Register Organization for Enhanced On-chip Parallelism

Rama Sangireddy
Department of Electrical Engineering
University of Texas at Dallas, Richardson, TX 75080, USA
rama.sangireddy@utdallas.edu

Abstract

Large register file with multiple ports is a critical component of a high-performance processor. A large number of registers are necessary for processing a larger number of in-flight instructions to exploit higher instruction level parallelism (ILP). Multiple ports for a register file are necessary to support execution of multiple instructions each cycle. These necessities lead to a larger register access time. However, register access time has to be minimal to enable design of high frequency processors. Analysis of lifetime of a logical to physical register mapping reveals that there are long latencies between the times a physical register is allocated, consumed, and released. In this paper, we propose a dual bank register file organization that exploits such long latencies, resulting in a large bandwidth with a reduced register access time. Implementation of one flavor of the proposed register file organization, as compared to a conventional monolithic register file, in an 8-wide out-of-order issue superscalar processor enhanced instructions per cycle (IPC) throughput up to 6% for Spec2000 applications while reducing register access time up to 22%. Another flavor of the register file organization, with a similar access time as the conventional monolithic register file, enhanced the IPC up to 15%. Thus a trade-off between register access time and ILP exploitation is shown.

1 Introduction

Wide issue processors require multiple ports in the register file which has an adverse affect on the register access time. Besides, a wide-issue processor is effective only if as many instructions as possible are issued during each cycle, which implies that the processor has to view a larger instruction window to achieve sufficient amounts of instruction level parallelism (ILP). Large instruction window implies a larger set of in-flight instructions requiring a larger number of physical registers. However, increasing the size of register file adversely affects the register access time.

Register access time plays a critical role in determining the processor clock cycle time [1]. As the width of the processor pipeline increases, its complexity increases, and a wide scale effort is underway to reduce the complexity and its effects in the processor design. Farkas *et al* [2] have shown that an 8-wide issue superscalar processor handling precise exceptions increases the average instruction throughput (IPC) as the register file size is increased up to 256. Besides, the portion of execution time during which there is at least one free physical register available, for logical register renaming at dispatch stage, increases considerably as the register file size increases. This is important for dispatching as many instructions as possible to instruction window to draw more ILP. However, the processor loses performance in terms of average number of billion instructions per second (BIPS) for a register file size beyond 128, due to the adverse impact of the large register file access time on the processor cycle time. On the other hand, Lebeck *et al* have proposed a fast instruction issue window assisted by a large wait instruction buffer (WIB) of size up to 2K entries for extracting higher amounts of instruction level parallelism (ILP) [3]. For such a large instruction window size to be utilized to the maximum extent possible, availability of a large number of physical registers is necessary to process as many instructions as possible at the dispatch stage.

In this paper, we propose an effective register file organization that performs well in meeting the main goal of providing a large number of registers to enable dispatching as many instructions

Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)
1063-6862/04 $ 20.00 IEEE
IEEE COMPUTER SOCIETY

as possible to issue window for extracting higher ILP. The scheme achieves that with a smaller register access time, compared to a conventional monolithic register file of same bandwidth, to enable a faster processor cycle time. The register file organization exploits long latencies involved, in between allocation of register to a logical value and actual consumption of the value by a functional unit, and then in between consumption of the value and actual freeing of physical register for next allocation. The proposed register file organization can be adapted to other processors designed to be application-specific, by suitably performing the register life time analysis of such processors in similar methodology as this paper.

The rest of the paper is organized as follows. Section II provides a detailed analysis of the register life time as a precursor for our proposed design. Section III discusses the related research. In Section IV, we present a detailed design of the proposed register file organization. The section also discusses the necessary modifications in the microarchitecture. Section V analyzes the performance of the architecture. Section VI concludes the paper.

2 Register lifetime Analysis

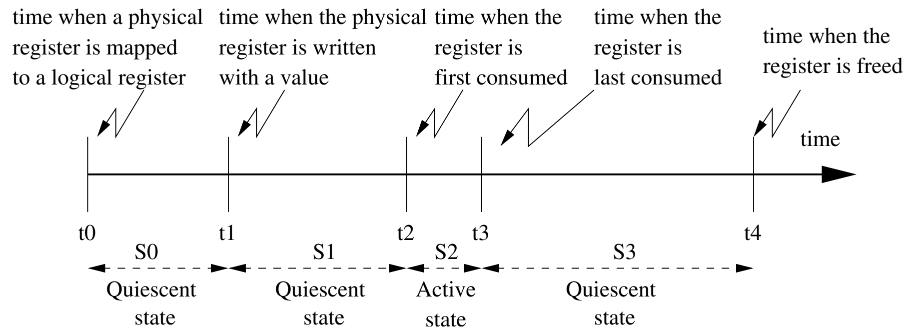

Figure 1. Various stages in the lifetime of a physical register, for a particular mapping to a logical register.

