A Dynamic Scheduling Logic for Exploiting Multiple Functional Units in Single Chip Multithreaded Architectures

Prasad N. Golla and Eric C. Lin
Computer Science & Engineering Department,
Southern Methodist University, Dallas TX 75275
{golla,ecl}@seas.smu.edu

Keywords—Tomasulo's Algorithm, Computer Architecture, Microprocessor, Multithreading, Threaded Architectures

Abstract—Previous techniques used for out of order execution and speculative execution use modified versions of Tomosulo's algorithm and re-order buffer. An extension to these algorithms for multi-threading is necessary in order to issue instructions from multiple streams of instructions dynamically and yet keep the consistency of state for the machine. We present an architecture which has a wide instruction issue rate and can issue from multiple threads at a time. This asynchronous architecture has a mechanism to dynamically resolve data dependencies and executes instructions out of order and speculatively without any special help from an optimizing compiler. Instructions from various threads are interleaved simultaneously to dynamically exploit ILP to the maximum that the architecture can provide. Consistency of state is maintained by precise interrupts and in-order commitment of instructions for all the threads in execution.

I. Introduction

Advancing semiconductor technology will make it possible to put a billion transistors on a single-chip in the coming years [19], [20]. This has prompted computer chip manufacturers to develop architectures which can utilize the extra transistors. Most of the extra chip space is being allocated to increased on-chip cache and more peripheral support rather than to provide more functionality to the CPU core of the architecture. Increased functionality does not scale up due to a single path of control. VLIW, Superscalar and Multiscalar are few such designs which have been proposed and implemented to date. There is another paradigm called the multithreaded paradigm which has been around since the 1950s but has recently come of age [15]. From the point of view of processor designers and manufacturers, it would be advantageous to have a corresponding performance gain in the processor with the increasing die size (or more specifically the decreasing feature sizes).

As the number of transistors on a die is becoming larger and larger, multithreading on a uniprocessor system is becoming more and more feasible. Such a single chip processor should have the ability to exploit ILP from multiple threads and switch between the threads in case of any latency operation. In this paper we present such a processor, which exploits parallelism from multiple threads. Not only does the processor architecture allow multiple User threads to execute concurrently and parallely on multiple logical pipelines, it allows User and Kernel threads to execute at the same time. Hence we named this architecture the Kernel-User Multithreaded Architecture (KUMA).

During the coming years, feature sizes will continue to decrease and clock speeds will continue to increase, making the wire delays a larger percentage of overall signal delay. Only a small percentage of the die will be reachable during a single clock cycle [21]. According to an estimate, only 16 % of the die length is reachable within one clock cycle for a 0.1 μm process at 1.2 GHz. This segregates an integrated chip into little isolated islands of logic. To deal with it, we need an architecture which is modularized and which has *self-timed units*. A proposed processor, Kin [22] is such an asynchronous architecture designed for future technologies. For the same reasons, KUMA is designed to be an asynchronous, modularized architecture.

A number of dynamic scheduling techniques have been proposed, some of which are: CDC 6600's Scoreboard [12], Tomasulo's algorithm [1], Decoupled execution [3], Register update unit(RUU) [2], Dispatch Stack [13], Deferred-*scheduling Register-renaming instruction shelf*(DRIS) [10]. These techniques are well suited for either scalar architectures or superscalar architectures with limited issue rate. Some of these techniques create stalls or locks in the architecture due to control and data dependencies. Also, some of them have the ability to support precise interrupts. Among the above schemes the Metaflow architecture [10] has the ability to issue instructions out of order and speculatively via DRIS but lacks the ability to provide support for multiple thread execution. The RUU method of allocating tags is a simplistic tag attaching method, however it reduces the amount of ILP that could be exploited because of the checking of dependencies at the issue stage. The architecture proposed by Hirata et al [17] concentrates mainly on multiprocessor multithreading systems and holds a thread on a single processor before a conditional branch could be resolved.

KUMA is a wide issue architecture issuing eight instructions in a single cycle from up to eight active threads. The hardware mechanism of KUMA automatically resolves dependencies and removes stalls at the execution stage. It supports out-of-order execution and completion and flushes speculatively executed instructions in case of a branch misprediction. Out of order and speculative execution is achieved through shelving similar to the Tomosulo's Algorithm. Depending on the number of threads active in the process queue, the utilization of the processor could be optimized for a given branch prediction accuracy by reducing the flushing of speculatively issued instructions. Since scheduling takes place at two stages in KUMA, the throughput for high priority threads could be managed. Scheduling of instructions at two levels, namely the thread level and the instruction level and elimination of context-switches among the sub-set of the active threads, may provide KUMA the ability to support Real-Time Operating Systems. Hence, KUMA is a

SAC '99, San Antonio, Texas
©1998 ACM 1-58113-086-4/99/0001 $5.00

good candidate for the new breed of multimedia uniprocessor computers. Our goal is to achieve a cost-effective MIMD uni-processor with the ability to have high utilization and throughput.

In this paper, we will concentrate on presenting the details of the logical mechanism of the architecture. The rest of the paper is organized as follows: in section 2, KUMA processor architecture and the issues related to its relevancy are addressed. Hardware organization and explanation of various units of KUMA are presented in section 3. Section 4 presents our empirical results and evaluation of the architecture, concluding remarks and future work are presented in the last section.

II. Processor Architecture

A. Machine Model

The most simplistic choice to the manufacturer of a uniprocessor system is to adopt the "Cookie-cutter" approach; when the semiconductor technology improves so much that the number of transistors on a single die is several times than that were used for a scalar or super-scalar architecture, replicating a scalar or superscalar architecture to fit a given die would be a plausible choice. Such a processor architecture is the 4 x two-issue multiprocessor proposed by Olukotun et al [5]. Depending on the amount of parallelism and granularity of the threads of the applications, they show that their multiprocessor is 10% to 100% better than a wide superscalar microarchitecture.

