Micro-threading: A New Approach to Future RISC

Chris Jesshope
C.R.Jesshope@massey.ac.nz

Bing Luo
R.Luo@massey.ac.nz

Institute of Information Sciences and Technology,
Massey University, Palmerston North, New Zealand

Abstract

This paper briefly reviews the current research into RISC microprocessor architecture, which now seems to be so complex as to make the acronym somewhat of an oxymoron. In response to this development we presents a new approach to RISC micro-architecture named micro-threading. Micro-threading exploits instruction-level parallelism by multi-threading but where the threads are all assumed to be drawn from the same context and are thus represented by just a program counter. This approach attempts to overcomes the limit of RISC instruction control (branch, loop, etc.) and data control (data miss, etc.) by providing such a low context switch time that it can be used not only to tolerate high latency memory but also avoid speculation in instruction execution. It is therefore able to provide a more efficient approach to instruction pipelining. In order to demonstrate this approach we compile simple examples to illustrate the concept of micro-threading within the same context. Then one possible architecture of a micro-threaded pipeline is presented in detail. At last, we give some comparisons and a conclusion.

1. Introduction

1.1 Where are we now?

When RISC processors first appeared, they focused on regular structure, good design and execution efficiency. In essence they were lean processors, optimised to make the common case as fast as possible and this approach has now been fully vindicated; even state-of-the-art CISC processors translate their CISC instructions into a more efficient RISC core, in order to obtain efficient pipelining. But what are the recent developments in the RISC approach? Although the instruction sets can no longer be called *reduced* the regularity is still present but the implementations are becoming more and more complex. The motivation is to obtain a larger IPC (Instructions executed per cycle), while executing sequential, legacy code. We look at some of these developments before introducing a new approach called micro-threading.

The first approach is super-pipelining, as used in the DEC Alpha. This technique deepens the pipeline by splitting the clock cycle. The gain is in asymptotic processing rate at the expense of a higher pipeline start-up or flush time. The major problem in microprocessor design however, is how to keep the pipeline full and super-pipelining does not address this issue at all. One can view super-pipelining as a vertically split pipeline. The alternative is a horizontal approach, or multiple-issue pipelines, which is used in most recent microprocessors. Here, instructions are issued in parallel to identical or functionally different execution units. Of course, it is possible to apply both super-pipelining and superscalar design to RISC microprocessors but the problem of keeping the pipeline(s) full are now much more severe. This problem is well understood and solved using speculation. Specific techniques used include dynamic instruction scheduling (with register renaming), dynamic branch predication, and more recently, even data-value speculation. Speculative instruction issue introduces write after read and write after write hazards, which would severely restrict instruction issue rates without register renaming. Also with wide or deep pipelines and out of order execution, dynamic branch prediction must be used to allay control hazards [11, 12]. These techniques do increase the processing power of modern RISC processor, but rarely by a factor proportional to the width of the pipeline used. Even with 4 or 8 way superscalar pipelines it is difficult to obtain an IPC count of very much more than 2. Moreover, speculation makes a computer's performance application dependent and large penalties are paid for miss-prediction both in execution time and, perhaps more importantly, in silicon area, as much logic is dedicated to the prediction mechanisms and to miss-prediction clean-up.

It seems possible therefore that RISC is coming to the end of its road, just like the CISC processor's situation about ten years ago. Researchers are now trying to find new ways to obtain more instruction-level parallelism (ILP) in RISC microprocessor design in order to avoid any application dependency in performance, the large penalties for speculative miss-prediction and the large design effort in complex speculative RISC processors.

1.2 Recent Research

In this section we will look at some alternatives before introducing micro-threading.

The Hydra project [3, 4] endeavours to place multiple CPUs onto a single chip. This approach is given the name of Chip Multi-processor (or CMP). The CMP uses a group of small, identical processors each sharing an L2 cache. Inter-processor communication is by bus. The CMP requires the compiler or programmer to find thread-level parallelism and uses both hardware and software speculation. CMP enlarges the instruction window size by exploiting thread-level parallelism.

