On the Use of Pseudorandom Sequences for High Speed Resource Allocators in Superscalar Processors

Srivatsan Srinivasan and Lizy Kurian John
Dept. of Electrical and Computer Engineering, The University of Texas at Austin, TX
{srivats, ljohn}@ece.utexas.edu

Abstract

With the increasing effort towards exploiting the maximum level of instruction level parallelism, modern microprocessors are designed to simultaneously issue and execute several instructions in the same clock cycle. A number of resource identifiers and tags are used in these superscalar processors to appropriately manage various resources in the processor, correctly identify and enforce data dependencies and to keep track of the instructions that are issued and completed. Structures whose delay is a function of issue window size and/or issue width are likely to become cycle time limiters and a hardware resource allocator is a potential candidate for investigation. The most straightforward technique to allocate and keep track of hardware resources in a processor is to use straight binary numbers as resource identifiers. In this paper, we investigate some alternate sequences especially, a pseudorandom sequence. The pseudorandom sequence is a 'maximal length sequence' that has some key properties which enable fast sequence generation using a Linear Feedback Shift Register (LFSR). We analyze the area and timing issues of various resource allocators using models constructed in Verilog hardware description language. Based on the timing optimizations in Synopsys targeting LSI Logic's 3.3v G10TM-p Cell-Based 0.29µ ASIC library, we conclude that the pseudorandom sequencer can enhance the clock speed by 15 - 20% when compared to the traditional straight binary sequencers at the expense of 1.1 to 2.2 times more area. Considering the fact that the resource identifier allocator is required for reorder buffer entry allocation, reorder buffer tag allocation, and any other internal resource allocation, and that all these units act in tandem, in reality better clock rate, and thus higher overall system performance, can be achieved by adopting the techniques presented in this paper.

1. Introduction

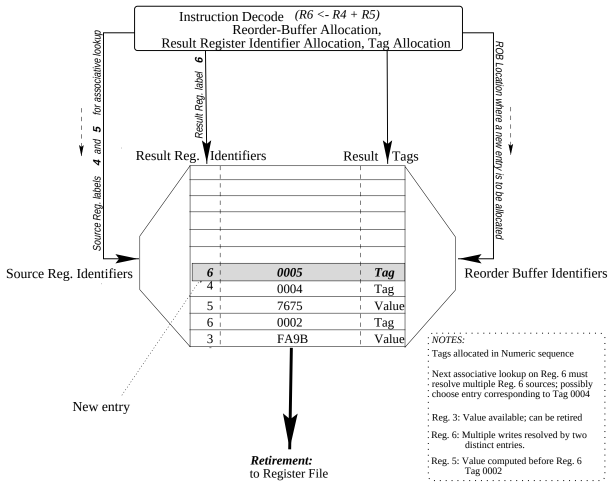
The performance of modern processors depend largely on their ability to exploit the instruction level parallelism (ILP). By and large, these processors work around the limited parallelism that a program sequence offers by executing instructions out-of-order. Parallelism in a program sequence can be extracted using a powerful compiler, or by providing sophisticated hardware, or both. With the availability of more silicon real estate, most of the modern processors provide additional hardware to perform dynamic scheduling and out-of-order execution. Typically, these processors look up a set of instructions in a window, identify instructions that can be executed in parallel, remove false dependencies, if possible, by renaming the logical registers in the instruction to a larger set of physical registers (*register renaming*), and retire instructions in program order [1]. A popular approach to perform register renaming and in-order retirement is the *reorder buffer* (ROB)+. For instance, Intel P6 architecture has a 40-entry reorder buffer [2] and the HP PA-RISC 8000 has a 56-entry reorder buffer [3].

The reorder buffer was first proposed for use in exception recovery [4], but has become the mainstay of out-of-order execution. Barring a few exceptions like Alpha 21264 [5][6], and IBM GHz processor [7][8], most of modern microprocessors incorporate a reorder buffer. Alpha 21264 implements register renaming by mapping instruction "virtual" registers to internal "physical" registers, and uses a scoreboard technique to handle out-of-order issue. Although the reorder buffer approach is more complex than scoreboarding, and hence, is normally expected to operate at slower clock rate, the reorder buffer is an elegant mechanism used to perform register renaming, exception handling, out-of-order execution and in-order retirement, all in one.

