A Simple Low-Energy Instruction Wakeup Mechanism

Marco A. Ramírez[1, 4], Adrian Cristal[1], Alexander V. Veidenbaum [2],

Luis Villa [3], Mateo Valero [1]

[1] Computer Architecture Department U.P.C. Spain, [2] University of California Irvine CA,
[3] Mexican Petroleum Institute, Mexico. [4] National Polytechnic Institute, México.

e-mails: {mramirez, adrian, mateo}@ac.upc.es, alexv@ics.uci.edu, lvilla@imp.mx
Phone: +34-93-401-69-79, Fax: +34-93-401-70-55

Abstract. Instruction issue consumes a large amount of energy in out of order processors, largely in the wakeup logic. Proposed solutions to the problem require prediction or additional hardware complexity to reduce energy consumption and, in some cases, may have a negative impact on processor performance. This paper proposes a mechanism for instruction wakeup, which uses a multi-block instruction queue design. The blocks are turned off until the mechanism determines which blocks to access on wakeup using a simple successor tracking mechanism. The proposed approach is shown to require as little as 1.5 comparisons per committed instruction for SPEC2000 benchmarks.

Keywords: Superscalar processors, Out of order execution, Instruction window, Instruction wake up, Low power, CAM.

1. Introduction

Modern high-performance processors issue instructions in order but allow them to be executed out of order. The out-of-order execution is driven by the availability of operands; i.e. an instruction waits until its operands are ready and then is scheduled for execution. An instruction queue (IQ) is a CPU unit used to store waiting instructions after they are issued and until all their operands are ready. Associated with the instruction queue are wakeup logic and selection logic [1].

The wakeup logic is responsible for waking up instructions waiting in the instruction queue as their source operands become available. When an instruction completes execution, a register number (tag) of its result register is broadcast to all instructions waiting in the instruction queue. Each waiting instruction compares the result register tag with its own source operand register tags. Three-address instructions are assumed in this paper and an instruction may have 0, 1 or 2 source operands. On a match, a source operand is marked as available. Thus, the instruction is marked ready to execute when all its operands are ready.

Selection logic is responsible for selecting instructions for execution from the pool of ready instructions. A typical policy used by the selection logic is oldest ready first.
A commonly used implementation of the instruction queue is shown in Fig. 1 and uses RAM-CAM arrays to store the necessary information [2]. The RAM section (solid area in the figure) stores the instruction opcode, the destination (result) register tag, and the busy bit indicating the entry is used. The CAM section stores the two source operand register tags and the corresponding ready bits.

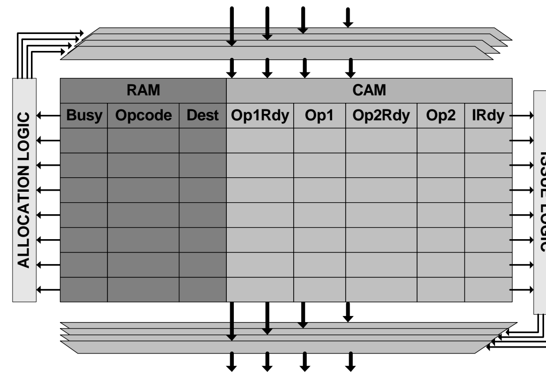

Figure 1: An Instruction Queue Architecture using RAM-CAM array (4-wide issue)

One or more of its operands may be ready as the instruction is initially placed in the instruction queue. In this case they are immediately marked ready by setting the corresponding flag(s). If one or more of its operands are not ready, the instruction waits in the queue until it is generated by the producing instruction. As instructions complete execution and broadcast their destination register tag, each CAM entry performs a comparison of that tag with operand1 tag and/or operand2 tag. On a match, operand1 ready flag (Op1Rdy) and/or operand2 ready flag (OpRdy2) are set. When both Op1Rdy and Op2Rdy are set, the Instruction Ready (IRdy) flag is set.

Figure 2 shows a more detailed view of the CAM section of the queue assuming an issue width of four instructions. Eight comparators are required per each IQ entry. All comparisons are ideally completed in one clock cycle to allow up to four instructions to wake up per cycle. It is clear from the figure that the wakeup logic is both time and energy consuming. This paper focuses on reducing the latter.

The energy consumption of the wakeup logic is a function of the queue size, issue width, and the design used. Overall, the energy used in the IQ is dominated by the total number of comparisons performed in the CAM portion of the IQ. Consider the CAM cell organization shown in Fig. 2b. The precharge signal enables the comparison in a given entry. Most of the energy dissipated during a comparison is due to the precharge and the discharge of the match line ML. Almost no energy is dissipated when the precharge of an entry is disabled.

