On-Chip Multiprocessing

Bernard Goossens and Duc Thang Vu

IBP-LITP, Université Paris 7 Denis Diderot, 2 place Jussieu
75251 Paris cedex 05, France
email: bg@litp.ibp.fr, vu@litp.ibp.fr

Abstract. This paper describes a processor design and gives its estimated performance through trace-driven simulation. The processor runs four threads in parallel and issues up to four instructions per thread per cycle. In order execution is assumed to keep the pipeline stages simple enough to have a very short cycle width. Moreover, all the arithmetic operators -adders, incrementers, shifters, multiplier and divider- have been sliced and pipelined with no execute stage including more than the equivalent of a 16 bits adder in its critical path. The first simulation results show a sustained rate of 5 instructions per cycle.

1 Introduction

To improve processor performance, there are actually two tendencies. Either the CPI is favoured or it is the cycle width, as illustrated by the recent processors designs: The DEC 21164 [3] is the last tenent of the second choice, while the MIPS R8000 and the Power2 [8] adopted the first one. By simply taking a look at the SPECint92 and SPECfp92 performances of these three processors [8], it seems that the short cycle choice is better than the low CPI one: 126 and 260 for the Power2, 100 and 310 for the R8000 and 330 and 500 for the 21164. Roughly twice better! However, it is known that there is far from the benchmark performance to the real one. The partisans of the low CPI assert that the technology limitates the ability to lower the cycle indefinitely. And the advocates of the short cycle answer that the lack of fine grain parallelism restricts the CPI decrease. Undoubtedly both arguments are right. Thus, improving performance requires decreasing both the CPI and the cycle.

What about the cycle width? Independently from the technology, architecture can do much on this side. From the first 2 pipeline stages RISC-I microprocessor to the actual 7 to 12 pipeline stages DEC 21164, many architectural improvements concerned the cycle width. In the first generation, the instruction and data fetches were performed off chip, which was by far the most costly operation, fixing the cycle. By integrating caches on chip and pipelining their accesses in todays processors, fetches and load-stores are no more critical events. The arithmetic operators have taken this place, one reason being the enlarging of the data path from 32 bits to 64 bits. As the DEC 21064 [2] design has shown, a 64 bits adder was very difficult to fit in a 5ns cycle. To space out stage design, we propose to slice these operators, computing an addition via 16 bits slices. This provides a potential 50% cycle width reduction over an actual 300Mhz rated processor as the DEC 21164. In a preceding paper [4] we have described such sliced operators, for integer computation and for competitive floating point emulation, including a multiplier and a divider.

Can the CPI be improved while keeping a very short cycle? Issuing and running more than one instruction per cycle requires a lot of hardware. Not that much from 1 to 2 instructions per cycle however as the DEC 21064 shows. But for more, out of order execution is essential, which implies reservation stations, registers renaming and so on. This is costly, both in space and time. The time cost explains the large cycle width of superscalar processors. To allow multiple instructions to be run per cycle, while keeping an in order execution and thus a simple enough architecture not to sacrifice the cycle width, we propose to have a multiprocessor on chip. Four threads will be handled simultaneously, each having up to four instructions issued and run per cycle. The operators will be shared and dynamically allocated to the requesting instructions. The compiler will be responsible for code ordering to reduce the dependencies in a single thread. In a certain sense, because we note there is not enough fine grain parallelism in an instruction flow for our architecture, so we get more by interleaving threads executions. We have measured a 0.2 sustained CPI on a trace driven simulation. This result must be compared to the 0.8 PowerPC 620 CPI [1].

