a parallel form in image coding, using Vector Quantization [1].

Other non-CAM approaches seek to avoid the high costs and insufficient performance of retrieving inexact matches using CAMs. A highly regular design is described in [2]. A basic cell containing a word of memory, a comparison unit and control logic is cascaded in a long pipeline. The requested word enters the pipeline as an input. Each cell compares the request with its own content and if it fits better than in the previous cell, the current cell will signal its own address to the next cell. This pipelined approach features a predictable, fixed response time and a high throughput rate. However, the latency is very high for large memories because the pipeline length is proportional to the memory size.

Finally, neural networks are an alternative for a best match lookup, in particular self-organising feature maps. VLSI implementations of neural networks are distributed processing systems with extensive connectivity. However, the fact that these connections have to be adaptable leads either to a reduced memory density or the use of non-standard VLSI fabrication techniques [2],[3],[4].

3. Architecture

The distinct feature of the proposed closest match lookup architecture is the use of a sorter tree, or "trie", to implement an associative memory, which is able to return either an exact or next smallest match. The term "trie" is derived from tree and re*trie*val. It was proposed by Fredkin [5] as a specialised search tree that stores multiple strings. This original structure can be adapted to solve a range of numeric lookup problems, e.g. finding the entry with the smallest Hamming distance to a given value. The number of levels in a trie determines the length of strings it can store. Its branching factor determines the number of literals of which the strings can consist. Since each level is usually accessed in one clock cycle, it is favourable to keep the number of levels low to reduce the latency of a pipelined implementation. Also, fewer levels will require less memory. To reduce the tree depth, two or more bits can be grouped together into a single literal and stored in one trie level (multi-bit trie, branching factor > 2). Fig. 1

1-4244-0054-6/06/$20.00

shows an example of a multi-bit trie with branching factor four (literals '00', '01', '10' and '11') and three levels, allowing strings with 3 literals or 6 bits.

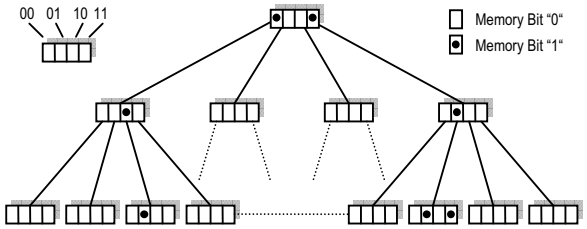


Figure 1: Multi-bit trie with values 001001, 110101 and 110111

Data is not stored by writing a value in memory, but by setting flags to indicate the presence of a value. The final trie level consists of one flag (bit) for each possible value the trie can store (4^3 = 64bits in Fig. 1). To store a string, one bit is set in each level of the trie. In the first level, the flag corresponding to the first literal of the string is set. Then, this tree branch is followed and in the next level the flag corresponding to the second literal is set. This goes on until the last level is reached. The advantage of this approach is that when retrieving data, the result is assembled literal by literal while passing through the tree levels. The policies used during the data retrieval process allow the required exact or next smallest value lookup to be achieved. In each level the desired search string literal is compared to the literal present in the trie and an exact or next smallest match is returned. If a non-exact match occurs, i.e. a smaller value than that requested is returned, all subsequent levels return their maximum value, so that if an exact match is absent, then the overall string returned is the closest value in the trie that is smaller than the desired search string. For example the closest match to "11 01 10" in the trie shown in Fig. 1 is "11 01 01".

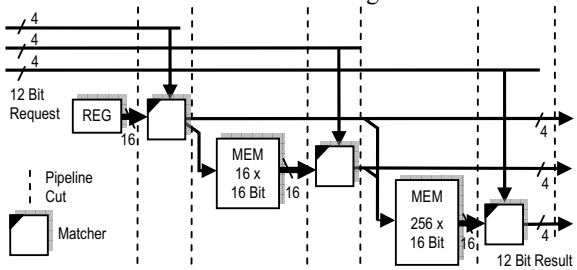
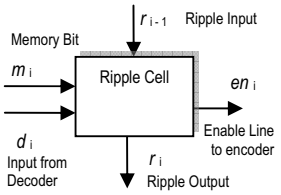


Figure 2: Implementation of a trie with branching factor 16.