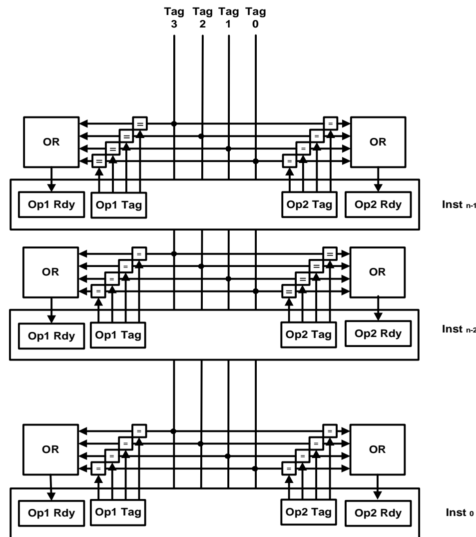

Figure 2a: Organization of the CAM Section of the Instruction Queue

The actual energy dissipated in a CAM cell depends on the cell design and the process parameters. It is, however, proportional to the number of comparisons performed. Therefore, the rest of this paper uses the number of comparisons to evaluate and compare different wakeup mechanisms. In general, a larger instruction queue is highly desirable in a dynamically scheduled processor as it can expose additional instruction level parallelism. This increases the energy consumption in the IQ even further. Several solutions have been proposed to address the IQ energy consumption.

This paper proposes a new solution at the microarchitecture level, which is simple, requires minimal CPU modification, and yet results in a very large reduction in the number of comparisons required and therefore the energy consumption of the instruction queue.

This solution can be used in conjunction with some of the previously proposed techniques to achieve even better results.

One of the commonly used IQ designs partitions the instruction queue into integer, floating point, and load/store queues. This reduces the destination tag fanout and the number of comparisons performed. The design proposed in this paper further divides each queue into a number of separate blocks. All queue blocks are normally disabled and perform no comparisons unless explicitly enabled (the precharge is turned on). A method for tracking which block(s) should be enabled upon an instruction completion is proposed. In addition, only valid entries are compared within each block as proposed in [3]. The proposed design is shown to lead to a large reduction in the total number of comparisons. The new design does not affect the IPC.

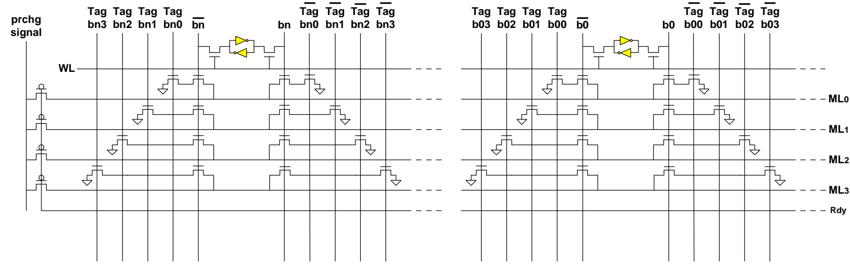

Figure 2b: A transistor-level design of an “n+1”-bit Entry CAM.

The rest of the paper is organized as following. Section 2 describes related research, Section 3 presents the proposed wakeup mechanism, and Section 4 shows the results obtained.

2. Related Work

The energy consumption of a modern dynamically scheduled superscalar processor is between 50 and 100 Watts. At the micro-architecture level, the issue logic is one of the main consumers of energy responsible for approximately 25% of the total energy consumption [3]. Many approaches to designing the wakeup logic have been proposed, both to reduce delay and to reduce energy consumption. [4] proposed a pointer-based solution, where each instruction has a pointer to its dependent instruction for direct wakeup. This was done for a singe-issue, non-speculative processor, however.

[5] extended the above solution to modern processors with wide issue and speculation. The effect of one, two, or three successor pointer entries per instruction was evaluated. Three approaches to deal with the case of an instruction with more successors than pointer entries were proposed. The first one stalled the instruction issue. The second one stopped recording successors and instead woke the instruction up when it reached the top of the instruction window. Both of these approaches lead to a loss of IPC. The third approach added a scoreboard to avoid stalls, an expensive approach to say the least. Finally, successor pointers were saved on speculated branches, which is quite expensive. Overall, the solution does not require the use of CAM and thus significantly reduces both the delay and the energy consumption of the wakeup logic.

[6] proposed a circuit design to adaptively resize an instruction queue partitioned into fixed size blocks (32 entries and 4 blocks were studied). The resizing was based on IPC monitoring. The use of self-timed circuits allowed delay reduction for smaller queue size. [7] further improved this design using voltage scaling. The supply voltage was scaled down when only a single queue block was enabled.

[8] used a segmented bit line in the IQ RAM/CAM design. Only selected segments are used in access and comparison. In addition, a form of bit compression was used and a special comparator design to reduce energy consumption on partial matches.