While undertaking the task of designing an effective register file for high performance processors, first we study and analyze the activity of a physical register during its lifetime of one logical to physical mapping. For this purpose, we consider a DEC Alpha 21264 processor [4] based microarchitecture. The Alpha 21264 processor consists of a deep pipeline with fetching and renaming of instructions performed in-order. When the source operands of an instruction waiting in issue stage are ready, the instruction is issued for execution. Once granted execution, the register tags of source operands of the instruction are used to access the register file in the register read stage of the pipeline. The operand values read from the register file are forwarded to the appropriate functional unit in the execute stage of the pipeline. If a dependent operation is issued for execution immediately following the current instruction, the dependent instruction will read a stale value from the physical register file. A bypass logic is provided in the execute stage to select between the incoming register operand, or a more recent value on the bypass bus. Dependent instructions that execute in immediate subsequent cycles will communicate via the bypass bus. All other instructions communicate through the physical register file.

The logical destination register for an instruction fetched is mapped to a free physical register at the dispatch stage. Subsequent instructions with the same logical register as their source operand are assigned to read from the mapped physical register. The logical to physical register mapping remains active until another instruction with same logical register as its destination enters the dispatch stage. The logical destination of that instruction is then mapped to another free physical register and the process continues. The dispatch stage in the pipeline is stalled when no free physical register is available. This scheme of logical to physical register mapping eliminates the write after read (WAR) and write after write (WAW) data dependencies. The earlier allocated physical register is freed only when the subsequent instruction with same logical destination is committed. This is

Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)
1063-6862/04 $ 20.00 IEEE
IEEE COMPUTER SOCIETY

done to enable the recovery from branch mis-predictions and handle exceptions precisely. The conditions for freeing registers are described in more detail in [2].

The life cycle of a physical register is identified as the time between its allocation to a logical destination at the dispatch stage and the time when it is freed. The various stages in the register lifetime, as illustrated in Figure 1, are:

t0: Time at which a free physical register R_p is allocated to a logical destination register R_l of an instruction I_k at the dispatch stage in the processor pipeline.

t1: Time at which R_p is written with a value. This happens when the instruction I_k is in the writeback stage.

t2: Time at which the value in R_p is first consumed. This happens when an instruction I_{k+x}, with a logical source operand of R_l, is issued for execution.

t3: Time at which the value in R_p is consumed for the last time. This happens when an instruction I_{k+y} $(y > x)$, with a logical source operand of R_l, is issued for execution. And, no further instructions use R_l as source operand until the R_l becomes a destination register for another instruction I_{k+z}, where $z > y > x$.

t4: Time at which the physical register R_p is freed and is ready for the next allocation. This happens when instruction I_{k+z} is committed to ensure the recovery from any precise exceptions.

A conventional monolithic register file maintains the mapping of the physical register throughout the life cycle of each logical to physical mapping. Note that at the microarchitecture level it is easy to identify when a register value is first consumed. However, it is not easy to determine when it is consumed for the last time. To do so, it requires a large overhead of keeping track of all the instructions that are potential customers for the operand using some sort of counting mechanism to track the instructions as and when they are executed. In this study, we identify the time of last consumption of a register value only for the purpose of analyzing the register activity during lifetime of its allocation to a logical register. Using that we develop our architecture where we do not have to keep track of time of last consumption of a register.

To study a relationship among these various times, we used Simplescalar-3.0 [5] for the Alpha AXP instruction set to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters depicted in Table 1. The instructions are traced along the various stages of the processor pipeline and the time intervals between various stages in the lifetime of a register are measured according to the above mentioned specifications. The time intervals measured are:

t1-t0: Time during which the register is waiting for the result to be written into it after it is allocated.

t2-t1: Time during which the register is waiting to be read by a functional unit after it is written into.

t3-t2: Time during which the register is active as supplier of an operand to functional units.

t4-t3: Time during which the register is waiting to be freed after it is consumed for the last time.

Table 1. Simplescalar simulation parameters.

Parameter	*Value*
Instruction cache	32KB, 2-way, 32B line, 1 cycle latency
Data cache	32KB, 4-way, 32B line, 1 cycle latency
Branch predictor	bimodal, 2K table size, 7 cycle mispred. latency
Instruction issue queue size	64 (INT and FP, each)
Load/store queue size and *ReOrder buffer size*	128
Decode,Issue,Commit width	2/4/8/16

The average time interval between each stage of register lifetime is shown for various Spec benchmarks in Figure 2. The upper chart shows the register lifetime in absolute number of cycles and the lower chart illustrates the same in terms of percentage of time for each interval. The benchmark programs are simulated on the superscalar processor with various issue widths. The register lifetime