The matcher is shared between nodes in a level since only one operation occurs at a time in each level. Therefore, each level consists of a memory and a "matcher". The structure (Fig. 2) can be pipelined and is scalable by either increasing the branching factor or by adding more levels. Since memory access is the most time dominant operation, the matcher delay must be as close to this time as possible to achieve optimum speed performance.

3.1 Matcher Architecture

The matcher requires a linear search to be performed to find the next smallest entry within a tree level. Due to its sequential nature the matcher normally determines the critical delay. The linear search is performed by ripple logic consisting of elements like the one shown in Fig. 3, which operates along the memory bits of the entries in each trie level. A decoder "injects" a logic '1' into the ripple path at the position of the requested value, which propagates through the ripple logic until it reaches a memory bit set to '1'. The ripple process then stops and the corresponding enable line is set to '1'. Finally, the resulting value is encoded in binary format. The critical path of the matcher circuit stretches across the input decoder, the full length ripple path and the output encoder.



m_i	d_i	r_{i-1}	r_i	en_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	*	*
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	*	*

Figure 3: The basic ripple element with truth table * don't care

The ripple logic is similar to a carry ripple adder and the basic compare element is similar to a full adder, allowing the definition of the "generate" and "propagate" conditions similar to carry chains in adders [6]. For an adder,

$$\text{generate} \quad g_i = a_i b_i \tag{1}$$

$$\text{propagate} \quad p_i = a_i \oplus b_i \tag{2}$$

$$\text{the next carry} \quad c_{i+1} = g_i + p_i c_i \tag{3}$$

For the matcher ripple logic,

$$\text{generate} \quad g_i = g_i \overline{m}_i \tag{4}$$

$$\text{propagate} \quad p_i = \overline{m}_i \tag{5}$$

$$\text{the ripple output} \quad r_{i+1} = g_{i+1} + p_{i+1} r_i \tag{6}$$

$$\text{enable signal} \quad en_i = m_i (d_i + r_i) \tag{7}$$

Most theorems developed to accelerate the carry chain in adders can be applied to the matcher circuit. In order to improve its performance, carry-look-ahead, block-carry-look-ahead and the combination of carry-skip and carry-look-ahead, carry-select and carry-look-ahead techniques have been applied and analysed.

3.2.1 Look-Ahead Approach

In this approach instead of each ripple signal r_i being dependent on the previous ripple signal, it is generated in parallel by trying to achieve a propagation delay complexity of $O(1)$. Equation (7) shows the dependency of the enable signal en_i from the ripple r_i, memory m_i and decoder d_i signals. For parallel generation, Equation (6) is extended for each i and it is found that (with $g_{-1} = r_{-1}$)

$$r_i = g_i + \sum_{\mu=0}^{i} \left(g_{\mu-1} \prod_{v=\mu}^{i} p_v \right) \tag{8}$$

Theoretically r_i can be obtained in two logic levels, the outer OR and inner AND. However, the number of gate inputs grows linearly with i, thus the AND and OR functions must to be split into multiple gates for higher i, such that the complexity of the propagation delay is not more than $O(\log i)$. Increasing i means a disproportionate increase of logic due to splitting gates into multiple stages, so this approach is only suitable for small ripple chains.

3.2.2 Block Look-Ahead Approach

To address the disadvantages of the look-ahead approach, a hierarchical structure is applied. Fig. 4 shows a functional block that performs the look-ahead operation for a restricted number of bits, generating the signals G (*group generate*) and P (*group propagate*). By introducing additional hierarchy levels, the input count and thus the hardware cost of the look-ahead-blocks can be reduced.