Tullsen and al. [9] have made some experiments on a simulated multithreaded superscalar processor architecture derived from the DEC 21164. They show that this type of architecture has a potential to achieve four times the throughput of a superscalar single-threaded processor. They have measured a potential 0.24 CPI with 4 threads and 16 functional units. Hirata et al. [5] present a multithreaded design and estimate its performance both with single and multiple issue techniques. Because they use multithreading for latency hiding, their conclusion is that their design is most suited to single issue. Their results for single issue are probably optimistic because they did not simulate the memory hierarchy. Prasadh and Wu [7] describe a multithreaded RISC processor design replacing single thread superscalar execution with thread interleaving. Their simulation gave a 0.13 CPI with 4 threads and 10 functional units. However, the architecture is poorly pipelined (3 stages; this lowers the dependencies and highers the single thread CPI) and thus its cycle width should be large.

2 An Overview of the Processor

2.1 The FIFO

Fetched opcodes are kept in a FIFO until they are issued. Four such FIFOs - one per thread- are provided. For each FIFO, the *tail* pointer designates where the newly fetched opcodes are written. The *issue* pointer separates non issued opcodes from issued ones up to the *head* pointer that delimits the FIFO free portion. Opcodes in the [*issue*, *head*[FIFO portion are issued but might still have to be restarted in case of a data cache load miss, while opcodes after the

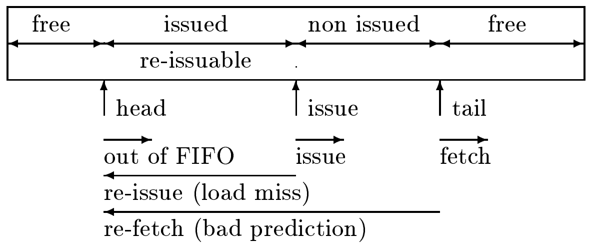

Fig. 1. The FIFO.

head pointer are definitely out of the FIFO. The *tail* pointer runs after the *head* one and when they meet, the FIFO is full. Fetch is suspended until new places have been freed. The *issue* pointer is moved down to the *head* one when re-issue is activated by a load miss (restart issue from the instructions following the load). The *tail* pointer is moved down to the *head* one -FIFO is emptied- after a control flow bad prediction has been detected (restart fetch from the new address). The fetch operation can be viewed as an opcode producer while the execution pipeline is an opcode consumer. The FIFO decouples the producer pipeline from the consumer one.

2.2 The Producer Pipeline

The producer gets its opcodes from the instruction cache. A full 8 words cache block (32 bytes) is fetched per cycle per thread.

Next program counter is speculatively determined in order to allow a new full cache block fetch every cycle. The speculation is based on a branch target cache (BTC). If no valid entry matches the actual PC value, predicted next PC simply points to the next cache block. Otherwise, the BTC gives the prediction. BTC access occurs during the first stage (while the PC-pointed block is being read) and next-PC choice is made at the end of the same stage, among the prediction, the actual PC after incrementation and a possible previous bad prediction correction. The PC incrementer must be especially designed to avoid a full 64 bits carry propagation in a single cycle. All addresses in the processor are 64 bits wide (virtual space), divided into four 16 bits slices. The single cycle incrementation process is restricted to the lowest slice. In case a carry out would be set, fetch would be suspended for one cycle to leave time for propagation.

In the second stage, runnable instructions are selected from the fetched block (see Fig. 2). First, according to the program counter lower bits, the left prefix of the fetched block not pointed to by PC is discarded. Second, if a valid prediction has been read, it matches one of the fetch block entries containing a control flow opcode. The entries on its right are also discarded (control flow break). If no valid prediction was obtained, all the opcodes except the left prefix are kept.

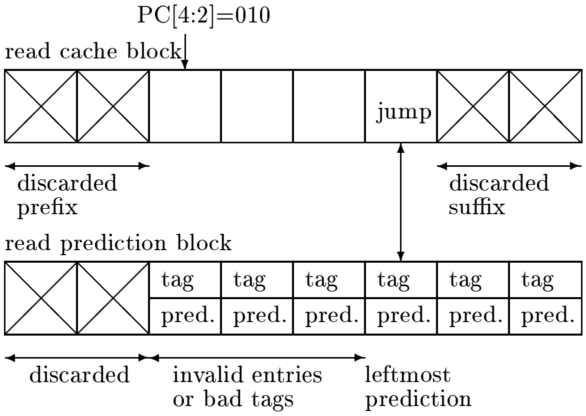

Fig. 2. Runnable Instructions Selection.