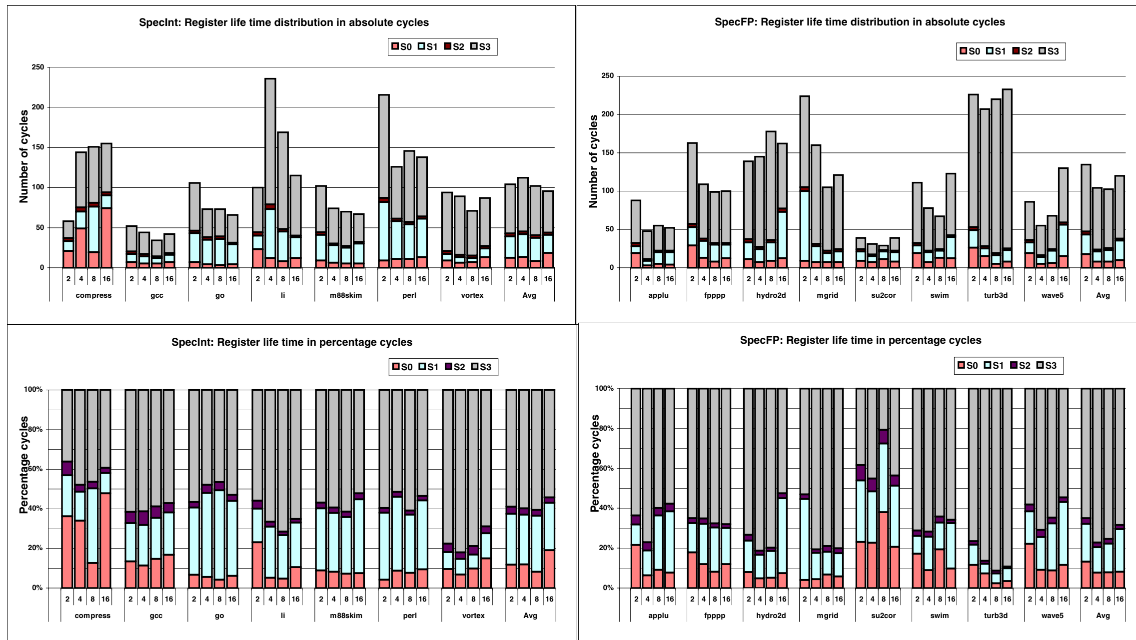

Figure 2. Physical register lifetime distribution in (above) absolute number of cycles (below) percentage of lifetime.

can be classified, as illustrated in Figure 1, into an active state (S2) where the register is supplying the values to functional units, and quiescent states (S0, S1, and S3) where the physical register is inactive waiting for some action to take place. It can be observed that the average active time of the register is exceptionally small (around 1% to 5%). This is mainly due to two reasons. First, it is observed that around 85% of the time a register value is read at most once. Second, some registers are never read as the value they hold are either supplied to their consumers through the bypass logic or even not read at all. The observations that emanate from the above analysis are:

- Physical registers are allocated at dispatch stage, early in the pipeline, and experience a long latency before consumption.
- Amount of duration when the register is an active supplier of values to consumers is very small as compared to its long lifetime.
- After the last consumption by a functional unit, there is a long latency before a register is freed up.
- A more aggressive logical to physical mapping at dispatch stage can be obtained by hiding the latency in freeing of registers, with the support of a mechanism to handle precise exceptions.

3 Related Research

Alpha 21264 microprocessor [4] uses a replicated register file organization to reduce the number of ports, wherein each copy can be accessed by only a few functional units. Reducing the number of ports is an effective technique to reduce register access time. Recently, Tseng *et al* [17] have examined the designs of such multiple bank with fewer ports to reduce power and area.

Cruz *et al* [16] used a multiple-banked organization for implementing a two-level register file. The level two (L2) register file is a large bank that holds all the register values and is used for logical to physical register mapping. The level one (L1) register file, a smaller bank and closer to the ALU, maintains a copy of those L2 registers that would be potential suppliers of values to the functional units. The scheme was proposed based on the analysis that at any instant only a small subset of

IEEE COMPUTER SOCIETY

registers in a large conventional register file actively supplies operand values to the functional units. This organization takes advantage of the latency in the quiescent states S0 and S1 when the register is waiting to be consumed and the small active state S2. The register value is fetched from L2 to L1 when ready to be consumed. The organization performs well in reducing the register access time by maintaining a small L1 register file. However, the scheme does not exploit the long latency in quiescent state S3, as the L2 register file still holds the values until the registers are freed. The L2 register file is used for logical to physical register mapping. Hence, this does not support the architectures that use aggressive techniques for processing a large number of in-flight instructions. Further, implementation of a large L2 register file results in a large latency in access from L2 to L1.

Balasubramonian *et al* [19] proposed and evaluated two orthogonal designs - two level and multi-banked register file. In the two-level organization, L1 structure is used for logical to physical register mapping and holds the values until they are consumed. After the consumption the register is moved to L2 and maintained until it is freed. This organization exploits the quiescent state S3 where the register is waiting to be freed. Such implementation helps in organizing a relatively smaller L1 register file as compared to a conventional single monolithic register file, and hence reduces the register access time. However, the microarchitecture keeps track for each physical register value when it is consumed for the last time. To do so, the microarchitecture requires a large overhead of keeping track of all the instructions that are potential customers for the operand using some sort of counting mechanism to track the instructions as and when they are executed. The approach in this case and in [18], with multiple interleaved register banks, results in difficulty in managing the complexity.