+ A content-addressable memory which stores non-committed instructions as described in [1]

This research was supported in part by the National Science Foundation under grant CCR-9796098. The authors are also supported by NSF grant EIA-9807112, State of Texas Advanced Technology program, Dell, Intel, IBM, AMD and Microsoft.

Figure 1 shows the snapshot of operation of a reorder buffer. During the instruction decode phase, the processor allocates a resource to an instruction by providing an identifier for the resource. For example, for the instruction $R6 \leftarrow R4 + R5$, the processor allocates a reorder buffer entry shown shaded in Figure 1. Also during instruction decode, the source operands or corresponding tags for each instruction have to be passed to the reservation station. To obtain operands, the reorder buffer is associatively searched using the source register identifiers of the decoded instructions. The source register identifiers are compared to result register identifiers of previous instructions stored in the reorder buffer. In Figure 1, the source register identifier for register $R4$ and $R5$ are compared to the previous result register identifier. If the register number is found and a value is available, the corresponding entry is obtained. However, if the value is not available, a result tag is obtained. In this case, the value for register $R5$, i.e., 7675, and the tag for register $R4$, i.e., 0004, are obtained. In case of multiple matches, the youngest matching entry is obtained. If the processor has a four instruction decoder, there should be four ports for result register identifiers, result tags and reorder buffer identifiers, and eight source register identifiers. If fewer ports than this number are used, arbitration will be required for port access.



Instruction Decode (R6 <- R4 + R5)
Reorder-Buffer Allocation,
Result Register Identifier Allocation, Tag Allocation
Source Reg. labels 4 and 5 for associative lookup
Result Reg. label 6
ROB Location where a new entry is to be allocated
Result Reg. Identifiers
Result Tags
Source Reg. Identifiers
Reorder Buffer Identifiers
6 0005 Tag
4 0004 Tag
5 7675 Value
6 0002 Tag
3 FA9B Value
New entry
Retirement:
to Register File
NOTES:
Tags allocated in Numeric sequence
Next associative lookup on Reg. 6 must resolve multiple Reg. 6 sources; possibly choose entry corresponding to Tag 0004
Reg. 3: Value available; can be retired
Reg. 6: Multiple writes resolved by two distinct entries.
Reg. 5: Value computed before Reg. 6 Tag 0002

Figure 1: Snapshot of a Reorder Buffer allocating entries for instruction $R6 \leftarrow R4 + R5$ (adapted from [1]).

To allocate new entries, to identify and match the existing entries, and to replace a tag with a value, the reorder buffer requires a number of hardware identifiers. Thus, one of the parameters that determines how fast the reorder buffer operates, is how fast these identifiers are generated. Figure 2 shows how these identifiers are generated. The resource allocation hardware that generates these identifiers, has multiple stages, with one stage per instruction simultaneously decoded. The input to the first stage is an identifier for the first available entry. If the first instruction requires the entry (assume certain types of instructions do not need a reorder buffer entry), the first stage uses this identifier and forms an identifier for the next available resource and passes it to the second stage; if insufficient resources are left for all four instructions, an overallocation signal is generated which results in decoder stall. Otherwise, the identifiers for each resource are passed to the corresponding instructions. The resources freed in each cycle are added to the pool of resources for the next cycle. All these stages operate in a single cycle. Resource identifiers are generally small. For example, a 32-entry reorder buffer requires only a 5-bit identifier. Assuming the top three entries in the reorder buffer are empty, the function of the reorder buffer allocator is to allocate these three entries to three out of four instructions being decoded.

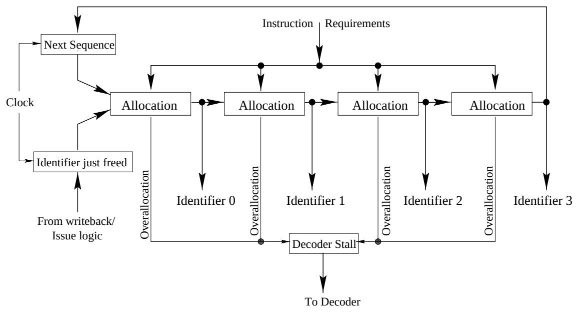


Figure 2: A typical four-port resource allocator (adapted from [1]).

