AN EVALUATION OF FUNCTIONAL UNIT LENGTHS FOR SINGLE-CHIP PROCESSORS

Matthew K. Farrens

Computer Science Division
University of California, Davis
Davis, CA 95616
(farrens@american.ucdavis.edu)

Andrew R. Pleszkun

Department of Electrical and
Computer Engineering
University of Colorado-Boulder
Boulder, CO 80309-0425
(arp@tosca.colorado.edu)

ABSTRACT

When designing a pipelined single-chip processor (SCP) with pipelined functional units of varying length, the processor issue logic must deal with scheduling of the result bus. In order to prevent serious performance degradation due to result bus conflicts, some pipeline scheduling techniques developed in the 1970's may need to be incorporated into the issue logic. Since this is a non-trivial complication of the issue logic, a set of simulations were performed in order to evaluate the effectiveness of the combination of multiple length functional units and scheduling techniques. Analysis of the simulation results indicates that providing relatively short multiple length functional units is not worthwhile. Multiple length functional unit configurations employing result bus scheduling do perform slightly better than uniform length configurations, but the difference is often less than 1%. Thus, the SCP designer should not waste valuable time improving the performance of each functional unit, but rather should produce a good design for the most complicated unit and design all other units to match it.

1. Introduction

When designing a single-chip processor (SCP) to support a particular instruction set architecture, the design engineer must carefully choose among the many different options available. The types and size of on-chip caches (if any), the type of instruction fetching strategy to pursue, the amount and degree of pipelining to incorporate, and whether or not to include on-chip floating point units are just a few of the design decisions that must be made. To aid in the option selection process, it is important to provide the designer with as much information as possible.

If the processor is to be pipelined, the designer must decide the structure and organization of the pipeline, including the extent the functional units will be pipelined. The first instinct of a high-performance processor designer is to make each individual functional unit as fast as possible, in order to avoid wasted clock cycles. For example, the more complex floating point units may be designed to take several clock cycles to produce a result, while the simpler logical unit may be able to produce its results in a single clock cycle. While on the surface this approach seems logical, closer inspection reveals certain problems with supporting multiple length functional units. When different length functional units are implemented with a CRAY-like issue strategy [Russ78], the issue logic must ensure that an instruction waiting to issue will not require the use of the result bus during a clock cycle when the result bus will be used by an already issued instruction. Being able to detect and deal with this result bus busy condition can significantly complicate the issue logic, and it is not clear that having multiple length functional units in a single chip processor provides a justifiable performance improvement.

In this paper, we will investigate the performance improvement resulting from using variable length functional units. This work is done in the context of SCPs where moderate amounts of pipelining are supported (most of today's SCPs). We first describe, by example, how pipeline scheduling for multiple variable length functional units leads to improved performance. Then, through simulation, we evaluate the performance of machines with different combinations of functional unit lengths.

2. Result Bus Scheduling

In a machine with multiple length functional units that are relatively short in length, conflicts over the use of the result bus frequently occur. Consider an instruction (Instruction A) that is prevented from beginning execution because the result bus will be in use during the clock cycle in which Instruction A wants to write its result back to the register file. By preventing Instruction A from issuing, the instructions behind it are also prevented from beginning. Overall throughput may be reduced if some of these instructions do not have the same result bus conflict

as Instruction A. Special pipeline scheduling techniques [PaDa76] exist that can be used to reduce the impact of these result bus conflicts and which are much less complicated than supporting out of order issue and execution [Toma67].

Figures 1 and 2 demonstrate how the technique of *result bus scheduling* improves the performance of a simple sequence of code. (In this example, we assume that arithmetic operations require two full clock cycles to complete, while logical and move operations require only one.) Without using pipeline scheduling, the code sequence shown in Figure 1 takes six clock cycles to complete. As can be seen in the figure, the OR instruction following the ADD cannot issue until the ADD completes, because at time 2, when the issue logic is first presented with the single cycle OR instruction, the issue logic knows that the result bus will be busy at the time the OR instruction will generate its result. This blocking of the OR instruction also delays the issue of the subsequent SUB instruction, which faces no such result bus conflict.

When result bus scheduling is employed, as shown in Figure 2, delays are inserted into the datapath forcing selected instructions to take longer to complete. The OR is issued at time 2, and although it can complete in the same clock cycle, the issue logic has it wait an extra clock cycle before writing a result. Since the OR has been issued at time 2, the issue logic can issue the SUB one cycle sooner. Thus, the blocked instruction and the ones behind it are able to proceed normally (assuming the issue logic detects no other hazards).

