Handling 16 instructions per cycle in a superscalar processor

Bernard Goossens

LIAFA, case 7014,
Université Paris 7 Denis Diderot,
75251 Paris cedex 05,
France
email: bg@liafa.jussieu.fr

We present a new partitioning of the processor data path based on a three level separation. First, as in a traditional processor, we separate integer computations from floating point ones. We include in both types data operators, registers, memory and address computation resources. Hence, integer computations are fully disjoined from floating point computations which reduces intra-chip communications. Second, data computation resources are also disjoined from address computation ones. This helps distributing the resource requirements on the data path and on the address path. Third, we separate each register file in two parts. The two sources of every instruction are taken from both parts. This two last separations help keeping the number of necessary resources to handle 16 Instructions Per Cycle (IPC) the same as what is needed today to handle 4 instructions per cycle. Our architectural proposal should allow the implementation of a superscalar degree 16 processor on tomorrow finer design processes.

1 Introduction

As digital systems engineers point out, signaling is next decade's challenge [4]. In a superscalar processor signaling is an essential issue both for technological and architectural reasons. Since 1995, the superscalar degree hasn't changed. It is stuck to 4 instructions per cycle. The main reason is that up to now, we had no way to provide more on-the-path instructions, due to the control flow breaks. With trace caches [13,14,7,12,1], we can expect that an average of 12 instructions per cycle can be delivered to the processor [12]. Figure 1 depicts a superscalar out-of-order processor fed with a trace cache. The trace cache contains blocks coming from the reorder buffer. The fetch unit reads both the instruction cache and the trace cache. In case of two hits, the trace

Preprint submitted to Elsevier Preprint *18 October 2003*

cache has precedence over the instruction cache. The multi-branch predictor [15,16,12] delimits the end of the fetched block (up to the first predicted taken branch for the block out of the instruction cache and up to the first branch predicted opposite way from the trace for the block out of the trace cache). After commit, committed instructions are grouped into blocks and update the trace cache. The committed branches update the multi-branch predictor.

Today, the main problem is to take care of so many instructions (say 16) without increasing any critical signal path [6,11]. Among the problems we are faced with, we find the register file ports, the reorder buffer length, the result forwarding data and control signals, the rename buffers control, the in-order commit logic and last but not least, the data memory access ports.

If today's architectures are simply expanded, register file read ports will increase from 8 ports per cell (superscalar degree 4) up to 32 ports per cell (superscalar degree 16). An n ports cell requires n data and n control wires, building an $(n-1)^2$ grid. A signal takes a time quadratic to the distance it travels. If we quadruple the number of wires on a register file (from 8 ports to 32 ports), we multiply the access time by 16. One way to prevent this for read ports is to duplicate the file as is done in the DEC 21264 [8]. Four copies with each 8 ports will keep the basic access time. However, this technique is not suitable for write ports because if read ports are distributed in the copies, write ports must be duplicated in each copy, making write access time a critical issue.

Data memory is also very critical. Among each group of 16 instructions, we have on the average 4 loads and 2 stores, requiring 6 memory access ports. For access time reasons, a multiple banks memory should be prefered to a multiple ports single bank memory. Nevertheless, a crossbar or similar routing technique is required to dispatch the memory requests to the banks. This kind of logic again has a complexity varying quadratically with the number of ports (i.e. banks).

To be able to absorb a rate of 16 instructions fetched per cycle, natural ILP has to be extended, mainly by dynamic reordering. Many instructions must be accumulated in a wait-before-firing buffer. For example, a SAXPY loop composed of 2 loads, one multiplication, one addition, one store, one displacement adjust, one vector limit check and one conditional branch, may be kept unrolled in the trace cache. Every cycle, 2 iterations are fetched. For each iteration, its dependent instructions (i.e. all of them except the two loads and the displacement adjust) are left in the wait-before-firing buffer until they can be fired to their functional unit (f.u.). If we assume that each f.u. has a single cycle latency, we need for the SAXPY loop a 32 slots wait-before-firing buffer. Of course, longer latencies would require a bigger buffer. The situation is worse for the reorder buffer due to the longer delay between fetch and commit. If we assume that fetching, running and committing each take one cycle (running includes out-of-order termination), the reorder buffer for the SAXPY loop should be 100 instructions deep.