[3] proposed a design, which divided the IQ into blocks (16 blocks of 8 entries). Blocks which did not contribute to the IPC were dynamically disabled using a monitoring mechanism based on the IPC contribution of the last active bank in the queue. In addition, their design dynamically disabled the wake up function for empty entries and ready operands[1].

[9] split the IQ into 0-, 1-, and 2-tag queues based on operand availability at the time an instruction enters the queue. This was combined with a predictor for the 2-operand queue that predicted which of the two operands would arrive last. The wakeup logic only examined the operand predicted to arrive last. This approach reduces the IPC while saving energy. First, an appropriate queue with 0-, 1-, or 2-tags must be available at issue, otherwise a stall occurs. Second, the last-operand prediction may be incorrect, requiring a flush.

[10] also used a single dependent pointer in their design. However, a tag comparison is still always performed requiring a full CAM. In the case of multiple dependent instructions a complex mechanism using *Broadcast* and *Snoop* bits reduces the total number of comparisons. The *Broadcast* bit indicates a producer instruction with multiple dependents (set on the second dependent). Each such dependent is marked with a *Snoop* bit. Only instructions with a *Snoop* bit on perform a comparison when an instruction with a *Broadcast* bit on completes. Pointer(s) to squashed dependent instructions may be left dangling on branch misprediction and cause unnecessary comparisons, but tag compare guarantees correctness.

[11] used a full bit matrix to indicate all successors of each instruction in the instruction queue. Optimizations to reduce the size and latency of the dependence matrix were considered. This solution does not require the use of CAM but does not scale well with the number of physical registers and the IQ size which keep increasing. [12] also described a design of the Alpha processor using a full register scoreboard.

[13] proposed a scalable IQ design, which divides the queue into a hierarchy of small, and thus fast, segments to reduce wakeup latency. The impact on energy consumption was not evaluated and is hard to estimate.

In addition, dynamic data compression techniques ([14], [15]) have been proposed as a way to reduce energy consumption in processor units. They are orthogonal to the design proposed here.

3. A New Instruction Wakeup Mechanism

The main objective of this work is to reduce the energy consumption in the wakeup logic by eliminating more unnecessary comparisons. The new mechanism is described in this section and is presented for a single instruction completing execution. The wakeup logic for each source operand in the IQ entry is replicated for each of the multiple instructions completing in the same cycle.

The approach proposed here takes advantage of the fact that in a distributed instruction queue with *n* blocks each source operand only requires one of the *n* blocks to be involved in instruction wake-up. In addition, a completing instruction typically has few dependent instructions waiting in the IQ for its operand [16]. For example, if an instruction has two successors then at most two out of the *n* IQ blocks need to be used for comparison. The remaining *n-2* blocks can be "disabled" and do not need to be involved in the wakeup process. This can be accomplished by gating their precharge signal. Clearly, the benefits of this method will grow with an increase in *n*, but at some point the complexity of the design will become to high. Four or eight blocks per instruction queue are a practical choice of *n* used in this paper.

Consider an instruction queue partitioned into *n* blocks such that each block B*i* can be enabled for tag comparison separately from all the others. All blocks are normally disabled and require an explicit enable to perform the tag search. A block mapping table, a key element of the proposed design, selects which block(s) hold the successor instructions and only these blocks are enabled to compare with the destination register tag of a completing instruction.

The block mapping table (BT) is a bookkeeping mechanism that records which of the *n* blocks contain successor instructions for a given destination register. It is organized as an *n* by *M* bit RAM, where *n* is the number of banks and *M* is the number of physical registers. It is shown in the upper left corner of Fig. 3. The table records for each destination register which IQ blocks contain its successor instructions. A BT row thus corresponds to a producing instruction and is checked when the instruction completes. It is read using the destination register tag of the completing instruction to find all the banks that contain its successor instructions. Such a row will be called a block enable vector (BE) for the register. An individual bit in BE will be called a block entry. For example, Fig. 3 shows BT[4], the BE for the physical register 4, with only entries for blocks 0 and 2 are set to "1". The 1's indicate that successor instructions using register 4 are in blocks 0 and 2.

The operation of the proposed design is as follows. When an instruction is decoded it is entered into the instruction queue with the source operand designators specifying the physical register sources. An IQ block to enter the instruction is selected at this time. Several block assignment algorithms are possible; the results presented in this paper are based on the round-robin assignment algorithm. The destination and source physical register tags are then entered in the IQ as per Figure 1. At this point two operations on the BT take place concurrently.

BE Allocation. The enable vector for the instruction's destination register Ri, BT[i], is cleared. It is guaranteed that by this time all dependent instructions have obtained a previous copy of this register.