Apart from the above, various other techniques have been proposed in the past for an effective utilization of register resources. Borch *et al* [15] have recently proposed the caching of registers. A software-controlled two-level hierarchical register file organization was implemented in Cray-1 [6]. Swensen and Patt [7] proposed a hierarchical non-inclusive register file for a statically scheduled architecture, where register allocation is performed by the compiler. Wallace *et al* [8] proposed a scalable register file architecture, that uses multiple banked register file and maps the result to a physical register at the write stage in the processor pipeline. Yung *et al* [9] proposed a *Register Scoreboard and Cache* scheme wherein a subset of the registers are cached in a fast bank with an LRU replacement policy. The register deallocation policies have been discussed in detail by Moudgill *et al* [10], where they proposed a mechanism that implements register renaming, dynamic speculation and precise interrupts.

The HP PA-8000 [11] processor implementation maintains a logical register file that holds committed values, and the rename registers are maintained in a separate buffer. To reduce the register file access time, Tremblay *et al* [12] proposed a *3-D Register File*. In that, it was shown that the data array portion of the register file can be significantly reduced by designing the register file in multiple planes, where a plane in 3-D is a set of registers in 2-D. Zyuban and Kogge [13] have developed energy models for multi-ported register files with a variety of architectural parameters, and assert that the centralized register files would become the dominating power component of next-generation superscalar computers. A *Split register file* architecture was proposed by them as an energy-efficient alternative design. Gonzalez *et al* [14] proposed a *virtual registers* architecture with a strategy to reduce the pressure on register file by delaying the allocation of physical registers until instructions complete, instead of doing it in the decode stage.

4 Proposed Register file organization

Using the observations from the register lifetime analysis, we develop a dual bank register file architecture as shown in Figure 3(a). A dual bank register file organization consists of two banks of physical registers with a heterogeneous structure, as shown in Figure 3(b). Each register bank consists of a different number of registers and different number of ports according to the architecture requirements as discussed later. Our register file architecture is similar in spirit to the two level register file architecture proposed by Balasubramonian *et al* [19], but the proposed implementation is different. We design the register file architecture to avoid the large overhead of book-keeping the instances of physical register value consumption.

IEEE COMPUTER SOCIETY

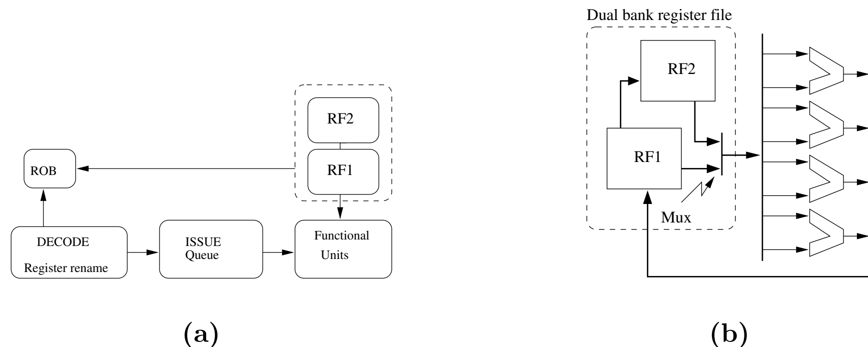

Figure 3. (a) A dual bank Register file in the pipeline of the processor, (b) A dual bank Register file organization.

The physical register bank RF1 consists of sufficient read and write ports to support instruction issue bandwidth. The physical registers in RF1 are used for logical to physical register mapping at dispatch stage. Results from the functional units are always written to registers in RF1.

From Figure 2, we observe that the average time during intervals t2-t0 (= S0+S1) and t4-t3 (S3) is around 45-50% each. Also the duration of the interval t3-t2 (S2) is negligible, as in most cases the register value is read only once or not read at all. Hence, we propose that there be an equal number of registers in RF1 and RF2 banks. The register values in RF1 are written to RF2 whenever RF2 has free registers and thus simultaneously freeing the corresponding registers in RF1. Thus, the register in RF1 is freed up much earlier than that is done in a conventional monolithic register file. Subsequently the freed register in RF1 joins the free pool of registers used to map to subsequent logical destination registers. This leverages the processor to process a larger number of in-flight instructions to draw higher ILP. The following subsections discuss in detail the process of transferring register values from RF1 to RF2, and conditions for freeing of registers in RF1 and RF2.

4.1 Register Transfer from RF1 to RF2

At the register renaming stage, a free physical register in RF1 bank is used to allocate to a logical destination register at time t0. The value produced after the execution of the corresponding instruction is written in to the physical register in RF1 at time t1. Since the two register banks RF1 and RF2 are maintained to be of same size, a direct-mapping (one-to-one) scheme is followed, i.e., transfer of values from RF1 to RF2 is done strictly in a one-to-one correspondence. The transfer of a physical register value in RF1 to RF2 bank will happen upon the satisfaction of both the following conditions:

1. The corresponding physical register in RF2 bank should be free, and
2. the physical register in RF1 bank must have already obtained its value, i.e., time t1 must have elapsed in its current lifetime.

