Further Pipelining and Multithreading to Improve RISC Processor Speed. A Proposed Architecture and Simulation Results

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 presents a new pipeline architecture which should improve both the number of Cycles Per Instruction (CPI) and the cycle width. The pipe stage critical path, imposed by the 64 bits integer unit, has been cut in half by data slicing and pipelining the Arithmetic and Logic Unit (ALU). Moreover, because this proposed pipeline stages division should impose a very long latency for CPU external accesses, a multithreading structure has been included. Up to four threads may be simultaneously run with a no delay context switch. Thus, multithreading is mainly used as a latency hiding technique for external accesses and internal dependencies. In order to estimate the real benefit of the construct, a simulator has been built. Simulation results show the impact of pipeline improvements without multithreading (between 24% and 32% according to cache size) and with it (from 56% to 65% with four threads and same caches).

1 The Three RISC Generations

Processor efficiency *Peff* is given by the following (and now famous) equation:

$$Peff = CPI * cycle * number\_of\_instructions \tag{1}$$

The *CPI* term in (1) has been improved in two steps: from Complex Instruction Set Computers (CISC) to Reduced Instruction Set Computers (RISC) and from RISC to superscalar. The evolution of the *cycle* term follows the evolution of RISC processors themselves. It can be decomposed in three successive steps corresponding to improvements in the pipeline organization.

The first so-called RISC processor, namely RISC-I [13], had a two stages pipeline: instruction fetch and register file read for the first stage, ALU computation and register file write for the second one. ALU could be used by three different types of instructions: register ones, load/stores and Program Counter (PC) relative branches. In the two latter cases, the ALU computed an address. In case of a load/store, a third cycle was needed to perform the memory access, fetch being suspended. In case of a branch, the target address was only known after a one cycle delay. However, the pipeline was not suspended in such a case and the instruction following the branch in memory, namely the delay instruction, was normally fetched and executed.

The two pipeline stages were speed-balanced because in the early 80's, a 32 bits ALU in nMOS technology had a speed comparable to a Dynamic Random Access Memory (DRAM) external access.

The next year appeared the MIPS processor [8]. It had two separate memory accesses for instructions and data and the pipeline included a special stage for data external accesses. By this mean, in contrast with RISC-I, every instruction would require the same time to be computed and no fetch suspension would occur during load/stores. The MIPS pipeline is said to have five stages but it merely corresponds to three stages, each stage being subdivided in two halves. The first half does the fetch with a ready address and simultaneously increments PC. The second half decodes the received instruction and reads the operands from the register file. This ends the first stage. The second stage uses the same ALU twice. During the first half, it computes a memory address, either for load/store or for PC-relative branch. Second half serves either for result computation (inter-register instruction) or to store a datum (store instruction). Third stage first half achieves load and register file write. Second half is not used.

MIPS and RISC-I are the first step (DRAM) in pipeline evolution. In both cases, the pipeline division is based on the fact that external access and ALU computation take the same time. Both processors can also be considered under-pipelined [10] with each stage including a register file access in addition to its main operation (ALU or memory access).

The second step is represented by the two industrial versions of the RISC and MIPS prototypes: the SPARC [12] and the MIPS R2000 [11]. The main innovation lies in the apparition of a Static RAM (SRAM) cache between the processor and the DRAM memory. This was imposed by the fact that processor speed had been more improved than DRAM speed, requiring more than a CPU cycle to perform an access (in fact, an instruction cache had already been provided in the first MIPS prototype). However, this had no impact on the pipeline because the SRAM speed still allowed a single cycle fetch. Thus, the only improvement in the pipeline design resided in the separation of the register file read from fetch stage and register file write from execute stage. This had a drawback: the execute stage was then no more adjacent to the fetch one. Computing branch target address through the ALU would impose a 2 cycles delay for control flow instructions. To keep the delay to a single cycle, an address adder was provided in the decode stage. In these designs, the cycle width had been improved a little but was still fixed by the external access time, close from the 32 bits ALU time.

The third step came when it was possible to include caches in the processor chip as in the MIPS R4000 [3] and in the Alpha 21064 [4]. Then, fetching, loading and storing no more needed an external access and could be done much faster. In the same time, the data width extended from 32 bits to 64 bits, imposing a 64 bits ALU. The computing stage became the most critical one.