As can be seen by comparing Figure 1 to Figure 2, the use of pipeline scheduling will improve the performance of this code sequence by over 16%. Based on the results of the small section of code presented above and the results presented in [HsPG84], it would appear that performing some sort of result bus scheduling in an SCP could be a valuable performance enhancing technique. Scheduling the result bus, however, requires hardware that is capable of dynamically recognizing the situations in which adding additional delays to an instruction will

Instruction	Time →					
	1	2	3	4	5	6
ADD	*Issue*	**Finish**				
OR			*Issue* **Finish**			
SUB				*Issue*	**Finish**	
MOVE						*Issue* **Finish**

Figure 1.

Without Pipeline Scheduling.

Instruction	Time →					
	1	2	3	4	5	6
ADD	*Issue*	**Finish**				
OR		*Issue*	**Finish**			
SUB			*Issue*	**Finish**		
MOVE				*Issue*	**Finish**	

Figure 2.

With Pipeline Scheduling.

improve program through-put. In our example, the OR instruction may take two cycles or one cycle to execute, depending on its context in the instruction stream. Its execution time is dynamically determined by the issue logic. Results to be presented later in this paper indicate that a much simpler and less hardware-intensive scheme can achieve nearly the same performance benefits.

3. The Simulation Environment

In order to study the effectiveness of result bus scheduling, a performance metric was needed. The one chosen for use in this study was the the total number of clock cycles required to execute a set of benchmark programs. This metric was chosen because the best way to evaluate the performance of various internal processor configurations is to monitor the amount of time the processor takes to complete a test program with a given configuration [Smit88]. It is important to remember that a solution that takes fewer clock cycles but requires significantly more complicated hardware may actually take more real time to execute than a simpler scheme with less complicated hardware. (This is, in fact, the basic argument behind reduced instruction set processors [Patt85].)

3.1. The Simulator

The simulator used in this study was a modified version of the PIPE simulator, which was written to facilitate the study of the PIPE processor [Farr89]. The PIPE processor, a single chip processor designed and built at the University of Wisconsin, is an outgrowth of the PIPE project [GHLP85] and employs a load/store register-register architecture similar to the CRAY and CDC architectures [Russ78, Thor70] with an elemental (or reduced) instruction set (designed both for ease of decode and to simplify the hardware issue logic), a 5-stage pipeline, a small on-chip instruction cache and sophisticated instruction fetch support logic [FaPl89], limited subroutine call support, a branch mechanism that allows the compiler to specify the number of delay slots after a branch, and architectural input and output queues at the processor/memory interface.

These I/O queues are perhaps the most distinctive feature of the PIPE processor, and are used to provide an insulating buffer between the processor internals and the outside world. This allows the processing elements of the chip to be clocked at a rate determined solely by the delays through the various processing elements, and prevents external effects such as memory speed from affecting the internal system clock. Making the queues visible to the programmer also makes it possible to schedule memory requests such that the impact of a slow external memory on processor performance is significantly reduced [YoGo84]. The use of I/O queues also impact the design of a processor in ways that are more subtle and less obvious. Throughout the rest of this paper we will attempt to point out some of these unexpected advantages.

In order to perform this study, the modifications to the PIPE simulator gave the processor issue logic the capability of doing a very simple type of result bus scheduling. The result bus scheduling was done as follows: If an instruction waiting to be issued requires the result bus at time t, and the issue logic determines that the result bus will be busy at that time, the issue logic checks to see if the result bus will be free at time t+1. If it *is* free at t+1, the output of the instruction in question is delayed by a single clock cycle, the result bus is marked busy at time t+1, and the instruction is allowed to issue. If, on the other hand, the result bus is also busy at time t+1, the instruction is not allowed to issue during that clock.

3.2. The Simulation Model

The memory system is modeled as a single large memory that services both instruction and data requests, connected to the processor chip by unidirectional input and output busses. The external floating point unit is memory mapped, so that a pair of data stores to the appropriate locations will cause a multiply to occur. The simulation model gives bus precedence to instruction fetches, followed by data loads and stores, with multiply results getting the bus whenever it is idle.

Giving instruction fetches top priority like this helps keep the instruction fetch unit ahead of the decode unit, and since the processor has been designed to tolerate slow memory, the extra clock cycles required by a data fetch that has been preempted by an instruction fetch are effectively hidden. This is an example of the less obvious advantages of using I/O queues.