Actual trends in processor design are to partition the layout. Embedded processors use a VLIW-like design, organized around a set of functional units. This is very efficient when a code using exactly the set of provided operators is run. The main drawback in such designs is that they are poorly scalable. The more operators on-chip, the larger the gap between optimal and real efficiency. Moreover, adding new operators has an impact on result forwarding logic (it increases quadratically with the number of functional units). Another choice is to partition the processor into multiple threads. When each partition contains a full processor, the design is called a CMP (on-Chip MultiProcessor [10,9]). The main problem in such a design is that the strict partitioning pattern isolates each thread from the others and each set of operators in a partition from the other sets. Moreover, the cache coherency requirements involve a permanent communication between the different partitions.

When all the threads share the out-of-order running engine (including the rename buffer, the wait-before-firing buffer and the functional units) we have an SMT-like architecture [17,18,5,19]. This time, the centralized nature of the functional units impacts the forwarding logic as in the VLIW-like design. Research works combining both CMP and SMT techniques are on their way [2,3].

In this paper, we propose a superscalar degree 16 design organized to partition quite evenly the fetched instructions and treat them in four isolated units, each requiring no more ports and logic than an actual degree 4 superscalar engine. Partitions are designed so that very few inter-partition communications are required.

The paper is organized as follows: next section presents the grouping of all the instructions in a computation around the type of the data manipulated and the separation in each type of the pointers computations from the data computations. Each typed register file is separated into a data part and an address part that resembles the 68000 data and address separated integer register files. Section 3 explains a second separation of the sources paths. Each register file is splitted in two disjoined parts and each instruction takes its two sources from both parts. This technique is comparable to the memory bank splitting to extend the bandwidth without extending the number of ports. Section 4 presents an implementation of the proposed architecture. Section 5 discusses the impact of the proposal on the ISA (Instruction Set Architecture) and on programming. Section 6 compares our work to the Multicluster architecture [6].

2 Data and pointers

Any computation a processor has to perform is composed of data loads, register computations, data stores and control flow logic. For example, the SAXPY loop can be written in an Alpha-like syntax as shown on table 2.

The trace of the loop contains a mix of the four mentioned categories. Each pair of iteration (16 instructions provided in a trace block) contains 4 loads, 2 stores, 8 register computations and 2 conditional branches. Notice that the distribution is roughly the one found in a standard benchmark: 25% loads, 12% stores, 50% computations (this one is a little high) and 12% branches (this one is a little low).

In an actual processor, the instructions are separated into integer and floating point ones. The loads and stores are considered integer ones because their address is computed from integer registers. A communication has to be performed between the integer data path and the floating point one for each load (twice per iteration). Another is required from the floating point data path to the integer one for each store (once per iteration). If two iterations are to be performed per cycle, the hardware should provide 6 buses linking the integer and the floating point resources. Even though the compiler partitions the code in two sets, the hardware partitions are not enough isolated from each other.

Partitioning has to take into account the fact that adjacent instructions mainly deal with the same data type. A floating point loop deals with floating point data. A string computation deals with characters or short integers. An integer numerical function deals with long integers. A graphical function deals with vectors of short integers (pixels). These data are obtained from memory, requiring address computation, i.e. mainly a constant base plus a varying displacement. The computation itself produces results of the same data type unless it is a conversion, in which case the computation is of the same data type but the result is of the converted type. In any case, the result is usually sent back to memory, requiring some more address computation. The control of the function is handled mainly by comparing the displacement to a limit. It may also rely on a check of the computed data.