The M-Machine[5, 6] exploits multi-level parallelism. Its computing nodes are connected with a 3-D mesh network and each node is a multithreaded processor (MAP: multi-ALU processor) incorporating 12 function units, on-chip cache, and local memory. A MAP processor chip contains four 64-bit, three-issue PEs (clusters). PEs and cache-banks are connected by a C-switch and an M-switch. The cache-banks are pipelined with a three-cycle read latency, including switch traversal. The compiler generates a sequence of long instruction words (LIW) called an H-thread which specifies the ILP on each MAP. Four H-threads are called a V-thread for coarser granularity in each MAP node. The M-Machine exploits thread-level and instruction-level parallelism in a relative coarser way. However, the latency in the cache and processor may be a bottleneck to performance.

SMT (Simultaneous Multi-threading) processor [14, 15] is designed to exploit both threading-level parallelism and instruction-level parallelism by adding an eight threads hardware context and eight-instruction fetch/decode mechanism to widen the instruction window size. The major innovation is in the fronted-end, which supplies instructions. Each thread can address 32 dedicated integer (and floating-point) registers. In addition to this SMT has an additional 100 integer and floating-point renaming registers. Because of the access time of the lager SMT register file, the SMT pipeline must be extended by using a two-cycles register reads and a two-cycle register write.

The DMT (Dynamic Multithreading Processor) [7] also uses an SMT pipeline to increase processor utilisation, except that the threads are created dynamically from the same program. The hardware breaks up a program automatically into loops and procedure threads that execute simultaneously on the superscalar processor.

The Trace Processor [1, 2] is derived from the Multiscalar processor [8]. The Trace Processor extends the instruction window size to a *trace*, where traces are dynamic instruction sequences constructed and cached by hardware. Trace Processors distribute the instruction window and register file to solve the instruction issue and register file complexity problems found in designs such as the SMT. Because traces are neither scheduled by the compiler nor guaranteed to be parallel, they still rely on aggressive control speculation and memory dependency speculation. The main difference between the Trace processor and the Multiscalar processor is that the traces in the Trace Processors are built as the program is executed, whereas the tasks in the Multiscalar Processor require explicit compiler support.

1.3 Compiler Solutions

There are two issues in exploiting ILP in microprocessor design. The first is the detection of concurrency and the second is the scheduling of concurrency. Detection of concurrency is well understood and there is a large body of literature on the topic, for example[17-19]. Concurrency can be enhanced by eliminating storage dependencies and by techniques such as loop unrolling and software pipelining. The unit of code normally analysed is the basic block or the largest unit of code having a single entry and single exit point. Techniques used include loop transformation, loop interchange (loop permutation), skewing, reversal, re-indexing (alignment or index set shifting), distribution, fusion, scaling, and statement reordering. Many of these techniques were developed for vector computers, where extensive parallelism is extracted from sequential code was. However, unless an ISA has explicit concurrency, this detection of concurrency can only be used to reorder instruction and provide slackness in the instruction sequence in order to aid dynamic extraction and scheduling of concurrency.

VLIW architectures, including the new Intel Merced architecture use compiler detected concurrency to perform *aggressive static code scheduling* but this solution also requires speculation although usually implemented through the use of guard bits instead of speculative branches. This does not solve the basic problem however, which is that some operations are inherently non-deterministic, such as conditional execution and memory access time. Only using hardware-based, asynchronous scheduling can the use of explicit concurrency overcome non-deterministic instruction execution. The question is can hardware scheduling be made sufficiently efficient so as not to cause large overhead?

1.4 The Micro-threading Approach

In early 1996, one of us proposed a new way to exploit more ILP in RISC microprocessor [9] using a technique called micro-threading. The fundamental assumptions of that work was to keep the micro-architecture simple by avoiding, where possible, any speculation. The pipeline was kept full by allowing instructions to be executed from one of two possible sources. Where instruction execution was deterministic, the next instruction could be provided by the current thread in the normal manner. However, in situations where speculation would normally be required, either through control or data-dependency, it is not possible to execute the next instruction from the same thread without using speculation. In these circumstances the micro-threaded pipeline issues the next instruction from a new thread **without having to flush the pipeline**. Because the threads are drawn from the same context, all thread-based operations have a very low overhead, which does not require a register context exchange.

