A High Out-of-Order Issue
Symmetric Superpipeline Superscalar Microprocessor

Stéphan Jourdan, Dominique Carrière, Daniel Litaize

Institut de Recherche en Informatique de Toulouse
118 route de Narbonne, 31062 Toulouse cedex, FRANCE

Abstract

Due to technology's evolution, the number of transistors that can be integrated in a same chip has become, at the dawn of the 21st century, more than sufficient to implement simple superscalar cores. This excess, nowadays generally used for on-chip caches, can also be utilized to improve core's performances, but mainly to increase the core's superscalarness degree. Although it now seems that a high degree is not justified, it could become useful in the future with progress in compilation.

Setting out from this observation, we describe in this paper a new superscalar architecture with a high out-of-order issue rate. This architecture implements, in particular, precise interrupt management and multiple branch prediction. Furthermore, the architecture's specification has taken into account the aspect of hardware implementation, and thus, temporal matching of pipeline's stages. We therefore assist to a finer partitioning of this pipeline, hence the additional superpipeline label.

1. Introduction

Nowadays, for the most part, two trends summarise processors architecture objectives: to move the clock rates from low to higher speeds, and to lower the mean cycle per instruction (CPI) rates [1]. They now have respectively led to superpipeline and superscalar notions. During the first years following the birth of microprocessors, those, known as CISC, acronym for *Complex Instruction-Set Computer*, had a high CPI. Indeed, an instruction often required more than 8 clock cycles to be executed [2]. The main solution to lower the CPI rates, best exploited in RISC, acronym for *Reduced Instruction-Set Computer*, has consisted in overlapping the different instruction's executing steps, using a technique called pipeline [3]. Pipelining exploits parallelism among the instructions in a sequential instruction stream. However, this technique suffers from what are commonly called dependencies, which degrade the optimum CPI rate of 1 [4]. Many improvements have nearly led to such a CPI rate. Successful mechanisms are: by-path of operands during the execute stage, Harvard architecture and delayed branching [5]. The next step in microprocessor's evolution has consisted in going beyond this limit of 1 CPI. The unit has hence become the IPC, i.e. the mean number of instruction per cycle. Such microprocessors have been called *superscalar* [6]. On the other hand, those dependencies lead to a higher degradation of the IPC because of the larger parallelism involved. Consequently, they also limit the issue rate. Objectives are therefore to reduce such degradation. Resource dependencies can be solved through the use of several functional units (FUs). Such an architecture, also based on Tomasulo's algorithm [7], minimizes data dependencies when no control dependency occurs. In such a case, with reservation stations (RSs) associated to each functional unit, the microprocessor implements an out-of-order issue policy. It then acts as a data-flow machine. An infinite RS number is required for such a treatment, but simulations have shown that it can be restricted without many penalties [8]. This kind of microprocessor, however, manages imprecise interrupts. Adding a reorder buffer allows for precise and fast interrupt management [9]. The problem hence lies in control dependencies. If the microprocessor stops issuing when a conditional branch is decoded, the effective IPC will be low. Branch prediction [8] is a method that improves IPC, depending on the authorized number of branches predicted simultaneously. A reorder buffer, here, allows such an implementation, as shown in section 5. Another important notion with control dependencies is branch delay, i.e. the time required to fetch the branch's targeted instructions to be decoded. The use of a branch cache, such as in the MC88110 [10], leads to a 0-cycle branch delay, as described in section 3. Hence an architecture featuring a reorder buffer and a branch cache, executes the instructions without stalls as if almost no control dependency occurred.

We describe in this paper, an architecture based on principles previously outlined. This architecture performs a high out-of-order issue rate, labelled N. The sections deal with a microprocessor based on such an architecture, called S3M, acronym for *Symmetric Superpipeline Superscalar Microprocessor*.

The outline of this paper is as follows. Section 2 presents the S3M's internal architecture. Section 3 describes instructions fetching. Section 4 recalls the operands forwarding. Multiple branch prediction is introduced in section 5, whilst the superpipeline aspect is dealt with in section 6. Section 7 presents the modelling, and results from simulation are discussed in section 8. Conclusions are summarized in section 9.