Each allocation stage generates an identifier, and collectively they form a sequence. The most straightforward sequence that one can use is a simple numeric sequence. However, there is no need to use a numeric sequence. If a non-numeric sequence such as *excess 3 code* or *Gray code* would result in the allocation hardware to be small and fast, one could certainly employ those.

Johnson presented the complexities of various hardware units at the architectural level [1], while Palacharla et al [9] presented a detailed analysis of some of the units in the pipeline of a processor for feature sizes 0.8μ, 0.35μ, and 0.18μ using Spice simulations. Palacharla et. al analyzed register renaming, instruction window wakeup and selection logic, and operand bypassing and concluded that the window wakeup and selection logic and bypass logic are likely to dictate the speed of the processor. However, the base model they analyzed does not have a reorder buffer. With reorder buffer based register renaming and out-of-order execution being a popular approach, it is imperative to know how well the reorder buffer and the associated hardware perform. Some design issues of a reorder buffer can be found in past literature [3][10], but the authors of this paper are not aware of any treatment on the hardware and timing issues of the resource allocation unit, which manages the allocation and de-allocation of the reorder buffer, and hence dictates how fast the reorder buffer can be managed. In this paper, we analyze the area and timing issues of some of the allocators, both numeric and non-numeric, by modeling them in Verilog hardware description language and synthesizing them in Synopsys targeting LSI Logic's 3.3v G10TM-p Cell-Based 0.29µ ASIC library. Based on the synthesis results of hardware description language models, we conclude that a pseudorandom sequence generator can enhance the overall performance by 15 - 20%. Considering the fact that the resource identifier allocator is required for reorder buffer entry allocation, reorder buffer tag allocation, and any other internal resource allocation, and that all these units act in tandem, in reality better clock rate, and thus higher overall system performance, can be achieved by adopting the techniques presented in this paper.

The rest of the paper is organized as follows. Section 2 describes the general structure of the resource allocator. In Section 3, we discuss the numeric and non-numeric sequences, while in Section 4 we present their implementations. Results and analyses are presented in Section 5. Section 6 provides concluding remarks.

2. General Structure of Resource Allocators

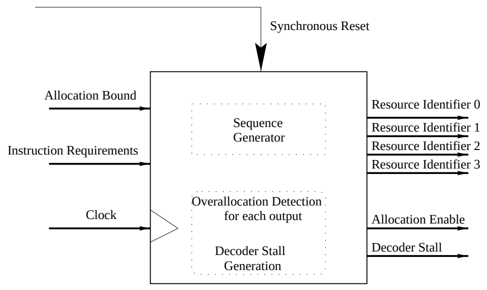


Figure 3: Symbolic view of a four-port resource allocator

Symbolically, a four-port resource allocator can be viewed as in Figure 3. The resource allocator generates the next set of identifiers considering the current set of identifiers. At the sensitive edge (rising or falling edge) of the clock, the overallocation circuit compares resource identifiers with the allocation bound (i.e., the bound beyond which no further resource allocation can be performed) to generate allocation enable (or disable) and decoder stall signals. As the allocation has to be performed in one clock cycle depending on the requirements of the instruction and the availability of resources, these stages are not pipelined [1]. *With widening issue-width and ROB size, the serial nature of resource identifier generation and allocation is bound to become the bottleneck and can limit the clock speed of the processor.* To minimize the critical path, the resource allocator should generate all the aforesaid signals as fast as possible.

3. Sequence Types

The resource allocator can generate the identifier sequence either in a numeric order or in some other order that enhances the speed of generating the identifiers. In this section, we discuss numeric sequences first and then we present non-numeric sequences.

3.1. Numeric Sequence

In this technique, the resource identifier generation is in numeric order. For example, for a 4-bit, 16-entry reorder buffer, the sequence could start from 0 and run through 15 and roll back to 0. While simple to understand, a straight binary encoding of a numeric sequence does not necessarily lead to the fastest counter implementation.

