I R I S A

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE

N° 763

OUT-OF-ORDER EXECUTION OF INTERRUPTIBLE CODES

YVON JÉGOU

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 38 38 32

Out-of-Order Execution of Interruptible Codes

Yvon Jégou

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet CALCPAR

Publication interne n ° 763 — Novembre 1993 — 19 pages

Abstract: The superscalar execution model extracts independent instructions from a restricted window. When pipeline latencies go beyond some limit, out-of-order execution becomes necessary to fully exploit the independency of instructions. On the other hand, multithreading merges the execution of independent instruction flows. The simultaneous use of these two techniques is expected to produce a high degree of concurrent activities in future processors. But, the codes must be interruptible and restartable. Classical program state construction is incompatible with out-of-order of execution. In this paper, we present a new processor architecture which basic execution model is out-of-order execution. In this new model, the state of some execution does not correspond to a possible sequential state any more: it represents the operations that remain to be executed. When the instruction flow is interrupted, a code sequence which contains the instructions that have been issued but which execution cannot be completed is built. On restart, the execution of this sequence restores the initial state.
Key-words: superscalar, out-of-order, multithread, pipeline, interruptible, multiple functional units

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Exécution dans le désordre de codes interruptibles

Résumé : Le modèle d'exécution super-scalaire permet d'extraire et d'exécuter des instructions indépendantes d'une fenêtre de largeur limitée. Ce modèle préserve l'ordre des modifications de l'état de la machine. Mais lorsque les longueurs des pipelines dépassent certaines limites, seule une exécution dans le désordre permet d'obtenir des performances satisfaisantes. D'autre part, le séquencement multi-flots –*multithreading*– augmente le parallélisme potentiel en combinant l'exécution de plusieurs flots d'instructions indépendants. L'utilisation combinée du multi-flots et de l'exécution dans le désordre devrait produire un parallélisme élevé dans les processeurs. Mais, les codes doivent pouvoir être interrompus puis relancés. Les techniques de sauvegarde d'état classiques sont incompatibles avec l'exécution dans le désordre. Nous présentons une architecture nouvelle de processeur dont le modèle d'exécution de base est multi-flot et dans le désordre. Dans ce nouveau modèle, un état d'exécution ne correspond pas nécessairement à un état séquentiel. Lorsque l'exécution d'un flot d'instructions est interrompu, le processeur construit automatiquement une séquence qui contient l'ensemble des instructions séquencées avant l'interruption mais dont l'exécution n'a pas abouti. L'exécution de cette séquence permet de relancer le code.
Mots-clé : pipeline, superscalaire, multi-flots, exécution dans le désordre, interruption, unités fonctionnelles multiples

1 Introduction

Instruction level parallelism is exploited on superscalar processors by simultaneously issuing groups of independent instructions. When applied at compile time, static scheduling tries to produce such sequences of independent instructions. However, many factors limit the real parallelism that is effectively found at run time. First of all, the execution time of all the instructions cannot be predicted. This is the case for the accesses to the memories which depend strongly on the caches behavior. Even when a cache miss does not stall the whole processor, the instruction sequencing is stopped as soon as the result of the cache access is needed. In order to overcome this limitation, dynamic scheduling techniques have been proposed [1]. These techniques detect the dependencies between instructions of an execution window during each cycle. The ready instructions are issued. In general, dynamic scheduling produces out-of-order execution of the instructions and the implementation of precise interrupts becomes a problem.

Another limitation of superscalar issue comes from the limited size of the execution window. Medium grain of parallelism — from independent basic blocs — is difficult to exploit because independant instructions are too distant in order to meet in the execution window. Moreover, the reuse of register names introduces false dependencies.

Medium grain parallelism might be exploited efficiently by multithreaded processors [3], [2]. Instructions from different threads are generally independent and should be candidate for parallel execution. But multithreading introduces new problems which must be solved. First of all, the interferences in the resource namings have to be avoided: e.g. two iterations of a single loop produce the same register traces. Some of the machine registers, for instance, should be shared between the threads, while other registers should be local to each thread execution. Moreover, the access to the register file might become the major bottleneck.

Multhreading improves processor resource usage in the presence of long latencies in instruction execution. The O3 (Out-of-Order) architecture we introduce in this paper is defined with multithreading in mind. The most important problem to solve is that the introduction of out-of-order execution of a mix of instructions seems necessary to reach some level of performance [1]. This paper concentrates on the management of out-of-order execution and proposes a new mechanism to deal with the lack of a simple program state. Section 2 presents a queue-based virtual processor and its instruction set. Section 3 gives the bases of a possible implementation of this processor with parallel out-of-order execution of the instructions. Then, the treatment of thread interruption and restart is described in section 4. The last section introduces multithreading.

2 The Qone virtual processor

For the compiler point of view, the visible part of a processor is mainly its instruction set architecture (ISA). This instruction set defines a virtual processor and hides many aspects of the implementation. For instance, the structure of the functional units can be hidden. The mapping of the virtual processor on the real one is produced by the instruction dispatcher.

The execution model of the Qone processor is based on the use of a data queue. In general the instructions of this processor extract their operands from the head of the data queue and append the results to the tail of this queue. In our examples, the operands and results of all instructions are implicit except for constant load and register (local memory) accesses. The long constant values are stored in the word following the referencing load in memory. For instance, an `add` instruction extracts two values from the head of the data queue and append the result to the tail of the queue. This instruction set is similar to a classical stack based instruction set. The main difference is in the order of evaluation in the expressions. The use of a stack exhibits locality: when evaluating an expression, the sub-expressions are first evaluated one after the other. On a queue-based architecture, the evaluations of all sub-expressions are merged and in general, the distance between the production of a value and its use is increased.