2.Presentation of S3M's internal architecture

S3M's internal architecture, illustrated in figure 2-1, is similar to that of Motorola's MC88110 [10], but for future files and a reorder buffer. Reservation stations are associated to each functional units, and the processor therefore implements an out-of-order issue policy. Each file is in fact a combination of a future file and a register file. Some instances are generic, integer, floating-point, graphic or status. Each being accessed by one or several FUs. Such an organization is similar to the MC88110's *Special Function Units* (SFUs). The reorder buffer's invalidation policy is total. Despite it diminishes the microprocessor's global performance, it offers however the possibility to easily implement a multiple branch prediction. In section 3 to 5, the pipeline used is simple, as shown in table 2-1. It corresponds to a basic partitioning of the processor's functionality, without caring for the temporal aspect of the implemented algorithms. Section 6 takes this into account, and leads to a finer partitioning.

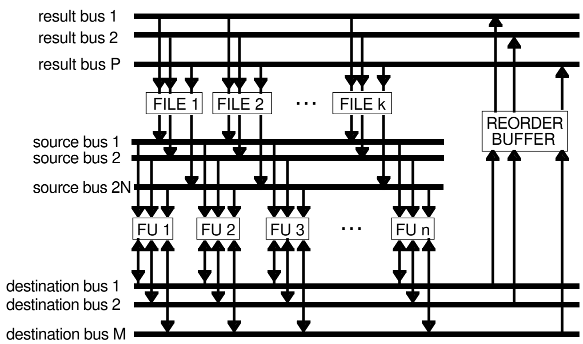

Figure 2-1. Data Path

Stage	Functionality
FE	(**FEtch**) Accesses caches and chooses the next N decoded instructions.
DE	(**DEcode**) Determines the number of issues, decodes and reads operands, and places instructions on source buses.
EX	(**EXecute**) Reads source buses and destination buses, arbitrate units for concurring RSs. Executes instructionsand then arbitrates the destination buses.
WB	(**Write Back**) Writes the results in future files and in the reorder buffer.
RC	(**Result Commit**) Dequeues the reorder buffer and updates register files via result buses.

Table 2-1. The pipeline stages

Basic parameters of such an architecture also concerns the number of common buses : source(2N), destination(M) and result(P). N corresponds to the maximum number of instructions issued per cycle. M usually equals P, otherwise, the smallest indicates the maximum number of instructions completed per cycle.

3. Fetching instructions

3.1 Instruction cache access

The fetch stage has to supply when possible the decoder with N valid instructions. Access to the instruction cache in a superscalar architecture, is problematic due to the processor's maximum number of issues (N). By means of simplification, we will not consider here any branch instructions, discussed in the next sub-section. The issuing of an instruction by the decoder can be omitted if by example there is no available reservation stations. Hence, the number of instructions issued is dynamic, and is only available during the decode stage, never at the beginning of the cycle. Therefore, the address of the last decoded instruction, that is the one preceding the first decoded instruction of the next cycle, remains unknown at the beginning of this cycle. It depends on the number of instruction issued by the decoder. The cache access is therefore delayed until the address is known.

One solution, implemented in S3M, is that the fetch stage memorises the N instructions supplied to the decoder during the previous cycle, and accesses to the cache for the following N instructions. Thus, whatever the number of instructions issued, N valid instructions from the 2*N considered will be supplied to be decoded in the next cycle, as shown in figure 3-1, for N=4.