The third stage is devoted to immediate jump resolution. The decoded absolute target address is compared to the predicted one. If they do not match, bad fetches are cancelled -two cache blocks- and target address is passed to the first stage. Such control flow instructions, if they do not save a link address, are removed from run flow. Remaining instructions are registered in the FIFO during the fourth stage if enough empty places are available. Otherwise, the producer pipeline remains frozen. These four fetch steps are handled for the four threads simultaneously, which implies four separate fetch paths.

2.3 The Consumer Pipeline

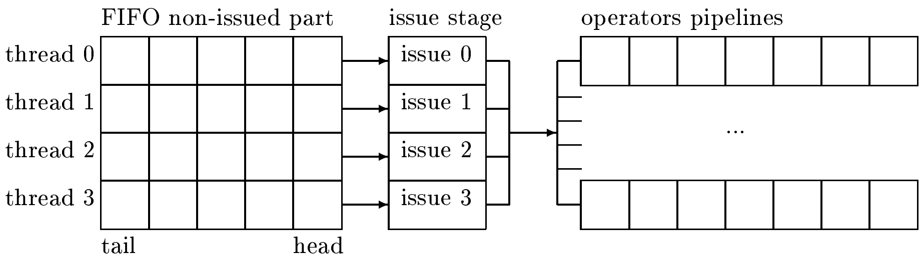

Fig. 3. Multithread Issue.

The Issue Part The consumer pipeline is separated in two parts: the issue and the execution. Head FIFO opcodes are selected to enter one of the execution pipelines, each associated to an operator (see Fig. 3). The four threads compete for free operator attribution and for each thread, the four FIFO head entries participate to the issue selection which is performed in four cycles. Each cycle is devoted to a single thread and the threads are priority ordered. The least priority one will only gain access to the still free resources.

Here are the six conditions an instruction must satisfy to be issued:

1. The preceding instruction in the FIFO must have been selected for issue (in order issue).
2. The source registers must not be locked (they may have been locked by a preceding instruction of the same thread; this concerns long latency operators like the multiplier and the divider; by-pass is provided so that ordinary operators don't lock their destination register).
3. Enough register read and write ports must be available, according to the instruction needs (e.g. no read ports for instruction using immediate values; no write ports for instruction writing in R0 register; ports are dynamically allocated during issue; simulations have helped to fix the number of provided ports: three read ports and two write ports are enough).
4. One copy of the requested operator type must be available.
5. For load instructions, neither previous load nor store must have been issued in the same cycle, for the same thread.
6. For store instructions, no previous store instruction must have been issued in the same cycle for the same thread, and a free store buffer must be available for the thread.

The Execution Part Issued instructions enter their execution pipeline which are all of the same length, corresponding to the longest path, i.e. seven stages for the multiplier and the divider. All results are registered in the last stage. This limits Write-After-Write hazards to opcodes issued in the same cycle. Their issue number ($0 <= n <= 3$) is used to solve the case. By-pass paths are provided between stages -but not for instructions issued in the same cycle- which helps to lower the operations latency.

Dependent instructions in a thread must be separated by the compiler with as many useful independent instructions as possible. However, no NOPs are inserted. A lock mechanism prevents dependent instructions to be issued too early (a 3 bits opcode field contains a delay constant that serves as the destination register lock when the instruction is issued). In the mean time, other instructions from the other threads have more chance to gain access to the operators they need. The static NOP insertion is replaced by a dynamic thread interleaving.

2.4 The Caches

Two cache levels are provided on chip, for instructions as for data. The first level is private to each thread and the second level is shared. Moreover, the first level caches are directly mapped and virtually addressed (with a full address including the context number to prevent synonyms problems). The second level caches are 4-way set associative and physically addressed. Both levels respect a MESI-like protocol for coherence.