The sequencing of the Qone processor is also queue based and is defined on basic blocks. An instruction basic block is defined by the address of the first instruction in memory. The last instruction of a basic block is tagged. This last instruction needs not be a control sequence instruction. The order of the basic blocks sequencing is defined by a block queue: when the last instruction of a basic block is sequenced, the address of the next instruction is extracted from the basic block queue. The sequence control instructions append block addresses to this queue. In general, each basic block contains a sequencing instruction which computes the address of the next block to be executed.

The architecture of a simple Qone processor is shown in figure 1.

Each instruction of a Qone code belongs to a sequencing group. A sequencing group is defined as a set of contiguous and data independent instructions which consume the results produced by the previous group and produce the operands for the instructions of the next group. There is no data dependency inside a sequencing group. The group separation is tagged in the instruction sequence. These tags are also set in the data queue, and help in the early detection of incoherence in the use of the data queue. Each instruction of the processor is store in a sixteen bits word. Fourteen bits allow the coding of parameterless instructions, of local memory accesses (short addresses) and of the constant loads. The two remaining bits are used for the end of basic block tag and for the sequencing group tag.

The main advantage of queue instructions over stack instructions is that contiguous operations are generally independent. This is always true for all instructions from the same sequencing group. This local independency favors

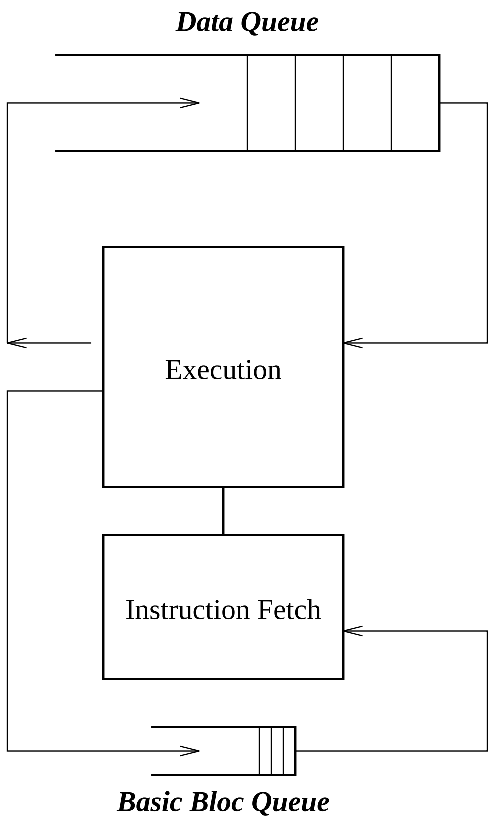

Figure 1: Architecture of Qone

superscalar type of execution. Stack codes do not present such a behavior. Another advantage of a queue based instruction set over a stack based one is that it is possible to interleave the execution of two or more code sequences on the same processor. Such an interleaving is produced at the sequencing group level. The consequence of this merge is larger sequencing groups, and thus more independent instructions.

Figure 2 shows the instructions generated from expression `a := b + c*10`. Variables `a`, `b` and `c` are located in main memory at absolute addresses `@a`, `@b` and `@c`. Constant 10 is loaded by a long constant load. Instruction `c32` is a long constant load, `arM` an absolute address memory read, `awM` an absolute address memory write. In this example, `/` separates instructions words from the constants.

```
c32,    ,     / @c
c32, arM, c32/ @b / 10
arM, mul,
c32, add,     / @a
awM
```

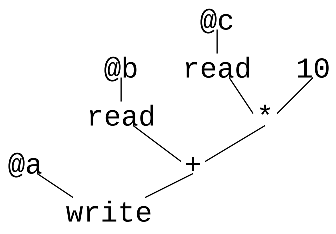

Figure 2: Instructions generated by expression `a := b + c*10`

3 Implementation of Qone architecture: the O3 processor

The introduction of a physical data queue in a real architecture would not be realistic as it would become the bottleneck in the data paths. In the parallel implementation O3, data flow directly between the functional units. Multiple internal data paths can then be used and the results can be consumed as soon as they are available. The data queue of the architecture is replaced by a labelling queue inside the instruction dispatcher and is used for the computation of the result destinations of instructions.

3.1 The O3 processor architecture

An O3 processor architecture contains one (or more) instruction dispatcher and functional units. These functional units exchange data directly. Each functional unit is specialized: the memory FU is in charge of the memory accesses and is responsible of the cache management, the register FU gives fast and compact access to local memories. The sequencing unit executes all the control sequence instructions. This unit is not responsible for dispatching the instructions on the functional units. It communicates basic block addresses to the instruction dispatcher. When this processor is introduced in a multiprocessor architecture, the memory functional unit is in charge of the global memory. Figure 3 shows the organization of an O3 processor.

3.2 The functional unit structure

The functional units of an O3 are independent. Each of these units contains some memory where instructions wait for execution. These instruction memories play the same role as the reservation stations in Tomasulo controlled pipelines [5]. The main difference is that in our case, the association of a result to the