When data, addresses and control values are grouped together in a single type, inter-type communication is rather rare. No such communication would be required for the SAXPY loop example. Even though we keep the usual integer/floating point separation to distinguish between integer data computations and floating point ones, we provide for each of these two types a set of resources to handle both data and addresses. This means that the floating point unit (depicted on figure 2 left part) includes both data and addresses registers (named FA and FD on the figure), floating point functional units as well as an address adder and a limit comparator. This is also true for the integer unit (figure 2 right part; IA and ID refer to the Integer Address register file and the Integer Data register file).

It is important to notice that this type extension has an impact on the instruction flow. For example, consider the SAXPY loop. In a traditional integer/floating point separation, the data computations are instructions run by the floating point unit whereas the loop control is taken in charge by the integer unit. The loads and stores are special in that they use both units. In our proposal, all the instructions in the SAXPY code are taken in charge by the floating point unit. Instead of using permanently both units, we use the integer one all along an integer data computation and the floating point one all along a floating point data computation. This is apparently bad because instead of partitioning the instruction flow on a fine grain basis, we partition it on a coarse grain basis. In other words, instead of distributing the instructions fetched every cycle, we keep them together in a single data path. This centralizing effect will be compensated by an intra-path distribution explained soon. It should be kept in mind that the reason for the type extension is to avoid as much as possible inter-type communication and to allow such an isolation in the layout.

This isolation also concerns the data memory. The floating point unit includes a first level cache that keeps the recently accessed floating point data only. The integer unit contains a separate first level cache that keeps the recently accessed integer data only. In the SAXPY loop example, each load and store being floating point ones as the instructions mnemonics show (LDS and STS), the data are loaded in the floating point unit first level cache. An integer computation would use integer loads and stores (LDL and STL in the Alpha syntax), involving the integer unit cache.

3 Distributing the ports requirements

Once types are isolated in per-type units, instruction handling within a unit should be organized in order to keep critical wires short. The first critical resource in a data path is the register file read ports. Table 3 shows how ports would be used in the SAXPY loop if the floating point unit would contain both data and address registers.

This table shows that the address part uses 2/3 of the ports. This is not suprising since we have 37% of memory accesses, each requiring 2 address read ports and half of the computation concerns the loop control involving displacement and limit that we consider as addresses. This unbalance should be present in any code. If two register files are provided in the floating point unit, one for data and one for addresses, 16 instructions, i.e. two iterations, would require 22 read ports on the address register file and 8 write ports on the data file. This is still far more than expected. We propose to separate each register file in two halves (see figure 3).

We would have four files, two for addresses and two for data. Each instruction works either on the address pair or on the data one. Its two sources may not belong to a single file. If the left source comes from the first address file, the right one must come from the second address file. At first sight, it seems constraining for the compiler. As the translation given for the SAXPY loop in table 3 shows, it is fairly easy to allocate registers to equilibrate each file use. In the SAXPY code, the names FA, FB, FC and FD refer to the four register files of the floating point unit (FA and FB for addresses and FC and FD for data). The general allocation scheme applied in this example assigns addresses and limits to the register file FA, displacements to the register file FB and data alternately to register file FC and FD.

One can notice that the left source and the right source may be swapped by the hardware. This is of course allowed only if the operation to be performed is commutative. If not (subtraction, division, shift), the constraint is that the left source should be in the A (or C) file and the right source in the B (or D) file. If this is still not the case, then a file transfer must be performed. Even though such a file transfer can be performed quickly (one cycle), the compiler should allocate the registers to minimize them. Moreover in certain cases, the ISA may be extended to allow two forms of non-commutative operations like $-y + x$ and $x - y$.

Table 3 shows the ports requirements for the four register files. As the table shows, no register file needs more than 4 write ports for 16 instructions. As for the read ports, a limitation to 8 read ports per file would not be too constraining (12 are required for the FA file and 10 for the FB file). Moreover, the separation of the registers into four files can be extended to the full computation path, including the wait-before-firing buffer, the functional units, the forwarding buses and comparators and the rename buffers. As next section will show with a figure of the micro-architecture we propose, the forwarding logic is much simplified because instead of routing all the results to all the waiting instructions, we route the results (4 write ports, hence 4 results) going to one register file (say FA) to the wait-before-firing buffer that contains waiting sources coming from the same register file (e.g. FA). We separate the buses and we reduce considerably the number of comparators.