BE Entry Modification. A BE entry for each source register of the instruction that is not ready is set. This will indicate for each register what blocks contain its dependent instructions. The source operand register tag is used to select a BT row to modify. The modification consists of setting the block entry in the BE to a 1. For example, an instruction allocated to IQ block "j" that uses a source operand register "i" sets entry "j" in BT[i] to "1". Note that multiple instructions in a block can all set the entry.

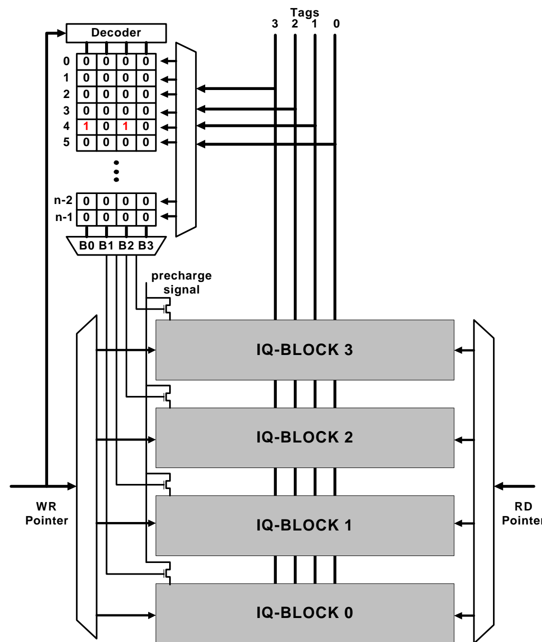

Figure 3: The Block Mapping Table and a 4-Block Instruction Queue

The BE is used when an instruction completes execution and produces a result. Its destination register is used to index the Block Table and to read out a corresponding block enable vector. Only the wakeup logic in blocks enabled by a "1" in the BE is going to perform comparisons. Furthermore, within an enabled block any of the previously proposed designs for wakeup logic in a centralized queue can be used to further reduce the number of comparisons. Only active entries are compared in each block in this paper.

The addition of the BT access in the path of the CAM precharge signal may lead to an increased delay. Pipelining can be used if a design overlapping the BT access with other parts of the CAM operation is impossible. The result tag can be broadcast to the

BT one cycle earlier and the BT access completed and latched before the CAM access. This is possible in current and future processors with deep pipelines that take several cycles to select/schedule an instruction after wakeup.

A final issue to consider is the impact of branch prediction. All instructions that were executed speculatively are deleted from the IQ on branch misprediction. Their BEs remain in the table, possibly with some entries set. However, this is not a problem since the deleted instructions will not complete and their destination register will eventually be re-allocated and cleared. BEs for instructions prior to the branch may, however, contain entries corresponding to mispredicted and deleted instructions.

Element	Configuration
Reorder Buffer	256 entries
Load/Store Queue	64 entries
Integer Queue	32-64 entries
Floating Point queue	32-64 entries
Fetch/decode/commit width	4/4/4
Functional units	4 integer/address ALU, 2 integer mult/div, 4 fp adders, 2 mult/div and 2 memory ports.
Branch predictor	16 K-entry GShare
Branch penalty	8 cycles
L1 Data cache	32 KB, 4 way, 32 byte/line, 1 cycles
L1 Instruction cache	32 KB, 4 way, 32 byte/line, 1 cycles
L2 Unified cache	512 KB, 4 way, 64 byte/line, 10 cycles
TLB	64 entries, 4 way, 8KB page, 30 cycles
Memory	100 cycles
Integer Register file	128 Physical Registers
FP Register file	128 Physical Registers

Table 1: Processor Configuration

These entries may cause unnecessary activation of IQ banks for comparisons. In the worst case, a deleted instruction may be the only dependent instruction in a block and cause its activation. This does not affect correctness or performance of a program, but may lead to unnecessary energy consumption. The impact of this, in our experience, is negligible.

4. Results and Analysis

The proposed approach was simulated using an Alpha 21264-like microarchitecture. The SimpleScalar simulator was re-written to accomplish this. The processor configuration is shown in Table 1. The reorder buffer with 256 entries was used to avoid bottlenecks.

As in the R10K design, separate integer and floating-point instruction queues were used. In addition, Ld/St instructions were dealt with in the Ld/St queue as far as wakeup was concerned. Only one source operand was assumed necessary for

memory access instructions, namely the address. All entries in the LSQ are compared to the integer result register tag. Data for stores is dealt with separately during commit when stores access memory. The integer and f.p. Instruction Queues were split into either 4 or 8 blocks.

SPEC2000 benchmarks were compiled for the DEC Alpha 21264 and used in simulation. A dynamic sequence of instructions representing its behavior was chosen for each benchmark and 200M committed instructions were simulated, with statistics gathered after a 100M-instruction "warm-up" period.

