Virtual Registers

Antonio González, Mateo Valero, José González and Teresa Monreal
Universitat Politècnica de Catalunya
Department d'Arquitectura de Computadors
Jordi Girona, 1-3 - Mòdul D6
08034 - Barcelona (SPAIN)
E-mail: antonio@ac.upc.es

Abstract

The number of physical registers is one of the critical issues of current superscalar out-of-order processors. Conventional architectures allocate in the decode stage a new storage location (e.g. physical register) for each operation that has a destination register. When an instruction is committed, it frees the physical register allocated to the previous instruction that had the same destination logical register. Thus, an additional register (i.e. in addition to the number of logical registers) is used for each instruction with a destination register from the time it is decoded until it is committed. In this paper we propose a novel register organization that allocates physical registers when instructions complete execution. In this way, the register pressure is significantly reduced since the additional register is only spent from the time execution completes until the instruction is committed. For some long latency instructions (e.g. load with a cache miss) and for parts of the code with small amount of parallelism, the savings could be very high. We have evaluated the new scheme for a superscalar processor and obtained a significant speedup.

1 Introduction

Out-of-order execution (dynamic scheduling) has been proved to be an effective approach to improve processor performance. A dynamically scheduled processor has the ability to identify dependences among instructions at run-time. This process must be restricted to a small window of the program, which is known as the instruction window, in order to keep the hardware complexity and the associated delay below certain limits. Those instructions that are free of dependences at each time are candidates to be issued.

The performance of out-of-order superscalar processors is mainly determined by a number of factors: instruction fetching mechanism, size of the instruction window, memory organization, and register organization are some of the most critical features.

Registers have a severe impact on the performance since they are used by the majority of instructions to store their source and destination operands. Dynamic register renaming plays a key role in out-of-order processors in order to eliminate name dependences [5] (also known as anti- and output dependences). In such processors, the number of physical registers determines the number of instructions that can be in the instruction window. Enlarging the physical register file is an obvious solution to deal with such limitation. However, the hardware cost of the register file is very high mainly because of the large number of ports that it has. In addition, larger register files have a longer access time, and this may increase the critical path length and penalize performance [1].

In this paper we propose a novel register organization that provides a significant improvement of the performance of current superscalar processors. Alternatively, the proposed organization can be used to reduce the cost of the register file without loosing performance. The key idea of the new organization is to allocate a physical register for the destination of an instruction at the time that its execution completes instead of doing it in the decode stage as it is usual. In this way, the register pressure is significantly reduced, specially in the presence of long latency instructions (e.g. loads with high miss penalty) and long dependence chains that force some instructions to wait long in the instruction window.

The rest of this paper is organized as follows. Section 2 reviews the traditional register management approaches. Section 3 presents the novel register organization that is called virtual registers. The performance of the new scheme is compared with the traditional one in section 4. Finally, the main conclusions of this work are summarized in section 5.

1094-7256/97 $10.00 © 1997 IEEE

2 Register renaming

Dynamic register renaming is a key issue for the performance of out-of-order execution processors and therefore, it has been extensively used since it was first implemented in the IBM 360/61 [12]. In this paper we focus on out-of-order processors that implement precise exceptions [8]. In such processors, instructions are committed in-order. After being decoded, instructions are kept in the instruction reorder buffer until they commit. The size of the reorder buffer determines the degree of reordering that can be achieved at run-time. The size of the reorder buffer is sometimes the instruction window size. In other words, the instruction window is defined as the set of instructions from the oldest not committed instruction to the latest decoded instruction

The objective of register renaming is to eliminate name dependences through registers (anti- and output dependences). This is achieved by allocating a free storage location for the destination register of every new decoded instruction. There are two different schemes based on the approach taken to implement these rename storage locations. In particular, the two following approaches are the most common solutions to provide the rename storage locations:

- The entries of the reorder buffer [9]. In this case, the result of every instruction is kept in the reorder buffer until it is committed. It is then written in the register file. The source operands that are available when an instruction is decoded are read either from the register file or from a reorder buffer entry. Those operands that are not ready at decode are forwarded from the execution units to the corresponding instruction queue entries (reservation stations) when they are produced. When an instruction commits, its result is copied from the reorder buffer to the register file. There is a slight variation that includes a register buffer used just for renaming and avoids to store the result in the reorder buffer(e.g. PowerPC 604 [10]).
- A physical register. In this case there is a physical register file that contains more registers than those defined by the ISA (instruction set architecture), which are referred to as logical registers. By means of a map table, each logical register is mapped to a physical register in the decode stage. The destination register is mapped to a free physical register whereas source registers are translated to the last mapping assigned to them. When an instruction commits, the physical register allocated by the previous instruction with the same logical destination register is freed. In this scheme, the operands are always read from the physical register file, which simplifies the operand fetch task when compared with the previous model.