In effect this approach represents a thread by just a program counter and because of this requires a very low overhead for thread-based operations, such as thread creation, synchronisation and termination. The non-deterministic events that cause a new micro-thread (program counter) to be executed will be: a cache miss, where the loading thread will be suspended pending the arrival of off-chip; explicit synchronisations instructions and conditional branches. As creation and scheduling overheads are insignificant, micro-threads can be rather small, possibly just a few instructions. The threads can easily be derived from within a single context using current compiler technology either from expression concurrency or from separate iterations of a loop. The overheads of thread creation, termination, and synchronisation are extremely low and comparable to that of a conditional branch, the normal means of executing multiple loop bodies.

Thus this approach is firmly in the category of compiler assisted rather than hardware assisted and requires source code to be re-compiled for any processor using it. At the very least compiled code must be pre-processed to introduce explicit concurrency.

In micro-threading threads are created dynamically by instructions that generate a new program counter when executed. A PC is the sole state of a micro-thread, which will be stored in a data structure to await execution. The instruction supply is therefore based on this queue of currently active program counters (continuation queue or CQ) and will be sufficient to feed the pipeline in most circumstances. As already described[9], this approach replaces the use of prediction in super-scalar design. Miss-prediction is avoided by PC interleaving. Thus the high cost of miss-prediction is avoided and performance will **not** be application dependent. Typical overheads are zero cycles for a context switch and just one cycle to create a new thread at run time.

Micro-threading uses a classical hardware supported, sleep-wakeup synchronisation mechanism to solve the problem of the micro-thread's synchronisation and interleaving. The mechanism makes use of two state bits associated with each register giving *empty*, *full* and *waiting* states. Program counters of suspended threads are stored in the empty register, when it is in the waiting state, pending data or a wake-up signal.

In Section 2, we give some examples of code generation for a micro-threaded microprocessor. Then in Section 3 we will introduce a possible architecture to exploit this form of micro-threading in current and next generation chips, where instead of using silicon area for complex speculation techniques it is used for multi-processing.

2. Micro-threading

2.1 An embarrassingly parallel example

In this first example we compile a simple parallel loop to a representative RISC ISA and then consider how the code may be micro-threaded. The C program we have assumed for the first example is as follows:

```
for ( i=0; i<63; i++)
    x[i] = x[i] +1;
```

We translate the C program using MIPS instruction set as below [10,11,12,13,and 16], assuming that there are 32 general registers; X-start is x[i]'s base address; 1 is stored in $2; i is stored in $4; and loop boundary number 63 is stored in $3.

```
        mv  $3, 63          # $3 =63, loop boundary
        mv $4, $0           # $4 = i = 0
        add $2, $4, 1       # $2 = 1
loop:   mult $5, $4, 4        # Byte offset for i
        lw  $6, X_start($5) # Temporary reg $7 = x[i]
        add $6, $7, $2
        sw $6,    X_start($5)# x[i] = x[i] + 1
        add $4, $4, $2        # i = i + 1
        blt $3, $4, loop      # if i<=63, continue loop
```

To compile to a micro-threaded MIPS ISA we use a modification of loop unrolling, where the unrolled loops comprise the separate thread, as shown in figure 1. Six threads are used and each increments the loop counter by 6 rather than 1. The main thread just creates and waits on the 6 threads used to execute the loop.