Another choice would be to increase the peripheral support or the on-chip cache size. Though this increases the performance of the processor by reducing the miss latencies etc, this does not directly correspond to the increase in processing power. Yet another approach would be to increase the number of functional units, which corresponds to the increase in processing power of the processor, provided the utilization of these units is high. The utilization of these added resources could be increased if they are shared among the virtual pipelines. Since the functional units are the heart of any architecture, to increase resource utilization and throughput, we have to keep these as busy as possible. As Hirata et al [17] explain, the utilization of a functional unit could be expressed as $U = \frac{N \cdot L}{T} \cdot 100\%$, where N is the number of invocations of the unit, L is the issue latency of the unit, and T represents the total execution cycles of the program. If the utilization of a functional unit or a set of functional units is 30%, by unifying three virtual pipelines to share the same functional unit or the set of functional units, the utilization can be brought up to 90% provided the conflicts between the threads sharing the functional units are properly resolved. Hence, the utilization of the functional units can be increased by unifying the virtual pipelines and running multiple threads simultaneously. The organization of such an architecture, and one which KUMA follows is as shown in Fig. 1. We try to unify two to eight virtual pipelines in KUMA.

KUMA sets to exploit both the instruction level parallelism of the programs and the fine-grain and coarse-grain parallelism of threads on a single-chip multithreaded processor. Multithreading was shown to have significantly improved the performance of commercial environments by switching between 3-5 active threads on cache misses [18].

Thread switching between the active threads takes place in KUMA. When multiple threads are issued at the same time, the instructions from these threads compete for the same functional units. These contentions are resolved on two criteria; in favor of threads which have high priority and and against a thread which has many instructions already executing speculatively.

KUMA issues eight instructions per cycle and hence needs high instruction bandwidth to sustain its aggressive hardware scheduling. Three major factors constrain the fetching of instructions [6]: instruction cache performance, branch prediction and instruction alignment. These issues have received much attention. In the following sections we present some of our results on these three main issues. KUMA decouples the instruction fetch hardware from the execution by using instruction queues, which reduces the impact of the fetch hardware on execution of the functional units.

Data bandwidth demands for KUMA would be also high because of its wide issue rate. Multi-port, non-blocking caches would be capable of supporting such high data bandwidth demands [4]. The cheapest way to build multi-port cache is to build a banked cache. Owing to the interconnects however, multi-port non-blocking caches add complexity and delay compared to single-port blocking caches. We assume that with sufficient number of banks and Miss information/Status Holding Registers(MSHRs) the bandwidth requirement placed on the data cache is met. For more details on the cache issues, we refer the reader to [4], [5], [6].

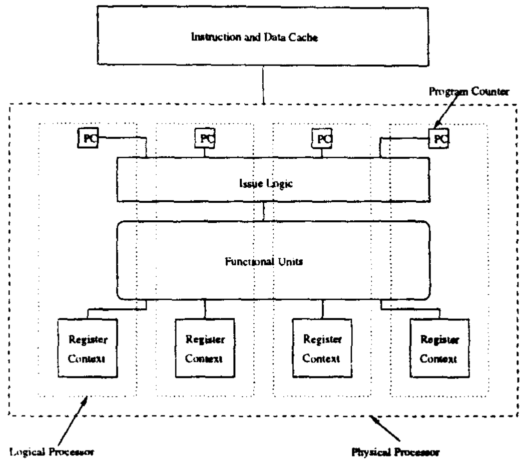

Fig. 1. Organization of the architecture.

B. Branch Prediction accuracy

The branch predictor performance becomes very crucial when we have a deep pipeline or have very wide issue rates because of the wastage of speculative work in case of a branch misprediction. In the face of limited accuracy of branch predictors, our simulations show that the utilization of the architectures could be increased by issuing from multiple threads simultaneously. To illustrate this point, we can use the "probability of commitment of a speculatively executed instruction" as an indicator for the performance and utilization of a processor. The probability of commitment of an instruction which is executing or was executed ahead of time, is directly proportional to the performance of the processor. Since performance per resources used gives us the utilization of the processor, a high probability of commitment of an instruction increases both the performance and utilization of the processor. However, when the turn around time of a critical application is of importance, one could forsake utilization by speculatively executing as many instructions as possible from that high priority thread. For general purpose computing and applications, our former argument holds, since most of the threads run at slightly varying priority and overall throughput of the applications executing is of importance. Our later argument holds particularly for time-critical and real-time applications where fast response time and turn around time are crucial.

Taking the average run length of a program to be 5 and the branch prediction accuracies of 97.2%, 93%, 90% and 85%, with different speculative execution window sizes, a table derived from a simple analytical model shows the probabilities of commitment of instructions in various execution window sizes. Table I shows these probabilities. For example, at 93% prediction accuracy, for a window size of 32, the average probability of commitment of the speculatively executing instructions is about 77% for a scalar architecture. At the same prediction accuracy and window size, if the architecture were running four threads, the average probability of commitment would be around 90% and for eight threads the corresponding probability of commitment would be 93%. These analytical results were confirmed by empirical results by running SPEC92 benchmark programs.(reference to our previous published paper)

C. Cache Memory Bandwidth

Cache memory bandwidth is as critical for a multithreaded architecture as is for any high issue rate processor. Since multiple threads are in execution at any given time, the architecture requires a significant bandwidth. To optimize the performance and to provide the required bandwidth, we need to find the proper correlation between the cache memory and the fetch unit. Our simulations were conducted using a set of benchmark program traces generated from the SPEC92 benchmark programs in the Dinero format. Consistently our results favored private instruction and data caches at the L1 level. At eight threads derived from different processes, the miss rates for private caches are about one third that of the miss rates for shared caches. Increased associativity decreases the miss rate up to a point and LRU is better than random replacement policy. The size of the block(line) has a limitation in its implementability; a longer block size will increase the number of interconnections. We show that as we increase the block size for about 64 bytes for data and 128 or even 256 bytes for instructions, the miss rates decrease. The Empirical results show the Cache Memory configuration best suited for a multithreaded architecture would be as follows: Private Data Cache, Private Instruction Cache, 64 bytes per line(block) for Data Cache, 64 bytes per line(block) for Instruction Cache, 4 way set associativity for each cache, Least recently used replacement policy, 64 k Data Cache and 64 k Instruction Cache. We noticed that the average of data and instruction miss rates even at eight threads do not go beyond 1.5% for the above configuration.