2 Further Improvements in Pipeline Design

The generation to come could further improve pipeline critical path by data slicing and pipelining the ALU and the next address compute operator. In [6] we have shown how integer arithmetic operations (addition, multiplication and division) can be sliced, by operating on 16 bits data slices. For example, a 64 bits addition can be computed through four pipeline stages, each containing a 16 bits adder, providing a 50% speed improvement compared to a 64 bits adder[1].

Figure 1 shows a 7 stages pipeline (*ic* stands for instruction cache read, *itag* for instruction cache tag check, *dc* and *dtag* have the same meaning for data cache, *rf rd* is the register file read stage, *rf wr* the register file write one and *exec* the ALU stage). Loads have a 3 cycles latency (from *rf rd* to *dtag*) and conditional branches must be predicted during *itag* stage. A bad prediction is detected during *exec* stage and annuls 2 instructions (the ones in *ic* and *itag* stages). This pipeline is derived from what can be found in actual superpipelined processors like the MIPS R4000 or the DEC 21064 if we leave apart the superscalar features.

ic	itag	rf rd	exec	dc	dtag	rf wr

Fig. 1. A seven stages pipeline

pf	fi	ti	ci	rr	x1	x2	x3	x4	td	cd	rw

Fig. 2. A twelve stages pipeline with a sliced ALU

Figure 2 shows what such a pipeline would become after ALU and next address compute operator slicing. This 12 stages pipeline gives load a 6 cycles latency and bad prediction branches annul 6 instructions. These stages are described below:

1. Prepare fetch address (*pf* on Fig. 2)
   Fetch address preparation is handled out of fetch stage to shorten fetch operation. The address is obtained via a by-pass selection of the previous incremented fetch address not yet registered in PC, and PC itself. PC register,

[1] In the same manner, a multiplication step operator giving a 8*64 bits result and a radix 16 division step operator have been proposed. The three operators are based on the same four stages pipeline, each stage critical path being limited to the crossing of a 16 bits adder.

however modified every cycle, is written only at the end of last pipeline stage, as are all the registers. This simplifies instruction cancellation and restart after load miss and bad branch prediction. The *pf* stage latched output is fetch address.

2. Fetch and compute next address lower part (*fi* on Fig. 2)
   Fetching means addressing the internal instruction cache with the lower part of the fetch address. This lower part has been prepared during *pf* stage. In parallel with instruction cache access, fetch address lower part is incremented. The incremented value is by-passed to the *pf* stage. In the same time, fetch address upper part is prepared through by-passing. The *fi* stage latched outputs are the next address lower part, the prepared upper part address and the set of tags and data extracted from the cache.
3. Translate the virtual address into a physical one and compute next address upper part (*ti* on Fig. 2)
   This involves an access to a Translate Lookaside Buffer (TLB), with a search of full fetch address. Only the upper part of the address is changed by the translation. In parallel, it is incremented and by-passed to the *fi* stage. The *ti* stage latched outputs are the next address upper part and either the physical address or a Memory Management Unit (MMU) miss indication.
4. Check the instruction cache tags (*ci* on Fig. 2)
   Cache tags check compares the tags output after *fi* stage to the physical address upper part output from *ti* stage. The *ci* stage latched output is either the fetched instruction or a cache miss indication.
5. Read the instruction operands 16 lower bits from register file (*rr* on Fig. 2)
   This stage decodes the fetched instruction. Unlike traditional pipelines, operands are not read in one shot for data dependency reasons. The latched outputs are the by-passed operands slices.
6. Compute result 16 lower bits and read operands bits [31:16] from register file (*x1* on Fig. 2)
7. Compute result bits [31:16], read operands bits [47:32] from register file and access data cache (for load/store instructions) (*x2* on Fig. 2)
   Data cache is addressed with the lower part of the computed virtual address, which is identical to the physical one. The outputs are the tags and data cached in the addressed set.
8. Compute result bits [47:32] and read operands bits [63:48] from register file (*x3* on Fig. 2)
9. Compute result bits [63:48] (*x4* on Fig. 2)
10. Translate the data virtual address into a physical one (for load/store instructions) (*td* on Fig. 2)
    This involves an access to the TLB, with a comparison of the computed address to each entry. The output is either the physical address or a MMU miss indication.
11. Check the data cache tags (for load/store operations) (*cd* on Fig. 2)
    Cache tags check compares the tags output from stage *x2* to the physical address output. This stage output is either the data read (load instruction)

or a write enable (store instruction) or a data cache miss indication.