When the above conditions satisfy, register value in RF1 is transfered to the corresponding register in RF2. Subsequently the register in RF1 is freed and is ready for a mapping to a new logical destination register. In direct-mapping policy, for applications in which latency of quiescent state S3 for a register is much larger compared to the latency in state S0+S1 during the lifetime of a logical to physical mapping, most of the time the register in RF1 will be written to RF2 sometime after it is consumed. In this case, the effective lifetime of a register mapping in the processor's view is the latency in freeing the register in RF1, which translates to the long latency of freeing a register from RF2.

However, if the latency in state S0+S1 is larger than that in state S3, a register value might be written from RF1 to RF2 even before it is consumed. This will not hinder the reading of operands as the value can still be fetched from RF2. However, there is a pitfall in this scenario. Consider a case when a value in physical register pr5 is moved from RF1 to RF2 even before it is consumed and

IEEE COMPUTER SOCIETY

the register pr5 in RF1 is freed for next allocation. Subsequently the register pr5 obtains another value corresponding to the next logical renaming. A following instruction that sources the logical operand corresponding to pr5 in RF2 (former mapping) or the logical operand corresponding to pr5 in RF1 (later mapping) has to read the value correctly. To illustrate the scenario more clearly, consider the example shown in Figure 4(a).

Sometime in between the first two instructions shown, the value in pr5 is moved from RF1 to RF2. Subsequently pr5 in RF1 is free to be allocated and thus gets mapped to lr4. For the next two instructions shown, the reading of operands lr6 and lr4 happens to be from pr5 and thus operands have to be read as per the correct mapping. This scenario can be addressed by the following mechanism. When the value in pr5 is moved from RF1 to RF2, the register mapping table in the dispatch stage is updated by setting a flag for the mapping $lr6 \longleftrightarrow pr5$, along with for all those instructions sourcing lr6 and are beyond dispatch stage in the processor pipeline. Thus any subsequent instructions that source lr6 will be indicated accordingly to read the value from pr5 in RF2, and instructions that source lr4 will be indicated to read the value from pr5 in RF1. After a while, when pr5 in RF2 is released and the value in pr5 in RF1 (corresponding to lr4) moves to RF2, the flag is set for the mapping $lr4 \longleftrightarrow pr5$. This always ensures the reading of the operands from the correct register bank.

4.2 Freeing of registers in RF1 and RF2

As discussed above, a register in RF1 is freed whenever the register value is transfered to the corresponding register in RF2 bank. A register in RF2 is freed according to the conditions followed in the case of a conventional monolithic register file. That is, for a current logical to physical register mapping, the physical register in RF2 is freed when a subsequent instruction with same logical destination commits. The freed register in RF2 is again ready to assume another register value from RF1.

The proposed architecture recovers from branch mis-predictions as follows. In a conventional architecture, when a branch mis-prediction occurs, the instructions that are not yet committed following the mis-predicted branch are squashed from the pipeline along with the corresponding values written in register file and the logical to physical register mappings due to those instructions. Subsequently, for instructions that are issued the source operands are read according to the logical to physical register mapping performed before the branch instruction. In the proposed architecture, the corresponding values existing in RF1 and RF2 are squashed when branch mis-prediction occurs. For new instructions issued, the source operand values are obtained from either RF1 or RF2, where ever they exist. Consider the example shown in Figure 4(b).

Code with logical registers	Code with renamed registers
lr6 ← ... ;	pr5 ← ...
.... ;	....
.... ;	....
lr4 ← lr2 ;	pr5 ← pr9
.... ;	....
... ← lr6 ;	... ← pr5 (in RF2)
... ← lr4 ;	... ← pr5 (in RF1)
.... ;	....

(a)

Code with logical registers	Code with renamed registers
lr6 ← ... ;	pr3 ← ...
... ← lr6 ;	... ← pr3
branch to LOOP ;	branch to LOOP
lr5 ← lr4 ;	pr8 ← pr9
lr6 ← ... ;	pr2 ← ...
... ← lr6 ;	... ← pr2
.... ;	....
LOOP: ... ← lr6 ;	... ← pr3

(b)

Figure 4. (a) Example code 1, (b) Example code 2.

Initially, the logical register lr6 is mapped to physical register pr3, and thus is read as a source operand for the next instruction. When the branch is predicted to be not taken and the following instructions are fetched, the logical register lr6 is renamed to a physical register pr2, different from the earlier mapping. The subsequent dependent instructions read from physical register pr2. However, when the branch is realized to be mis-predicted, the instruction in the correct path after the branch

IEEE COMPUTER SOCIETY

requires the logical value from lr6 which actually refers to pr3. Thus when the recovery mechanisms are initiated and the instructions are processed on the correct program path, instruction requiring the value in physical register pr3 will find the register value either in RF1 or RF2, depending on the timing of transfer of the value from RF1 to RF2.