Both register renaming schemes are being used in the latest microprocessors. The first one is used by the Intel Pentium Pro [2], the PowerPC 604 [10], and the HAL SPARC64 [3]. The MIPS R1000 [13], and the DEC 21264 [4] are current implementations of the second approach. In this paper, we focus on the second scheme. A comparison of both approaches in terms of cost-effectiveness could be an interesting study but it is beyond the scope of this paper.

A number of physical registers significantly higher than the number of logical registers is required for performance. This is so because when the instruction window is empty (e.g., after a branch misprediction), each logical register is mapped to a physical register. Thus, the minimum number of physical registers that are used is equal to the number of logical registers. In addition, for every instruction whose destination operand is a register, an additional register is allocated when it enters the window and a physical register is released when it leaves the window.

3 Virtual registers

This section presents the novel register renaming scheme proposed in this paper. This new organization is proposed for both integer and FP register files. Since it is implemented in the same way for both files, only one file is considered in the explanation of this section.

The motivation for the register organization that is proposed here comes from the observation that the conventional register renaming scheme based on a physical register file allocates a new physical register for every instruction with a destination register. This register is allocated from the time when the instruction is decoded until the next instruction that has the same logical destination register is committed.

However, the register does not hold any value until the instruction execution finishes (i.e., when it is written in the write back stage). For sections of code with little parallelism and/or long latency operations, the time from decode to write back can be very large for some instructions, especially when the instruction window is large.

For instance, long latency operations may be the case of load instructions that miss in cache when the cache miss penalty is high. Due to the increasing gap between processor and memory speed, this factor can have an increasing impact in the future. A large instruction window is desirable in order to increase the instruction level parallelism that can be exploited at run-time.

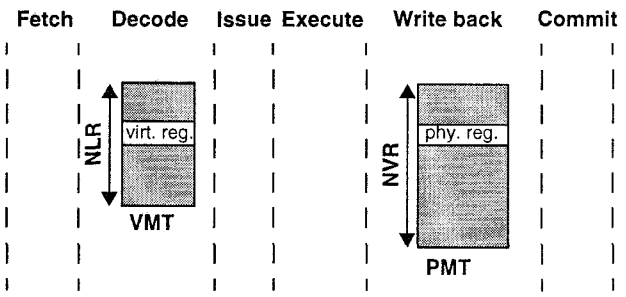

Figure 1: Virtual register organization. Each table is depicted in the stage in which it is updated

Notice that the reason why logical registers are mapped to physical registers at decode stage is mainly to keep track of dependences among instructions. In fact, what is just required to keep track of dependences is a tag that identifies the last producer for every logical register. These tags are used to determine from where the source operands are to be read.

The register organization that we propose allocates a new physical register for every instruction when the instruction execution completes (in the write back stage). Therefore, when compared with the conventional scheme, this new organization saves one physical register for every instruction in the instruction window that has not completed. As a result, the register pressure is significantly reduced. This benefit can be used in two ways: a) design processors with a smaller register file and reduce the hardware complexity, without loosing performance; or b) increase the performance of the processor by building a larger instruction window without increasing the register file.

The new organization (see figure 1), which is called *virtual registers*, is based on adding a new type of registers that are called virtual registers. The registers referenced by the instructions of the ISA are referred to as *logical registers*. When an instruction is decoded, its destination register is mapped to a free virtual register. These registers are not related to any physical storage location and therefore they are merely tags. This mapping is done by means of a table, which is called the *virtual map table (VMT)*, that is indexed by the logical register number and contains its last virtual register mapping. Since virtual registers are not related to any particular storage, the processor may use as many of them as it wants. In practice, the decode stage will stall when the instruction window is full, so the number of virtual register (NVR) to guarantee that the processor never stalls because of the lack of them is the number of logical registers (NLR) plus the instruction window size (WS). The size of each entry of the VMT is then $\lceil \log_2(NLR+WS) \rceil$ bits. There is a free pool of virtual registers that identifies those that can be allocated by a new mapping at each moment. In the decode stage, the source operands of every instruction are renamed using the VMT. The instruction is then dispatched to the instruction window with all its source and destination operands referring to virtual registers.

The wakeup and selection logic is basically the same as that of a conventional register organization. Every time a result is produced, the virtual register associated to this result is broadcast to all the instructions in the instruction queue. If an instruction is using such virtual register as a source operand, then it is marked as available. An instruction can be chosen for issuing when all its operands are available.