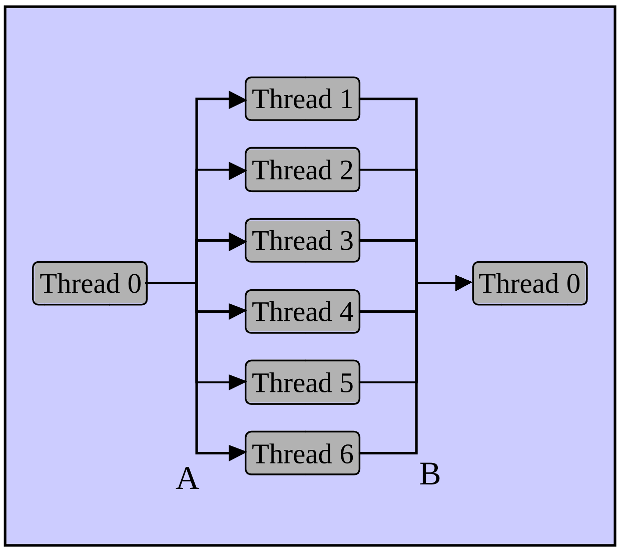

Figure 1. Thread flow diagram

Upon initial execution there will only be a single program counter. The other program counters will be generated on execution of thread create instructions: **crt, crteq**, **crtne**. The create instructions are analogous to a branch instruction **j, beq, bne** in the MIPS instruction set. They use the address literal to generate a new program counter, while executing the next instruction from $PC = $PC+4.

Each thread must have a separate set of register resources allocated for local storage. In this example, the first 4 threads will calculate 11 iterations each and the last two 10 each. In terms of concurrency, we could have created an thread for each iteration but the number of threads we can allocate (to a single processor) will depend on the resources we have available (assumed to be 32, as in the MIPS ISA). Note that one register for each thread is used for synchronisation.

```
th0:  reset_all              # sets all registers to empty
          mv  $3, 15          # $3 = 63, loop boundary
          mv $4, $0           # $4 = i = 0 for thread 1
          add $2, $3, 1       # $2 = 1
          mv $28, 6           #$28 = 6 skip for each thread
          crt th1             # create thread_1
          add $8, $4, 1       # $8 = i  for thread 2
          crt th2             # create thread 2
          add $12, $4, 1      # $12 = for thread 3
          crt th3             # create thread 3
          add $16, $4, 1      # $16 = i for thread 4
          crt th4             # create thread 4
          add $20, $4, 1      # $20 = i for thread 5
          crt th5             # create thread 5
          add $24, $4, 1      # $24 = i for thread 6
          crt th6             # create thread 6
          wait $7             # wait for thread 1 to complete
          wait $11            # wait for thread 2 to complete
          wait $15            # wait for thread 3 to complete
          wait $19            # wait for thread 4 to complete
          wait $23            # wait for thread 5 to complete
          wait $27            # wait for thread 6 to complete
          …                   # main thread continues
th1:      mult $5, $4, 4      # Byte offset for i
          lw  $6, X_start($5)# Temporary reg $7 = x[i]
          add $6, $6, $2
          sw $6, X_start($5)# x[i] = x[i] + 1
          add $4, $4, $28     # i = i + 6
          blt $3, $4, th1     # if i<=64, continue loop
          set $7              # set synchronisation register
          kill                # terminate thread_1
th2:      mult $9, $8, 4        # Byte offset for i
          lw  $10, X_start($9)# Temporary reg $7 = x[i]
          add $10, $10, $2
          sw $10, X_start($9)# x[i] = x[i] + 1
          add $8, $8, $28     # i = i + 6
          blt $3, $8, th2     # if i<=64, continue loop
          set $11             # set synchronisation register
          kill                # terminate thread_2
th3:      mult $13, $12, 4        # Byte offset for i
          lw  $14, X_start($13)# Temporary reg $7 = x[i]
          add $14, $14, $2
          sw $14, X_start($13)# x[i] = x[i] + 1
          add $12, $12, $28# i = i + 6
          blt $3, $12, th3    # if i<=64, continue loop
          set $11             # set synchronization register
          kill                # terminate thread_3
          etc.
```

Bold instructions are those that have been added to the MIPS instruction set and instructions in italics are non-deterministic and cause a transfer to a new thread from the CQ, rather than executing the next instruction in the current thread. It can be seen that the code in the body of the original loop is exactly the same as that in each thread. Any non-deterministic instructions cause the next instruction to be taken from the PC at the head of the CQ. When a dependent instruction attempts to read the register, it either finds it empty and suspends the thread by putting its PC in the register or finds it full, in which case the PC goes back to the CQ to continue execution at some stage (note that this happens at the register read stage of the pipeline).