4.3 Handling Precise Exceptions

The Alpha 21264 processor [4] implements a precise exception model using in-order retiring. The programmer does not see the effects of a younger instruction if an older instruction causes an exception. The retire mechanism also tracks the internal register usage for all in-flight instructions. Each entry in the mechanism contains storage indicating the internal register that held the old contents of the destination register for the corresponding instruction. This (stale) register can be freed for other use after the instruction retires. After retiring, the old destination register value cannot possibly be needed. An exception causes all younger instructions in the in-flight window to be squashed. These instructions are removed from all queues in the system. The register map is backed up to the state before the last squashed instruction using the saved map state. The map state for each in-flight instruction is maintained, so it is easily restored. The registers allocated by the squashed instructions become immediately available.

In the dual bank register file organization, the speculative values can exist in any of the two register banks. A register map table is maintained with the status of each of the register in both banks, as against the maintenance of a register map table for a conventional monolithic register file. When an exception occurs, all younger instructions in the in-flight window will be squashed. These instructions are removed from all queues in the system, and the register map state is rolled back to the state before the exception causing instruction. The registers in RF1 allocated by the squashed instructions are immediately available for logical register mapping, and the registers in RF2 holding the values by squashed instructions are made available for the transfer of values from corresponding registers in RF1. If registers in RF1 and RF2 with the same entry are cleared, the register in RF1 is soon allocated with a new mapping at dispatch stage. When the register in RF1 obtains the value, only then is the entry transferred to the corresponding free register in RF2. Until then the register in RF2 remains vacant.

Table 2. Configurations for various register file organizations simulated. Access time is measured at 0.18μ.

Index	*Configuration (IW = Issue Width)* read port (rp) , write port(wp), access latency	$(IW) = 4$ *access time (ns)*	$(IW) = 8$ *access time (ns)*
C1: conventional	RF = 128 registers, rp = 2*IW, wp = IW	1.0614	1.4873
C2: two-level organization	RF1 = 16 registers, rp = 2*IW, wp = IW, 1 cycle RF2 = 128 registers, rp = IW, wp = IW, 2 cycles	0.8046 0.9428	0.9791 1.2302
C3: Dual bank organization	RF1 = 64 registers, rp = 2*IW, wp = IW, 1 cycle RF2 = 64 registers, rp = 2*IW, 1 cycle number of buses from RF1 to RF2 = IW	0.8922 0.8687	1.1552 1.0844
C4: Dual bank organization	RF1 = 128 registers, rp = 2*IW, wp = IW, 2 cycles RF2 = 128 registers, rp = 2*IW, 2 cycles number of buses from RF1 to RF2 = IW	1.0704 0.9428	1.5072 1.2302

5 Performance Evaluation

5.1 Simulation Methodology

We used Simplescalar-3.0 [5] for the Alpha AXP instruction set to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters summarized in Table 1, with a few modifications as below. In Simplescalar the instruction issue queues and the re-order buffer (ROB) constitute one single centralized circular structure called the Register Update Unit (RUU). The simulator has been modified to model the issue queues that are smaller than the ROB size. Besides, an Alpha 21264 processor [4] based architecture is implemented with split integer and

IEEE COMPUTER SOCIETY

floating-point physical register files and issue queues for a 4-wide and an 8-wide out-of-order issue processor. The configurations for four different register file organizations are used for the analysis as shown in Table 2.

The configuration C1 is a base processor implementation. In C1 the monolithic register file is implemented as a single cycle register file with one-level bypass logic. The configuration C2 is implemented in line with the two-level register file design proposed by Cruz *et al* [16]. The configuration C3 is used to evaluate the performance of the dual bank register file organization with RF1 and RF2 constituting 64 physical registers as against a conventional monolithic register file with 128 physical registers. This constitutes an even-handed comparison of the proposed scheme with C1 and C2 in terms of number of physical registers available for data storage. However, note that the physical register bandwidth available for logical register mapping in C3 will be half of that available in case of C1 and C2. Subsequently, to measure the performance of proposed scheme with same logical to physical register mapping bandwidth, we also evaluate the configuration C4. RF1 and RF2 banks in C3 are implemented with a single cycle latency, while RF1 and RF2 in C4 are implemented with a two-cycle latency.

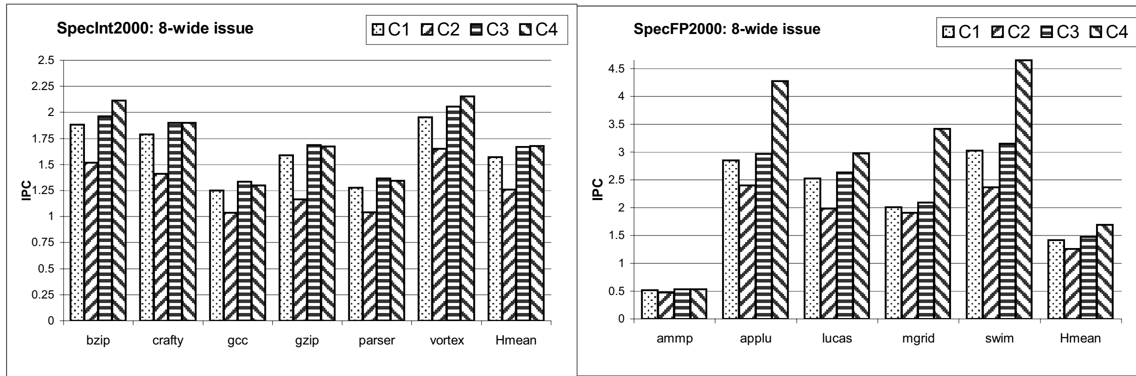

Figure 5. Instructions per cycle (IPC) throughput for various register file configurations in the 8-wide issue processor.

