Simultaneous Multithreading RISC Processor with Non-blocking Load/Store

Hao-Sheng Chen, Fang-Yu Shiu, Yi-Chao Chan, Tien-Fu Chen
Department of Computer Science and Information Engineering,
National Chung Cheng University
Chia-Yi, Taiwan (R.O.C)
{chs89u, hfy94, cych91u, chen}@cs.ccu.edu.tw

ABSTRACT

This paper proposes a simultaneous multi- threading RISC processor with non-blocking load/store. Many applications exhibit multi-tasking characteristics, such as parallel data operations in video and audio codec. But traditional RISC processor can not take advantage of this inherent parallelism. Instead, multithreading technique enables more than one instruction string to be active in the CPU. Its ability to share hardware resource and hide memory latency would improve performance and efficiency.

While multithreading processor is good at multi-processing, it has the limit on issue- bandwidth and throughput. In this paper, we design a 4-way, 2-issue SMT RISC processor to improve that with low design complexity and low area increment incurred. Besides, we also provide non-blocking load/store to hide memory latency. Finally, the clock rate of SMT RISC processor can reach of 210MHz.

Keyword:

Helper RISC, SMT (Simultaneous Multi-Threading), non-blocking load/store

1: INTRODUCTION

Our simultaneous multithreading RISC processor is like a general RISC processor. It could support main RISC to accelerate the complex computation. Therefore, this SMT RISC processor could mitigate the burden of computation of main RISC. We implement it in a core, named VisoMT (VLIW, SMT with Open configurable Multi-Threading). The VisoMT is a programmable embedded processor core aiming at multimedia application. It has two major clusters, main RISC cluster, and heterogeneous multithreading cluster. The main RISC cluster supports operating system and controls the application program flow. Heterogeneous multithreading cluster accelerates the complex computation, and is composed of four accelerate function units (AFU) and a light RISC. There are four hardware threads in heterogeneous multithreading cluster, and each composes an AFU and a light RISC in light weight VLIW.

In VisoMT, there are several accelerative threads derived from main thread. Besides the AFUs taking charge of complex computation, a light RISC processor is essential to handle all other tasks of the threads, such as: program control, load/store, and specific configuration.

We denominate a name, Helper RISC, for this light-weight RISC processor. As the Figure 1, we divide Helper RISC to five stage pipeline, and add the three stage pipeline of Front-end, the total is eight stage pipeline. Helper RISC is shared by multi-thread, maximum is four thread, so each stage would be different thread. Furthermore, on the inside of Helper RISC, we partition it into two sides. The right side is the operation of load/store and special move engine, the left side is computation and program control engine (see Figure 3).

Figure 1: Pipeline stage

In general condition, we could simultaneously issue two operators of different thread. This 2-issue SMT Helper RISC could alleviate the competition of the usage of main RISC processor among the threads. We also provide non-blocking load/ store to prevent from harmful cache miss stall. Furthermore, repeat instruction is supported to allow load/store instruction to be repeated up to, say, 32times. It avoids loop overhead, and is useful in moving block of data in multimedia application.

Each thread has two banks of data register, one stores source data, and the other stores data after computation. Owing to the characteristic of multimedia, data after computation of one thread may be used by another thread. We can switch the data banks between the two threads instead of exchanging data via memory. We provide large data register banks and the switch circuit to accomplish this. And each thread could set its mapping of data bank via the instruction of Helper RISC.

The remainder of this paper is organized as follows. In Section 2, we review these related works of multithreading processor design. After that, we propose our design of Helper RISC in Section 3. Section 4 presents the some experimentation. Finally, in Section 5, we draw our conclusions.