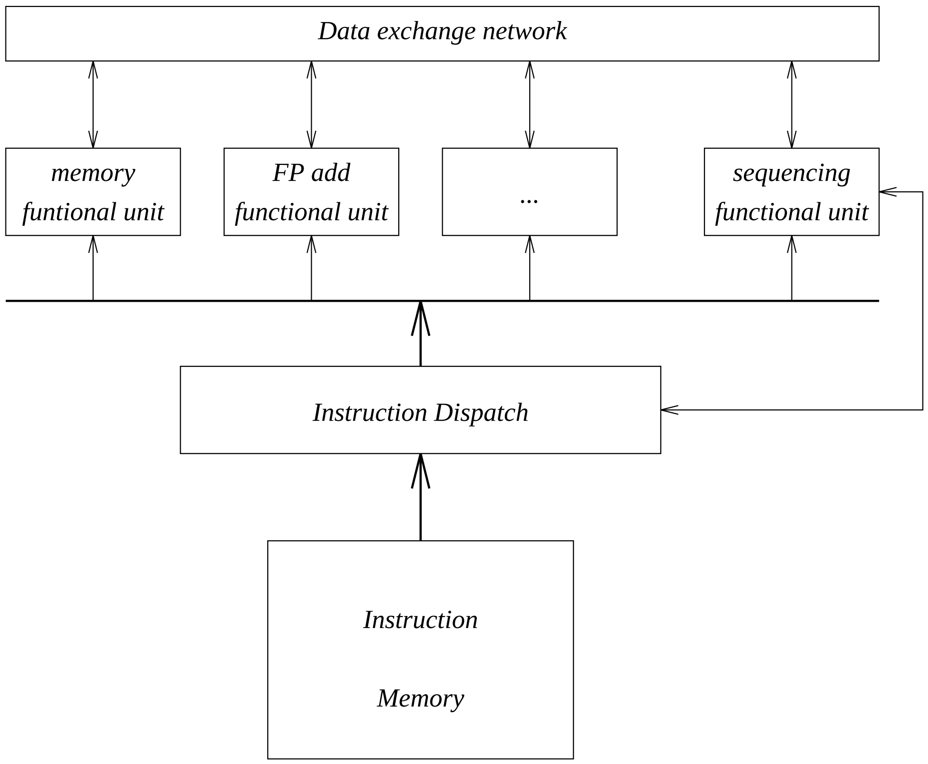

Figure 3: O3 architecture

instruction operands is direct: the producer knows the location of the consuming instruction.

Each cell of these waiting stations contains an operation field, operand fields and destination fields. The operation field contains the instruction code to be executed. Each operand field can store a parameter for the instruction. The addresses where the results are to be forwarded are stored in the destination fields. The operation and the destination fields are loaded by the instruction dispatcher. The operand fields are loaded directly by the functional units. When a program instruction is dispatched, it is written in the operation field of a waiting station on some functional unit. An instruction is executed when all its operands are present. Then the results are forwarded to the operand cell of a functional unit specified in the destination fields.

An instruction can be elected for execution as soon as all its parameters are present. In general, there is no implicit order for the execution of the instructions. However, more strict rules can be applied in some cases. For instance, the dispatching order is significant on the sequencing unit. On the memory units, addresses dependencies can be considered, based on dispatching order. Note that the execution of an instruction can be initiated as soon as its operands are present, even in the case the resutt destinations are not already known.

3.3 Instruction dispatch

The data exechanged between the functional units must be tagged because the execution order of the instructions is knot globally known. The instruction dispatcher is in charge of sequencing of instructions and in the computation of the thread states. Each time an instruction is executed by a functional unit, an execution status is sent back to the instruction dispatcher. The thread state is computed from the execution status. This aspect of the O3 architecture is treated in section 4.

The data tagging used in our system is different from the tagging used in Tomasulo based systems: in our system, the tag identifies the consumer of the result whereas in Tomasulo based systems the tag identifies the producer. The main advantage we see in our system is that a result can be routed directly to its destination reservation station and need not be seen by all the stations. The functional units can communicate through multiple parallel data path. The drawback, however, is that, in case a result is consumed more than once, it must be duplicated explicitly.

The instruction dispatcher of O3 repeatedly extracts a new basic block address from the block queue and scans the block to the end. During this scan, the instructions are separated from the constants. Constant folding needs a slight analysis of instruction codes. The instructions are packed in the instruction window and the constants are stored in a queue. The instructions are distributed on the dispatching stations, one group at a time. The number of dispatching stations defines the maximum number of instructions that can be sequenced during the same cycle. In our examples, we consider three dispatching stations.

An instruction issue by a dispatch station contains six major steps as shown in figure 5. Some of these steps are independent and are run in parallel. Steps of successive instructions are overlapped. In the following explanations, we consider the code sequence of figure 2 and we show the flow of computation inside each dispatcher stage.

partial decode and FU selection

$$fu_i = FuOf(Inst_i)$$

The instruction is partially decoded and the destination functional unit is selected.