We used SPEC2000 benchmarks, and evaluated both the integer and floating-point programs. We used the access time models of CACTI-2.0 [20] at 0.18μ technology, with necessary modifications to generate cycle times for multiported register files, to evaluate the complexity of proposed register file structures in comparison to the baseline organization. The use of CACTI-2.0 was greatly expanded, since the tool is made to analyze caches with a fewer ports. Here we have made use of it for register files (which typically do not use sense amps like caches) with a larger number of ports. We compute the access time of the RF1 and RF2 register banks while accounting for the multiplexer logic used to select values from either of the banks.

5.2 Results and Analysis

The results obtained with the simulation of various register file configurations in an 8-wide out-of-order issue superscalar processor are shown in Figures 5 and 6. Similar results were also obtained for a 4-wide processor, but are not shown due to space limitations. Figure 5 shows the IPC throughput for various integer and floating-point Spec2000 programs for an 8-wide processor. The degradation in IPC for configuration C2 as compared to configuration C1 is in line with the analysis given by Cruz *et al* [16]. It can be seen that the proposed register file configurations C3 and C4 perform either similar or better as compared to the base configuration. An enhancement in IPC by 4% and 5% for a 4-wide processor, and by 6% and 5% for an 8-wide processor, is seen with configuration C3, for SpecInt and SpecFP programs, respectively. A reduction in register access time by 22% is seen with this implementation.

On the other hand, implementation of configuration C4 enhances IPC by 5% and 12% for a

IEEE COMPUTER SOCIETY

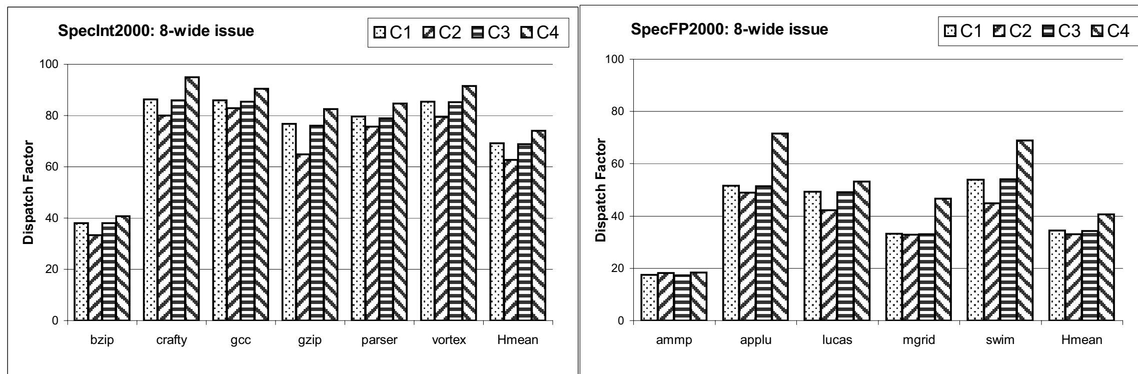

Figure 6. Percentage of run time during which at least one free register exists for various register file configurations in the 8-wide issue processor.

4-wide processor, and by 7% and 15% for an 8-wide processor for SpecInt and SpecFP programs, respectively, while having a similar register access time as the conventional register file. For certain integer benchmarks like crafty, gcc, gzip, and parser, C3 is observed to be performing slightly better than C4. This is due to the larger access cycles for register banks RF1 and RF2 in C4 as compared to those in C3. Hence, this performance difference can vary in either way depending on the architecture implementation technology and the other factors that govern the access time of a register bank. Thus the configuration C3 focuses on reducing the register access time while slightly enhancing the IPC throughput. And, configuration C4 focuses on keeping the register access time at similar level as that of a conventional register file, even while significantly enhancing the IPC throughput by exploiting more instruction level parallelism.

The advantage gained by the inclusion of RF2 register bank, used to retain the register values for a long latency before being freed, is explained as follows. Cruz *et al* [16] have shown that a large level two register bank (with a large access time) and a smaller level one bank results in IPC loss due to increased pipeline latency, though instruction throughput per second (IPS) is gained. We have shown that splitting the large register bank into RF1 and RF2 provides the same memory bandwidth for dispatch stage while reducing pipeline latency improving both IPC and the register access time. This is a significant contribution.