D. Instruction Fetching and alignment

Rather than fetch only one instruction per access, an advanced processor is generally implemented to fetch four or eight contiguous instructions. Conte et al [6] do a cost and performance trade-off study of various schemes for instruction realignment. KUMA takes the fine-grain multithreading approach and issues eight contiguous instructions from one thread in each clock cycle. Therefore, the entire issue bandwidth is filled with instructions from the same thread on a single clock cycle. This would avoid the need for stride accessing and memory interleaving. KUMA does not take any help from the compiler to issue the instructions and does not keep track of the availability of the functional units. Instead it issues these eight contiguous instructions every cycle without any empty slots at the issue stage. Issuing of instructions from multiple threads based on functional units availability is not easy to implement. It requires compiler support and even leads to wasted issue slots per cycle.

Also by eliminating the need for instruction dependency checking or functional unit availability checking at the fetch stage, we could fetch a whole block of instructions from the same thread in one clock cycle without any restrictions. Some of the fetched instructions in this case might be lost after re-alignment. We believe that this reduces the complexity of the memory configuration and the fetch unit. We present the design of the instruction fetch unit to support fine grain multithreading in KUMA. The instruction fetch unit is connected to the branch prediction unit (BPU). The BPU updates the program counters(PCs). If we have eight threads active at any time, we need eight PCs as shown in the Figure 2. Further, the BPU can be divided into two units: *the scheduler* and *the predictor*, according to the functionality.

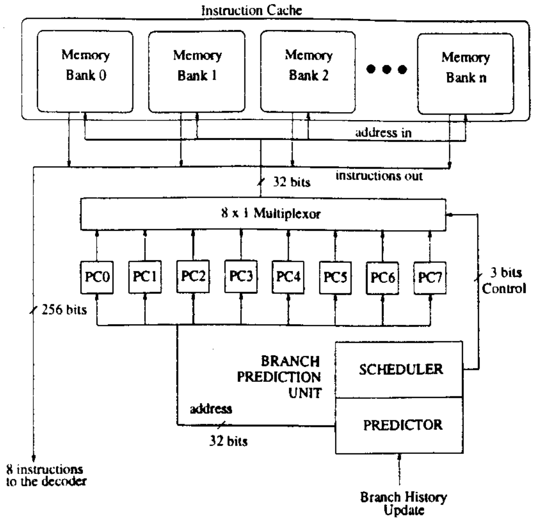

Fig. 2. Organization of Instruction Fetch Unit in KUMA

D.1 BPU-Scheduler

The Scheduler sends the control bits (three bits for eight threads) to show which thread the next block of instructions should be fetched from. Hence, it makes the decision on which thread the next block of instructions will come from and can avoid fetching from threads whose PC requires to be updated by the scheduler. In this category fall the threads which have just encountered a branch misprediction, threads which are waiting on a cache-miss, or threads which have just encountered an execution "block" (waiting) due to remote memory access, I/O operation, or made a blocked system call, or waiting for synchronization. A thread whose instructions are fetched in a particular cycle is automatically excluded from being scheduled the next clock cycle. This is because it takes one more clock cycle at least for the decoder to decode and update the branch history table to predict the next block of instructions. This exclusion is not strictly enforced, but if multiple threads are ready for execution then this technique is definitely preferred. If all the threads are of equal priority and there are eight threads in the ready queue, the scheduler will schedule these threads in a *round-robin* fashion, so that there will be eight clock cycles before another block from the same thread is brought in. This will lighten the burden on the Branch Predictor. If there were eight cycles between fetching of each thread, we can even eliminate branch misprediction altogether, since the operation on which the branch is waiting might be already resolved for small branch frequencies. Also, out of the ready threads the Scheduler can get the thread which has the highest priority indicated and set by the operating system. It is important to support priorities for threads especially for KUMA, since issues both Kernel and User threads simultaneously. As is the practice, Kernel threads are given higher priority than the User threads and are scheduled ahead of User threads.

D.2 BPU-Predictor

The predictor makes all the branch predictions based on the history buffer it keeps for all the threads. It sends out the predicted address to the PCs and indicates to the BPU-Scheduler that the particular PC slot is valid. Only one access is made for a block of instructions, the fetch logic fetches and re-aligns these instructions. The PC increments at the stride of eight or the address of the last instruction fetched for that thread as indicated by the fetch logic. There are two copies of each PC for each thread. One at the fetch stage, which indicates the instructions fetched and the other in the context of that particular thread, which indicates the instructions already committed. Whenever a bunch of in-

TABLE I

AVERAGE PROBABILITY OF EACH INSTRUCTION COMMITTING IN THE WINDOW FOR DIFFERENT BRANCH PREDICTION ACCURACIES BASED ON THE ANALYTICAL MODEL. CALCULATIONS BASED ON ASSUMPTION OF AN AVERAGE RUN LENGTH OF 5.

Window Size	Scalar				Multithreading at 4 threads				Multithreading at 8 threads			
	97.2%	93%	90%	85%	97.2%	93%	90%	85%	97.2%	93%	90%	85%
8	0.962	0.906	0.866	0.802	0.972	0.930	0.900	0.850	0.972	0.930	0.900	0.850
16	0.942	0.859	0.803	0.716	0.972	0.930	0.900	0.850	0.972	0.930	0.900	0.850
32	0.901	0.770	0.689	0.572	0.962	0.906	0.866	0.802	0.972	0.93	0.900	0.850
64	0.772	0.627	0.512	0.387	0.942	0.859	0.803	0.716	0.962	0.906	0.866	0.802
128	0.681	0.437	0.328	0.218	0.901	0.770	0.689	0.572	0.941	0.859	0.803	0.716
256	0.515	0.253	0.175	0.110	0.770	0.627	0.512	0.387	0.901	0.770	0.689	0.572

structions are flushed from a thread, the BPU loads the committed PC of that thread into the fetch unit PC. Hence the amount of speculative execution for a thread could be found from the difference of these two PCs. Explanation as to how the state of the machine is kept consistent is given later in this paper.