4 A superscalar degree 16 processor

Figure 4 depicts a general overview of the superscalar degree 16 processor. Nothing is much different from the processor shown on figure 1 except the inside of the two computing units. This makes two differences on the instruction flow. First, the floating point instructions entering the floating point unit include the floating point data structures management instructions (loads, stores, address and displacement computation and limit checking). Second, the commit process is distributed. The reorder buffer keeps track of the ongoing instructions in-order. Each unit updates a status field per instruction each time an instruction is terminated. Each unit receives a copy of the status of the instructions to be committed. It stops committing when a non terminated instruction is encountered, whatever its type. Commit proceeds in the unit for any instruction of its type. Other instructions are simply discarded (and committed in parallel by the other unit). The connection between the integer unit and the floating point one is a bidirectional bus allowing the transfer of a datum between any integer register file and any floating point register file. Special transfer instructions are used to control this bus. A single bus is enough because inter-types transfers should be rare (conversion function result). The branches outcomes are part of the status informations transmitted back to the reorder buffer by each unit. The reorder buffer checks every terminated branch at commit time and updates the predictor.

Figure 4 depicts the floating point unit. The integer unit has a similar design. From the 16 instructions per cycle fetched, we assume half of them concern pointers and the other half concern data. In other words, on the average the address part of the fp unit receives 8 instructions per cycle and the data part receives also 8 instructions per cycle. Each instruction reads its two sources in two different register files. For example, the $MULSFC8, FD1, FC8$ instruction reads file FC for its left source and file FD for its right source. The files addressing logic as depicted on figure 3 allows a left source to be taken from either file. Hence, each group of 8 instructions uses no more than 8 read ports per file.

The sources, either ready or still being computed, are placed in a per-file wait-before-firing buffer. The set of 8 functional units computes 8 results per cycle, separated in two halves, according to the destination register file. For example, the $ADDLFB8, \#4, FB8$ instruction drives its result along the FB path while the $SUBLFA3, FB8, FA8$ instruction drives its result along the FA path. Thus, each rename buffer needs 4 write ports and each wait-before-firing buffer includes $4n$ comparators for n being its number of entries.

Loads and stores instructions are separated in two parts (as if two instructions had been fetched). The address computation goes in the address part and the memory access goes in the data part. For example, the $STSFD8, FB8(FA2)$ store instruction is separated into an address computation $(FA2 + FB8)$ and an access instruction reading register FD8. The data cache itself is one of the fp data functional units. The connection between the two parts is done by result forwarding. The dependent part (the memory access) waits in its wait-before-firing buffer the result from the other part (the address computation). When this result is received, the second part is fired. As such events are frequent, connections between fp address functional units results and FC and FD wait-before-firing buffers are provided. Notice that as the compiler distributes loads and stores on both files FC and FD, the addresses are hardware distributed on both wait-before-firing buffers. In the SAXPY loop example, the $LDSFC8, FB8(FA1)$ instruction sends the address to the FD wait-before-firing buffer. The $LDSFD8, FB8(FA2)$ instruction sends it to the FC wait-before-firing buffer and the $STSFD8, FB8(FA2)$ instruction sends it to the FC buffer and the datum to be stored in the FD buffer.

The same technique can be applied to inter-type and intra-type transfer instructions. More connections are required but because such transfers should be rare, a shared bidirectional bus could be enough (linking FC and FD to FA and FB, used for example when a pointer is loaded, and any data file from the integer unit and the floating point unit, used for a conversion). Such supplementary links are not drawn on the figure. Of course, transfers between FA and FB files are straightforward, as are transfers between FC and FD.

5 Other impacts on ISA and programming

The instructions codification should be adapted to the set of available registers. If we assume 64 registers are provided per type, divided into 4 equal files of 16 registers each, we need 12 bits to point to 2 sources and 1 destination, locally in each file. As sources are distributed on two files, we need 2 more bits to fix the left source and the destination file. The operation code indirectly designates either the address or the data space (or both for loads and stores) and it also designates the type (integer or floating point). Figure 5 shows the instruction code general format.