3.2. Non-numeric Sequences

Since the primary goal of the design is to generate identifiers at a fast rate, a candidate sequence must use minimal *levels* of logic. Though there are a number of alternate sequences available, not all of them can be used to build a fast sequence generator. For example, Gray code and weighted codes are not suitable for sequence generation as they make slower counters (or incrementers) and at best can only match up with the straight binary (numeric sequence) counter.

3.2.1. Pseudorandom sequence

Table I: *Complete 4-bit LFSR sequence. All-zero state "0000" is inserted after "1000"*

0001
0011
0111
1111
1110
...
0100
1000
0000

A pseudorandom sequence is a maximal length sequence formed by a characteristic polynomial for a given *n*-bit number that can be easily realized by Linear Feedback Shift Register (LFSR) and a few Exclusive-OR (XOR) logic gates [11]. The characteristic polynomial for *n*-bits has the property of generating 2^n-1 numbers (as an all-zero value cannot be generated using the LFSR and the XOR circuit without additional hardware). Using zero-insertion circuit, it is possible to generate all the 2^n numbers in a non-numeric sequence [12]. Maximal length sequences have found application in pseudo-exhaustive and exhaustive testing and pseudo-random testing [12] and in the generation of store addresses with on-line fault-detection capability [13].

Table I lists the 4-bit complete LFSR sequence generated using the characteristic polynomial $x^4 + x + 1$. As per the polynomial, XORing the most and the least significant bits produces the least significant bit of the successor; while the three most significant bits of the successor are obtained by left-shifting the three least significant bits of the present stage. All-zero state is inserted so that the hardware requirements are minimum [12].

4. Implementation

Any sequence can be implemented in a variety of ways depending on the amount of logic and storage used. Depending on the area and speed constraints of the design, logic can be traded for storage, and vice versa. In this section, we present two designs for both numeric and non-numeric sequences.

4.1. Numeric sequence generation

4.1.1. Partially-stored numeric sequence generation

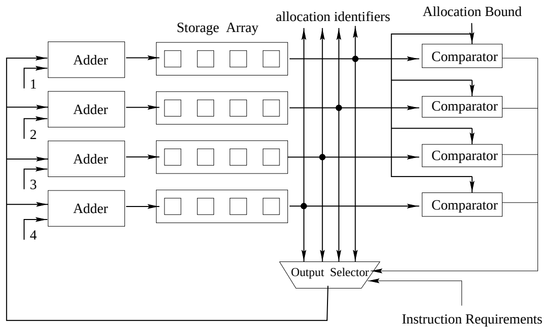


Figure 4: Four-bit, four-port resource allocator using partially-stored parallel numeric sequencers.

As the name indicates, in this technique we start with a partial set of numeric sequences and generate the next set of identifiers based on certain requirements. Figure 4 shows the organization of a partially-stored 4-bit, four-ported resource allocator which uses adders to generate the next set of identifiers. To facilitate fast identifier generation, fast adders like Carry Look Ahead adder (CLA) could be used. It may also be observed that one operand for each adder is constant and special optimization techniques for fast addition can be applied. The four adders operate in parallel to generate the next four identifiers following the highest value allocated in the current clock cycle (if 0 follows 15, 0 is considered to be higher of the two). The output selector chooses the highest value that gets allocated. The adders write the identifiers back to the four storage arrays, which in turn output these identifiers depending on the requirements of the instruction. In each clock cycle, depending on the allocation bound, and the requirements of the instructions, new set of identifiers are generated and stored in the storage arrays.

4.1.2. Fully-stored numeric sequence generation

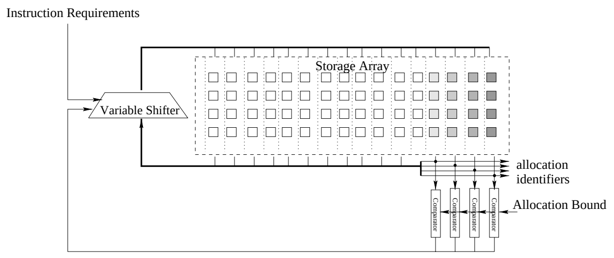


Figure 5: Four-bit, four-port resource allocator using fully-stored sequencer.