A queue design that disables comparison on ready operands and empty IQ entries (collectively called active entries), similar to [3] but without dynamic queue resizing is referred to as Model A. The impact of the proposed queue design, Model C, is evaluated using the number of wakeup comparisons per committed instruction and is compared with model A. Model C also uses only active entries in comparisons. The proposed design is utilized only in the integer and f.p. queues, the Ld/St queue are identical in both models. The number of comparisons performed in the Ld/St queue is thus the same in both models and is not reported.

Comparisons are divided into those that result in a match on each wakeup attempt (necessary) and all the rest (unnecessary) in the analysis. The former are labeled "Match" in the graphs and the latter labeled "No-Match".

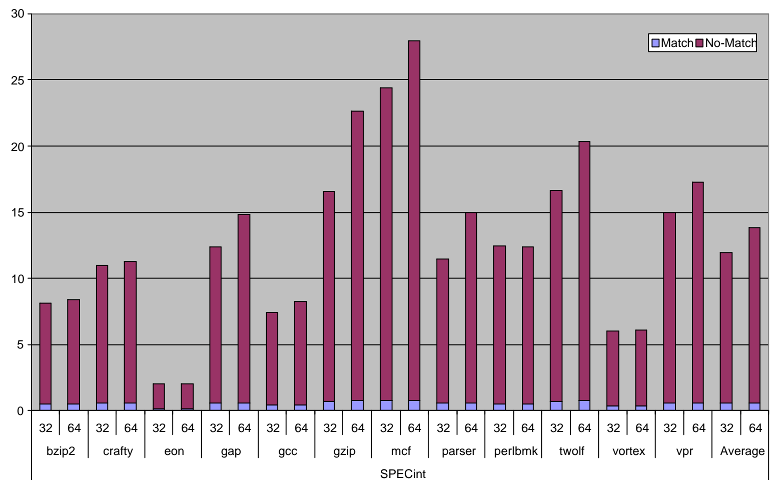

Figure 4: Comparisons per Instruction for Model A and Integer Benchmarks

Figs. 4 and 5 show comparisons per committed instruction for integer and f.p. benchmarks, respectively, and the Model A queue design. They show that most of comparisons are unnecessary and that, on average, the integer and f.p. queues have similar behavior. The number of comparisons per committed instruction is 13 and 15,

on average, for integer queue size of 32 and 64, respectively. The averages are 12 and 17 for the same f.p. queue sizes.

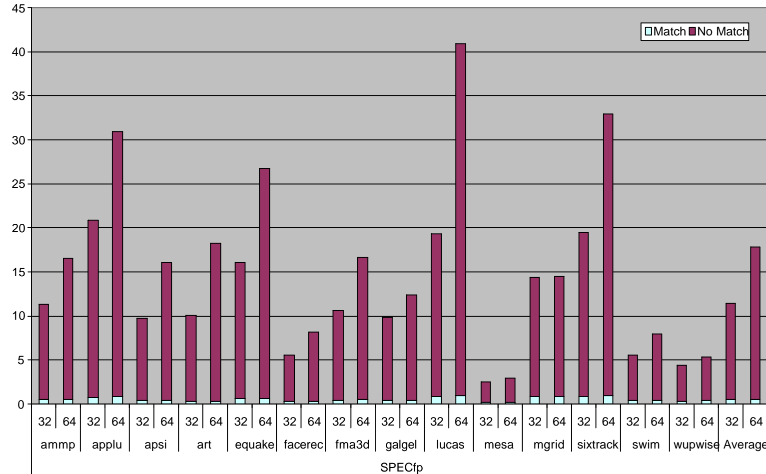

Figure 5: Comparisons per Instruction for Model A and F.P. Benchmarks

Figs. 6 and 7 compare the two different queue designs and the effect of the queue size and of the number of blocks. The results are averaged over all integer and all f.p. benchmarks, respectively, and show the total number as well as the fraction of necessary comparisons per committed instruction. The results for Model C are always significantly better than Model A results for both 4- and 8–block queue organization. They are even better for a 1-block Model C design, which is a single queue, because some instructions do not use a destination register and the use of the BT allows wakeup to be completely avoided in this case. In general, results improve with the increase in the number of blocks.

The 4-block IQ design achieves a 78% improvement, on average, over model A for the integer queue and 73% for the f.p. queue. The relative improvement is almost identical for both queue sizes. In absolute terms, the number of comparisons performed is higher for the larger queue size.

The 8-block design results in fewest comparisons. It reduces the number of comparisons in Model A by 87% for both the 32- and the 64-entry IQ for integer benchmarks. For f.p. benchmarks the reduction is 85%. More than one third of the total comparisons are necessary indicating the potential for further improvement.