The instruction cache is accessed during the first fetch stage. Tags are checked during the second fetch stage. If a miss is signaled (see Fig. 4), the thread producer pipeline is frozen (but not the consumer one if FIFO is not empty) and access to the shared second level cache is requested. Second level misses are handled off chip. As soon as a hit is detected, caches are updated and fetch pipeline is restarted. A first level cache miss followed by a second level cache hit gives a 4 cycles penalty.

The two levels of data caches are organized the same way. First level cache access occurs during the second execute stage with the low part of the computed address. Tag check is performed during the fifth execute stage, when the full address has been computed by a sliced adder. If a load miss is detected, all instructions issued two or more cycles after the load are restarted (the locking feature ensures that instructions issued in the same cycle and in the next one are surely independent from the loaded datum). Their re-issue is obtained by moving the FIFO *issue* pointer down to the *head* one. The producer pipeline is not affected by such events.

Accesses to the shared second level data cache are handled exactly the same way as for the instruction cache. For stores, buffers are used. No thread suspension occurs unless all buffers are busy. In this case, the locking mechanism prevents a new store for the same thread to be issued.

2.5 Conditional Branches and Indirect Jumps

These control flow instructions, because they need computation, have to cross the execution pipeline. The target address is computed in four cycles through a sliced adder. Then, the prediction is checked and if a bad one is detected, the instructions following the offending branch and yet executed are cancelled and the FIFO is emptied. The prediction cache is updated if the branch is taken and the prediction was incorrect.

3 The Simulator

In order to estimate the design performance, we have built a trace-driven simulator. It includes the whole pipeline, with the four stages fetch part, the FIFOs, the four stages issue part and the seven stages execution pipelines. All the caches have been simulated (except the virtual-physical TLBs) Level two misses have been simulated as accesses to a shared output bus (4 words per cycle) and 4 cycles latency external SRAMs.

For now, we have just started measurements, thus our results (see Table 1) must be considered application-specific. The program trace was run for one million cycles corresponding to the middle execution of the simulator itself. The

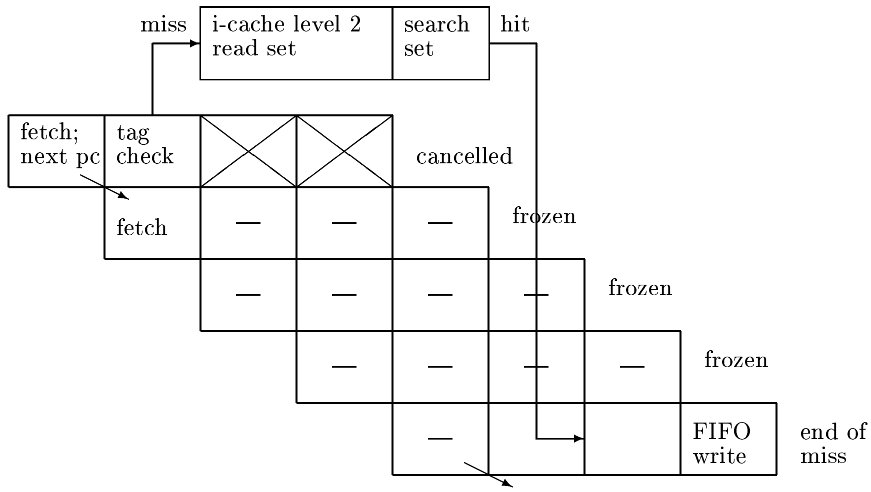

Fig. 4. First Level Instruction Cache Miss Handling.