The design shown in Fig. 2 is the over all logic of the fetch unit for KUMA and does not include the instruction alignment logic or the cache memory configuration. A huge block will increase the hardware complexity of the fetch unit, alignment logic and cache memory. We show the cost, performance trade-offs of the issue width for KUMA.

III. Hardware Organization of KUMA

The hardware organization of KUMA is shown in Fig. 3. As was pointed out in the last section, the issue and the execution stages are decoupled by the use of instruction queues. For eight virtual pipelines we have nine instruction queues; one for each of the pipelines and one for temporary shelving of instructions which the dispatch unit cannot issue in a particular cycle. The length of the queues depends on the instruction fetch unit implementation and the cache memory hierarchy. Each of these queues is tagged with the thread number, so that the dispatch unit knows which thread it is fetching from. The dispatch unit has the same issue width as the fetch unit.

A. Instruction Pipelines and Opcode

The instruction pipelines in KUMA are as shown in Fig. 4. The second fetch cycle is for the dispatch-decode unit. The instructions are decoded twice. First decode is done by the dispatch-decode unit to find the function of the instruction. The second decode is to find out the actual operation. The instruction set format is as shown in Fig. 5. KUMA uses load-store, RISC instruction format. The instructions that do not have the second decode are executed by the Branch Prediction Unit(BPU) as explained later. The instruction opcode in Fig. 6 shows broadly the division between the functional unit and branch/transfer operations. All instructions are 32 bit wide. We did not implement the "load register" instruction since the values are read automatically by the dispatch unit.

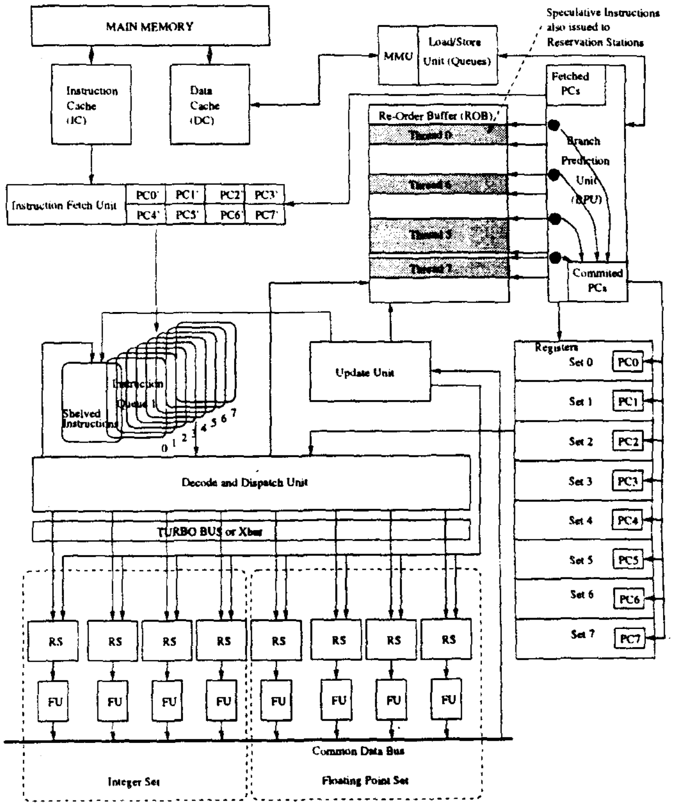

Fig. 3. Hardware Organization of KUMA.

KUMA is designed as an asynchronous architecture and hence is modularized in such a way that each module has it own control logic. This entails the use of self-timed units which are coupled to one another without the strict synchronization of a global clock. Hence each stage of the pipeline as shown in Fig. 4 may not have the same width. Elimination of the global clock has more benefits than are obvious. A global clock is generally implemented as a huge gate (in fact it is the biggest gate on a processor) which switches on and off every cycle irrespective of whether the clock is needed (even when the processor is idle). Usually this is allowed since it is cumbersome to start and stop such a huge clock. None the less, this switching contributes to the maximum dissipation of power (heat) than any other component on the processor (in some cases the heat generated by the clock is more than 50% of the total heat generated and is becoming a higher percentage as the clock speeds are increase and the feature sizes are decrease). Another problem is the clock skew; distributing a centralized clock signal to all the units on a die will not be possible when the signal at high clock speeds (above 1 GHz) can propagate less than $\frac{1}{5}$th of the die. An asynchronous architecture will solve the above problems; it was suggested that a centralized control combinatorial logic on a processor will be an architecture of the past in the coming years.

B. Dynamic Scheduling in KUMA

Dynamic scheduling in KUMA closely follows the Tomasulo's algorithm. This part of KUMA is the heart of the design. It follows the "Data Flow" principles of execution and resolves the conflicts in the hardware. It does out-of-order and speculative execution at the instruction level and is the most scalable part of the design. The negative factors are the increased complexity of interconnects and increase in associative searching.

Interconnects could be drastically reduced by replacing crossbars with busses. The disadvantage of using busses is

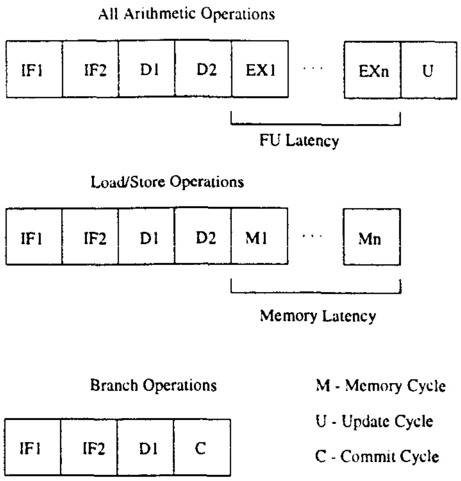

Fig. 4. Instruction Pipeline.