The total number of comparisons per committed instruction is 1.45 and 1.72, on average, for integer queue sizes of 32 and 64, respectively, and eight blocks. The averages are 1.59 and 2.59 for f.p. benchmarks for the same queue sizes.

Overall, a larger instruction queue has more valid IQ entries, which explains the increase in the number of comparisons per instruction. For example, the average number of IQ entries per cycle is ~41% higher in the 64-entry/8-block queue for f.p. benchmarks as compared to the 32-entry queue. The difference is smaller for integer benchmarks, approximately 20%.

The associated IPC increase is small, only about 13% for f.p. codes and near 0 for integer codes (using a harmonic mean instead of an average).

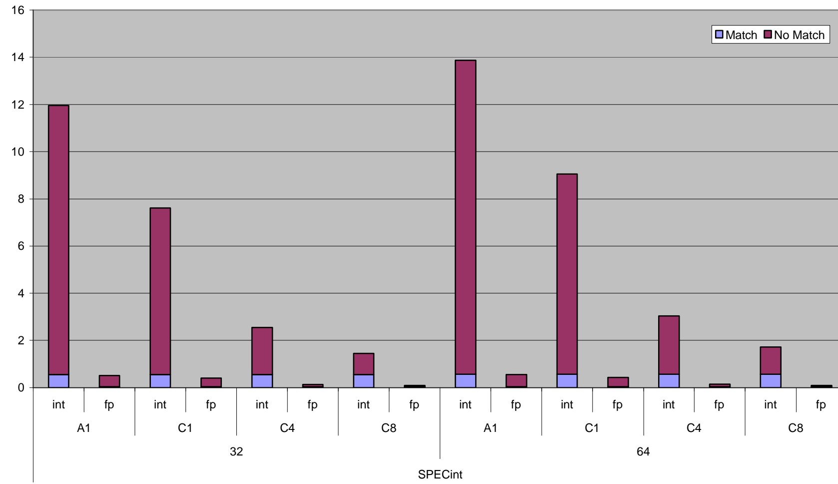

Figure 6: Average Number of Comparisons per Instruction for Integer Codes

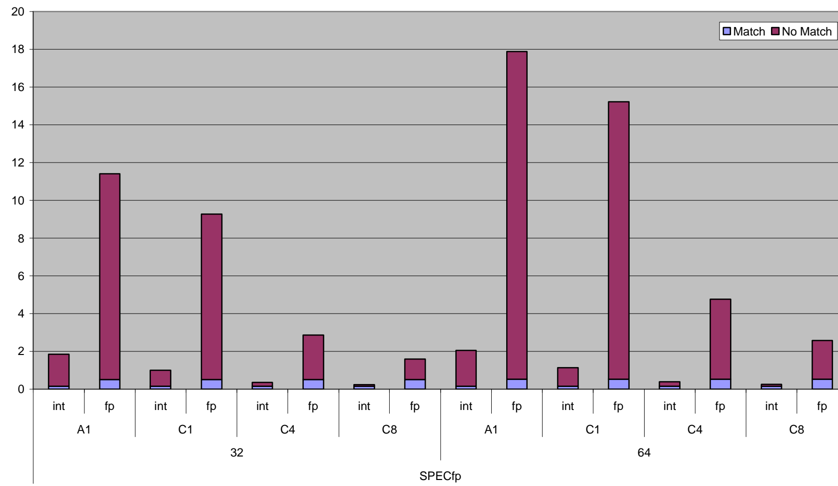

Figure 7: Average Number of Comparisons per Instruction for F.P. Benchmarks

5. Energy Consumption of the Wakeup Logic

The results presented above evaluated the change in the number of tag comparisons performed by various organizations of the instruction queue. This section presents an estimate of the energy consumption and shows the optimal IQ configuration to minimize the energy consumption.

The energy consumption of the wakeup logic was estimated using models of RAM and CAM from the Wattch [17] simulator. The RAM is the Mapping Table that is organized as R entries of B bits, where R is the number of physical registers in the CPU (128) and B is the number of blocks in the IQ. The RAM has 8 write and 4 read ports since it can be accessed by four instructions each cycle. Each instruction can update 2 entries when issued and each completing instruction reads an entry to find successor IQ blocks.

The CAM is the portion of the IQ storing tags and has (IQS / B) entries, where IQS is the number of IQ entries. An entry contains two separate source register tags of 7 bits each and has 4 write and 4 read ports. The total CAM size is 32 or 64 entries, but an access only goes to one block of the CAM.

The number of blocks in the IQ was varied from 1 to 64 in order to find the optimal organization. An IQ with one block represents the standard design. The IQ with block size of 1 does not even require a CAM; the Mapping Table entry identifies each dependent instruction in the IQ.