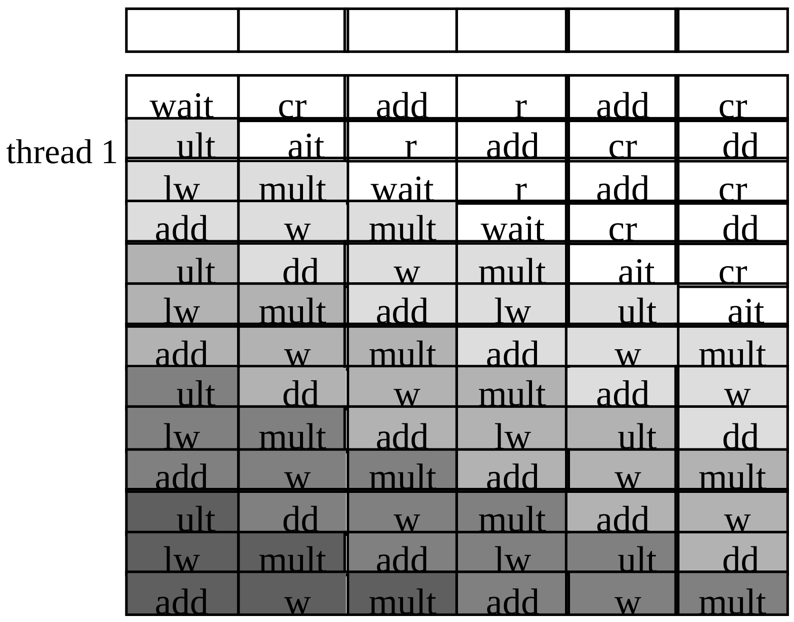

Figure 2. Shows the interleaving of threads in the pipeline on non-deterministic instructions (the adds). Each lw has 16 cycles to complete without stalling.

Initialising, creating, killing and synchronising a thread gives a five cycle overhead, in this case amortised over 10 iterations of a six instruction loop: one cycle to compute the start value of i for the thread, one to create the thread, **crt**, one to send a signal when complete, **set**, one to kill the thread, **kill,** and a **wait**. In the main thread. In the worst case we could have one iteration per thread 9 cycles per thread (the last two instruction in the loop would be redundant in this case), giving a 50% overhead for this very simple loop. In [9] we show how the single cycle for the thread create and thread kill could be eliminated altogether by using an encoding scheme, not backward compatible with a base instruction set. This would reduce the overhead to 7 cycles per loop in the worst case, one more than the original loop.

Although there is an overhead, this code will never stall on a cache miss or on a conditional branch, unless all 6 threads are awaiting data or a branch condition. Figure 2 shows the thread interleaving from the suspension of thread 0 onwards. It shows that each lw has 16 cycles to complete i.e. 3 instruction in each of 5 other threads plus one. Thus if all lw instructions missed the cache, which may be the case on the first iteration, 16 clock cycles are available before the pipeline stalls. The bnes only 3 cycles as the branch condition is computed at stage 3 of the pipeline. This code therefore will never stall, unless memory accesses take more than 16 clock cycles.

2.2 A Non-parallel Loop Example

The second example contains a dependency between loop iterations and would cause a vector compiler to generate sequential code. We show how threaded parallelism can still be used to hide the memory latency. The C code is as follows:

```
A[0] = 0; a[limit+1] = 0;
For (I = 0; i <= limit; i++)
        A[i] = c1*a[i-1] + c2*a[i] + c3[a[i+1];
```

Again the code is compiled to 6 threads, each representing one iteration of the unrolled loop. A register allocation for a[i] is given in the table below, there being a dependency between the third row of each column and the top row of the following column, shown for i = 1 -> 2. a[i]' is the new value of a[i] and this register is used to synchronise between threads.