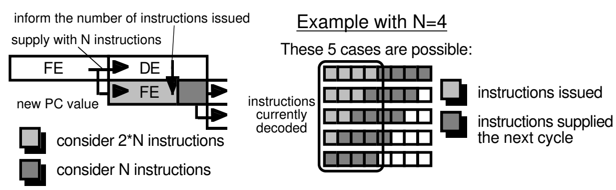

Figure 3-1. Fetch and Decode Stages

Another problem lies in the instruction's alignment with cache blocks. When a superscalar microprocessor is used, the fetch stage needs to access consecutive blocks. If B is the number of instructions in a block, then the worst case number of blocks accessed is given by the equation 3-1.

$$\begin{cases} 1+\left\lfloor \frac{N-1}{B} \right\rfloor & \text{if } \operatorname{mod}\left(\frac{N-1}{B}\right)=0 \\ 2+\left\lfloor \frac{N-1}{B} \right\rfloor & \text{if } \operatorname{mod}\left(\frac{N-1}{B}\right)\neq 0 \end{cases}$$

Equation 3-1. Number of Cache Blocks Acceded

3.2 Unconditional branches

If there is a branch, the decoder does not issue following instructions, and waits for the fetch stage to supply the targeted instructions. It is important here to notice that the number of stages preceding the decode stage cannot increase as each additional stage would lead to an increase of instructions to be considered, hence, an increase of the possible number of branches. However, *delayed branching* cannot be efficient in a superscalar architecture when N is important [8] since it is difficult to fill the N slots. When N=1, the slot can only be filled 70% of the time [11]. A better mechanism consists in the use of a branch cache enabling, in case of a hit, a 0-cycle branch delay. The branch cache is indexed with the same address than that of the instruction cache. A hit is searched for the N addresses starting from the given one. With each valid branch cache entry is associated, along with the index branch instruction address, the first N instructions starting from the address specified by this branch. One must remember here that the branch instruction involved is being decoded while the cache is accessed. At the fetch stage level, the 2*N instructions coming from the instruction cache, as well as the N instructions of the branch cache in case of a hit, are considered. The fetch stage then needs to wait for the number of issued instructions from the decoder. Indeed, a branch instruction that is detected in the branch cache can be omitted if by example, the preceding instruction is not issued. It can then choose one of the sequences, and supply the decoder with N instructions for the following cycle. On the other hand, the decoder does not issue any instruction after a branch one. Refer to figure 3-2 for summary.

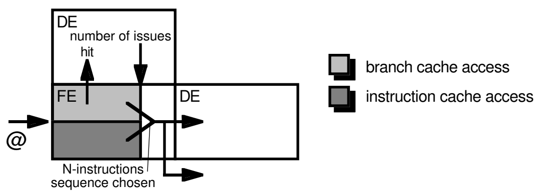

Figure 3-2. The Fetch Stage

3.3 Branch prediction

Furthermore, the branch cache can also be used for conditional branches: it can contain informations about past execution of these branches so as to carry out dynamic predictions. This leads to the implementation of multiple prediction algorithms, from the N-bits counter [12] to the Adaptive Scheme [13]. A survey of these algorithms is presented in [14]. A choice however still needs to be made concerning default prediction:

- First, it is possible to use any of the static prediction algorithms. It is to be noted that a branch predicted taken will lead to a lost cycle due to the miss in the branch cache.
- Then, the choice for the sequence can be defined dynamically through correlation of the predictions brought into effect for the last issued K branches.
- Finally, through static-dynamic association, e.g. correlation along with annotation.

To sum up, the branch cache described here performs a 0-cycle branch delay in case of a hit, as well as contributing to dynamic prediction of conditional branches. It has now become commonly used in recent microprocessors, e.g. the MC88110's Target Instruction Cache (TIC) [10]. Similarities with the PPC601's instruction buffer can also be noticed [15]. In fact in our solution, instructions which are not issued, are stored in the fetch unit, in spite of being stored in a specific buffer.

4. Operands forwarding

4.1 Introduction

S3M, as previously described, implements out-of-order initiation/out-of-order completion policies, and a precise interrupts management. This is enabled thanks to the use of reservation stations and a reorder buffer. Subsequent sub-sections, *states* and *buses*, describe the ways by which operands are provided and forwarded.