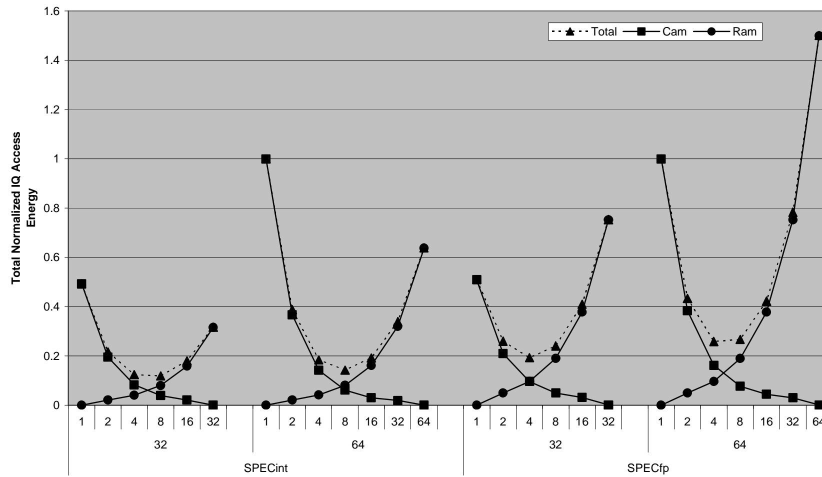

Figure 8: Normalized IQ Energy Consumption for Integer Queue

The energy consumption in the wakeup logic is computed as follows. For each access to the instruction queue a CAM lookup is performed. If more than 1 block is used than the RAM access energy is also included for Mapping Table access, but a smaller CAM is accessed in this case. The total lookup energy is the sum of RAM and CAM access. The lookup energy per program is the total lookup energy per access multiplied by the total number of accesses in the program.

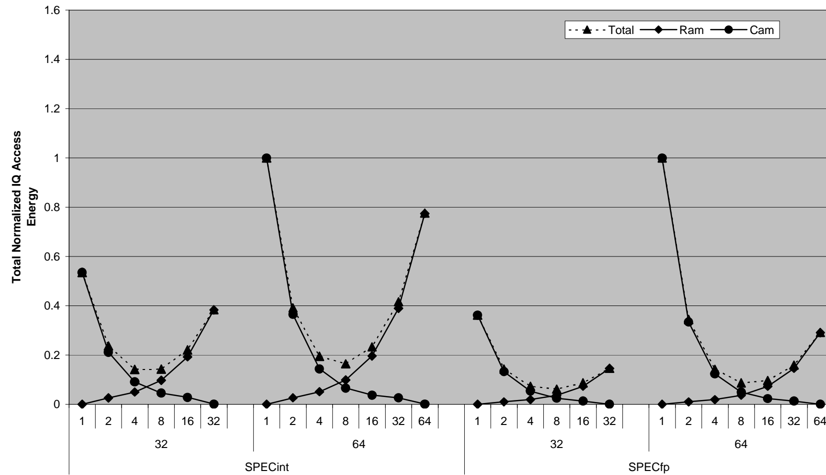

Figure 9: Normalized IQ Energy Consumption for Floating Point Queue

The total wakeup energy per program for each IQ configuration is normalized to the energy consumption of the IQ with 64 entries and 1 block. Figure 8 shows the total normalized energy for the Integer Queue and Figure 9 for the Floating point Queue. The total normalized energy is averaged over all integer and all floating point benchmarks, respectively, for IQs with 32 and 64 entries. In each case the number of blocks is varied from 1 to 32 or 64. The RAM and CAM energy consumption are shown separately.

As the number of blocks is increased, the CAM energy consumption is reduced because only one block is accessed at a time and the number of entries accessed is reduced. At the same time, the Mapping Table RAM entry has more bits and requires more energy to access. The optimal configuration varies between 4 and 8 blocks depending on the total IQ size. The difference between the 4- and 8-block configurations is small.

6. Conclusions

The instruction queue architecture presented in this paper uses a multi-block organization and delivers a large reduction in the number of comparisons per instruction. The major contribution of this work is the use of the block mapping mechanism that allows fast determination of blocks to activate for wakeup when an earlier instruction completes execution. It does not require prediction or adaptivity to achieve the energy savings. This multi-block design performs approximately one and a half comparisons per committed instruction for a 32-entry instruction queue organization with eight blocks (recall that integer *and* f.p. queues are separate).