12. Write result in register file or in data cache. Update PC and other special registers (*rw* on Fig. 2)

Result is written in one shot to allow precise interrupt handling. This implies a propagation of result slices from the stage they are output to the write back stage.

In every stage, each operation propagation delay is kept lower than a 16 bits adder one. This is the case for the PC-slices incrementers. This is also the case for internal cache access (mainly an address decoding), tag check (equality comparator), MMU translation (also equality comparators) and register file read and write. By-pass mechanism must also be carefully designed: it is built with a cascade of multiplexers, passing the latched results. The last multiplexer is the only one in the critical path, passing the only unlatched result computed during the same cycle. This concerns all the registers, including PC.

3 Improvement Impact on Processor Cycle: Simulation Results

However such an improvement in actual pipelines should allow a 50% speed increase as expected at the beginning of Sect. 2, the real benefit is limitated by the penalty imposed to the processor in case of cache misses. This penalty should be more important for the proposed architecture because the speed difference between the processor and the SRAM secondary cache is increased. The extension of the pipeline also increases loads and control flow instructions latencies which should further reduce the gain.

In order to give an accurate estimation of the real improvement, and hints to fine tune the pipeline (reduce the number of lost cycles), we have built a simulator. The simulation process is based on random drawings of the instruction flow. Different instruction types have been distinguished to take care of external accesses latency and of idle cycles induced by loads and jumps latencies. Their distribution has been choosen according to [9]: 25% of loads, 13% of stores, 14% of conditional branch instructions, 2% of other control flow instructions and 46% of miscellaneous instructions. Each load has been charged a two cycles penalty in the 7 stages reference pipeline (Fig. 1) and a four cycles penalty in the 12 stages proposed pipeline (Fig. 2) (we have considered that on the average, 2/3 of the delay slots were filled with No OPeration instructions (NOP); this estimated rate is not based on any measure).

The simulation parameters include the instruction and data internal caches sizes, given in Table 1 in Kilo Bytes (KB), the external bus size and the external unified cache size. Another important parameter is the SRAM/CPU speed ratio. It has been fixed to 6 for the proposed processor and to 3 for the reference one (the reference ratio corresponds to 15ns SRAM linked to a 5ns CPU like the DEC 21064).

The miss rates for caches have been taken from [9] and appear in Table 2.

Table 1. Memory hierarchy parameters used in the simulator

Cache size	Int. cache size (KB)	Ext. cache size (KB)	Ext. bus size (bits)
small	16+16	512	64
medium	32+32	1024	64
large	64+64	2048	128
very large	128+128	4096	128

Table 2. Cache miss rates applied to the simulations

Cache size	Instruction cache	Data cache	External cache
small	0.036	0.053	0.0072
medium	0.022	0.040	0.0045
large	0.014	0.028	0.0039
very large	0.010	0.021	0.0022

Simulation results shown on Table 3 are based on runs of 10000 instructions. It has been checked that larger runs do not change results due to the uniformity of the random drawings. It should be noticed that the cycles in the 12 stages case are half of the ones in the 7 stages case. This is why Table 3 third column shows the ratio 6 raw results and in parenthesis the ratio 3 scaled results.

Table 3. Number of cycles to run 10000 instructions

Cache size	7 stages, ratio 3	12 stages, ratio 6	Improvement
small	31202	46192 (23096)	26%
medium	25732	39064 (19532)	24%
large	19156	26752 (13376)	30%
very large	17665	23899 (11950)	32%

The simulations results show, as expected, that the most part of pipeline improvements is cancelled out by external accesses and dependency latencies. Even if caches and buses are highly enlarged, the CPI remains high.

4 Branch and Load Latencies in a Highly Pipelined Processor

Loads and control flow instructions are long latency and frequent operations. For these two reasons, they must be especially adapted to deep pipelines.

Control flow instructions can be separated in two types: PC-relative and absolute. PC-relative branches have always represented a hardware complication: they need an adder to be performed. In our proposal, this adder would have to be sliced as the ALU is. Then, next instruction address would not be complete before the end of stage *x1* (fetch followed by a four slices addition), delaying cache tag check until this moment. Late cache miss detection is not favorable since it leads to the annulment of more instructions.