When an instruction is issued, it always reads its register operands from the physical register file. This is done by means of a second table, which is called the *physical map table (PMT)*. This table has as many entries as the number of virtual registers and it is indexed by the virtual register number. Each entry identifies the last physical register to which that virtual register has been mapped. The size of each entry is then $\lceil \log_2(NPR) \rceil$, where NPR denotes the number of physical registers. Using this table, the source virtual registers are translated to physical registers and the operands are then obtained from the physical register file. Alternatively, this map table could be implemented by means of a CAM (content-addressable memory) with a number of entries equal to number of physical registers, which is much lower than the number of virtual registers. This approach is for instance used by the DEC 21264 [4] to implement the logical to physical map table.

Every instruction whose destination is a register allocates a new physical register when its execution completes. At this time, a new physical register is taken from a free pool of physical registers (the solution to the lack of free physical registers is considered in the next section; for the sake of simplicity we assume now that this event never happens). The destination virtual register is mapped to this free physical register and this mapping is reflected in the PMT.

When an instruction commits, the previous virtual register allocated to the logical destination register is returned to the free pool. The identifier of this virtual register is obtained from the VMT in the decode stage, before being updated with the new mapping. This information is kept in the reorder buffer. In addition, the physical register allocated to that virtual register is also released. This information is obtained from the PMT when the instruction commits and therefore it does not need to be kept in the reorder buffer.

In case of an exception or a branch misspeculation, a precise state can be obtained by undoing the mappings

performed by instructions that follow the offending one. This can be done by popping out the entries of the reorder buffer from the newest until the offending one. For each instruction, the reorder buffer stores the destination logical register and the virtual register that was allocated to the previous instruction with the same destination logical register. Using the logical register identifier, the VMT is read to obtain the current virtual register mapping and then the entry is restored with the previous virtual mapping. Using the current virtual mapping, the PMT is read and the physical register allocated to that virtual register, if any, is freed. Notice that a mechanism based on checkpointing similar to the one used by the R10000 [13] could be used to recover from branches in just one cycle.

3.1 Avoiding deadlock

A virtual register organization may be designed with any number of logical, virtual and physical registers. The number of logical registers is a feature of the ISA and therefore remains fixed for different implementations of the same ISA. The amount of virtual registers has a small impact on the hardware cost: it determines the number of entries of the PMT ($\lceil \log_2(NPR) \rceil$-bit wide) if it is not implemented through a CAM, and it has a logarithmic impact on the width of the VMT. On the other hand, the number of physical registers has a very high impact on the hardware cost as discussed in the introduction. In consequence, the number of physical registers will be lower than that of virtual registers.

In this case, it may happen that when a instruction completes there is no a free physical register. The obvious approach to deal with this situation is to squash such instruction. However, in this situation, the eldest instruction in the window will not be able to commit because when its execution completes it will also find that there is not any free physical register. Under this circumstances, no instruction will be allow to commit and therefore no physical register will be freed, which will result in a deadlock.

This deadlock can be avoided by a slight modification in the register management policy. In particular, it is enough to reserve a given number of registers for the oldest instructions that require them. Such number is referred to as the *number of reserved registers (NRR)*. When an instruction completes, it allocates a new physical register as previously described, provided that there are more free physical registers than those reserved for the oldest NRR instructions that require them or it is one of the NRR oldest instructions in the window with a destination register. Otherwise, the instruction is squashed and kept in the instruction window and re-issued later on.

This is implemented by means of a register that points to the youngest entry in the reorder buffer with a register reserved (PRR), a register that indicates the number of instructions below (older than or equal to) PRR that have a destination register (Reg), and how many of such instructions have already allocated a physical register (Used).

When an instructions with a destination register commits, PRR is moved to point from the current location to the next instruction with a destination register or until the tail of the reorder buffer. If the new instruction pointed by PRR has not yet allocated a physical register, then Used is decreased; otherwise it is left unchanged. If the new instruction pointed by PRR has not a destination register (PRR points to the tail of the reorder buffer), then Reg is decreased; otherwise it is unchanged.

When an instruction with a destination register is decoded, if Reg is lower than NRR, then Reg is increased and PRR is updated to point to the location of such instruction in the reorder buffer.

Notice that this parameter may have a strong influence on the performance. A higher value of the NRR means that there are less physical registers to be used without the constraint of being one of the oldest instructions. However, a higher value of NRR also implies that the processor speed when there are only NRR free registers or less is higher.