It is hard to compare results across designs due to the differences in processor configuration, benchmarks, and compilers used. However, an estimate of the design complexity can be made. The design proposed in this paper does not affect the IPC and only reduces energy use. It uses simple hardware and does not require modification outside the instruction queue itself. Compared to other multi-block queue designs it does not require prediction or monitoring. For example, multi-queue design of [9] requires management of multiple queues and a predictor for the last operand. The pointer-based designs are usually more complex. Approaches presented in [5] and [10] require modifications in the register renaming logic and quite complex pointer/bit vector manipulation to add successors. Finally, the full dependence matrix is not scalable and has its own overheads [11] [12]. The design proposed here is a simpler alternative delivering a very large reduction in energy consumption. Its results can be further improved by using compression and other techniques.

7. Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain, under contract TIC-2001-0995-C02-01 by the CEPBA, the program of scholarships SUPERA-ANUIES and COTEPABE-IPN from Mexico, and, in part, by the DARPA PAC/C program under contract F33615-00-C-16 32 in the USA.

8. References

[1] S. Palacharla, "*Complexity effective Superscalar processors*" PhD Thesis, University of Wisconsin, Madison 1998.

[2] Alper Buyoktusunoglu, Stanley E. Shuster. David Brooks, Pradid Bose, Peter W. Cook, and David H. Albonesi, "*An Adaptive Issue Queue for Reduced Power at High Performance*", Workshop on Power Aware Computer Systems, in conjunction with ASPLOS-IX, November 2000.

[3] Daniel Folegnani and Antonio González, "*Energy Effective Issue Logic*", Proceedings of 28th Annual of International Symposium on Computer Architecture, 2001. Page(s): 230-239, Göteborg Sweden.

[4] Shlomo Weiss, James E. Smith, "*Instruction Issue Logic for Pipelined Supercomputers*", Proceedings of 11th Annual International Symposium on Computer Architecture, 1984 Page(s): 110-118.
[5] Toshinori Sato, Yusuke Nakamura, Itsujiro Arita, "*Revisiting Direct Tag Search Algorithm on Superscalar Processors*", Workshop Complexity-Effective Design, ISCA 2001.
[6] Alper Buyoktusunoglu, Stanley E. Shuster. David Brooks, Pradid Bose, Peter W. Cook, and David H. Albonesi, "*An Adaptive Issue Queue for Reduced Power at High Performance*", Workshop on Power Aware Computer Systems, in conjunction with ASPLOS-IX, November 2000.
[7] Vasily G. Moshnyaga, "*Reducing Energy Dissipation of Complexity Adaptive Issue Queue by Dual Voltage Supply*", Workshop on Complexity Effective Design, June 2001.
[8] Gurhan Kukut, Kanad Ghose, Dmitry V. Ponomarev, Peter M. Kogge, "*Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors*", Proceedings of International Symposium on Low Power Electronics and Design August 2001 Huntington Beach California, USA.
[9] Dan Ernst, Todd Austin, "*Efficient Dynamic Scheduling Through Tag Elimination*", Proceedings of 29th Annual of International Symposium on Computer Architecture, 2002.
[10] Michael Huang, Jose Renau and Josep Torrellas, "*Energy-Efficient Hybrid Wakeup Logic*", Proceedings of ISLPED August 2002 Page(s): 196-201, Monterrey California, USA.
[11] Masahiro Goshima, Kengo Nishino, Yasuhiko Nakashima, Shin-ichiro Mori, Toshiaki Kitamura, Shinji Tomita, "*A high-Speed Dynamic Instructions Scheduling Scheme for Superscalar Processors*" Proceedings of 34th Annual International Symposium on Microarchitecture, 2001.
[12] James A. Farrell and Timothy C. Fisher, "*Issue Logic for a 600-Mhz Out-of-Order Execution Microprocessors*" IEEE Journal of Solid State Circuits Vol. 33, No. 5 , May 1998. Page(s): 707-712.
[13] Steven E. Raasch, Nathan L. Binkert and Steven K. Reinhardt, "*A Scalable Instruction Queue Design Using Dependence Chains*", Proceedings of 29th Annual of International Symposium on Computer Architecture, 2002 Page(s): 318-329.
[14] L. Villa, M. Zhang M. and K. Asanovic, "*Dynamic Zero Compression for Cache Energy Reduction*", Micro-33, Dec. 2000.
[15] Vasily G. Moshnyaga, "*Energy Reduction in Queues and Stacks by adaptive Bit-width Compression*",. Proceedings of International Symposium on Low Power Electronics and Design, August 2001 Page(s): 22-27 Huntington Beach California, USA.
[16] Manoj Franklin, Gurindar S. Sohi "*The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism*", Proceedings of 19th Annual of International Symposium on Computer Architecture, 1992 Page(s): 58-67.
[17] D. Brooks, V. Tiwari, and M. Martonosi. "*Wattch: A framework for architectural-level power analysis and optimizations*. Proceedings of 27th Annual International Symposium on Computer Architecture, June 2000.