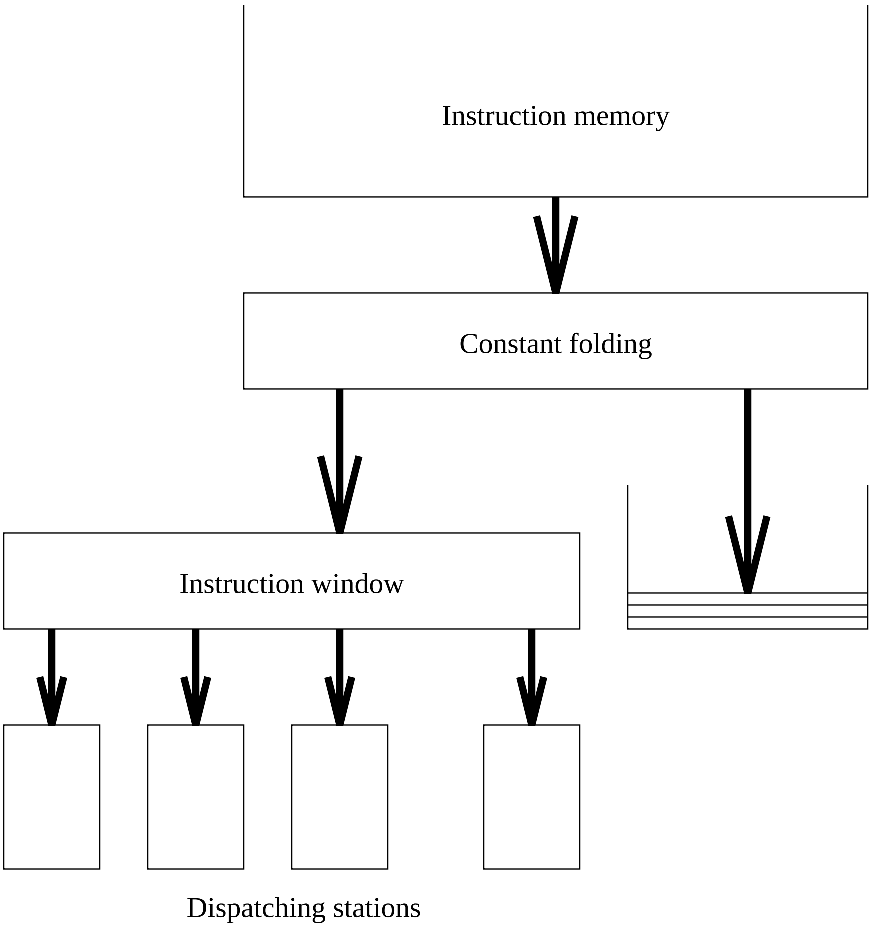

Figure 4: Instruction fetch and constant folding

instruction labelling

$$FuLabel_i = NewLabelFrom(fu_i)$$

An instruction label identifies globally an instruction waiting for execution in the local memory of a functional unit. This label is associated to all the data exchanges. Once the instruction is executed, its label is returned back to the dispatcher and can be reallocated. Such a label is constructed from a functional unit identification and the address of a cell in the waiting station of this FU. The dispatcher maintains a list of free labels for each functional unit. These lists

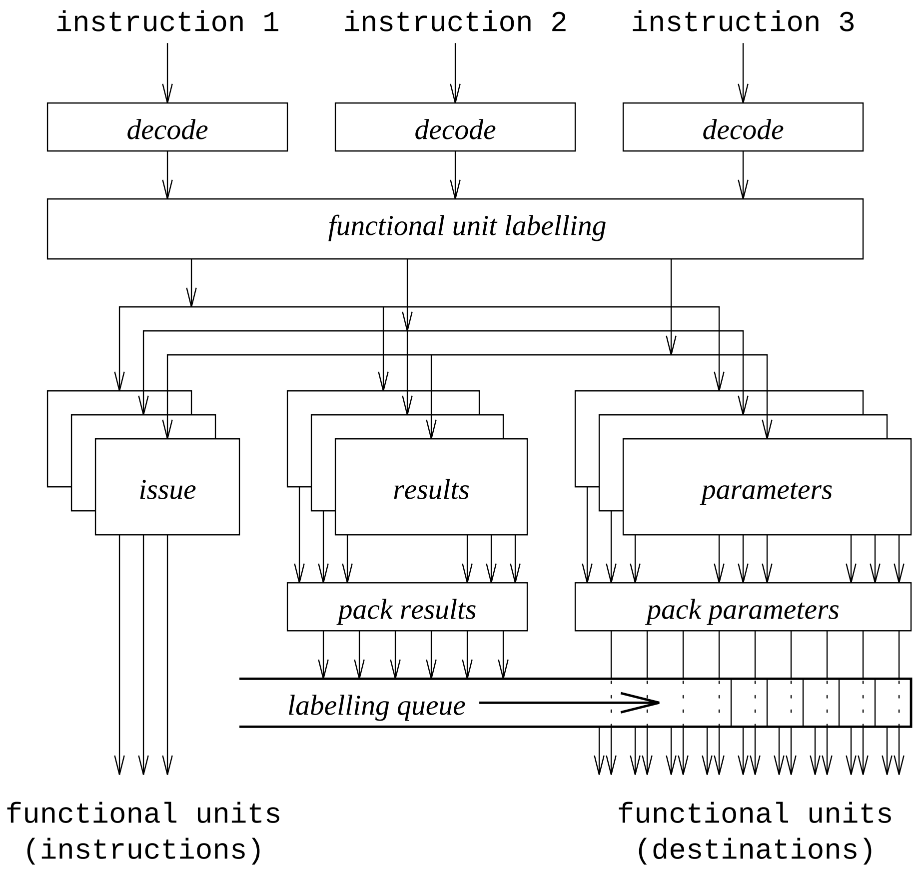

Figure 5: instruction issue

can be shared by the dispatch stations or can be partially distributed in order to limit conflicts. This scheme does not force any ordering in the allocation of the slots. One can also imagine simpler schemes where the memory slots are allocated and freed cyclically. Such a scheme simplifies the hardware for the label management (only counters are needed) but a higher number of slots may be necessary if the latency variations are high. In our examples, the labels are allocated cyclically starting from 0.

instruction issue

$$Inst_i \Rightarrow fu_i.Mem[FuLabel_i].op$$

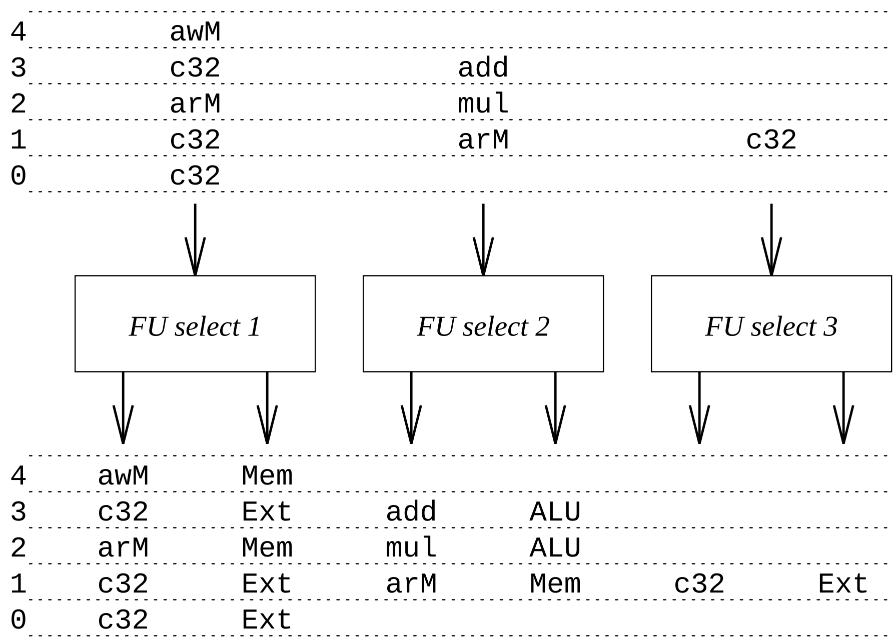

Figure 6: functional unit selection

The instruction are sent to the selected functional unit. It can be stored immediately in the instruction memory of the FU. This operation can be delayed in case two or more instructions destinated to the same functional unit are issued during the same cycle. At instruction issue time, the destination fields are not known.