3.3. The Benchmark Program

The benchmark programs selected were the first 14 Lawrence Livermore loops as defined in [McMa84]. The loops were first compiled by the SUN4 optimizing compiler, in order to get a feel for the kinds of optimizations a compiler could perform. The loops were then completely hand-written using the output of the SUN compiler as a guide. A serious effort was made not to hand-optimize the loops, however. The loops are not "tuned" to increase performance, as this might limit the information that could be gained from the interpretation of the results.

In an effort to make the results obtained from the study applicable to the general case of processors that do not use I/O queues, an additional set of Lawrence Livermore loops were created that do not take advantage of the I/O queues. In these loops, a data load instruction is followed immediately by the instruction that requires the data item. Since the entire PIPE architecture and

instruction set is designed to take advantage of these I/O queues, the simulation results will only approximate the general case. The PIPE processor can only behave *similarly* to a machine without queues. It can behave similarly enough, however, to allow some general conclusions to be drawn.

The 14 loops were assembled as one large program, so that each loop would run until finished and then fall through to the next loop. The variant of each loop was modified so that each loop executes approximately 10,000 instructions. This was done to balance the impact of the different loops on the results. A total of 139,608 instructions are executed in a single run through the original benchmark program, and 140,267 are executed in a run through the modified benchmark that does not take advantage of the I/O queues.

4. Discussion of Simulation Results

In this section, we evaluate the performance of various functional unit lengths and the impact of result bus scheduling techniques. The simulation results necessary to perform this evaluation were generated by executing the two suites of benchmark programs (those with and without queues) on the PIPE simulator, varying the following parameters:

1. The number of clock cycles required for the different functional units.
2. The ability of the issue logic to do dynamic bus scheduling.
3. The time necessary to preform a floating point multiply.

The remaining simulation parameters were set so that the simulation results would highlight the effects of result bus scheduling and minimize the number of clock cycles lost due to conditions not related to functional unit length (such as the Instruction Register being invalid). In order to accomplish this, the instruction cache size was set large enough to contain the entire program and it was started warm. In addition, the memory delay was set to a single clock cycle to guarantee there would be no processor blocking due to waiting for a data item from memory.

The results of these simulations are presented in Tables 1 and 2. Table 1 contains the results of the simulations with a four cycle external multiply unit and the processor queues fully enabled, while Table 2 contains the simulation results for a four cycle external multiply unit and the processor queues *not* fully utilized.

The simulation results are presented in tabular form because, while looking at a graph is generally preferred to reading numbers out of a table, this data does not lend itself to graphical representation. A number of different graphical formats were created, but in each case the information of interest was easier to extract from the table itself than from the graph. Therefore, the reader should be prepared for frequent references to entries in Tables 1 and 2 throughout the following discussion.

The best place to begin is to look at where the implemented PIPE processor falls in Table 1. PIPE employs a two cycle arithmetic unit, single cycle shift and logical units, fully utilizes its I/O queues, and does no bus scheduling. Looking under the appropriate entry in column 1 of Table 1, we see that the PIPE processor takes a total of 172,776 clock cycles to execute the benchmark program. Going across the table we see that providing the ability to do result bus scheduling would provide a 10% performance increase.

The impetus behind this study was initially to try and determine if replacing PIPE's restricted single cycle shift unit with a fully functional two cycle shift unit would have a serious negative affect on performance. (The PIPE processor was implemented in single layer metal nMOS, and the amount of polysilicon necessary to build the shifter prevented a fully functional design.) Interestingly, Table 1 shows that the as-implemented PIPE processor would actually perform slightly *better* (1.7% better) with a two cycle shifter than it does with a one cycle shifter. This is separate from the implementational difficulties of producing a single cycle shifter; these simulation results show that the total number of clock cycles used *decreased* with an increase in the time required to do a shift. This result is not what one would first expect, and has to do with the result bus scheduling problem. When result bus scheduling is not being used, the more uniform the lengths of the functional units, the less time that is lost to result bus busy conditions.

Since making the functional units more uniform in length seems to provide a slight increase in performance, the next logical step is to look at how the processor performs when all functional units lengths are the same. Looking at the results in Table 1 we see that the processor actually performs best under these circumstances. This says that the PIPE processor would perform 9.3% faster if the shift and logical units were *slowed down*. This is perhaps the most significant and initially counter-intuitive result to come out of this study, and implies that an SCP designer may be able to significantly improve the performance of a processor by slowing down the different functional units until they all function at the same speed. Doing this has the added advantage of removing the need for result bus scheduling.