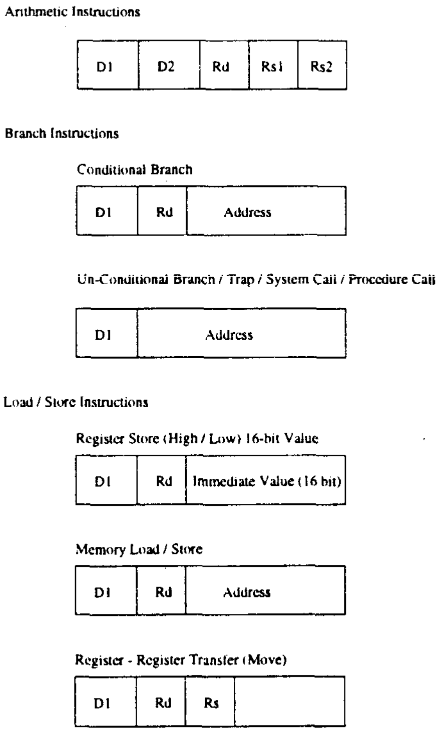
Arithmetic Instructions
D1
D2
Rd
Rs1
Rs2
Branch Instructions
Conditional Branch
D1
Rd
Address
Un-Conditional Branch / Trap / System Call / Procedure Call
D1
Address
Load / Store Instructions
Register Store (High / Low) 16-bit Value
D1
Rd
Immediate Value (16 bit)
Memory Load / Store
D1
Rd
Address
Register - Register Transfer (Move)
D1
Rd
Rs

Fig. 5. Instruction Set and Format.

the classic case of resolution of bus contention. Contention could be reduced by having multiple busses. Since having dedicated busses to connect units is not feasible, the functional units could be appropriately divided into groups to reduce number of busses and contention on them.

Associative searching could also be reduced by dividing the functional units and the contexts into logical sets/groups. We are presently investigating how this sub-grouping of the search space would impact the performance in KUMA.

C. Register Renaming

To facilitate simultaneous execution of threads and to avoid costly context switches, KUMA makes use of multiple register contexts. At the least it uses one context for each virtual pipeline. Hence for an architectural setup of eight virtual pipelines, there are eight contexts. Each register is tagged (except the special purpose registers like the stack pointer, program counter etc) with two counters. One is the "Commit Counter Tag(CCT)" and the other is the "Dispatch Counter Tag(DCT)". The DCTs are with the dispatch unit and the CCTs are with the instruction commit unit of the Branch Prediction Unit(BPU). Logically both the CCTs and the DCTs belong to the context of a particular virtual pipeline.

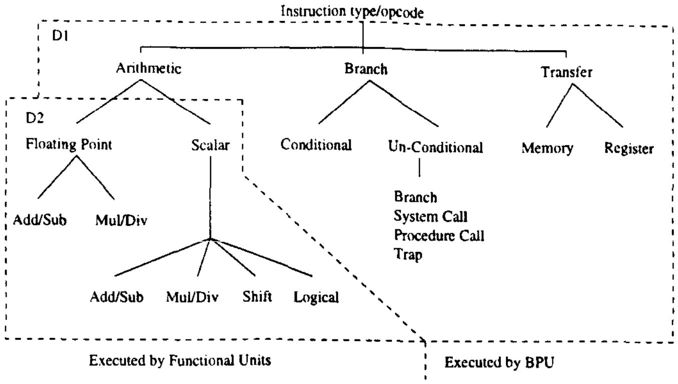

Fig. 6. Example Instruction Opcode for KUMA.

The dispatch unit increments the corresponding DCT of the destination register of any instruction that it is issuing. This is regardless of the prior instructions issued or the source registers of the instruction being issued. Since the dispatch unit issues multiple instructions per cycle, it increments the DCTs of the destination registers of the instructions that it is issuing in that cycle in a sequential manner. Hence a DCT of a particular register could, in the worst case be incremented by the issue width if all the instructions had the same destination register and all the instructions issued have destination registers (for example, arithmetic instructions).

The source operands are requested by the dispatch unit by the register number and the corresponding DCT at the moment of issuing of that instruction. If the DCT matches the corresponding CCT, the contents of the register have the "valid" contents and hence are read into the reservation station entry where the instruction is issued. If the DCT does not match the CCT of the particular register either in the Re-order Buffer (ROB) or the particular register context, the instruction is still issued to the reservation station and the ROB, except that the operand is not made available (and is so indicated by the state of the entries).

With the above scheme, the dispatch unit does not check for the dependencies between the instructions. For a machine type (2,1) [1] which we use for KUMA, Table II shows the number of source operands needed and the number of registers written to per cycle for various issue rates. The table also shows the number of comparisons between the registers that need be done to resolve the dependencies among the issuing instructions. For simplicity we assume all are register-register instructions. If the machine was checking for dependencies to rename the registers and resolve the conflicts, for a window size of 36 instructions, it has to do 1296 comparisons. At the dispatch stage in KUMA, the number of comparisons for a eight issue configuration is 16. The comparisons are done between the counter tags for the corresponding registers. The comparisons are done to check for the availability of the "correct" operands and to fetch them (read) if the counter tags match. The comparisons to resolve dependency conflicts are completely avoided. The dispatch unit does not block the issue of instructions in KUMA due to dependency conflicts. However, when either

[1] If a general machine type is (n, m) , where n,m are always $\geq$ 1, such a machine has instructions reading at most "n" operands and writing results to at most "m" storage components. Machine type (2,1) corresponds to reading at most two operands and writing results to one storage component per instruction.

TABLE II
NUMBER OF OPERANDS AND DEPENDENCIES VERSUS THE ISSUE RATE FOR A MACHINE TYPE OF (2,1).