	I =1 etc	i=2 etc	i=3 etc	i=4 etc	i=5 etc	i=6 etc
a[i-1]	$17	$7	$9	$11	$13	$15
a[i]	$18	$8	$10	$12	$14	$16
a[i] '	$7	$9	$11	$13	$15	$17
a[i+1]	$8	$10	$12	$14	$16	$18
i	$19	$20	$21	$22	$23	$24

In this second example the main thread executes iteration I=1,7, etc. and 5 other threads are created.

```
     reset_all            # sets all registers to empty
     lw$25 limit($0)      #$25 = limit
     sll $25, $25, 2      #convert to byte address
     lw$4 c1($0)          #$4 = c1
     lw$5 c2($0)          #$5 = c2
     lw$6 c3($0)          #$6 = c3
     sw $0, a($0)         #a[0] = 0
     sw $, a($23)         #a[ limit] = 0
     mv $19, 1            # i=1 for thread 0
     mv $26, 24           #$26 = skip for each thread
     crt th1              # create thread_1
     add $20, $19, 1      # i=2 for thread 1
     crt th2              # create thread 2
     add $21, $20, 1      # i=3 for thread 2
     crt th3              # create thread 3
     add $22, $21, 1      # i=4 for thread 3
     crt th4              # create thread 4
     add $23, $22, 1      # i=5 for thread 4
     crt th5              # create thread 5
     add $24, $23, 1      # i=6 for thread 5
     lw $17 a($0)         #a[i-1]
     lw $18 a($19)        # a[i]
th0: lw $8 a+1($19)       # a[i+1]
     mul $18, $18, $5     #c2*a[i]
     mul $17, $17, $4     # c1*a[i-1] - wait th5
     add $17, $17, $18    #1st two terms
     mul $18, $8, $6      #c3*a[2]
     add $7, $17, $18     #new a[i] set here - signal th1
     reset $17            #reset $17  for next round
     sw $7, a($19)        #store new value of a[i]
     add $19, $19, 24     #increment i by 6(*4)
     blt $19, $25 th0     #loop if I < limit
     …                    # main thread continues

th1: lw $10  a+1($20)     # a[i+1]
     mul $8, $8, $5       #c2*a[i]
     mul $7, $7, $4       # c1*a[i-1] - wait th0
     add $7, $7, $8       #1st two terms
     mul $8, $10, $6      #c3*a[2]
     add $9, $7, $8       #new a[i] set here - signal th2
     reset $7             #reset $7  for next iteration
     sw $9, a($20)        #store new value of a[i]
     add $20, $20, 24     #increment i by 6(*4)
     blt $20, $25 th1     #loop if I < limit
     kill
th1: lw $12  a+1($21)     # a[i+1]
     mul $10, $10, $5     #c2*a[i]
     mul $9, $9, $4       # c1*a[i-1] - wait th0
     add $9, $9, $10      #1st two terms
     mul $10, $12, $6     #c3*a[2]
     add $11, $9, $10     #new a[i] set here - signal th2
     reset $9             #reset $9  for next round
     sw $7, a($19)        #store new value of a[i]
     add $19, $19, 24     #increment i by 6(*4)
     blt $19, $25 th1     #loop if I < limit
     kill
     etc…
```

2.3 Micro-threading on a multi-cpu

There are a number of different problems in designing a multi-threaded multi-cpu, which share L2 cache. The major principle decision is whether a single or multiple register file should be used. Using a single register file means a potentially slow register access and possibly two pipeline cycles for an access, as in the SMT[14,15]. In addition there will a large area overhead for multiple read ports to support all of the cpus. On the other hand there are also difficulties in implementing multiple register files, one per cpu. Firstly there is a requirement for inter-cpu synchronisation and possibly data transfer. This is not so difficult to implement, however.

What poses the greater problem is the requirement to statically allocate resources as shown in the code above, where register identifiers are allocated by the compiler for different threads executing the loop body. This means that the compiler would effectively have to make a decision regarding thread allocation to cpu, without regard to the resolution of non-determinism and hence issues of load balancing. Clearly this is not a good solution.

Consider the code above, effectively we have 6 instances of exactly the same loop (excepting the register resources not shared by the parallel threads). Ideally we would like to write one parametric loop body and then instance it as many times as the compiler thought necessary (i.e. depending on the number of cpus). This situation is analogous to register renaming, where additional registers are used to remove write after read and write after write hazards. Thus we propose a two level CQ. The first level holds ready threads that have not yet been allocated to a cpu, there register requirements are generic subject to the limit of registers in one cpu's register file. From here threads are allocated to a cpu but only when that cpu would otherwise have stalled. Once allocated, the thread runs to completion on that cpu and is held in the cpus own CQ, the second level. Thus there is a pool of threads in the master CQ and within each cpu a CQ holding threads allocated to it. The allocation must ensure that register resources are available on the cpu for the thread being allocated and must rename the thread's local registers from their generic form to the actual registers allocated.