result labelling

$$ResLabel_i^j = (FuLabel_i, j)\mathbf{if} present, \emptyset\mathbf{otherwhise}$$

An instruction of O3 can produce 0, 1 or 2 results. Each of these results is consumed by aninstruction of the next sequencing group and must be associated to an instruction entry. This is the purpose of the result labelling. The active result field addresses (if any) of the instruction are built. Such an address contains the instruction label and the result number. The result identifications are appended to the labelling queue.

$$\mathbf{for} i, \mathbf{for} j, AppendToLabelQueue(ResLabel_i^j)$$

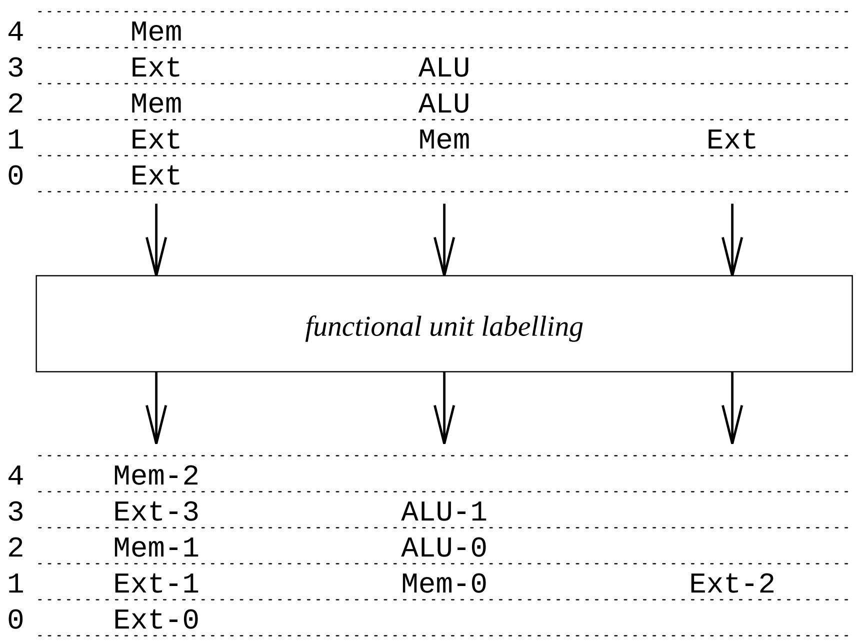

Figure 7: instruction labelling

parameter labelling

$$ParLabel_i^j = (FuLabel_i, j)\mathbf{if} present, \emptyset \mathbf{otherwhise}$$

Each parameter label identifies uniquely one operand of one instruction waiting inside the functional units: in our implementation, it is an address in the local memory of a functional unit. When an instruction is executed by a functional unit, each result is stored at the location defined by its label. This label identifies a functional unit, a cell in its waiting station, and a parameter field of this cell.

The parameter labels are computed when instructions are dispatched. Each label is then associated to the instruction which produces this operand.

$$ParameterList = \{ParLabel_i^j\}$$

parameter association

The parameter association connects each producer to the corresponding consumer. A parameter list is constructed by appending the parameter labels produ-