4.2 States

Such an architecture implements a kind of *data-driven* mechanism to achieve *data-flow* computations. But unlike Tomasulo's algorithm, it allows for precise interrupt management through the use of a reorder buffer. As described in [8] [16], register files maintain the *in-order state*, used for exception recovery. Future files maintain the *out-of-order state*, required for instruction issuing (register accesses). The reorder buffer contains informations on all issued instructions, starting from the oldest uncompleted one to the last decoded one. Register files are updated by dequeuing the buffer when possible. So as to manage precise interrupts, exceptions are only treated when the entry corresponding to the faulting instruction is dequeued from the reorder buffer.

4.3 Buses

Source buses transport the issued instruction's operands to provide reservation stations. These contain either a value or a station number for pending values. A control bus containing the target reservation station number, a functional unit treatment number and the destination register number, is associated to a pair of source buses.

Destination buses are used by functional units to place their results. An arbitration is required because of disproportion between the number of buses and the number of units. These buses transport the following information which will be subsequently considered by the reorder buffer: a destination register number, a result value, a produced exception number and the used reservation station number to compute the result, required for forwarding.

Result buses provide register files with completed instruction results, placed by the reorder buffer. The different fields consist of: the register file number, the number of the register to be updated and the value.

4.4 reservation stations

Due to the presence of several RSs per functional unit, an arbitration, based on the instruction issuing order, is needed to issue the instruction's execution. As shown in the following section, the reorder buffer maintains this order. To each RS is hence associated a field which indicates the placement of the corresponding entry in the buffer.

Moreover, if the issue rate is high, an old instruction can become blocked in a RS. Indeed, if an instruction ahead of a decoded instruction group cannot be executed because of dependencies, and another more recent one issued in the same FU is executed, it is required to wait for this last one to obtain a destination bus so that the first one be executed. This wait can become important if many instructions are fetched between the two. Here, the reorder buffer hence stops the dequeuing, and following instructions cannot go into the RC stage : there is cycle losses. An example is the instruction group described in figure 4-1. In that case, if the second and the last add are issued in the same FU, the last one keeps the FU occupied as long as its results cannot be placed on result buses : the second add cannot be executed, and then the instructions between the two adds cannot go into the RC stage. A solution lies in structuring FUs so as to render them non blocking through the adjunction of an additional field result to each RS. The arbitration of destination buses is then executed on these fields.

instruction group { add r4, r2,r0
add r6,r4,#10

add r12,r10,r0 }

Figure 4-1. An instruction group

5. Multiple branch prediction

5.1 Reorder buffer

The reorder buffer is described in [8] [10]. In summary, it is managed as a FIFO queue. Instruction decoding can proceed as long as entries are available. With the issue of one instruction is associated one entry at the top of the queue to receive its execution result. When an entry reaches the bottom of the buffer, it is dequeued if the corresponding instruction result has been notified. This result will then modify the register file the next cycle when in the RC stage. Hence the reorder buffer maintains the *look-ahead state*, i.e. items about all instructions starting from the oldest uncompleted to the last decoded. In fact, the reorder buffer orders the modification of register files through the out-of-order issued instruction execution. It therefore establishes an order relation for their issuing, that is for all instructions being treated. This property has been previously utilized in section 4. The reorder buffer's most important parameter is its number of entries: it measures the level of anticipation that characterises the processor.

5.2 Precise interrupt management

With an architecture implementing out-of-order completion policy, two ways of managing interrupts are possible: imprecise or precise. With imprecise management, the handler cannot consider the microprocessor's correct state at the time of the interrupt, except in the event of a possible rebuilding. However, the interrupt recovery process is similar to the one for bad prediction. Therefore, a branch prediction implementation enables a precise interrupt management for free. The reorder buffer allows for this. Thus, S3M accomplishes this precise management. Concerning the reorder buffer, chosen policy is *total invalidation*. The exceptions are not considered immediately, but only when the corresponding entry has to be dequeued. The reorder buffer, the future files and all RSs are then entirely discarded. Store instructions are treated by only issuing them when their entry is dequeued. On the other hand, external interrupts are dealt with immediately. Execution will resume with the first uncompleted instruction after resources are discarded.