Table 1: A comparison of key hardware complexity features of various models

Model	Register port	Inter-inst Dependence checking	Forwarding Logic	Instruction Scheduling onto FUs
Fine-Grain	H	H	H/L	L
SMT: Single Issue	L	None	H	H
SMT: Dual Issue	M	L	H	H
SMT: Four Issue	M	M	H	H
SMT: Full Issue	H	H	H	H
SMT: Limited Connection	M	M	M	M
Our SMT RISC	L	None	M	M

2: RELATED WORK

Alpha EV8[5] is the most powerful and ambitious microprocessor yet proposed. It is a 4-way multithreaded and 8-way superscalar RISC microprocessor with simultaneous multithreading. However, it is cancelled prior to release for manufacturing. Niagara[3] is a 32-way multithreaded SPARC processor. It supports 32 hardware threads by combing ideas from chip multiprocessor and fine-grained multithreading. The architecture organizes four threads into a thread group. Each of the thread group shares the pipeline and operates like a 4-way multithreaded processor. Eight groups result in 32 threads on the CPU. It is effective in commercial server applications such as database and Web servers, which tend to have workloads with large amount of thread level parallelism. It is important to us to find a suitable architecture for our multithreading RISC[6][11][1]. Previous research has defined several machine models for simultaneous multithreading, spanning a range of hardware complexities.

In [10], it shows some important difference in hardware implementation complexity between these machine models, as shown in Table 1.

In Table 1, H means high complexity. Fine-Grain model means only one thread issues instructions each cycle, it doesn't match simultaneous multithreading. From SMT: Single Issue to SMT: Full Issue, the 4 models have different limit on the number of instructions each thread can issue per cycle. We can see that the more instructions a thread can issue, the more complexity it will have in hardware feature. Limited Connection in the last row represents that each hardware context is connected exactly to particular functional units. It's medium complexity due to its limited connection.

In this paper, in order to minimize hardware complexity, we choose SMT: Single Issue, that is, each thread can issue only one instruction per cycle. Under the circumstances, it is impossible for two instructions of the same thread to access the same register file.

Consequently, register ports have no need to grow. Besides, inter-instruction dependence checking can be absolutely omitted.

3: HELPER RISC

Helper RISC is a shared function unit. It is designed particularly for our proposed heterogeneous SMT architecture. In this paper, we focus on design the Helper RISC and effective utilization of the resource, particularly the load/store and arithmetic instruction.

3.1: Overview of Helper RISC

Helper RISC performs program control and load/store option of that thread. The four thread can executes in 2-way VLIW. Figure 2 shows the diagram of the heterogeneous SMT.

In our VisoMT (VLIW, SMT with Open configurable Multi-Threading), accelerative threads are started by cluster1 RISC processor. After that, the program flows of those threads are controlled by Helper RISC in their life period. However, the internal pipeline stages of Helper RISC would be comprised of instructions coming from different threads. The ability to recognize which thread is in which pipeline stage, and which context should take into account is critical in our architecture design.

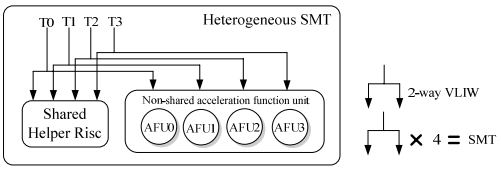

Figure 2 Diagram of heterogeneous SMT

Because the numbers of read/write port is critical to the design of cache and memory, it is unreasonable to let each thread has its own read/write port. Instead, we unify the load/store interface through Helper RISC. In other words, load/store operations of all the threads are accomplished by Helper RISC. Furthermore, we provide repeat load/store to repeat the operation with only one instruction to reduce code size. And we provide non-blocking load/store to prevent the harmfulness of stalling the whole machine just because of cache miss from one thread.

In our VisoMT architecture, each thread has two banks of data register, one stores source data, and the other stores data after computation. Owing to the characteristic of multimedia, data after computation of one thread may be used by another thread. We can switch the data banks between the two threads instead of exchanging data via memory. We provide large data register banks and the switch circuit to accomplish this. And each thread could set its mapping of data bank via the instruction of Helper RISC.