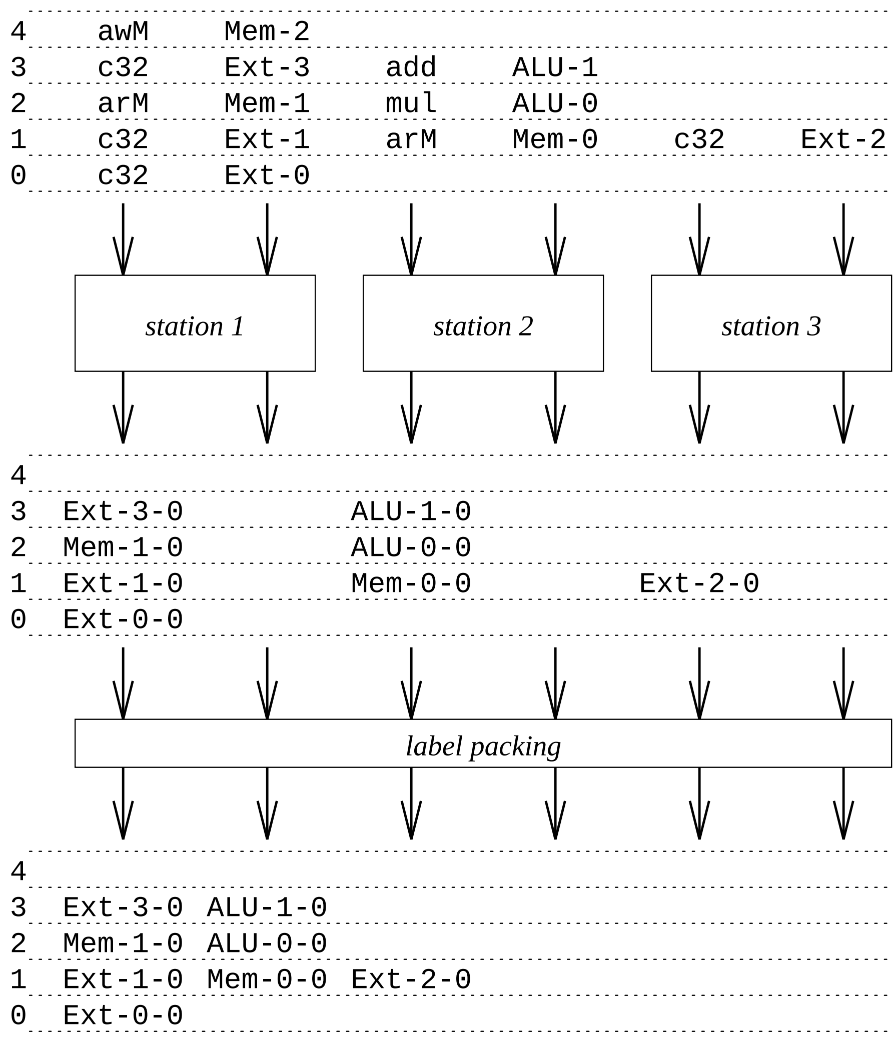

Figure 8: result labelling

ced by each dispatch station. A one to one mapping of these parameters to the result labels in the label queue produces the parameter association.

$$ParameterList_i \Rightarrow LabelQueue_i$$

A result label is in fact the address of a field in a waiting station where the corresponding parameter label is stored. As it was the case for the instructions,

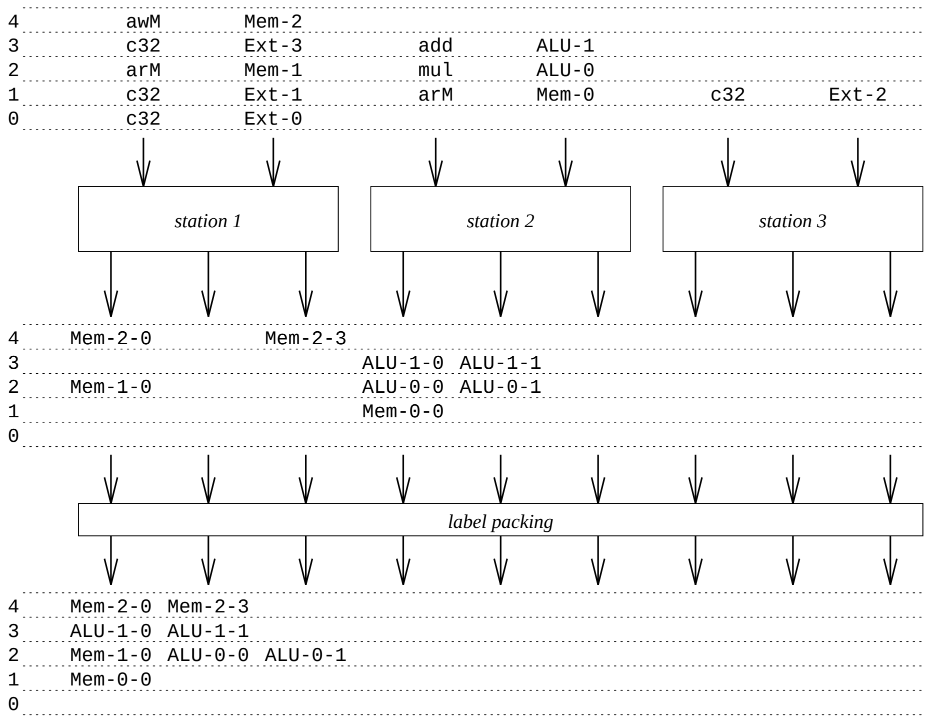

Figure 9: parameter labelling

more than one identification may be sent to the same functional unit during a machine cycle. But these addresses need not be stored immediately.

These steps can be fully pipelined. The only data dependency that can be found in this algorithm is in the extraction of the result labels produced by the dispatch of the previous sequencing group. But, in fact, the computation of these result labels does not depend on the values of the parameter label. In general, these results can be appended to the label queue even before the parameter of the same instruction have been labelled.

4 Program state, post-sequencing

The state of a code running on a processor must be computed each time its execution must be suspended. This computed state is used to restart the code when it can be activated. On classical sequential processors, such a state is the current program counter value, the processor registers and the memory. The

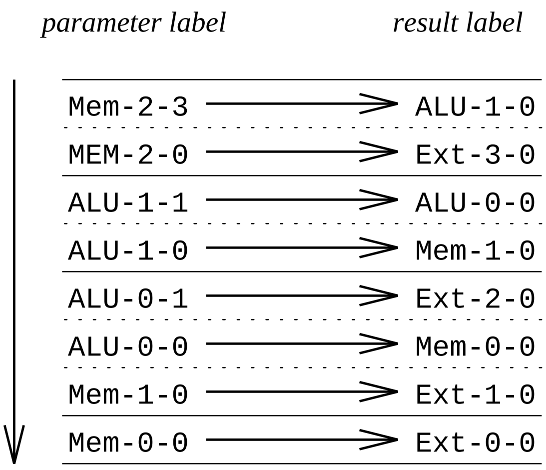

Figure 10: parameter association

sequential nature of processor sequencing allows a simple computation of the program state. But as soon as some parallelism is introduced in code execution, more than one instruction is active during one cycle, and so, the program state does not correspond to the register / memory values during this cycle. Some action must be taken in order to reach a *sequential* state [4]. The fact that such a sequential state must be always computable limits the effective parallelism of the computation.

The O3 model does not define such a sequential state because it would limit the usefulness of out-of-order execution. The O3 model defines restartable states which take into account the instructions which execution has not been completed among the instructions that have been issued. In order to build such a state, the origin of the interruptions is considered: external interrupts arriving from outside the process are distinguished from internal interrupts which are raised by the process activity.