PC-relative branches have been invented for two purposes: give a relocatable property to the code and more compactly encode immediate target address. The first purpose is no more actual since the MMU ensures code relocation through memory paging. The second purpose is becoming less important because the instruction operation code (opcode) width is increasing: in a 16 bits opcode, operation type and branch condition encoding leave very little space for target immediate address. This field may not be interpreted as an absolute address. In a 32 bits opcode, target address is still too small to be considered absolute. But in a 64 bits opcode, an at least 48 bits target absolute address may be encoded. The price is an enlarged instruction codification but the benefit is a simplified hardware with a one cycle delay unconditional branch.

In the proposed pipeline, branching is always absolute and handled in two stages. First, the branch opcode is fetched during *fi* stage. Then, a pre-decoding is done during *ti* stage so that any break in the instruction flow will be effective after a one cycle delay.

In case of a conditional branch, the opcode will contain a prediction bit: go on with target instruction for backward conditional branch and do not branch for a forward target (a prediction bit is needed since absolute target field in the opcode does not directly indicate if the target point follows or precedes the branch instruction; with a relative displacement, this can be deduced from the field sign bit; anyway, this prediction technique should be efficient because loop branching instructions opcodes give a correct prediction for each loop but the last one; for *if then else* statements, prediction hit is enhanced if the rare case is placed in the *else* part). In case a false prediction is made, falsely fetched instructions will be annulled (at most 6: see Fig. 3; prediction check is done during the last ALU stage and delay instruction is not annulled).

Indirect jumps may need an ALU operation to provide target address. In order to keep branching delay to one cycle, the target address preparation will have to be done in a preceding inter-register instruction. The address preparation and the indirect jump will have to be separated by at least 6 cycles (see Fig. 4; a 64 bits target address will be obtained at the end of fourth ALU stage; in a 7 stages pipeline like the one on Fig. 1, only 2 separation cycles would be needed; a 6 cycles delay is not a handicap since indirect jumps are unfrequent instructions). The target address is read from a source register during *ti* stage, requiring a special read port. It is by-passed to the *pf* stage (a selection is made between the *fi* incrementer output -no branch- and the *ti* target address output -branch- based on prediction bit; see Fig. 5). Return from subroutine is a special case of indirect jump. It will be handled the same way.

Load instructions have a longer latency in the 12 stages pipeline than in the 7 stages one, due to the more stages between the register file read stage and the data load stage. Loaded datum may be by-passed to register file read stage after

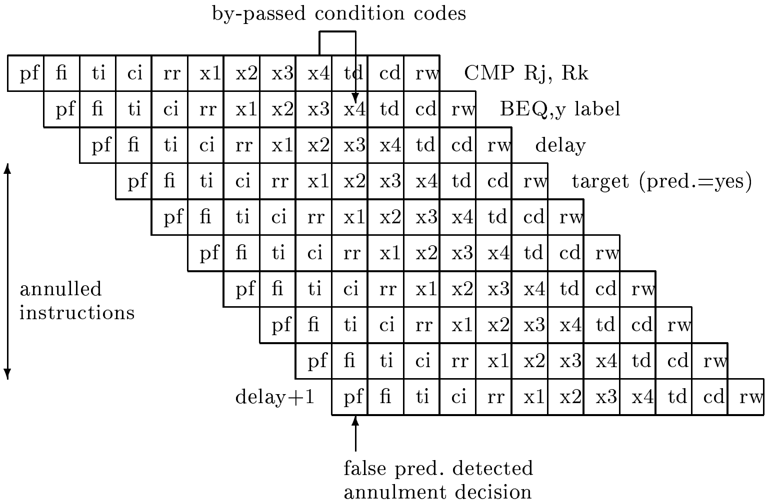

Fig. 3. Conditional branch

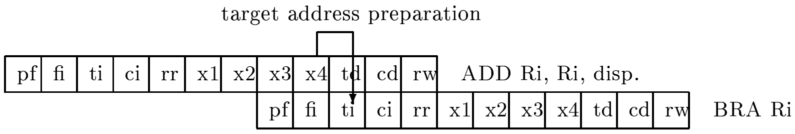

Fig. 4. Indirect branch

data cache tag has been checked. This is a very long latency (6 cycles for the 12 stages pipeline and 3 cycles for the 7 stages one) for a so frequent instruction type: if it is considered that 25% of the executed instructions are loads, it means that groups of loads should occur at least every 10 cycles. This leaves very few instructions that may be inserted in the loads delays. It is better to allow loaded datum by-passing as soon as it has been read from cache, even though a miss would be later detected. The data cache must then have a direct organization so that there is a single word and tag output. The load latency is now given by the distance between the register file read stage *rr* and the data cache read stage *x2*, i.e. 2 cycles only. However, instructions might have to be annulled each time a miss is detected. But, because load hits are more frequent than load misses, it is still an improvement.