Figure 6 illustrates *dispatch factor*, the percentage of runtime when at least one free physical register is available for logical register mapping. It is noteworthy that for architectures provided with large instruction processing structures (issue queues, ROB, and LSQ), only a very large register file would be able to obtain a dispatch factor of 100%, i.e., a free physical register is available for mapping all through the run time. The implementation of configuration C4, wherein a register bank is viewed by the processor for logical to physical mapping and another register bank is provided to hold the values during the latter part of their lifetime until they are freed, enhances the overall performance of the processor. Also, this design performs better, as each of the two register banks will have a smaller access time as compared to a single large register bank designed to provide the large memory bandwidth.

6 Conclusions

Design of an effective register file architecture is becoming a bottleneck to the enhancement of the performance of a high frequency processors. The two main goals in designing an effective register file organization is to provide a small register access time to enable a faster processor cycle time, and provide a large number of registers to enable dispatching as many instructions as possible to issue window for extracting higher ILP. To meet these two goals simultaneously, we developed a dual bank register file organization, a novel architecture that exploits the quiescent states in the lifetime of a logical to physical register mapping. Implementation of one flavor of the proposed

IEEE COMPUTER SOCIETY

register file organization, as compared to a conventional monolithic register file, in an 8-wide out-of-order issue superscalar processor enhanced instructions per cycle (IPC) throughput up to 6% for Spec2000 applications while reducing register access time up to 22%. Another flavor of the register file organization, with a similar access time as the conventional monolithic register file, enhanced the IPC up to 15%. Thus a trade-off between register access time and ILP exploitation is shown. The proposed register file organization is adaptable to other processors designed to be application-specific, by suitably performing register life time analysis of such processors in similar methodology as this paper. Besides, the significant contribution of our proposed register file architecture is that it enhances IPC, when earlier work in designing effective register file has resulted in IPC degradation.

References

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors", *Proc. 24th Annual International Symposium on Computer Architecture*, 1997, pp. 206-218.

[2] K. I. Farkas, N. P. Jouppi, and P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors" *Proc. Second International Symposium on High-Performance Computer Architecture*, 1996, pp. 40-51.

[3] A. R. Lebeck, J. Koppanalil, Tong Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses", *Proc. 29th Annual International Symposium on Computer Architecture*, 2002, pp. 59-70.

[4] R. E. Kessler, "The Alpha 21264 microprocessor", *IEEE Micro*, Volume: 19, Issue: 2, March-April 1999, pp. 24-36.

[5] Doug Burger and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0", Computer Sciences Department Technical report # 1342, University of Wisconsin-Madison, June 1997.

[6] R. M. Russell, "The Cray-1 Computer System", in *Reading in Computer Architecture*, Morgan Kaufman, 2000, pp. 40-49.

[7] J. Swensen and Y. Patt, "Hierarchical Registers for Scientific Computers", *Proc. International Conference on Supercomputing*, 1988, pp. 346-353.

[8] S. Wallace and N. Bagherzadeh, "A Scalable Register File Architecture for Dynamically Scheduled Processors", *Proc. Conference on Parallel Architectures and Compilation Techniques*, 1996, pp. 179-184

[9] R. Yung and N. C. Wilhelm, "Caching Processor General Registers", *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '95*, 1995, pp. 307-312

[10] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: An Alternative Approach", *Proc. 26th Annual International Symposium on Microarchitecture*, 1993, pp. 202-213.

[11] A. Kumar, "The HP PA-8000 RISC CPU", *IEEE Micro*, Volume: 17, Issue: 2, March-April 1997, pp. 27-32.

[12] M. Tremblay, B. Joy, and K. Shin, "A Three Dimensional Register File for Superscalar Processors", *Proc. 28th Hawaii International Conference on System Sciences*, 1995, Vol. II, pp. 191-201.

[13] V. Zyuban and P. Kogge, "The Energy Complexity of Register Files", *Proc. International Symposium on Low Power Electronics and Design*, 1998, pp. 305-310

[14] A. Gonzalez, M. Valero, J. Gonzalez, and T. Monreal, "Virtual registers", *IEEE Fourth International Conference on High-Performance Computing*, 1997, pp. 364-369

[15] E. Borch, E. Tune, S. Manne, J. Emer, "Loose loops sink chips" *Proc. Eighth International Symposium on High-Performance Computer Architecture*, 2002, pp. 270-281.

[16] J. L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, "Multiple-banked Register File Architectures", *Proc. 27th Annual International Symposium on Computer Architecture*, 2000, pp. 316-325.

[17] J. H. Tseng and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors", *Proc. 30th Annual International Symposium on Computer Architecture*, 2003.

[18] I. Park, M. D. Powell, and T. N. Vijayakumar, "Reducing Register ports for higher speed and lower energy", *Proc. 35th Annual International Symposium on Microarchitecture*, 2002.

[19] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors", *Proc. 34th ACM/IEEE International Symposium on Microarchitecture, MICRO-34*, 2001, pp. 237-248.

[20] S. E. Wilton and N. P. Jouppi, "An Enhanced Access and Cycle Time Model for On-chip Caches", DEC WRL Research 93/5, July 1994.

IEEE COMPUTER SOCIETY