The adder delay can be eliminated if, instead of generating the next identifier every time, the sequences are stored completely and are output as per the requirements. This gives rise to the fully-stored numeric sequencer implementation illustrated in Figure 5. The storage array stores all the sequences in order and are indexed appropriately every cycle to generate the next identifier. As the next identifiers are not computed but are only indexed from the storage array based on the highest identifier allocated in each cycle, the bottleneck lies *only* in determining the highest identifier allocated in a given cycle. Thus, this implementation is expected to produce faster sequence generation.

Consider the design of a four-bit, four-port fully-stored numeric sequencer shown in Figure 5. The four fully shaded locations on the right side of the figure output the four resource identifiers every clock cycle. Depending on the requirements of the instruction and overallocation, if any, the identifiers for next cycle are determined. Since all elements in the sequence are stored in the storage array itself, next identifiers can be generated by shifting the array by an amount equal to the number of resource identifiers allocated in the current clock cycle. The variable shifter performs one, two, three or four shifts depending on the number of resources that need to be allocated in the cycle. The speed of this sequence generation depends on the speed of the shifter implementation is. Unlike partially-stored array implementation of Figure 4, this design needs a big multiplexer. The size of the multiplexer depends on the number of entries in the storage array and the number of bits in each array. A large multiplexer is usually composed of a number of smaller multiplexers, thus giving rise to larger delay. For a 4-bit, 16-entry storage array, the best timing optimization in Synopsys, targeting the 0.29µ library indicates that the maximum clock speed of the design is 471 MHz. As the design scales up to accommodate more entries in the reorder buffer, the design may not meet the timing constraints.

4.2. Non-numeric sequence generation

4.2.1. Partially-stored pseudorandom sequence generation.

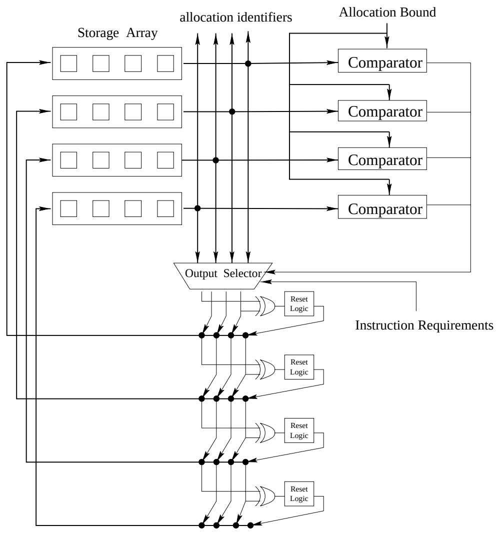


Figure 6: Four-bit, four-port resource allocator using partially-stored LFSR sequences.

As discussed in Section 3, the non-numeric pseudorandom sequences can be generated from a characteristic polynomial using XOR logic. Figure 6 illustrates a partially-stored 4-bit, 16-entry, four-port pseudorandom sequence generator that realizes the characteristic polynomial x^4+x+1 using XOR for next identifier generation and reset logic for zero-insertion. Here again, the output selector selects the highest identifier that was allocated in the current cycle and passes it to the next identifier generator, namely the XOR and the zero-insertion logics as shown in Figure 6. As in Table I, the zero-insertion logic inserts an all-zero state after the identifier "1000", and inserts a "0001" state after the all-zero state. In this design, there are two major delay components, namely, the *output-selector delay* and the *XOR-cum-reset logic delay.* This implementation is not particularly attractive because of serialization of the XOR-cum-reset logic circuitry. While the output-selector delay is unavoidable, the XOR-cum-reset logic delay provides scope for improvement.