As we have said in Sect. 2, to handle annulments, the PC register is written at the end of the last pipeline stage, as are the other registers. A by-pass mechanism provides the temporarily incremented value to the following fetch cycles. It also filters values coming from annulled instructions. For a miss-predicted branch, all the instructions following the branch except the delay one are annulled. For a load miss, all the instructions including the load are annulled (the load is restarted when the missing block is in the cache).

Simulations of these improved pipelines have been made (in the preceding simulation results, branch prediction was already included, with a 20% bad prediction rate; so, these new results show the impact of loads improvement). The results appear in Table 4 with the improvements from the preceding pipelines.

Table 4. Enhanced pipelines simulation results

Cache size	7 stages, ratio 3	Improvement	12 stages, ratio 6	Improvement
small	29430	6%	45547 (22774)	1%
medium	23439	9%	33918 (16959)	13%
large	17287	10%	21897 (10949)	18%
very large	15769	11%	19268 (9634)	19%

More detailed results explain why the improvement cannot be important: the lost cycles are mainly due to CPU suspensions during cache block refill. For example, the unimproved 12 stages pipeline is idle (cache block refill latency) during 25578 cycles, i.e. 71% of all the lost cycles with a small cache and during 4323 cycles, i.e. 31% of the lost cycles when the cache is very large. The main improvement should come from external access latency hiding techniques.

5 Multithreading as a CPI Improving Technique: External Access Latency Hiding

In order to hide external access latency, one can use at least two techniques: multithreading and anticipation. Anticipation is essentially a software technique because a good anticipation of cache loads, whether for instructions or data, is program dependent. Multithreading in constrast is mainly a hardware problem. A multithreaded processor runs more than one thread in pseudo-simultaneity. To do so, it is provided with several register files and program counters. The pipeline is shared.

The described multithreaded computers may be separated in two types: every cycle, no delay context switch (HEP like [14], [7]) and suspending event context switch (APRIL and SPARCLE like [1], [2]). In HEP, the pipeline may not contain more than one instruction per thread. The consequence is that a reasonable efficiency may be obtained only when enough threads are available, at least as many as pipeline stages. In APRIL and SPARCLE, the pipeline may not contain instructions from more than one thread. Thus, it must be emptied upon each context switch: the lengthier the pipeline, the more the suspension cycles.

Moreover, because internal caches are still small today, suspending events (cache misses) should be quite frequent.

Our proposal is in between HEP and SPARCLE. The pipeline will contain instructions from ready threads. If only one thread is ready, the processor will perform as SPARCLE and as a single threaded processor. If many threads are ready, one will be selected for fetch.

The maximum number of threads has been limitated to 4. The reason is that if more than one thread is often suspended at a time, this probably means that the suspension time is more important than the mean time between two suspensions. Then, progressively all the threads but one will be suspended. This situation being not favorable, it is better to build the processor and its memory hierarchy so that an external access lasts less than the mean time between two suspensions. In such a case, two threads are enough to hide latencies. However its probability should be low, two close suspensions may concern the two threads. A third one would then be useful. This other thread may also be convenient to reduce the dependencies between instructions. One more thread may give a little more flexibility.

Each thread slot in the processor may be in one of four states: empty slot, ready thread, suspended thread (cache block refill) or long latency operation in progress (load or conditional branch).

PC by-pass mechanism is provided for each thread so that four prepared fetch values are latched at the end of *pf* stage. In order to keep cache and MMU accesses simple, only one fetch may occur per cycle. The selection of the fetching thread is done during *pf* stage. A configuration of four output enable bits is formed and latched. These bits are used at the beginning of *fi* stage to enable one fetch latched value among the four to address the instruction cache. If no thread is ready, no fetch takes place. The selection process is based on a round robbin priority, leaving away from the selection the suspended threads stopped by a cache miss and the empty slots. Between a ready thread and a thread running a long latency operation, the first one is selected. This feature has the advantage to interleave other ready threads when a load or a conditional branch is being run, so that in case of a late load miss or bad prediction detection less annulments will occur.