A processor using result bus scheduling in conjunction with minimal length functional units is somewhat faster than a processor using uniform (slower) functional units, as one would expect. Comparing the implemented PIPE processor with result bus scheduling enabled to a processor configuration with all functional units of length two shows that the PIPE setup is in fact slightly faster. However, the difference is just over 0.6%. It is highly

Table 1. Multiple Length Functional Unit Simulation Results, Processor Utilizing Queues.

With I/O Queues, Floating Point Unit Delay = 4 clocks				
Functional Unit Delays (in Cycles)			Result Bus Scheduling OFF	ON
Arith. = 1	Log. = 1	Shift=1	140631	140631
Arith. = 2	Log. = 1	Shift=1	172776	157044
		Shift=2	169945	157602
	Log. = 2	Shift=1	162979	157817
		Shift=2	158079	158079
Arith. = 3	Log. = 1	Shift=1	195284	185075
		Shift=2	195131	185827
		Shift=3	198574	189595
	Log. = 2	Shift=1	194168	181811
		Shift=2	192946	182267
		Shift=3	196394	184395
	Log. = 3	Shift=1	189027	184381
		Shift=2	185397	185074
		Shift=3	186776	186776
Arith. = 4	Log. = 1	Shift=1	217493	209466
		Shift=2	219614	212643
		Shift=3	222402	215402
		Shift=4	225689	219029
	Log. = 2	Shift=1	221605	211881
		Shift=2	221296	213250
		Shift=3	224261	215688
		Shift=4	227827	219457
	Log. = 3	Shift=1	222576	208494
		Shift=2	222495	212438
		Shift=3	224391	214457
		Shift=4	227962	216726
	Log. = 4	Shift=1	218024	213552
		Shift=2	215773	215450
		Shift=3	217910	217710
		Shift=4	219551	219551

Table 2. Multiple Length Functional Unit Simulation Results, Processor Not Utilizing Queues.

Without I/O Queues, Floating Point Unit Delay = 4 clocks				
Functional Unit Delays (in Cycles)			Result Bus Scheduling OFF	ON
Arith. = 1	Log. = 1	Shift=1	184058	184058
Arith. = 2	Log. = 1	Shift=1	235403	218499
		Shift=2	232097	219266
	Log. = 2	Shift=1	228211	222511
		Shift=2	222705	222705
Arith. = 3	Log. = 1	Shift=1	253847	244444
		Shift=2	253497	245832
		Shift=3	255762	247878
	Log. = 2	Shift=1	255825	244969
		Shift=2	254427	244971
		Shift=3	256349	246380
	Log. = 3	Shift=1	253873	249060
		Shift=2	250292	249183
		Shift=3	250014	250014
Arith. = 4	Log. = 1	Shift=1	292939	285769
		Shift=2	294574	287333
		Shift=3	296294	289439
		Shift=4	299001	292147
	Log. = 2	Shift=1	295548	288817
		Shift=2	294376	288865
		Shift=3	295891	291013
		Shift=4	298426	293207
	Log. = 3	Shift=1	299846	288941
		Shift=2	299007	290610
		Shift=3	299474	291519
		Shift=4	301666	293053
	Log. = 4	Shift=1	299812	295002
		Shift=2	295759	294823
		Shift=3	296695	295856
		Shift=4	296687	296687

unlikely that a 0.6% performance improvement can justify a decision to choose incorporating the more complicated result bus scheduling technique into the issue logic over the much easier task of equalizing all functional unit lengths.

The final thing to note in Table 1 is that as the length of the functional units increases, the advantage gained by having uniform length functional units decreases. Once the functional unit length has reached four clock cycles, several of the designs with unequal lengths actually outperform the uniform length configuration. (However, the difference is less than 2%.) This implies that when functional units become long enough maximizing the performance of individual units may again become a useful approach. (This coincides with what is actually done in machines like the CRAY.)

Moving to Table 2, some similar results are evident. Table 2 contains the results of simulations in which the I/O queues were not fully utilized, and in these results we see again the impact of I/O queues on performance. As was the case in Table 1, the performance of the PIPE processor configuration does increase as the functional unit lengths become more uniform, but the performance increase is not as appreciable.