Issue Rate	Registers accessed		Dependencies
(i)	Destination ($m \cdot i$)	Source ($n \cdot i$)	($n \cdot (i^2 + i)/2$)
1	1	2	2
2	2	4	6
4	4	8	20
8	8	16	74
16	16	32	272

the Reservation Stations are full or the ROB is full, the dispatch unit shelves the instruction(s) and tries to issue it in the next clock cycle or when the thread the shelved instructions are from is scheduled. Our simulations have shown that for a particular issue rate and the number of functional units with their latencies, the sizes of the reservation stations and the ROB could be increased to eliminate this shelving at the dispatch unit altogether. The efficiency of the machine reduces nominally even if the instructions were allowed to be shelved by the dispatch unit.

The request for source operands should be done not only to the register context but also to the ROB of the corresponding thread. The ROB needs to be searched for result values corresponding to the register number and DCT. The search is minimal since only the instructions issued but not committed need to be screened.

D. Branch Prediction Unit

The BPU keeps track of the issued instructions and the committed instructions. In case of a misprediction or termination of execution of a thread, it flushes the issues instructions from the ROB and the Reservation Stations. It commits the instructions in-order from the executed instructions in the ROB and updates the program counter of the thread's context. It checks the flags of the executed instructions for any exceptions and calls the appropriate handler in case of exceptions or interrupts. Hence, precise interrupts are supported by the in-order commitment by the BPU. Under some circumstances the BPU flushes the speculatively issued instructions of that thread from the ROB and the Reservation Stations. The BPU also handles the loads and stores. It issues the requests for loads and stores in-order to the load-store unit. The load-store unit does speculative loads and load/store bypassing using queues. The loads and stores are issued as the BPU is committing them to keep the consistency of data. To avoid the data cache to be a bottle-neck, we need multi-port, non-blocking caches [4].

The state of the thread is kept consistent as we mentioned before by the in-order commitment of the instructions in the ROB by the BPU, even though the instructions are executed out-of-order. The entry of a ROB is shown in Fig. 8. Each entry in ROB is in one of four states at any time; Empty, Issued, Executed or Committed. Empty and Committed states suggest that the ROB entry is free and could be filled by another instruction. The ROB is maintained as a "circular buffer" for each thread. After the result of an operation is made available, the entry holds the contents and the flags of that execution till the BPU commits it. The conditions raised by an instruction executed are checked at the time of commitment according to the "Direct Check Concept". The BPU increments the CCT of the particular register that it writes to. Once committed by the BPU, the instruction is completely executed and the state of that thread is permanently changed.

E. Reservation Stations, Functional Units and Common Data Bus

An entry of a reservation station is shown in Fig. 7. The status entry indicates four states: both operands are not available, first operand is available, second operand is available, and both operands are available (ready for issue). The functional units read from the shelved instructions from the reservation stations either using a round-robin scheme or a simple priority scheme. In the case of the latter, instructions from the threads whose priority is higher are issued to the functional units before the lower priority threads. There is also a mechanism to by-pass at the reservation stations as there is at the dispatch stage when the instructions are by-passed from the instruction queues.

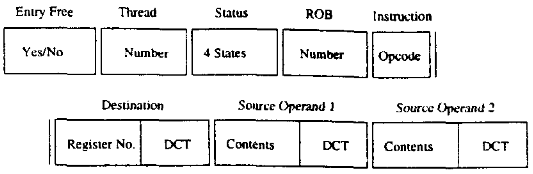

Fig. 7. An entry in the Reservation Station.

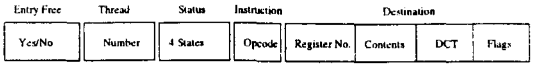

Fig. 8. An entry in the Re-Order Buffer.

The functional units could themselves be pipelined. They take out an entry from the reservation station whenever they are ready (depending on that particular functional unit latency) to process another instruction. In case of long latency functional units, there could be a mechanism to abort the processing of an instruction in the middle of its execution. Though not a necessity, this would allow the BPU to terminate the execution of instructions of a particular thread from the functional units in case the thread was terminated.

The results are put out to the Common Data Bus(CDB) by the functional units. Fig. 9 shows the format of the CDB. The update unit reads from the CDB and updates the entries in the ROB and the reservation stations. To keep the machine from running into a dead-lock, the update unit also needs to attach the results to the shelved instruction queue (not the instruction queues that the dispatch unit has not attempted to issue before) if the source registers and the corresponding DCTs match for a particular thread. If this is not done, the temporarily shelved instructions in the instruction queue might miss the temporary state of a particular register and depending on the instruction dependencies of the thread the machine would eventually run into a dead-lock.

At the update stage, the CDB could be implemented as a crossbar. However the complexity of the interconnections would be high for a large number of functional units. The best case would be to implement the transfer of results from the functional units to the ROB and reservation stations as multiple busses. This would reduce the bus contention with an ideal case of a bus per functional unit, to avoid contention logic on a bus altogether.

F. Program Run on KUMA

Without a mechanism for register renaming or shelving the free-flow of execution is largely hampered in any processor. When a particular register is reserved by a functional unit while it is executing, so that it can write the results of the operation to that register, issuing of any further instructions which make a read or write access to this register would be stalled without the above mechanisms. Correctness of scheduling depends on timing of three actions: issuing, reading of operands, and writing of results.

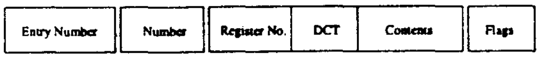

Fig. 9. The Format of the Common Data Bus.

R3 = R1 + R2	add r3, r1, r2
R4 = R4 / R3	div r4, r4, r3
R3 = R1 * C0	mul r3, r1 #100
R5 = R5 + R3	add r5, r5, r3
R3 = R3 + R3	add r3, r3, r3

Fig. 10. A sample program.