An instruction cache miss is detected during *ci* stage. The thread is marked suspended until the cache controller achieves the missing cache block load. The falsely fetched instruction must be annulled. If successive instructions of the same thread have yet been fetched (at most 3 in *fi*, *ti* and *ci* stages while the missed instruction is in its *rr* stage), they are also annulled. The execution of the thread restarts from the missing instruction as soon as it is available (notice that the suspended thread's PC will not be updated neither by the missing instruction nor by the annulled ones).

In case of a data cache miss caused by a load instruction, the cache controller performs the load while the processor keeps running other ready threads. Because the thread is suspended during this operation, its register file is no more in use. Nevertheless, the CPU is not suspended so that writing in the register file is still possible. Thus, the cache controller may procede to the register write and terminate the load, which need not be restarted. Moreover, successive instructions that have been fetched before the miss was detected, as long as they are independent from the load, may be continued. In a single thread processor, the compiler would have to reorganize the code and possibly insert NOPs to force independency. In a multithreaded CPU, it is better to rely on threads interleaving to ensure independency. Thus, the compiler never inserts NOPs and can no more guarantee independency. In such a case, the hardware inserts the NOPs (SPARC like internal NOP opcode) only when needed, i.e. when no other thread is in the ready state and next instruction is a dependent one. This is why, since the load has a two cycles latency, the two slots following it surely contain independent instructions (naturally independent, inserted NOP or belonging to another thread). They may procede, even though the thread is suspended. Annulments concern at most 8 instructions but surely less than that, thanks to threads interleaving.

Control flow breaks must also be adapted to the multithreaded case. The reason is that we cannot be sure that the delay instruction will be run one cycle after the control flow instruction (an instruction of another thread may be inserted in between). The instruction type and prediction bits are pre-decoded during *ti* stage and form the break bit (break bit is asserted if and only if opcode type is control flow and opcode prediction bit is asserted). The target address is latched at the end of the *ti* stage. The break bit and the target address are forwarded all along the pipeline and registered at the end of the *rw* stage. A by-pass mechanism (see Fig. 5) provides the target address and the branch decision to the *fi* stage, which passes either the incremented fetch address or the target address to the *pf* stage, according to the by-passed break bit. Figure 5 low part shows how fetch address is by-passed when threads are interleaved in the pipeline. The by-passed latched value is either the latched incremented one (no branch) or the latched target address (branch). If the delay instruction is itself a branch instruction with an asserted branch prediction bit, it will reassert break bit. Otherwise, break bit will be cleared as any non control flow instruction does.

We have adapted the simulator to the multithreaded architecture with a varying number of threads from 1 to 4. The simulation results are given in Table 5. For the simulations it has been considered that threads use disjoined memory areas. Thus, caches are shared and miss rates are fixed according to the share size. Runs were still limited to 10000 instructions, i.e. for 4 threads, roughly 2500 each. Table 6 shows the improvements in performance, due to multithreading (m percentage column: comparison between the 12 stages pipeline result in Table 4 and the results from Table 5) and to both multithreading and ALU slicing (m+s percentage column: comparison between Table 4 seven stages pipeline results and Table 5 results). We can see that even though cache size is increased, multithreading keeps paying. A second result that can be deduced from the two tables is that the most part of the multithreading improvement is due to the adding of a single second thread.