4.2.2. Fully-stored pseudorandom sequence generation.

A fully-stored pseudorandom sequencer eliminates the XOR-cum-reset logic delay by avoiding the next-identifier generation, and storing the complete set of sequences in an array. An efficient storage array can be realized using an important property of pseudorandom sequences. Revisiting Table I, it can be seen that only the least significant bit of the next element needs to be computed, whereas the other three bits can be obtained by shifting the current element. This key property enables an elegant implementation for this pseudorandom sequence by storing only the least significant bit as in Figure 7. Besides savings in storage, this design significantly reduces the size and complexity of the shifter, thereby improving the timing characteristics. *Such an optimization is not possible for numeric sequencers*; this explains the size and complexity of implementation in Figure 5 in comparison to the implementation in Figure 7.

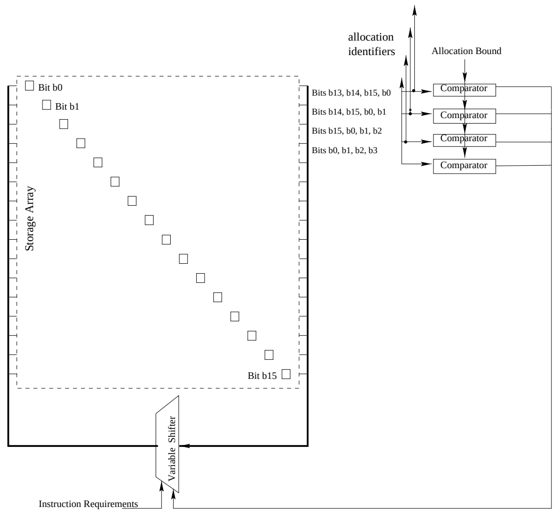


Figure 7: Four-bit, four-port resource allocator using fully-stored pseudorandom sequencer.

5. Results and Analyses

Resource allocators for various reorder buffer specifications were modeled in Verilog and synthesized in Synopsys [14][15] targeting the 0.29µ ASIC library. The results presented in this section correspond to the highest level of optimization that Synopsys could perform to minimize critical paths. Four-ported and eight-ported designs were implemented. Reorder buffer sizes of 16, 64 and 128 were considered. Preliminary results from the implementation in Figure 6 confirmed that the serial nature of the circuitry is a performance limiter. Hence, only the design trade-offs of Figure 4, 5 and 7 are presented in the forthcoming subsections.

5.1. Comparison of numeric and non-numeric sequence performances

Table II lists the results of best timing optimizations for various reorder buffer specifications. It can be clearly seen that the fully-stored pseudorandom sequencer has better timing compared to the other two. In particular, the fully-stored pseudorandom sequencer is, on an average, 17% faster than the partially-stored numeric sequencer. In contrast, the fully-stored pseudorandom sequencer requires 1.1 to 2.2 times more area than the partially-stored numeric sequencer. The fully-stored sequencer uses storage cells as wide as the number of entries in the ROB and a few multiplexers. For example, the ROB identifier for a 128-entry ROB requires only 128 pseudorandom storage bits and a few multiplexers. Thus, the area overhead is very negligible in modern processors, and the performance advantage of a fully-stored pseudorandom sequencer offsets any area issues. Fully-stored numeric sequencer, on the other hand, is significantly poor in area usage and yields mediocre timing characteristics. It may be noted that the clock rates in Table II are not particularly impressive compared to frequencies of modern processors, however, it should be remembered that we use 0.29µ process technology.

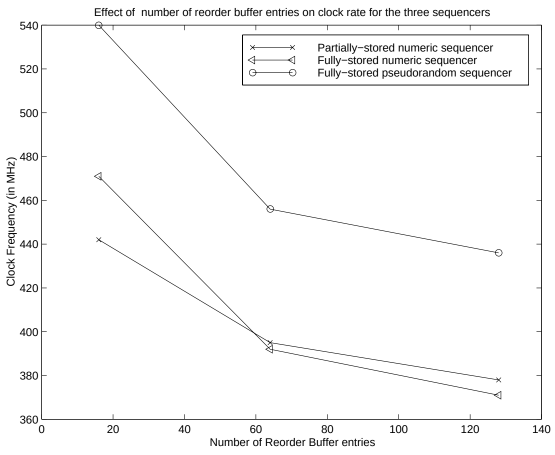


Figure 8: Effect of number of reorder buffer entries on clock rate for the three sequencers.