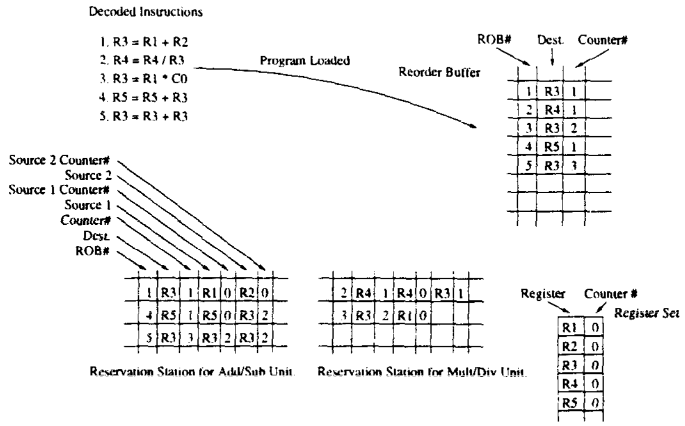

Fig. 11. Dynamic Dependency Resolution in KUMA.

In KUMA we use both register renaming and shelving dynamically to keep the correctness of scheduling. The state is kept consistent by the in-order commitment by the ROB. The dynamic scheduling in KUMA works as a data-flow execution machine entirely dependent on the availability of data. Our aim was to make the execution mechanism in KUMA work for a general case of eight-pipelined architectural model, hence requiring a robust implementation for multithreading.

We take a sample program code as shown in Fig. 10 to show how KUMA resolves the dependencies automatically in hardware. This particular sequence of instructions is shown to run into a deadlock with the original Tomosulo's algorithm by Muller et al [23]. Fig. 11 shows the actual program run. The example shows reservation stations of two functional units: the adder/subtracter and a multiplication/divider. For clarity we show that same ROB entry numbers as the sequence of instructions. Instructions 1, 4 and 5 are issued to the adder/subtracter while instructions 2 and 3 are issued to the multiply/divider. The counter tags with the reservation stations and the ROB are DCTs while the register context counters are the CCTs. There are three references to the register number 3 in the sequence and its corresponding DCT is incremented in sequential order as seen in the ROB. Though register number 3 is read four times in this sequence of instructions, the fifth instruction need not have the two source registers as register number 3 for the original Tomosulo's algorithm to run into a dead-lock. It could be observed that the instructions issued to the reservation stations have the corresponding DCTs of the destination registers when they were issued. This mechanism feeds the right operands to the functional units. The executed instructions are committed in-order at the ROB and the register context holds the appropriate CCTs (the counter associated with the ROB entry is also written to the register context along with the register contents). This scheme works quite efficiently and exploits the ILP of a thread to the maximum, the negative factor being only the associative searching of the thread number, counter tags, and the register numbers. However, total exhaustive searching could be reduced by reducing the size of the thread and counter tags and grouping of reservation stations and sub-dividing register contexts. We are presently investigating this.

TABLE III
INSTRUCTIONS EXECUTED PER CYCLE (IPC) FOR VARIOUS NUMBER OF THREADS AND ISSUE RATES FOR KUMA.

Virtual Pipelines (No. of Threads)	Issue rate			
	1	2	4	8
1	0.66	0.66	0.63	0.58
2	0.79	1.27	1.11	0.88
4	0.88	1.59	2.53	1.01
8	0.94	1.77	3.18	4.5
16	0.96	1.88	3.54	3.28

IV. Estimation

A. Simulation Model

We built the simulator for KUMA as both behavioral and structural models using the Verilog Hardware Description Language. The simulator was again rewritten using C++Sim [24]; KUMA related results presented in this paper were all derived from this simulator. Though this simulator could keep the timing constraints strictly, one negative aspect of this simulator is its relative slowness. We feel that the speed could have been vastly improved if the model was more abstract than conforming to the actual structure of the machine. However, our design followed the structure of the machine so as to give us more scope to validate the architecture. The slowness of the machine is compounded by the simultaneous execution of multiple threads making it time consuming to execute huge benchmark programs. Though our simulator is portable, we have it only running on DEC Alpha with OSF1 because the C++Sim library available to us is only for this platform.

B. Empirical Results

We divided the task of testing the architectural design into validation and performance analysis. To test the design, we ran different programs we wrote. These programs had different granularity of instruction dependencies. If an instruction's result is used in the very next instruction as a source, then we called it a granularity of one. If the result is used by the second instruction after it, the granularity is two, and so forth. For each run of a program of a particular granularity, we check if the state of the context is what we expect. The state of the machine at the end of the execution of a program with any granularity and of any configuration of the machine should be consistent and correct. The machine should not either run into a dead-lock or the state at the end of the execution be incorrect. The architecture of KUMA was thus validated for different programs we handcrafted. KUMA resolves the dependencies dynamically as we explained in the previous sections.

To analyze the performance of the architecture we wrote a program with a granularity of 16. Further, we assumed perfect branch prediction. We avoided the load-store and branch instructions and ran programs with arithmetic instructions. We took a base architecture of 256 entries for Re-Order Buffer, 16 Functional Units with latency of 1 and 64 entries for each Reservation Station. We chose 1 bus each for each of the functional units to update the ROB and the Reservation Stations. Round-robin policy was used to issue instructions at the issue and the dispatch stages. The functional units take out the instructions from the reservation stations in a round-robin fashion too. Though by-passing was used, we did not test the other issuing policies. The choice of the above configuration was to avoid "stalling of issue of instructions" at the dispatch stage, because either the reservation stations or the re-order buffer are full. Table III summarizes our results. The issue rate is varied from 1 to 8 for different number of execution of threads, which are varied from 1 to 16.

The results show that the Instruction Executed per Cycle (IPC) increases as the issue rate and the number of threads increases. In general by increasing the issue rate of the instructions and the number of threads executing simultaneously on the processor, we could greatly enhance the through-put of the processor. The fall of IPC at the edges (for example issue rate of 8 and 4 threads) is because of the configuration parameters of the KUMA chosen and the round-robin policy of scheduling at the execution stage, these cases could be avoided by a proper configuration and issue rate.