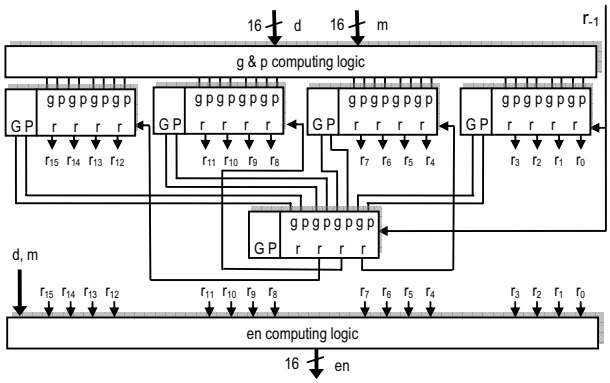


Figure 4: 16 bit Block Look-Ahead structure, block size is 4 bit.

For an *m*-bit wide block with *m*-1 inputs G and P are

$$G = g_{m-1} + \sum_{\mu=0}^{m-2}\left(g_{\mu-1} \prod_{v=\mu+1}^{m-1} p_v \right) \quad (9) \qquad P = \prod_{v=0}^{m-1} p_v \quad (10)$$

The maximum value for *i* in Equation (8) is the length of the matcher. In comparison, *m* in Equation (9) is a fraction of this, so the overall hardware cost and propagation delay are reduced, compared to the look-ahead approach.

3.2.3 Skip & Look-Ahead Approach

The carry-skip adder delivers a good trade-off between delay and area cost. An optimised skip structure is proposed [7] using a look-ahead technique within the skip blocks using variable block sizes. This approach can be transferred to the matcher problem. A number of bits are grouped into a block that bypasses the ripple (carry) signal for the block if all propagate inputs are '1'. Fig. 5 shows the basic skip block and Fig. 6 the resulting chain of skip blocks. The ripple initiating blocks and the block where the ripple signal ends cannot be bypassed, all other blocks can be bypassed, thus the ripple's maximum path is the first and last block of the skip & look-ahead architecture, bypassing all other blocks. Furthermore, since the lengths of the blocks, other than the first and last, do not contribute to the critical path, they can be increased as long as alternative paths through the system and enlarged blocks cannot become longer than the existing worst case path. Increasing the block size reduces the number required and thus the critical path delay is also reduced.

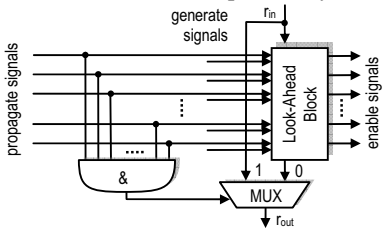


Figure 5: Ripple Bypass using the propagate condition

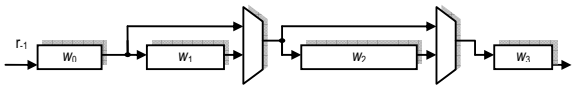


Figure 6: Variable block size Skip Matcher

3.2.4 Select & Look-Ahead Approach

Two acceleration techniques are again combined in this approach proposed for adders [8]. The matcher can be significantly simplified compared to a carry-select adder. The main difference between an adder carry chain and the matcher ripple chain is that once the ripple signal has "left" the chain (changed back from '1' to '0'), it will not change to '1' again. The rest of the ripple chain can then be ignored unlike an adder where another carry could be generated. The ripple chain is divided into blocks but this time all blocks calculate their result simultaneously. Within the blocks, the ripple process is again accelerated using a look-ahead technique. The ripple inputs *r_in* are controlled by the "result control" circuit (Fig. 7). Unlike an adder, which can have multiple possible carry propagations, the matcher has only a single search path and can therefore perform a true parallel result calculation.