3. Micro-threading Architecture

In this section, we present a possible micro-threading architecture. A single micro-threaded CPU is shown in figure 3. This is conventional, except in instruction fetch, where the continuation queue provides a window of ready to execute threads. Each cpu also has a bus to the global register file. Figure 4 shows the multiple CPU organisation. Notice that each cpu shares the level 1 data cache but contains its own L1 instruction cache. The L2 cache is also shared, as is the interface to external RAM. It is expected that all components, with the possible exception of RAM and possibly L2 cache are on a single silicon die.

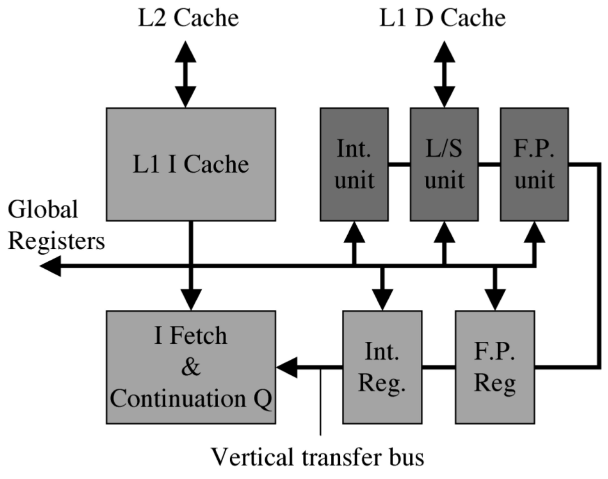

Figure 3 Micro-threading processing unit (PU) internal structure

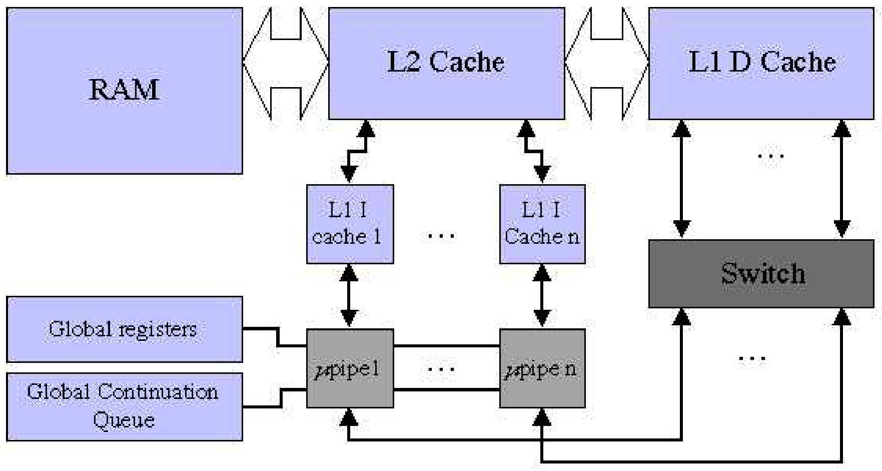

Figure 4 Micro-threaded multiprocessing CPU architecture

From Figure 3 and Figure 4 it can be seen that there is a two level instruction supply mechanism, the global and local CQ. The register file also comprises a global and local register file. The global register file contains the global temporary variables. It can be read simultaneously but must be written in proper order. The local registers on the other hand act as a generic RISC register file and stores the local temporary variables of the threads running on that cpu. Both local and global registers have synchronisation bits associated with them.

Stage 0 is the most critical in the design of the micro-threaded pipeline because of the PCs in two-levels of CQ. It may require a further prefetch stage to be added to maintain performance. This stage would be responsible for thread allocation and renaming.