Precise interrupt management does not require any additional information in the reorder buffer. Basic fields are the same as in destination buses. Moreover, instead of an additional field to store the instruction address for each entry, a local PC can be utilized to resume program execution when interrupts occur. Except for jumps, it is increased each time an entry is dequeued. In that case, the branch target address is assigned to the PC. This address can be stored in the value field of the buffer entry, since this field is not required for a jump instruction.

5.3 Multiple branch prediction

So as to check predictions made during the issuing of branch prediction, a branch FU is required. When such an instruction is issued, the decoder sends to this unit the operands as well as an indication on the prediction made. This can be done through the modification of the op-code of the instruction (treatment number). The RS number of this unit then indicates the number of branches that can be simultaneously checked. In fact, as previously shown, bad prediction recovery is identical to that of interrupts. Thus, the result produced by the branch unit is only an interrupt: no interrupt in the case of correct prediction, else an interrupt labelled *incorrect prediction*. Furthermore, the recovery process requires in the reorder buffer a new information, the *not-predicted* address. It is to be noticed that one of the two addresses still is the sequence. Hence, similarly to the jump instruction, the value field always contains the branch target's address. An extra bit is still required to indicate whether the prediction as been the sequence or the target. Figure 5-1 summarises the recovery process in case of bad prediction.

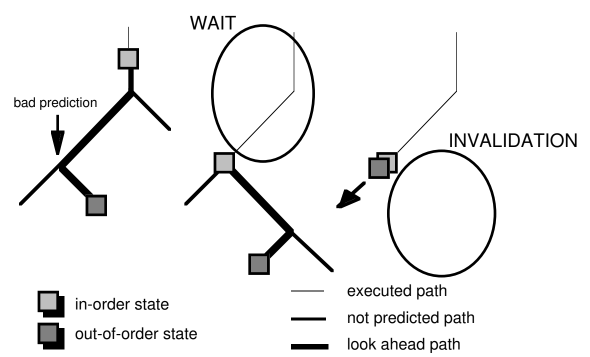

Figure 5-1. Recovery from Mispredicted Branch

6. The superpipeline aspect

The superpipeline notion first consisted in the partitioning of each pipeline's stages in m independent sub-stages, but has now derived to all processors featuring a deep pipeline. The temporal dismatching of the S3M pipeline, mainly comes from 3 stages, we are hence going to split them up :

- The FE stage first, because the cache access usually constitutes a stage by itself, besides, it often is the slowest one. It then sets the clock frequency.
- Then the DE stage, that must determine the number of issued instructions before decoding them.
- Last, the EX stage, which operates 2 arbitrations on top of the execution.

First concerning the FE and DE stages, it is important to minimize as indicated in section 3, the number of stages between the one operating the cache access and the one computing the number of issued instructions: none when possible. Furthermore, the choice of the instruction group cannot be made outside of the FE stage, as it must be fetched in the instruction cache the following cycle. It is therefore not possible to split up this stage. The DE stage, however, can be divided meaning a new stage called NI. Table 6-1 shows the functionalities of each of the 3 stages.

Stage	Functionality
FE	(**FEtch**) Accesses to the caches and chooses the next N decoded instructions.
NI	(**Number of Issues**) Determines the number of issues.
DE	(**DEcode**) Reads operands, inserts in the buffer the corresponding entries. Places instructions on source buses.
RA	(**Read and Arbitrate**) Reads source and destination buses to modify RSs. Arbitrates available FUs between the different RSs.
EX	(**EXecute**) Executes instructions and places their results in the RSs. Arbitrates to obtain the destination buses.
WB	(**Write Back**) Writes the results in future files and in the reorder buffer.
RC	(**Result Commit**) Dequeues the reorder buffer and updates register files via result buses.