Call and return instructions should also be typed in order to save and restore the return address in the proper register file. All the caller's instructions concerning the callee's arguments should place them in the callee's type register file. The return address itself should be placed in the callee's A file. This means that most of the calling preparation instructions have the callee's type. If arguments type conversions are needed, transfer instructions must be added (the transfer instruction itself can include in its operation field a conversion to be applied during the transfer). An array of function pointers should belong to the common result type memory space. For example, an array of integer function pointers should belong to the integer memory space. In case the array would contain a mix of floating point and integer functions pointers, the array can be arbitrarily placed in the floating point memory space, requiring a transfer for each integer function call (this is to avoid caching the same memory block both in the integer and in the floating point caches, which would lead to a potential incoherency). As such transfers should be slow, such a mix should be avoided or kept rare.

The same technique can be applied to structures and unions either statically or dynamically allocated. For example, a complex number structure grouping 2 floating point numbers is placed in the floating point memory space. A structure combining data with pointers (as in a list) is placed in its data memory space. A structure or union combining a mix of data types (i.e. at least one integer and one floating point) can be arbitrarily in the floating point memory space (involving transfers for integer data accesses). Another possibility is to rely on the compiler that can distribute the structure fields on the different memory spaces according to the data types. This should be particularly useful for object oriented structures.

The two basic types considered in this paper may be extended. For example, an SIMD data type used to handle arrays of fixed point data (like pixels) can be added. For each new type, a new set of register files, operators, rename and wait-before-firing buffers and a data cache should be provided. The communication bus linking the integer unit to the floating point one is extended to the new units.

6 Related works: the Multicluster architecture

The Multicluster architecture [6] partitions the resources of the processor into multiple clusters. Each cluster contains private resources among which are a renamed register file, a dispatch queue (our wait-before-firing buffer) and execution units. Renaming is performed as in the R10000 [20] with a dynamic mapping of the registers pointed by the code of each instruction onto the renamed ones. Inter-cluster communication is involved every time the renamed registers of an instruction are located in two different clusters. The memory is represented by a data cache shared by all the clusters. Instructions are distributed to the cluster they need. For instructions with two clusters involved, two dual instructions are issued to perform the communication, like our load and store instructions computing the address in the address partition and accessing the memory in the other partition.

In the Multicluster architecture, the cluster partitioning helps keeping the number of ports low, the dispatch queues sizes small and the forwarding logics reasonable. However, the main problem the architecture has to face is inter-cluster communication. To reduce the communication needs, the architecture uses a static balancing algorithm that tries to place independent instructions in different clusters. Hence, a computation involving chained instructions are likely to be placed in the same cluster. This algorithm also takes care of equilibrating the clusters loads.

To compare the Multicluster architecture to our proposal let us first assume they both have two clusters, one for floating point computations and one for integer computations. But in the Multicluster case, rather than giving each cluster some address computing facilities as in our proposal, each cluster is strictly devoted to data of its type as in a traditional processor architecture. Then, the balancing algorithm for a code like the SAXPY loop has to handle the many inter-cluster communications involved by the address computations and the loop control. Moreover, as the data cache is shared by the two clusters it becomes a centralized resource, thus critical.

The main difference between the Multicluster architecture and our proposal is that the Multicluster architecture first partitions and then tries to avoid communications. In our proposal, we first group dependent instructions (address and data computations of the same type including memory accesses) and then we partition them inside their type cluster (separate addresses from data and separate left source from right source). Grouping avoids most inter-cluster communications and separating simplifies the hardware.