V. Conclusions and Future Work

Simultaneous execution of threads on a single chip architecture raises a lot of interesting questions but provides a number of possibilities to the designers. The interactions of the operating system and architecture become more intricate and intertwined. The execution of both User and Kernel threads concurrently, eliminates the need for context switches, at least amongst the active threads. Since modern and some commercial operating systems (Windows NT, Solaris etc) are threaded and modularized, and since more applications are being written using threads, an architecture such as KUMA would be a prime candidate to make use of this trend. And also, KUMA could capitalize on the recent boom in multimedia applications. We briefly discuss some of the issues that affect KUMA.

Owing to simultaneous issue of threads the process structure of the operating system becomes an integral entity of the architecture. The kernel threads could be allowed to execute on one or two virtual pipelines or all of them at the same time. In the same way, processing of interrupts and system calls could be processed exclusively by one or two virtual pipelines or all of them (asymmetric vs symmetric multithreading).

Since scheduling of instructions takes place at couple of stages in the architecture (instruction fetch, dispatch and execution stages) the scheduling policies at these stages should also be managed by the operating system, if they are not already hardwired. In some of the commercial processors available now a days, owing to the complexity of page table entries, at every context switch the TLB is flushed and updated, this lengthens the context switch time. A robust Virtual Memory System maintained by the Memory Management Unit (MMU) under the control of the kernel which allocates the Page Table Entries (PTEs) for all the multiprogramming environment might reduce the time wasted in allocation and deallocation of these tables. However, we perceive a possible increase in virtual memory complexity and protection problems.

In this paper we presented a novel architecture that executes threads simultaneously on a single chip architecture. By doing so, the functional units and resources are shared to get high through put and utilization. This architectural model needs further testing and analysis to make it a feasible alternative to the existing architectural paradigms such as VLIW and Superscalar.

References

[1] R.M.Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units,"IBM Journal, pp. 25–33,Jan 1967.

[2] G.S.Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", "IEEE Transactions on Computers", volume:39, pp. 349-359, March 1990.

[3] James E. Smith, "Decoupled Access/Execute Computer Architectures," Conference Proceedings - The 9th Annual Symposium on Computer Architecture, ,pp. 112–119, April 1982.

[4] Gurinder Sohi and Manoj Franklin, "High Bandwidth Data Memory Systems for Superscalar Processors," Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS-IV), pp. 53-62, April 1991.

[5] K.Olukotun B.Nayfeh L.Hammond K.Wilson and K.Chang, "The Case for a Single-Chip Multiprocessor," Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS - VII), pp. 2–11, October 1996.

[6] T.M.Conte K.N.Menezes P.M.Mills and B.A.Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, pp. 333–344, June, 1996.

[7] Mike Johnson, "Superscalar Microprocessor Design," P T R Prentice-Hall, Inc. Englewood Cliffs, New Jersey 07632, 1991.

[8] Manoj Franklin, "The Multiscalar Architecture - Technical Report 1196," University of Wisconsin Madison, Computer Sciences Department, Madison , WI 53706, 1993.

[9] T-Y Yeh and Y.N.Patt, "Two–Level Adaptive Branch Prediction", Proceedings of the 24th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture. pp. 51–61,November 1991.

[10] V.Popescu M.Schultz J.Spracklen G.Gibson B.Lightner and D.Isaman, "The Metaflow Architecture", IEEE Micro, pp. 10-13 63-72, June, 1991.

[11] J.Lee and A.J.Smith, "Branch Prediction Strategies and Branch target Buffer Design," IEEE Computer, pp. 6–22, Jan 1984. Mountain Research Laboratories, Boulder, Colo., personal communication, 1992.

[12] J.L. Hennessy and D.A.Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, INC., San Mateo, CA, 1990.

[13] H.Dwyer and H.C.Torng, "An Out-of-order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts", Proceedings of the 25th Annual International Symposium on Microarchitecture(MICR-25), pp. 272-281, 1992.

[14] W.W.Hwu T.M.Conte and P.P.Chang, "Comparing Software and Hardware Schemes for reducing the cost of branches", The 16th Annual Symposium on Computer Architecture,pp.224–233,1989.

[15] Peter Song, "Multithreading Comes of Age," MicroProcessor Report, pp. 13-18, July 14th,1997.

[16] Tera Computer Company, "Press Releases," http://www.tera.com/,1997.

[17] H.Hirata K.Kimura S.Nagamine Y.Mochizuki A.Nishimura Y.Nakase and T.Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads", "Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia", pp. 136–145,May,1992.

[18] R.J.Eickemeyer R.E.Johnson S.R.Kunkel B.H.Lim M.S.Squillante and C.E.Wu, "Evaluation of Multithreaded Processors and Thread-Switch Policies", "International Symposium on High Performance Computing (ISHPC '97), Fukuoka, Japan", pp. 75–90, November, 1997.

[19] D.Burger and J.R.Goodman, "Billion-Transistor Architectures", IEEE Computer, pp. 46-49,September,1997.

[20] Y.N.Patt S.J.Patel M.Evers D.H.Friendly and J.Stark, "One Billion Transistors, One Uniprocessor, One Chip", IEEE Computer, pp. 51-57, September, 1997.

[21] Dough Matzke, "Will Physical Scalability Sabotage Performance Gains?", IEEE Computer, pp. 37-39, September, 1997.

[22] Rakefet Kol and Ran Ginosar, "Kin : A High Performance Asynchronous Processor Architecture", To appear in ACM International Conference on Supercomputing, 12 -17 July 1998, Melbourne, Australia.

[23] S.M.Muller and W.J.Paul, "Making the original scoreboard mechanism deadlock free", Proceedings of the 4th Israeli Symposium on Theory of Computing and Systems(ISTCS), IEEE Computer Society, 1996.

[24] Little, M. C., D. L. McCue, "Construction and Use of a Simulation Package in C++", Computing Science Technical Report, University of Newcastle upon Tyne, Number 437, July 1993 (also appeared in the C User's Journal Vol. 12 Number 3, March 1994).