Table 6-1. The pipeline stages

Concerning the EX stage, the problem lies in the fact that it is not possible to split it up in m stages without operand forwarding for each. We hence divide it in just two substages, forwarding of the second stage being still taken care of by the destination buses. There is no forwarding between the two. One can notice that the execution and the arbitration on destination buses can be parallelized in anticipation of the arbitration. We therefore regroup them within the same stage: the new EX stage. Reading buses and the first arbitration hence constitutes a single stage labelled RA.

7. Experiments

The results presented in this paper are for nine loops from the Livermore Loops (1 3 5 7 9 10 11 12 and 24), which are FORTRAN loops from production codes run at Lawrence Livermore National Laboratory, and they represent the type of computation kernel typically found in large-scale scientific computing [17]. These loops were translated into C, and then were compiled for the M88000 instruction set, using the `gcc` 2.4.5 compiler with all optimizations turned off.

FU type	Latency	Instructions	Files
Integer	1	INT add and logic OPs	I
FP Add	3/1*	FP add and comparison*	FP/I
Multiply	3	INT and FP multiplication	FP/I
Divide	18/26	INT and FP division	FP/I
Load / Store	1	Memory accesses	FP/I
Branch	1	Prediction checking	I
Bit Field	1	Shift, and bit testing	I

Table 7-1. FU Types and Latencies

Table 7-1 shows functional unit types and the latencies of instructions belonging to these types. Each type is listed with a description of the instructions that belong to it, and the register file on which it accesses. The selected latencies are the MC88110 ones [10]. Each FU is fully pipelined and mutually independent, except the divide FU.

The simulated machine FU configuration is listed in table 7-2. The Instruction and Data caches are assumed to be infinite, while the Branch cache features 32 fully-associative entries. The reorder buffer size is variable.

FU type	Number	RS per unit
Integer	var.	var.
FP Add	1	4
Multiply	1	4
Divide	1	4
Load/Store	1	var.
Branch	1	4
Bit Field	1	4

Table 7-2. Functional Unit Configuration

The modelling of S3M has laid to a simulator which is source-driven. The simulator was written in C, and it simulates all parts of the microprocessor. The simulation process works as follows : First of all, benchmarks have to be compiled into assembly code using `gcc`. Our S3M simulator then reads in a configuration file which describes the internal architecture to be simulated and begins processing the assembly code instruction stream produced by `gcc`. The simulator performs a cycle by cycle simulation, in the same manner as a real microprocessor does. Our simulator makes no simplifying assumption except the infinite size of the data and instruction caches : they are not modelled. There are several key parameters that define the internal architecture being simulated :

- The superscalarness degree
- Number of destination and result buses
- The reorder buffer size
- FUs configuration and the number of associated RSs

8. Simulation Results

8.1 Introduction

For a given processor configuration, the measurements are presented as the average of the nine benchmarks computed using the harmonic mean. Therefore, each with a performance value of K_i, the harmonic mean of all values is given by the equation 8-1.

$$\text{harmonic mean} = 9 \times \left(\sum_{i=1}^{9} \frac{1}{K_i} \right)^{-1}$$

Equation 8-1. Harmonic mean

The use of the harmonic mean lowers results, but it reflects more accurately an hypothetical reality. Therefore, the IPC of the best S3M configuration is 2.88 (with the harmonic mean), while computed with the arithmetic mean, the IPC would be 3.19. Table 8-1 lists the best IPC value for each of the nine benchmarks.

Name	IPC	Name	IPC	Name	IPC
loop 1	3.943	*loop 7*	3.736	*loop 11*	2.051
loop 3	4.57	*loop 9*	3.347	*loop 12*	2.261
loop 5	2.613	*loop 10*	1.884	*loop 24*	4.373

Table 8-1. Best IPC

Our studies have been performed accordingly with 4 axis :

- The reorder buffer size
- The number of destination buses
- The number of RSs associated to the Load/Store FU
- The number of integer FUs.

Lastly, each of the figures plots performances for a superscalarness degree through 1 to 16.

8.2 Analysis

The reorder buffer

Figures 8-1 plots the performance, given in IPC, for a reorder buffer size varying from 1 to 128. The number of integer FUs is fixed to 4, with each 4 RSs. 16 Load RSs and 16 Store RSs are associated to the Load/Store FU.