Figure 8 shows the plot of clock frequency of the three sequencers for different number of entries of four-ported reorder buffer. Fully-stored pseudorandom sequencer is superior in timing characteristics to fully-stored and partially-stored implementations of the numeric sequencer.

Table II also charts the degradation in the clock rates of the sequencers as the number of ports are increased. As the number of ports increases from four to eight, the maximum clock speed drops by about 18% for both the partially-stored numeric sequencer and the fully-stored pseudorandom sequencer. This is definitely a cause for concern if the future processors exploit more ILP by providing more number of ports in the reorder buffer. The technique discussed in this paper does illustrate an elegant and effective technique to boost the clock by 15-20%.

In the analyses presented so far, we have characterized the timing of the three sequencers based on the results of synthesis. In practice, many circuit tricks can be adopted to optimize the critical paths. For example, the GHZ processor from IBM [7][8] uses several such tricks and dynamic logic to achieve its high clock rate.The results in Table II do not involve any hand optimizations. Although one could argue that the results presented here depends on the specific design tool and methodology we adopted, we are confident about the usefulness of the results and their broad applicability based on the reasoning that the property of LFSRs reduces the number of *levels* of logic, thus operates at a higher speed than any other topology.

5.2. Effects of adder topology on numeric sequence generation

We analyzed different implementations of the design in Figure 4 by varying the adder circuitry. As noted earlier,

Table II: Result of synthesis (targeting LSI Logic's 3.3v G10TM-p 0.29µ ASIC library) of various resource allocators

Reorder Buffer Specification	***Design Characteristics***	***Partially-stored numeric sequencer*** (Figure 4)	***Fully-stored numeric sequencer*** (Figure 5)	***Fully-stored pseudorandom sequencer*** (Figure 7)
Four-ported 4-bits 16-entries	Critical Path Timing (ns) (Max. Clock Speed (MHz))	2.26 (442)	2.12 (471)	1.85 (540)
	Total Area[a]	832.375	2074.85	930.175
Four-ported 6-bits 64-entries	Critical Path Timing (ns) (Max. Clock Speed (MHz))	2.53 (395)	2.55 (392)	2.19 (456)
	Total Area[a]	1040.35	12977.075	2504.95
Four-ported 7-bits 128-entries	Critical Path Timing (ns) (Max. Clock Speed (MHz))	2.64 (378)	2.69 (371)	2.29 (436)
	Total Area[a]	1482.65	30368.5	4840.5
Eight-ported 7-bits 128-entries	Critical Path Timing (ns) (Max. Clock Speed (MHz))	3.25 (307)	-[b]	2.92 (354)
	Total Area[a]	2630.5		7722.325

[a] Equivalent gates; 1 equivalent gate = 1 two-input nand gate
[b] Synthesis proved to be extremely time consuming, hence, result is unavailable.

one input to each adder in Figure 4 is a constant (1, 2, 3 or 4) and this allows Synopsys to highly optimize these circuits.

Table III compares the results of timing optimizations performed on two partially-stored numeric sequencers (for four-ported, 16-entry reorder buffer), one realized using the best automatically synthesized adders, and the other realized using optimized Carry LookAhead adders (CLA). The synthesized adders were observed to have better timing characteristics compared to the CLA.

Table III: Comparison of the results of synthesis of 4-bit, 16-entry partially-stored numeric sequencers of Figure 4 using best synthesized adders and Carry LookAhead adders

	With the best synthesized adders	***With Carry LookAhead adder***
Critical Path Timing (ns) (Max. Clock Speed (MHz))	2.26 (442)	2.55 (392)
Total Area[a]	832.375	1218.7

[a] Equivalent gates; 1 equivalent gate = 1 two-input nand gate

This being the case, further tuning of the highly-optimized adder may not be possible. On the contrary, the fully-stored sequencers offer a possibility of further reduction in timing, and thus improvement in performance more than what is indicated in Table II.