7 Conclusion

In this paper, we have presented a partitioning of the processor that helps reducing intra-chip communications and that is able to handle a 16 IPC fetch rate. We did not include any simulation results. It would be useful to measure if the distribution of the address and data instructions is not more unbalanced than estimated in the paper and if the inter-cluster communications are really low. A simple simulation, done from a trace out of an existing processor would not be probing since the code would not be organized according to our partitioning. Too many transfers might be necessary either between A and B file or C and D file for source swap reasons or between the fp unit and the integer unit for mixed data structures access. A more probing simulation requires a precise definition of the ISA and a new compiler adapted to this ISA, minimizing file or unit transfers. Moreover, to make a fair comparison of our proposal with traditional architectures we need to take into account the impact on the processor cycle. This should provide the frame of a future work.

Nevertheless, the simple hand coding of very basic functions -as the ones in the C standard library- that we have experimented, shows that the partitioning does not constraint the compiler, that transfers for correct sources placement are rare and that address computations represent from 1/2 to 2/3 of the total instructions. Of course in these functions quite no inter-cluster communications take place.

Another important issue that has not been fully addressed in this paper is the memory partitioning. Our proposal separates the data memory into types. This is good to avoid communications between clusters but it does nothing to lower the ports needs. A simple solution is a banked organization but other solutions involving a separation between data and pointers as what has been proposed for the register files should be investigated.

Eventually, one can notice that our partitioning has the particularity to make the processor use its units one at a time. It can constitute a consumption reduction for the chip or a multithreaded version of such a processor can take advantage of this to run typed threads in parallel in the different units. This is another hint for a possible future work.

References

[1] B. Black, B. Rychlik, J.P. Shen, *The block-based trace cache* (ISCA26 1999).

[2] J. Burns, J-L. Gaudiot, *The SMT fetch bottleneck* (MTEAC/HPCA5 1999).

[3] J. Burns, J-L. Gaudiot, *Quantifying the SMT layout overhead. Does SMT pull its weight?* (HPCA6 2000).

[4] W.J. Dally, J.W. Poulton, *Digital system engineering* (Cambridge University Press, 1998).

[5] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, D. Tullsen, *Simultaneous multithreading: a foundation for next-generation processors* (IEEE Micro, sept/oct 1997).

[6] K. Farkas, P. Chow, N. Jouppi, Z. Vranesic, *The multicluster architecture. Reducing cycle time through partitioning* (ISM30, 1997).

[7] D. Friendly, S. Patel, Y. N. Patt, *Alternative fetch and issue policies for the trace cache fetch mechanism* (ISM30, 1997).

[8] L. Gwennap, *Digital 21264 sets new standard* (Microprocessor report (10)14, 1997).

[9] B. Nayfeh, L. Hammond, K. Olukotun, *Evaluation of design alternatives for a multiprocessor microprocessor* (ISCA23 1996).

[10] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chang, *The case for a single-chip multiprocessor* (ASPLOS7 1996).

[11] S. Palacharla, N. Jouppi, *Complexity-effective superscalar processors* (ISCA24 1997).

[12] S.J. Patel, M. Evers, Y.N. Patt, *Improving trace cache effectiveness with branch promotion and trace packing* (PACT 1998).

[13] E. Rotenberg, S. Bennett, J.E. Smith, *Trace cache: a low latency approach to high bandwidth instruction fetching* (ISM29 1996).

[14] E. Rotenberg, Q. Jacobson, Y. Sazeides, J. E. Smith, *Trace processors* (ISM30 1997).

[15] A. Seznec, S. Jourdan, P. Sainrat, P. Michaud, *Multiple block ahead branch predictors* (ASPLOS7 1996).

[16] J. Starck, M. Evers, Y.N. Patt, *Variable length path branch predictors* (ASPLOS8 1998).

[17] D.M. Tullsen, S.J. Eggers, H.M. Levy, *Simultaneous multithreading: maximizing on-chip parallelism* (ISCA22, 1995).

[18] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, *Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor* (ISCA23, 1996).

[19] S. Wallace, B. Calder, D.M. Tullsen, *Threaded multiple path execution* (PACT 1998).

[20] K.C. Yeager, *The MIPS R10000 superscalar microprocessor* (IEEE Micro, 16(2), 1996).