First of all, one can see that the best IPC peaks at 2.88. This seems to be the maximum instruction-level parallelism we can reach for these benchmarks with no optimization. The best of them is *loop 3* with a 4.57 IPC, while *loop 10* is the worst with a 1.88 IPC. Furthermore, there is little improvement with a superscalarness degree higher than 4, and no one with any higher than 8. This proves the limitation of the current compiling processes, which is not adequate for producing better instruction-level parallelism. In the same manner, due to this limitation, there is no improvement in performance with a reorder buffer size higher than 64. One must remember here that the reorder buffer holds all kind of instructions which have not modify register files yet. Thus, this size characterises the out-of-order level of anticipation. Therefore, it seems that a reorder buffer of 64 entries gives the best results with a 4-issues S3M. Lowering the size below 64 affects perceptibly the IPC. A 32 entries reorder buffer is a good compromise with a 2-issues S3M. Figure 8-2 and 8-3 show the average utilization of the reorder buffer capacity. The best ratio is 32 entries for a 4-issues S3M, but 64 entries gives a good average fill rate of 73%.

There is two other important things to note in figure 8-3.

First, the curve rapidly decreases when the size is higher than 32. Indeed, the issuing depends on two factors : the reorder buffer size and the number of RSs. The issuing limitation comes from this last factor when the size grows, and then lowers the reorder buffer capacity utilized. However, increasing this number of RSs does not improve the processor performance, in IPC, because this performance is limited by the poor instruction-level parallelism.

Last, the utilized capacity decreases when the size is below 32, while one can wait for a nearly 100% use with a reorder buffer size of 1. In fact, when an instruction is issued (stage NI), the corresponding entry in the buffer is not allocated (it will be in the DE stage). Between two successive instructions, the buffer is therefore empty during one cycle : in general, it represents one cycle out of 5, i.e. 80%. Indeed, the ratio is lower due to invalidations.

Destination buses

Figures 8-4 through 8-6 give the same results as figures 8-1 through 8-3 but in function of the number of destination buses instead of the reorder buffer number of entries. FU configuration is the same than in the previous sub-section, except for the reorder buffer size which was fixed to 64. Figure 8-4 shows same IPC results when the number of buses is higher than 7. Therefore, seven destination buses seems to be the minimum number for a performing operand forwarding. A loss of 2.5% is made when the buses number lowers to 5, and another 2% to 4, which seems to be the optimum number for a 4(or more)-issues S3M. Therefore, number of buses higher than 4 gives slightly better results, due to the register writing back serialization in the reorder buffer. One can consider here the implementation of only 3 destination buses, but at the cost of a 10% loss. In term of fraction of bus capacity utilized, the use of 4 buses gives a 64% result while 3 buses 79%.

Load/Store Reservation Stations

Figure 8-7 shows the importance of the number of RS in the Load/Store Unit. As a matter of fact, it affects for a great part, the S3M capacity of anticipation by blocking the issuing. A number of RSs greater than 16 shows no benefit, and there is only a 6% loss with 8 RSs for each Load and Store instructions.

Integer Functional Units

Figure 8-8 plots the performance for a number of integer FUs varying from 8 to 1. Integer FUs does not affect performance as much as Load/Store RSs, but there is a 19% loss in performance between 4 and 2 integer FUs.

9. Conclusions

In this paper, we have ended at the specification and the modelling of a superscalar and superpipeline monoprocessor architecture, featuring a high out-of-order issue rate. This elaboration has been developed in order to make the IPC as close as the superscalarness degree. This work has also taken into account temporal aspects of each of the algorithms characterizing the pipeline stages. Moreover, through out the paper, we have highlighted the difficulties encountered with a high issue rate implementation. The proposed solutions have led to the utilization of:

- a branch cache performing a 0-cycle branch delay as well as the possibility of implementing any dynamic prediction algorithms.
- reservation stations, especially in the FU's output, in order to implement out-of-order issue/out-of-order completion policies.
- a reorder buffer which has a variety of uses. It allows precise interrupt management, multiple branch prediction and it supplies a strict order relation required to the different arbitrations.