4.1 External interrupts

An external interrupt is in fact a request for acquiring processor resources. Such a request is treated locally by a code dispatcher of O3. In order to execute an interrupt thread, some processor resource must be reclamed from a currently active instruction flow. The sequencing of instructions is stopped on the next instruction word and the dispatcher associates a *save destination* to each label in the label queue. The purpose of this special destination is to consume all the data identified by the label queue. The treatment of the save destination is described in subsection 4.4. The state of the process contains the data produced on the save destinations and the basic block queue — the first element of this queue is the current program counter value. The instructions that where dispatched but where not already executed when the external interruption was raised are not removed from the pipeline: they continue normal execution and can still produce internal interrupts. As soon as the label queue has been emptied, the sequencing of another processes can be started. And during some time, instructions from the interrupted process and instructions from the thread one can be mixed on the functional units. External interrupts produce sequential states.

4.2 Internal interrupts

Internal interrupts are generated by the activity of the thread. These interrupts can have several origins. In many cases, they correspond to the fact that some instruction could not be executed immediatly and, in general, needs some software handling. This case happens when some instruction is not implemented in hardware. It can also happen when instructions are partially implemented — some rounding of floating point operations. Another typical example is the translation from virtual to physical addresses through a software managed TLB cache. If the translation is not found in the cache, software code must be run. Physical page miss is another classical example where a memory access need some treatment.

Internal interrupts can also be raised explicitly by software. In general, Unix systems calls are treated this way, mainly because the context and the capabilities of the handler code are not the same as the user's ones.

In O3, all the internal interrupts are treated in a uniform way by a system we call post-sequencing. The basic idea behind post-sequencing is partial reduction. The instructions dispatched on an O3 develop the computation graph corresponding to the code. Each node defines an operation and each edge corresponds to a data communication. After the execution of each instruction, the corresponding node in the graph is transformed and reflects the result of this execution: a new operation code is associated to the node. When the new operation code is empty, the node is removed. The reduction is safe if no pending edge appears: an instruction cannot be reduced if one of the instructions that produces its operands was not previously reduced. At any time, the reduced graph represents the remaining computations of a thread.

In the same way a code sequence corresponds a computation graph, the reduced graph corresponds to a code sequence: the reduced code sequence. In the O3 architecture, the post-sequencer is in charge of the production of the reduced code sequence. In fact, the computation graph is never built in the architecture: the reduced code is generated directly from the execution status of the instructions. The sequencing model of an O3 is shown in figure 11. In this model, the instruction are dispatched on the functional units by the seauencer. The execution status of each instruction is returned to the post sequencer which in

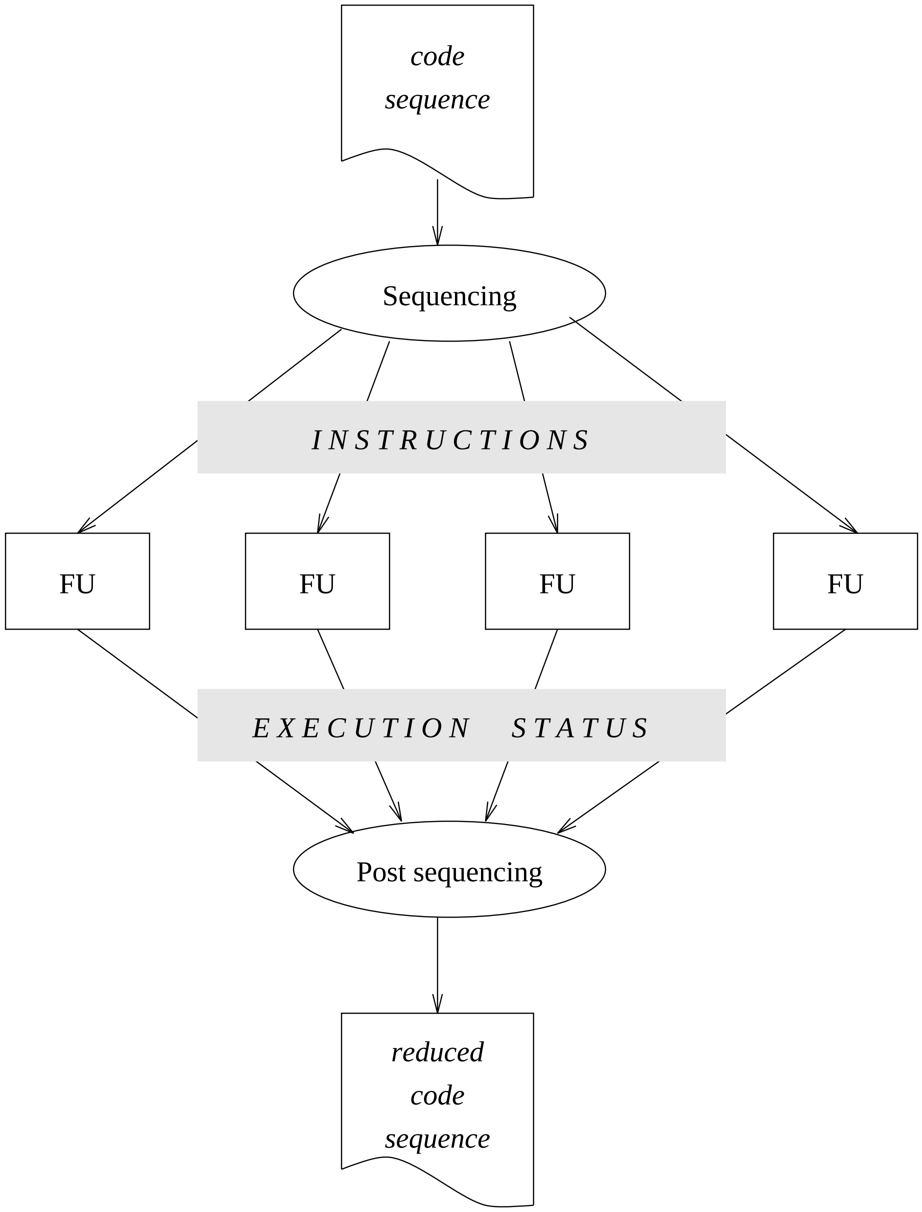

Figure 11: Sequencing model

turn computes the reduced instruction sequence. A validation tag – one bit – is associated to each datum exchanges by the functional units. In general, invalid tags are produced by unreduced instructions. Whether its operands are valid or not, an instruction can be executed as soon as all its operands are present. If at least one of its operands is invalidated, an instruction cannot be completly reduced, and so, produces invalidated results. This algorithm ensures safe reduction of the computation graph. It guaranties also that all the instructions that are dispatched on the functional units are executed within a finite delay.