4 Performance evaluation

This section presents a performance evaluation of the virtual register organization. The objective of this section is to demonstrate the benefits of the novel register organization. An exhaustive evaluation of the impact on the cost and performance of the different design parameters for different processor architectures is left for future work.

The evaluation of the new scheme is performed by comparing the execution time of an aggressive superscalar processor with a conventional register organization with that of the same processor with the virtual register organization. In both cases it is assumed the same amount of physical registers.

4.1 Experimental framework

A trace-driven simulator of a out-of-order superscalar processor has been developed to evaluate the proposed register organization. Two different register organizations have been simulated. The first one is the conventional register scheme used by the R10000 [13] among others. The second one is the virtual register organization proposed in this paper. In both cases, a typical superscalar architecture has been assumed. The execution of an instruction consists of the following stages:

- Fetch: Every cycle, up to eight instructions can be read from the Instruction cache. A perfect instruction cache is assumed. In this stage, branch prediction is performed using a 2048 entry Branch History Table with a 2 bit up-down saturated counter per entry.
- Decode: Up to eight instructions are decoded every cycle. Their logical registers are renamed and mapped onto physical registers for the conventional organization and they are mapped to virtual registers for the virtual register organization. Then, the instructions are dispatched to a global Instruction Queue and to a global Instruction Reorder Buffer (IRB). The size of both, which corresponds to the size of the instruction window, is 128 entries. Instructions are kept in the Instruction Queue until they are issued and they remain in the IRB until they are committed. If the IRB is full or there are not free physical/virtual registers, then the fetch and decode stages are stalled. There is one physical register file for integer data and another for FP data. Both have 64 registers with 16 read ports and 8 write ports. The number of virtual registers is 160 for integer and 160 for FP.
- Issue: Every cycle, the issue logic searches the Instruction Queue for instructions that are free of dependences. If there are more ready instructions than available resources, the oldest instructions are selected. Register operands are obtained at this moment from physical register file. Table 1 shows the number of functional units and their latency.

Functional Unit	Count	Latency	Repeat rate
Simple Integer	3	1	1
Complex Integer	2	9 multiply 67 divide	1 67
Effective Address	3	1	1
Simple FP	3	4	1
FP Multiplication	2	4	1
FP Divide and SQR	2	16 divide 35 SQR	16 35

Table 1: Functional units and instruction latency.

- Execute: Instructions are executed in the corresponding functional units. Loads and stores execute in the effective address computation units and then they are entered into the load/store queue. Instructions wait in the load/store queue until they commit. Load instructions access the cache when a cache port is available. Three memory ports and the memory disambiguation scheme implemented in the PA-8000 [6] have been assumed in this experiment.
- Write-back: Instructions are completed and the results are written into the register file. For the virtual register organization, the virtual to physical register mapping is performed at the beginning of this stage.
- Commit (or retire, or graduate): Instructions are committed in-order so that a precise state [8] can be recovered at any time. Store instructions send their request to memory in this stage.

The processor has a lookup-free data cache [7] that allows up to 8 different cache lines with outstanding misses. The cache size is 16 KB, and it is direct-mapped with 32-byte line size. Cache hit latency is 2 cycles and the penalty for a cache miss is 50 cycles. An infinite L2 cache is assumed and a 64-bit data bus between L1 and L2 is considered (i.e., a line transaction occupies the bus during four cycles).

Our experimentation methodology is trace-driven simulation. The object code, previously compiled with full optimization for a DEC AlphaStation 600 5/266 with and Alpha AXP 21164 processor, is instrumented using the Atom tool [11]. The instrumented program is executed and the trace generated feeds the processor simulator. A cycle-by-cycle simulation is performed in order to obtain accurate timing results. Because of the detail at which simulation is carried out the simulator is slow, so we have simulated 100 million of instructions for each benchmark after skipping the first 2 billion of instructions. Five floating-point (apsi, swim, mgrid, hydro2d, wave5) and four integer (go, li, compress, vortex) SPEC95 benchmarks have been selected for this study. Each program was run with the largest input set available for that benchmark.

4.2 Results

Figure 3 shows the speedup gained by the new register renaming scheme in front of the traditional one. The speedup is measured as the execution time of the traditional scheme divided by the execution time of the new organization.

It can be seen that in general the highest improvement is achieved by the highest value of NRR (i.e., 32). The speedup decreases significantly when NRR decreases for FP codes, except for hydro2d, whereas for integer codes, the difference is not so high. In fact, for two out of four integer codes, the optimal NRR is 4. Intermediate values of NRR like 16 and 24 also outperform the traditional scheme for all the benchmarks except for compress.