4. Conclusion

In this paper, we present a micro-threaded RISC processor that can potentially achieve more ILP than a standard RISC pipeline and also more than in the use of superscaler enhancements to a conventional RISC pipe. This is because speculation is totally removed and replaced by thread interleaving. Speculation has high cost in silicon area and in execution time, when prediction fails. This of course is application dependent but most research shows ILP to be limited in the superscaler approach. It has been shown here that concurrency can be extracted from loops, even when there exist dependencies between iterations, as a compiler can almost always find some instructions in each loop which can be executed prior to the dependency being encountered.

The main problem with thread interleaving is normally in the overheads asociated with either the interleaving or the execution of the additional instructions supporting threading. We have studied the cost of micro-threading in [9], which show the cost is very small. This is illustrated in the hand-compiled code in this paper.

We are currently studying multiple cpu micro-threading architectures in detail and are developing a simulator for detailed evaluation and comparison.

5. References

[1] E. Rotenberg, Q. Jackson, Y. Sazeides, and J.Smith, “Trace Processors”, The 30th International Symposium on Microarchitecture, pages 138-148, December 1997

[2] James E. Smith and Sriram Vajapeyam, “Trace Processors: Moving to Fourth-Generation Microarchitectures”, pp. 68-74, IEEE Computer, Vol. 30, No. 9, September 1997

[3] L. Hammond, B.A. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor”, IEEE Computer, 29(12): 84-89, December 1996

[4] Kunle Olukotun, Lance Hammond, and Mark Willey, “Improving the Performance of Speculatively Parallel Applications on the Hydra CMP”, Proceedings of the 1999 ACM International Conference on Supercomputing, Rhodes, Greece, June 1999.

[5] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee, “The M-Machine Multiprocessor”, The 28th International Symposium on Microarchitecture, pages 146-156, November, 1995

[6] S.W. Keckler, W.J. Dally, D. Maskit, N. P. Carter, A. Chang and W.S. Lee, “Exploiting fine-grain thread level parallelism on the multi-ALU processor”, The 25th Annual International Symposium on Computer Architecture, June 1998

[7] H. Akkary, M. Driscoll, “A Dynamic Multithreading Processor”, 31st Annual ACM/IEEE International Symposium on Microarchitecture, Nov. 30 - Dec. 2, 1998

[8] G. S. Sohi, S. Breach, and T. N. Vijaykumar, “Multiscalar Processors”, 22nd International Symposium on Computer Architecture (ISCA-22), 1995

[9] A.Bolychevsky, C.R.Jesshope, V.B.Muchnick, “Dynamic scheduling in RISC architectures”, IEE Proc.-Comput. Digit. Tech. Vol.143, No.5, September 1996

[10] R10000 Microprocessor User's Manual, Copyright 1996, 1997, MIPS Technologies, Inc. -- 09 DEC 96, Version 2.0

[11] PowerPC 604/604e User's Manual, MPC604EUM/AD, Motorola Inc., 2/24/98

[12] J L Hennessy & David A Patterson, “Computer Architecture-A Quantitative Approach”, 2nd ed., 1996

[13] David A. Patterson and John L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann Publishers, Inc., San Francisco, California, February 1999

[14] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen, “Simultaneous Multithreading: A Platform for Next-generation Processors”, IEEE Micro, September/October 1997, pages 12-18.

[15] Jack L. Lo, "Exploiting thread-level parallelism on simultaneous multithreaded processors", Ph.D thesis, Department of Computer Science and Engineering, University of Washington, 1998

[16] R4400 Microprocessor User's Manual, Copyright 1996, MIPS Technologies, Inc. -- 21 MAR 96, 2nd Edition

[17] Amy W. Lim , Monica S. Lam, “Maximizing parallelism and minimizing

synchronization with affine partitions”, Parallel Computing 24 1998 445–475, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA, Parallel Computing, Vol. 24, Issue 3-4, May 1998, Pages 445-475

[18] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, “Compiler Technology for Future Microprocessors”, Proceedings of the IEEE, Vol. 83, No. 12, December 1995, pp. 1625-1640.

[19] Jack Lo, Susan Eggers, Henry Levy, Sujay Parekh, and Dean Tullsen, “Tuning Compiler Optimizations for Simultaneous Multithreading”, In 30th Annual International Symposium on Microarchitecture (Micro-30), Dec. 1-3, 1997, p. 114-124.