Detailed results are presented in Table 7. In each cell, the values are the

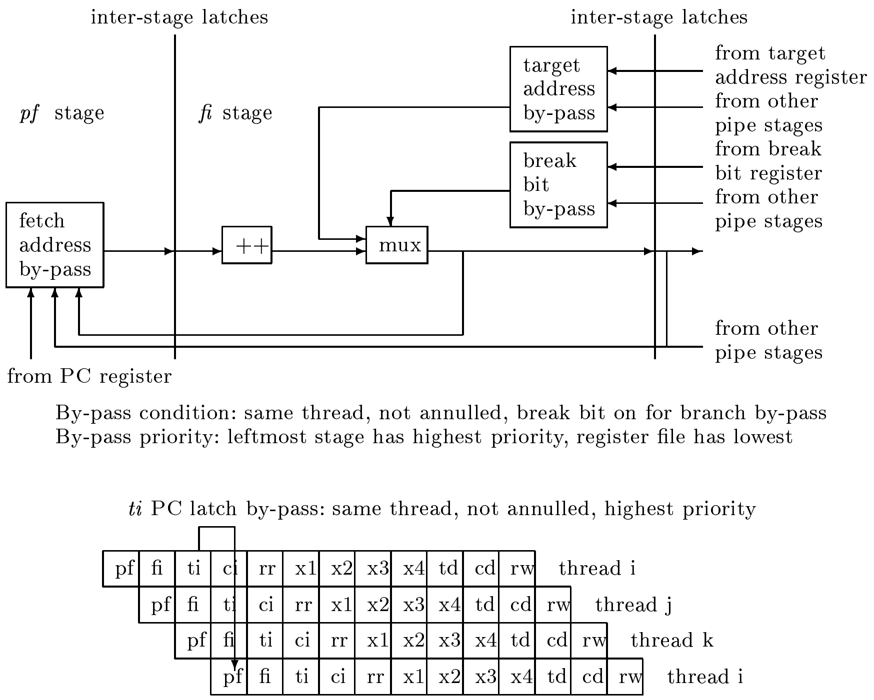

Fig. 5. PC and branch address by-pass mechanism

Table 5. Multithreaded CPU simulation results

Cache size	One thread	Two threads	Three threads	Four threads
small	43637	31915	27554	25787
medium	33157	23664	19810	17830
large	21400	15136	12976	12130
very large	19068	13971	11960	11197

Table 6. Multithreading performance improvement over monothreading

Cache size	One thread		Two threads		Three threads		Four threads	
	m	m+s	m	m+s	m	m+s	m	m+s
small	4%	26%	30%	46%	40%	53%	43%	56%
medium	2%	29%	30%	50%	42%	58%	47%	62%
large	2%	38%	31%	56%	41%	62%	45%	65%
very large	1%	40%	27%	56%	38%	62%	42%	64%

number of lost cycles due to fetch miss, load miss and inserted NOPs, bad branch prediction and idle CPU (no thread ready).

Table 7. Multithreaded CPU detailed simulation results

Cache size	Loss type	One thread	Two threads	Three threads	Four threads
small	fetch	1452	2301	2887	3445
	load	3549	3071	2720	2500
	branch	1205	932	858	733
	idle	27421	15601	11079	9099
medium	fetch	852	1334	1649	1976
	load	3478	2780	2227	1703
	branch	1259	939	741	649
	idle	17558	8599	5183	3492
large	fetch	552	687	803	947
	load	3278	2161	1250	638
	branch	1290	866	569	326
	idle	6270	1412	344	209
very large	fetch	400	447	434	517
	load	3046	1940	875	389
	branch	1280	825	496	236
	idle	4332	749	145	45

6 Shared Pipeline: Improvement to Further Reduce the CPI

Dependency situations can be further reduced through pipeline sharing. For this purpose, we separate the pipeline in two parts (see Fig. 6). Stage *pf* to stage *ci* form the fetch part, producing checked and ready to be run opcodes. Stage *rr* to stage *rw* constitute the run part, consuming opcodes. These opcodes are stored at the tail of one among four First In First Out structures (FIFO) (one per thread). The *ci* stage serves for the run part as the *pf* stage does for the fetch part: a selection occurs among the ready threads to choose the FIFO head entry that will enter the *rr* stage for run. This mechanism ensures that when a fetch miss is detected, however no new opcode is registered in any FIFO, if at least one is not empty, its head entry will enter the run part instead of a bubble. Moreover, annulments in case of a fetch miss will no more correspond to idle cycles if the ready opcodes stock is not empty.

In addition to the opcodes coming from the fetch part, the FIFO may receive ready to be run opcodes from the run part after a load miss. Among the 8 annulled instructions, five of them are in the run part from *rr* to *x4* stage. All those belonging to the same thread re-enter its FIFO.

Each FIFO has 5 places. Each stage keeps track of the FIFO entry number where the opcode it takes care of resides. When a load miss occurs, the head