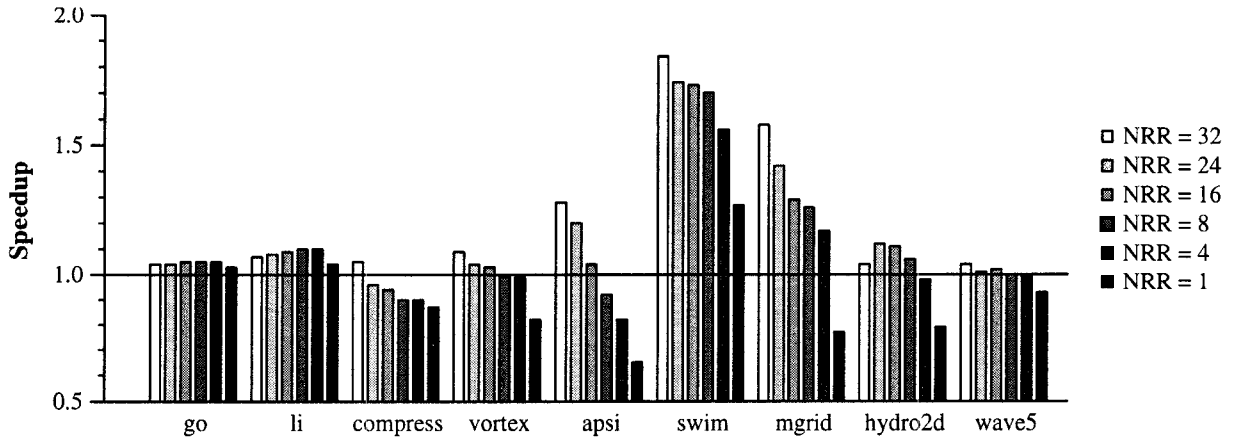

Figure 3: Speedup of the virtual register organization when compared with the conventional scheme for a varying value of the NRR.

With NRR equal to 32, the novel register renaming scheme provides an average speedup of 1.21 for the analyzed programs and it goes up to 1.84 for swim.

5 Conclusions

We have presented a novel register organization for out-of-order execution processors. The key idea behind the new organization is to delay the allocation of physical registers until instructions complete, instead of doing it in the decode stage, in order to reduce the register pressure.

The new scheme is based on introducing a new concept that is called virtual registers. Virtual registers are not related with any storage location but they are merely tags that are used to keep track of register dependencies.

A preliminary evaluation of the proposed approach has shown speedups of 21% in average when compared with a conventional register organization with approximately the same hardware cost. The improvement for FP codes (36%) is much higher than for integer codes (6%).

Acknowledgments

We would like to thank Jim Smith for his comments and suggestions on early versions of this paper.

This work has been supported by the Spanish Ministry of Education under grant CICYT TIC 429/95 and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1996FI-03039- APDT.

The research described in this paper has been developed using the resources of the CEPBA.

References

[1] K.I. Farkas, N.P. Jouppi and P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors", in *Proc. of the Int. Symp. on High Performance Computer Architecture*, pp. 40-51, 1996

[2] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design", *Microprocessor Report*, 9(2), Feb. 1995

[3] L. Gwennap, "HAL Reveals Multichip SPARC Processor", *Microprocessor Report*, 9(3), March 1995

[4] L. Gwennap, "Digital 21264 Sets New Standard", *Microprocessor Report*, 10(14), Oct. 1996

[5] J.L Hennessy and D.A. Patterson, *Computer Architecture. A Quantitative* Approach. Second Edition. Morgan Kaufmann Publishers, San Francisco 1996.

[6] D. Hunt, "Advanced Performance Features of the 64-bit PA-8000", in *Proc. of the CompCon'95*, pp. 123-128, 1995.

[7] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization", in *Proc. 8th International Symposium on Computer Architecture* (1981) pp. 81-87

[8] J.E. Smith and A.R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", *IEEE Tranactions on Computers*, 37(5), pp. 562-573, May 1988.

[9] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", *IEEE Transactions on Computers*, 39(3), pp. 349-359, March 1990

[10] S.P Song, M. Denman and J. Chang, "The PowerPC 604 Microprocessor", *IEEE Micro*, 14(5), pp. 8-17, Oct. 1994

[11] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools", in *Proc of the 1994 Conf. on Programming Languages Design and Implementation*, 1994.

[12] R.M. Tomasulo, "An Effient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development*, 11(1), pp. 25-33, Jan. 1967.

[13] K.C. Yeager, "The MIPS R10000 Superscalar Microprocessor", *IEEE Micro*, 16(2), pp. 28-40, April 1996