All along the study which lead to this document, we have designed and adapted a simulator of this architecture, which can be programmed for different configurations. Experimentations on this tool have guided us to certain choices as for given solutions, e.g. the new RS's result field, problem which has been raised for an N=16 configuration. Subsequently, this problem also lies whatever the issue rate.

Nevertheless, the limits concerning IPC which was shown by the simulator on large-scale scientific computing, indicates the unsuitability of recent compilators. At the present time, a 4-issues superscalar microprocessor featuring a 64-entries reorder buffer is sufficient to take into account the instruction-level parallelism produced. Similar remarks was made in [8] concerning general-purpose applications.

This study has been conducted jointly to the *serially multiported shared-memory multiprocessor* project labelled M3S [18]. Its aim is to characterize an utilization's possibility of a larger processor-memory bandwidth at the microprocessor level. We have therefore noticed this possibility in the utilization of the anticipation mechanism supported by multiple branch prediction, this with a high issue rate.

10. Acknowledgements

We like to thank P. Magnaud, A. Mzoughi, P. Sainrat and C. Rochange, for their contribution to our research work as well as F. Vialet for his help in drawing up this paper.

11. References

[1] ***Hwang, K.*** Advanced Computer Architecture: Parallelism, Scalability, Programmability. *Mac Graw-Hill, 1993*

[2] ***Motorola Semiconductors.*** Advance Informa-tion. *ADI-814R2, Motorola Inc., 1981*

[3] ***Kogge, P. M.*** The Architecture of Pipelined Computers. *Mac Graw-Hill, 1981*

[4] ***Stone, H. S.*** High-Performance Computer Architecture. *Addison-Wesley, 1990*

[5] ***Hennessy, J. L.*** *and* ***Patterson, D. A.*** Computer Architecture A Quantitative Approach. *Morgan Kaufmann Publishers, 1990*

[6] ***Agerwala, T.*** *and* ***Cocke, J.*** High Performance Reduced Instruction Set Processors. *Technical Report RC12434 (#55845). Yorktown, NY: IBM Thomas J. Watson Research Center, 1987*

[7] ***Tomasulo, R. M.*** An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal, Vol. 11, 1967*

[8] ***Johnson, M.*** Superscalar Microprocessor Design. *Prentice-Hall, 1991*

[9] ***Smith, J. E.*** *and* ***Pletzkun, A. R.*** Implementation of Precise Interrupts in Pipelined Processors. *Proceedings of the 12th Annual International Symposium on Computer Architecture, 1985*

[10] ***Motorola.*** MC88110 Second Generation RISC Microprocessor User's Manual. *Motorola Inc., 1991*

[11] ***McFarling, S.*** *and* ***Hennessy, J. L.*** Reducing the Cost of Branches. *Proceedings of the 13th Annual International Symposium on Computer Architecture, 1986*

[12] ***Smith, J. E.*** A Study of Branch Prediction Strategies. *Proceedings of the 8th Annual International Symposium on Computer Architecture, 1981*

[13] ***Yeh, T.*** *and* ***Patt, Y. N.*** A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. *Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993*

[14] ***Stjernfeldt, C. O., Czeck, E. W.*** *and* ***Kaeli, D. R.*** Survey of Branch Prediction Strategies. *Technical Report CE-TR-93-05, 1993*

[15] ***IBM***, POWER, POWER PC, *IBM F 465, 1993*

[16] ***Jourdan, S.*** Les Microprocesseurs Super-scalaires : Prospectives, *DEA report, 1993*

[17] ***Feo, J.T.*** An Analysis of the computational and parallel complexity of the Livermore Loops, *Parallel Computing 7, 1988*

[18] ***Litaize, D., Mzoughi, A., Rochange, C. and Sainrat, P.*** Towards a Shared-Memory Massively Parallel Multiprocessor. *Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992*