6. Conclusion

This paper addressed the hardware and timing issues of hardware resource allocators for superscalar processors. We described the traditional numeric sequence method and compared it with non-numeric sequence generation techniques, in particular, a pseudo-random sequence. We analyzed the area and timing issues of some of the numeric and non-numeric sequence generation modeled in Verilog hardware description language and synthesized in Synopsys targeting LSI Logic's 3.3v G10TM-p Cell-Based 0.29µ ASIC library. Based on the results of timing optimizations, we found that a fully-stored pseudorandom sequence generator can enhance the clock speed by 15 - 20%. This improvement was possible due to some unique properties of the chosen pseudorandom sequence. The fully-stored pseudorandom sequence generator consumes approximately 1.1 to 2.2 times the area of the best numeric sequencer implementation. The fully-stored sequencer uses storage cells as wide as the number of entries in the ROB and a few multiplexers. For example, the ROB identifier for a 128-entry ROB requires only 128 pseudorandom storage bits and a few multiplexers. Thus, the area overhead is very negligible in modern processors, and the performance advantage of a fully-stored pseudorandom sequencer offsets any area issues.

As architects propose machines with wider widths and aggressive dynamic instruction scheduling techniques, it is essential to consider techniques that yield complexity effective designs. While optimizations and tricks based on dynamic logic are an option to boost the clock, any improvements that can be obtained using higher level design tactics enables designers to use synthesis tools, and use static logic, and still achieve high clock rates.

References

[1] Johnson M., *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[2] Colwell, R.P., and Steck, R.L.,"A 0.6µ BiCMOS processor with dynamic execution", *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, Feb. 1995, pp.176-7.

[3] Gaddis, N.B., Butler, J.R., Kumar, A., and Queen, W.J., "A 56-entry instruction reorder buffer," *IEEE International Solid-State Circuits Conference*. *Digest of Technical Papers*, 1996, pp.212-3.

[4] Smith, J.E., and Pleszkun, A.R., "Implementation of Precise Interrupts in Pipelined Processors," *Proc. of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 36-44.

[5] Leibholz, D., and Razdan, R., "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor," *Proc. of IEEE Compcon 97*, Feb. 1997, pp. 28-36.

[6] Kessler, R.E., McLellan, E.J., and Webb, D. A., "The Alpha 21264 Microprocessor Architecture", *International Conference on Computer Design: VLSI in Computers & Processors*, Oct. 1998, pp. 90-5.

[7] Nowka, K. J., and Galambos, T., "Circuit Design Techniques for a Gigahertz Integer Microprocessor", *International Conference on Computer Design: VLSI in Computers & Processors*, Oct. 1998, pp. 11-6.

[8] Posluszny, S. et al, "Design Methodology for a 1.0 GHz Microprocessor", *International Conference on Computer Design: VLSI in Computers & Processors*, Oct. 1998, pp. 17-23.

[9] Palacharla, S., Jouppi, N.P., and Smith, J.E., "Complexity-Effective Superscalar Processors", *Proc. of the International Symposium on Computer Architecture*, May 1997, pp. 206-18.

[10] Wallace, S., Dagli, N., and Bagherzadeh, N., "Design and Implementation of a 100 MHz Reorder Buffer," *Proc. of the 37th Midwest Symposium on Circuits and Systems*, vol.1, 1994, pp. 42-5.

[11] Wang, L.-T., and McCluskey, E.J., "Hybrid Designs Generating Maximum-Length Sequences," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 1, Jan. 1988, pp. 91-9.

[12] Wang, L.-T., and McCluskey, E. J., "Complete Feedback Shift Register Design For Built-in Self-Test," *IEEE International Conference on Computer-Aided Design*, 1986, pp. 56-9.

[13] Hsiao, M.Y., Patel, A.M., and Pradhan, D.K., "Store Address Generator with On-Line Fault-Detection Capability," *IEEE Transactions on Computer*s, vol. c-26, no. 11, Nov. 1977, pp. 1144-7.

[14] Synopsys Online Documentation, *Guidelines and Practices for Synthesis.*, v.1997-08.

[15] Synopsys Online Documentation, *Design Compiler Reference Manual.*, v.1997-08.