thread 0	1458092	68503	4.7%	342557	23.5%	128674	8.8%	918358	63%
thread 1	1396411	66508	4.8%	327335	23.4%	122458	8.8%	880110	63%
thread 2	1238596	58533	4.7%	290776	23.5%	108961	8.8%	780326	63%
thread 3	958790	45199	4.7%	225444	23.5%	84140	8.8%	604007	63%
total	5051889	238743	4.7%	1186112	23.5%	444233	8.8%	3182801	63%
	run	c. flow	rate	loads	rate	stores	rate	misc.	rate

Table 1. Simulation Results for 1 Million Cycles.

executed trace contains 4.7% of control flow instructions (this is a very low value, specific to the application), 23.5% of loads, 8.8% of stores and 63% of inter-register instructions (high application-specific rate). Table 2 gives the caches miss rates with 16%, for the prediction cache wich is comparable to known rates [6]. The simulation consisted in the same program run for the four threads, but started at different addresses.

The simulated processor contained 8KB first level instruction caches, 16KB first level data caches, a 64KB second level instruction cache, a 128KB second level data cache (for a total of 288KB of caches: 4*(8KB + 16KB) + 64KB + 128KB), four 1024 entries branch target caches, six adders, three control flow operators (for conditional branches, indirect jumps and calls), three logic units, one left shifter, one right one, one multiplier and one divider (16 operators altogether). The FIFOs were fixed to 24 entries each. Two write ports and three read ports were provided for each register file access and eight store buffers per

th 0	275488	10985	3.83%	445216	26016	5.52%	240658	45815	16%
th 1	264811	10902	3.95%	423238	26555	5.90%	233471	42242	15.32%
th 2	233907	9324	3.83%	373428	26309	6.58%	204723	38508	15.83%
th 3	180438	7077	3.77%	284006	25579	8.26%	155377	32138	17.14%
total	954644	38288	3.86%	1525888	104459	6.41%	834229	158703	15.98%
	i hit	i miss	i rate	d hit	d miss	d rate	p hit	p miss	p rate

Table 2. Caches Hits and Misses.

thread for data cache access. Such a processor is estimated at 35M transistors. We obtained a CPI of 0.198, i.e. a little more than 5 instructions per cycle (1.26 instructions per thread per cycle).

4 Conclusion

We have presented a design intended to be an implementation of a multi-processor in a single chip. The different parts of the processor have been investigated, from the fetch data path to the execution one and including the internal memory hierarchy. We did not describe the floating point computation model. This has been presented in [4]. The first simulations of our design shows that a sustained rate of more than 5 instructions per cycle can be achieved without the need of out of order execution, thus allowing a shorter cycle.

References

1. T.A. Diep, C. Nelson and J.P. Shen: Performance Evaluation of the PowerPC 620 Microarchitecture. 22nd AISCA, 1994
2. D.W. Dobberpuhl et al.: A 200 Mhz 64 Bits Dual Issue CMOS Microprocessor. IEEE Journal of Solid State Circuits, vol. 27-11, 1992
3. J.H. Edmondson, P. Rubinfeld, R. Preston and V. Rajagopalan: Superscalar Instruction Execution in the 21164 Alpha Microprocessor. IEEE Micro, april 1995
4. B. Goossens and D.T. Vu: Une Unique Unité de Calcul RISC pour les Entiers et les Flottants. 2nd Real Number and Computers Conference, 1996
5. H. Hirata et al.: An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. 19th AISCA, 1992
6. J.K.F. Lee and A. J. Smith: Branch Prediction Strategies and Branch Target Buffer Design. Computer, 1984-01
7. R.G. Prasadh and C.L. Wu: A Benchmark Evaluation of a Multithreaded RISC Processor Architecture. ICPP, 1991
8. A. Seznec, Y. Mével: Etude des Architectures des Microprocesseurs DEC 21164, IBM POWER2 et MIPS R8000. Rapport de Recherche INRIA 2553, 1995
9. D.M. Tullsen, S.J. Eggers and H.M. Levy: Simultaneous Multithreading: Maximizing On-Chip Parallelism. 22nd AISCA, 1995