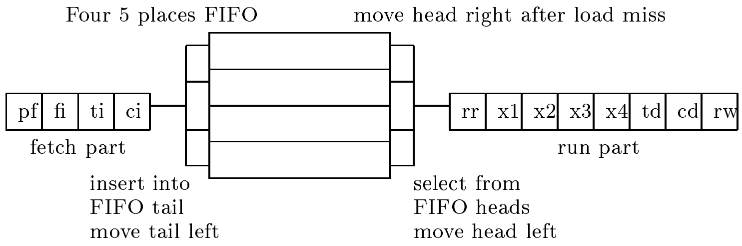

Fig. 6. Two parts pipeline

pointer is updated with the oldest FIFO entry number by-passed, corresponding to the annulled instruction next to the load (the pointer moves backward of at most 5 positions). In the same time, the tail pointer gets the old head pointer value. Instructions of the same thread in the pipeline fetch part are also annulled. For bad branch predictions, the annulled instructions will not be restarted and they are not inserted in the FIFO. Finally, a thread with a full FIFO does not compete for fetch selection. A last simulation has been made and the results appear in Table 8, with the number of cycles (*nbc.*) needed to run 10000 instructions and the improvement (*imp.*) with the non separated pipeline. Two reasons explain the improvement values obtained: first, by increasing the caches sizes, we decrease the number of misses and also the number of restorable lost cycles due to fetch annulments and the number of FIFO inserted opcodes after load annulments; second, less annulled opcodes are inserted in the FIFO when the number of threads is increased, because thread interleaving is forced while a load is in progress.

Table 8. Two parts pipeline simulation results

Cache size	One thread		Two threads		Three threads		Four threads	
	nbc.	imp.	nbc.	imp.	nbc.	imp.	nbc.	imp.
small	39220	10%	27867	13%	24297	12%	21589	16%
medium	29015	12%	21672	8%	17674	11%	16069	10%
large	18507	14%	13437	11%	11922	8%	11574	5%
very large	16119	15%	12381	11%	11324	5%	10950	2%

7 Conclusion

In [5], we have proposed another multithreaded architecture. This proposal differs in many ways: firstly, by the simulation results not available in [5]; secondly, in the architecture. As the simulations have shown, the new proposal performs better with a more simple hardware.

In this paper, we have proposed a new pipeline architecture, where all the stages have a critical path close to a 16 bits ALU delay. To obtain such a speed, the ALU has been sliced, giving a deeper pipeline. In order to tolerate the increased speed difference with its external memory hierarchy, the CPU has been given a multithreading facility, allowing the interleaved execution of up to four threads. A simulator has been built and simulation results show that multithreading is essential to really improve performance through superpipelining, whatever the internal cache size is.

Extending the proposed pipeline to a Very Large Instruction Word (VLIW) structure is possible, but the regularity of the pipeline division must not be broken. In a certain sense, to build an efficient VLIW or superscalar processor, first build an efficient superpipelined processor as was done for the DEC 21064.

Such a processor can also serve as the main building block in a multiprocessor. Multithreading helps to tolerate external communications latency and should perform well in a Cache Only Memory Architecture (COMA).

References

1. A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz: APRIL a processor architecture for multiprocessing. Proceedings of the 17th AISCA, 1990
2. A. Agarwal, J. Kubiatowicz, B Kranz, B.H. Lim, D. Yeung, G. D'Souza, M. Parkin: SPARCLE: an evolutionary processor design for large scale multiprocessor. IEEE Micro, june 1993
3. A. Bashteen, I. Lui, J. Mullan: A superpipeline approach to the MIPS architecture. IEEE, 1991
4. Digital Equipment Corporation: Alpha architecture reference manual. DEC, 1992
5. B. Goossens and M. Akil: MT a multithreaded 64 bits RISC CPU. Proceedings of PaCT-93, Obninsk 1993
6. B. Goossens and D.T. Vu: Data slicing arithmetic operators. Real numbers and computers, St. Etienne, april 1995
7. R.H. Halstead and T. Fujita: MASA, a multithreaded processor architecture for parallel symbolic computing. Proceedings of the 15th AISCA, 1988
8. J.L. Hennessy, N. Jouppi, F. Baskett, A. Strong, T. Gross, C. Rewen and J. Gill: The MIPS machine. Proceedings of COMPCON, feb. 1982
9. J.L. Hennessy and D.A. Patterson: Computer architecture. A quantitative approach. Morgan Kaufmann, 1990
10. N.P. Jouppi, D.W. Wall: Available instruction level parallelism for superscalar and superpipelined machines. Proceedings of the 3rd ASPLOS, april 1989
11. G. Kane: MIPS R2000 RISC architecture. Prentice Hall, 1986
12. N. Namjero, A. Agarwal: Implementing SPARC: A high performance 32 bits RISC microprocessor. SUN microsystems, 1988
13. D.A. Patterson, C.H. Sequin: RISC-I: a reduced instruction set VLSI computer. Proceedings of the 8th AISCA, 1981
14. B.J. Smith: Architecture and applications of the HEP multiprocessor computer system. SPIE proceedings, 1981