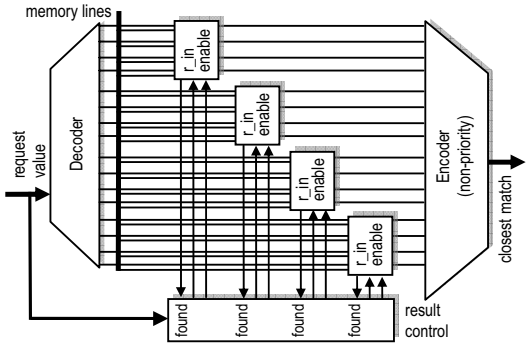


Figure 7: Select & Look Ahead structure for 16 Bit

4. Synthesis and Circuit Analysis

The circuits were described in VHDL and synthesised using Quartus II for Altera Stratix II FPGA technology. Bi-directional matchers were implemented, consisting of two matcher circuits. The first searches for the next smallest match and the second for the next highest. This is necessary to avoid a "nil" return if a smaller match is not present. In this case, the next highest match is returned. The designs have been scaled over a range of word lengths and branching factors, although not all are suitable for all word lengths. In particular the skip and select approaches require a minimum length and the look-ahead is expensive beyond 64 bits. The block look-ahead approach is only useful for lengths equal to $(blocksize)^m$.

4.1 Matcher Results

The post-layout synthesis results for the matcher architectures are presented in Table 1 and Figures 8 and 9.

	t_{pd} [ns] for different word lengths					
Implementation	**4**	**8**	**16**	**32**	**64**	**128**
Ripple Cells	2,3	3,9	6,2	8,8	14,2	
Skip & Look-Ahead			5,5	7,6	11,4	
Look-Ahead	2,4	4	5,8	7,7	9,5	
Block Look-Ahead	2,4		5,8		8,5	
Select & Look-Ahead		4,4	5,2	7,1	8,8	10,2

Table 1: Synthesis results for different matcher architectures. If different parameter values are possible, the optimum is shown.

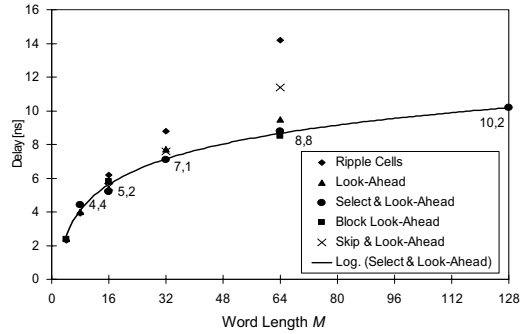


Figure 8: Matcher delay comparison

For small word lengths (4-8bits), the classic ripple carry is best because the simple structure can be mapped to a small logic path compared to the complex look-ahead approaches. The hardware cost does not significantly vary in this range. However, as the word length grows, the cost and path-delay become significant. The area cost for the look-ahead grows rapidly at approximately $O(M^2)$. The improvement gained by introducing the block look-ahead approach is clear for a 64-bit word length, where the matcher circuit is up to 3 times smaller than the classical look-ahead approach. The reduced complexity also gives a slightly smaller delay. The combined skip & look-ahead approach gives a further reduction in area by trading off ripple delay performance. For word lengths greater than 32-bit, it is slower than all the other look-ahead architectures. The best trade-off, for area versus delay performance, is achieved by the select & look-ahead architecture for word lengths greater than 8 bits. It is the most area efficient amongst all look-ahead based approaches, while maintaining the smallest ripple delay.

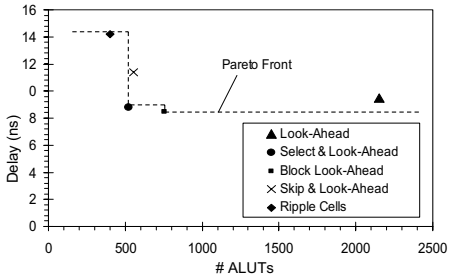


Figure 9: AT Diagram for 64 Bit matchers.

For M=64, the Pareto front [9] in Fig. 9 shows the optimum results to be the ripple cells, select & look-ahead and block look-ahead architectures. For low numbers of ALUTs the ripple approach dominates all others. For a medium number of ALUTs, select & look-ahead has the shortest propagation delay. The lowest delay can be achieved using the block look-ahead approach.

4.2 Synthesis of the Lookup Trie

Based on the results, the lookup trie (Fig. 1) was implemented using the select & look-ahead approach. Three tries with word lengths of 12, 16 and 20-bit were implemented with a pipeline scheme. For each word length, different combinations of branching factor and number of trie levels are possible. Due to the low latency required by the application, branching factors of 8 and 16 were chosen to keep the number of levels as low as possible.