This difference in performance is due once again to the inherent properties of I/O queues. Programs for processors using I/O queues try to maximize the distance between a request for a data item and its consumption, based on the assumption that some unspecified amount of time after the request has been issued it will be serviced. What is causing the delay is immaterial; whether the delay is due to slow memory or multi-staged functional units, it is hidden by the proper use of these queues.

5. Summary and Conclusions

The goal of this paper was to determine if a processor should employ functional units of varying lengths. The natural instinct of many designers is to maximize the performance of each individual functional unit, which leads to multiple length functional units. If a processor does use multiple length functional units, it must be able to deal with the added issue condition of a busy result bus. Some pipeline scheduling techniques developed in the 1970's may also need to be incorporated into the issue logic to prevent serious performance degradation due to the result bus conflicts. Since this is a non-trivial complication of the issue logic, a set of simulations were performed in order to evaluate the effectiveness of the combination of multiple length functional units and this scheduling technique.

Analysis of the simulation results indicates that having relatively short multiple length functional units is not worthwhile. Multiple length configurations employing result bus scheduling do perform slightly better than uniform length configurations, but the difference is less than 1%. Since configurations with all functional units the same length consistently outperformed configurations with functional units of different lengths when result bus scheduling was not enabled, the added complexity of supporting this technique is not justified.

These results indicate that an SCP designer should not waste valuable time squeezing performance out of the various functional units, but rather should produce a good design of the most complicated unit and design all other units to match it. However, it should be pointed out that these results are for a single set of hand-written benchmark programs, and processors without I/O queues could only be emulated, so the results may or may not be directly transferable to designs without queues. However, the results themselves are very interesting, and indicate that there is much more work that can be done in this area.

Acknowledgements

This work was supported by National Science Foundation Grants DCR-8604224, CCR-8706722, and CCR-9011535. We would also like to extend a special thanks to the members of the PIPE project, for creating an excellent research environment.

References

[Farr89] M. K. Farrens, *The Design and Analysis of a High Performance Single Chip Processor*, Ph.D. Thesis, Department of Electrical and Computer Engineering,, Madison, Wisconsin, (August 1989).

[FaPl89] M. K. Farrens and A. R. Pleszkun, ''Improving the Performance of Small On-Chip Instruction Caches'', *Proceedings of the Sixteenth Annual International Symposium on Computer Architecture*, vol. 17, no. 3 (June 1989), pp. 234-241.

[GHLP85] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter and H. C. Young, ''PIPE: a VLSI Decoupled Architecture'', *Proceedings of the Twelveth Annual International Symposium on Computer Architecture*(June 1985), pp. 20-27.

[HsPG84] J. T. Hsieh, A. R. Pleszkun and J. R. Goodman, ''Performance Evaluation of the PIPE Computer Architecture'', Computer Science Department Technical Report #566, University of Wisconsin-Madison , Madison, Wisconsin (November 1984).

[McMa84] F. H. McMahon, *LLNL FORTRAN KERNELS: MFLOPS*, Lawrence Livermore Laboratories, Livermore, California, (March 1984).

[PaDa76] J. H. Patel and E. S. Davidson, ‘‘Improving the Throughput of a Pipeline by Insertion of Delays’’, *Computer Architecture News (ACM-SIGARCH)*, vol. 4, no. 4 (January 1976), pp. 159-164.

[Patt85] D. A. Patterson, ‘‘Reduced Instruction Set Computers’’, *Communications of the ACM*, vol. 28, no. 1 (January 1985), pp. 8-21.

[Russ78] R. M. Russell, ‘‘The CRAY-1 Computer System’’, *Communications of the ACM*, vol. 21, no. 1 (January 1978), pp. 63-72.

[Smit88] J. E. Smith, ‘‘Characterizing Computer Performance with a Single Number’’, *Communications of the ACM*, vol. 31, no. 10 (October 1988), pp. 1202-1206.

[Thor70] J. E. Thorton, *Design of a Computer - The Control Data 6600*, Scott, Foreman and Co, Glenview, Illinois, (1970).

[Toma67] R. M. Tomasulo, ‘‘An Efficient Algorithm for Exploiting Multiple Arithmetic Units’’, *IBM Journal*, vol. 11 (January 1967), pp. 25-33.

[YoGo84] H. C. Young and J. R. Goodman, ‘‘A Simulation Study of Architectural Data Queues and Prepare-to-Branch Instruction’’, *Proceedings of the IEEE INT Conference on Computer Design: VLSI in Computers*, Port Chester, New York (October 1984), pp. 544-549.