4.3 Execution status

The last step in the execution of an instruction by a functional unit is the execution status production. An instruction status contains a replacement instruction and parameter descriptions. The replacement instruction is associated to the node in the execution graph. A `noop` replacement instruction means that the node can be removed. When the instruction is not a `noop`, the parameter description is present in the message. In general, some or all of the parameters of an unreduced instruction are valid. The valid operand must be saved in order to be regenerated when the code is restarted. The parameter description specifies, for each operand (one bit) of the instruction, if it is received from another node (the corresponding parameter was tagged in the previous execution), or if its value is already known. The valid values of the parameters are transmitted to the post-sequencer with the execution status.

4.4 Save-destination

When the dispatching of a thread is stopped, a save destination is associated to each result label in the label queue. When the execution of some instruction completes on a functional unit, two cases can appear depending on the presence of a save destination or not. If the replacement instruction is a `noop` and the destination of a result is a save destination, the `noop` is replaced by a constant load instruction. By this way, when the reduced code is restarted, the description of this result is appended to the label queue. If the replacement operation is not a `noop`, this result is ignored.

4.5 Reduced code

The reduced code sequence produced by the post-sequencer is stored in the instruction memory of the processor. The instruction dispatcher and the post-sequencer are independent. The fact that some operation cannot be reduced does not necessarily stop the instruction dispatching. However, within some delay after the production of a non `noop` replacement code is detected, the sequencing of the current thread should be stopped: an external inperrupt is generated, and a new state is computed.

4.5.1 page miss

In general, the memory management is handled directly by the system kernel. When a virtual page miss is detected by the memory functional unit, the kernel receives a signal, the access parameters and the thread identification. Because it cannot be executed, the faulting instruction produces a non `noop` status. The execution status may contain the faulting instruction, but this is not necessary. For instance, a faulting based address memory read can be replaced by an absolute addressing instruction. Once the page fault has been treated, the thread can be activated again. The first basic block of the thread is the reduced basic block and its execution is followed by the execution of the basic blocks from the block queue of the thread. Notice that, as such faults do not stop immediately the execution, more than one fault might have been generated.

4.5.2 system call

A system call cannot be executed atomically on a functional unit. The execution of such an instruction produces an execution status containing a result retrieval instruction. Once the thread is activated again, the execution of this instruction produces the result(s) of the call in the pipeline. Note that the system call does not stop instruction dispatching of the active thread. Moreover, a basic block can contain more than one system calls which in turn can be executed in parallel.

4.5.3 floating point exception handling

In general, these exceptions are handled through the insertion of kernel calls in the reduced code. Once the reduced basic block is executed, these system calls are transmitted to the kernel which can take appropriate decision. This two-levels treatment of the exceptions allows out-of-order execution of instructions with in order exception handling.

4.5.4 thread communication

The thread communication instructions are defined in the O3 ISA. But their execution is, generally, not possible on a single functional unit. In fact, in our model, it is based on post-sequencing as it involves kernel routines. At the hardware level, there are no significant difference between threads communication and system calls.

4.6 The post sequencer

The post sequencer is in charge of producing the reduced basic blocks in an O3 architecture. It communicates with the dispatcher and with the functional units through a system close to the labelling of the data. A post sequencing label is associated to each instruction that is dispatched to a functional unit for execution. Because of the sequential nature of sequencing and post sequencing, the post sequencing labels are allocated cyclically. This label is used by the functional units to tag the execution status.

The post-sequencer contains a local memory and the post-sequencing label are addresses in this memory. Each cell of this memory can contain one instruction code, a parameter specification and three constants. When the execution of an instruction is completed, the functional unit stores the execution status at the address defined by the label in the post-sequencer memory. This status contains an instruction code and the parameter definition. The parameter definition defines the number of parameters and, for each parameter, indicates its origin — known or received. The known parameters of the instruction are stored in the constant fields of the cell.

The post-sequencer scans its local memory in the dispatching order. Each instruction field is analysed and the `noop` status are removed. This scan produces a list of instructions, a list of constants, and a list of parameter origin. The list of instructions and the constant loads are merged according to the parameter specification and produce the reduced basic block. The reduced basic block is stored in the instruction memory from where it is fetched when the execution is restarted.

5 Conclusion

We have presented a new processor architecture which integrates pipeline, superscalar, out-of-order execution of instructions and which allows efficient implementation of multithreading. The main originality of this architecture is a new exception handling system which accepts true out-of-order execution of instructions. Precise sequential state recovery is no more necessary prior to restarting a thread execution.

Many aspects of this new model have still to be refined. Extensive simulations have to be run in order to evaluate the real potential performance of this architecture.

References

[1] Harry Dwyer and H.C. Torng. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, ACM, SIG MICRO NEWSLETTER, dec 1992.

[2] R. Govindarajan P. Lenir and S.S. Nemawarkar. Exploiting instruction-level parallelism: the multithreaded approach. In *Proceedings of the 25th Annual International Symposium on Microarchitecture, MICRO-25*, pages 189–192, ACM, SIG MICRO NEWSLETTER, dec 1992.

[3] Burton J. Smith. Architecture and application of the hep multiprocessor computer system. In *SPIE Real-Time Signal Processing IV*, pages 241–248, 1981.

[4] G.S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, mar 1990.

[5] R.M. Tomalulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, jan 1967.