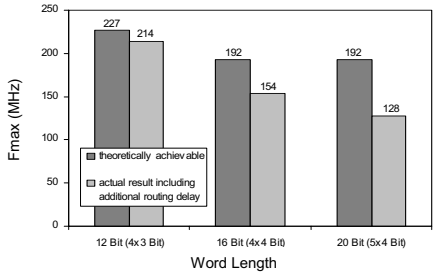


Figure 10: Theoretical matcher delay compared to actual maximum operating frequencies for different data word lengths

Theoretically, the critical path of the circuit is determined only by the branching factor. For example a trie with branching factor 16 has a critical path delay of 5.2 ns allowing an implementation at $f_{max} = {}^{1}/_{5.2\ \mathrm{ns}} = 192$ MHz. In reality this cannot be achieved due to additional routing delays. Experimental synthesis results showed that for a 4 level, 16-bit trie with a branching factor of 16, only 154MHz is achieved. Figure 10 shows how the additional routing delay dependents on the branching factor. This is because the trie memory is implemented using the M4K memory blocks, which are located at the centre of the Altera Stratix II FPGA. As expected, the place and route tool places the matcher circuitry around the central memory, which has implications for the overall routing delay. As the number of matcher circuits that need to be arranged around the centre increases, the routing between them and the memory gets increasingly longer.

5. Conclusions

The architecture and implementation of a closest match lookup circuit has been explored. The design was carried out in the context of a search trie for a network processing application, although it is applicable to a range of lookup applications. An architecture based on a search trie is used to obtain the closest match among a number of entries, given a desired value. The detailed study reveals a number of design issues concerning associative memory design for the lookup problem and identifies the matching circuit to be the bottleneck in the architecture. Comparable bottlenecks in arithmetic circuits, which have been resolved using look-ahead approaches, have been investigated and modified for the matcher circuit design for different word lengths and branching factors. The post layout synthesis results for given word-lengths have been analysed in terms of scalability, critical path and hardware cost. For word lengths greater than 8-bits, the select & look-ahead design achieves the best trade-off for area versus delay performance. The study also reveals that the select & look-ahead is the most area efficient architecture among the look-ahead designs because it is possible to simplify the original architecture when applying it to the matcher design. Although the design is suitable for a range of different high speed lookup applications, the architecture is required for a specific packet scheduling architecture using 16-bit word lengths and a 4x4 bit branching factor. This design enables it to support an operation frequency of up to 154MHz using standard FPGA technology. The resulting lookup circuit can retrieve up to 40 million IP packets per second for service.

References

[1] S. Panchanathan, M. Goldberg, "*A Content-Addressable Memory Architecture for Image Coding Using Vector Quantization,*" IEEE Transactions on Signal Processing, Sept. 1991, pp. 2066 – 2078.

[2] L. T. Clarc, R. O. Grondin, "*A Pipelined Associative Memory Implementation in VLSI*," IEEE Journal of Solid-State Circuits, 1989.

[3] T. Kohonen, "*Self-Organization and Associative Memory*," Springer- Verlag, 1984.

[4] H. P. Graf, P. d. Vegvar, "*A CMOS Associative Memory Chip Based on Neuronal Networks,*" ISSCC 87, Feb. 1987, pp. 304-305, 437.

[5] E. Fredkin: *Trie Memory*. Communications of the ACM, 3(9): 490-499, Sept. 1960.

[6] B. Parhami: *Computer Arithmetic*, *Algorithms and Hardware Design*. Oxford University Press, 2000.

[7] M. J. Schulte, K. Chirca et al, "*A Low Power Carry Skip Adder with fast saturation"* Proc. of the IEEE International Conference ASAP '04, pp. 269 – 279, 2004.

[8] Y. Wang, C. Pai, X. Song, "*The Design of Hybrid Carry-Lookahead/Carry-Select Adders,"* IEEE Transactions on Circuits and Systems Vol. 40 No. 1, 2002.

[9] M. Brayton, R. Spence: *Sensitivity and Optimization*, Elsevier, Amsterdam, 1980.