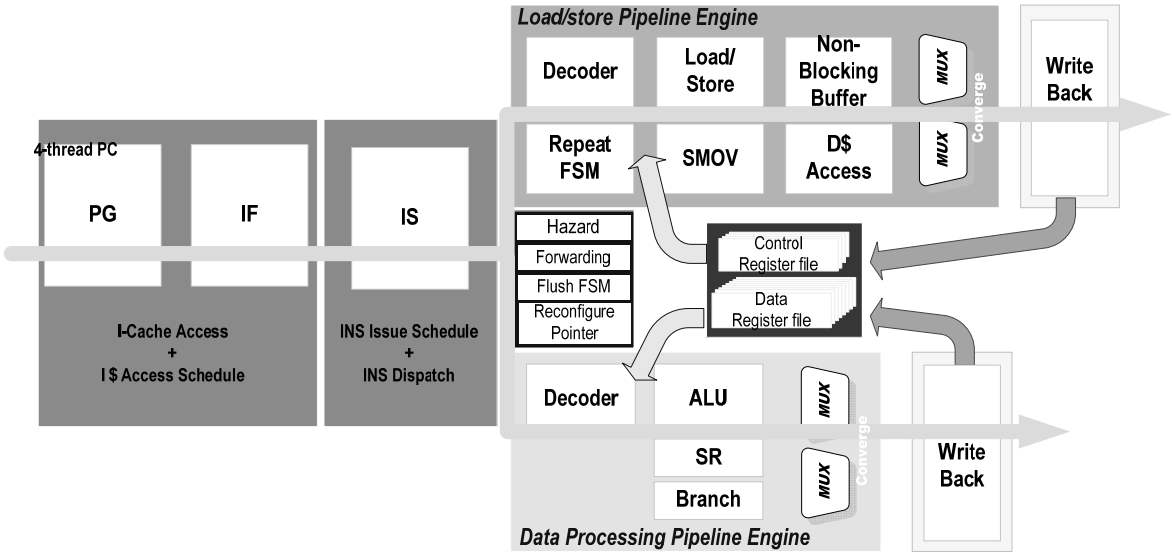

Figure 3: Architecture of Helper RISC

3.2: Front-end

As Figure 3, Front-end is composed of three stages: PC generation, instruction fetch, and dispatch. Each thread has its own program counter. In PC generation stage, each program counter is updated according to the information such as: create a new thread, branch, or usual program counter update. A thread select multiplexer determines which of the four thread program counters would perform instruction cache access.

In instruction fetch stage, 128bits instruction string is fetched from instruction cache and store into instruction queue. The 128 bits instruction string includes two instruction bundles which is 64bits. An instruction bundle contains three instructions with 1bit space unused. An instruction is 21bits, bit[0] determines instruction type, value 0 represents Helper RISC instruction, and value 1 represents AFU instruction. Figure 4 shows instruction string composition described above.

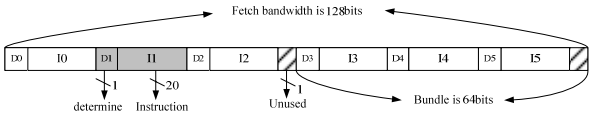

Figure 4: Instruction string composition

In dispatch stage, instructions which in the instruction queue are issued in VLIW mode. The VLIW issue packet contains 2 instructions. It may be a Helper RISC instruction with an AFU instruction, a Helper RISC instruction with a NOP, or an AFU instruction with a NOP.

Furthermore, four threads can issue instructions concurrently, that is, instructions in the four issue packets are issued concurrently[7]. However, we adopt heterogeneous SMT instead of conventional SMT to reduce design complexity. In heterogeneous SMT, the partition of functional units among threads is less flexible than conventional SMT. In our design, Help RISC is shared, so Helper RISC instructions of all the 4 threads are issued to it. But AFU is non-shared, it is dedicated to particular thread. For example, AFU instructions of thread 0 can only be dispatched to AFU 0.

3.2.1: Dynamic Scheduling. In our 4 way multithreading architecture, each thread can issue Helper RISC instruction concurrently. Consequently, there are at most 4 Helper RISC instructions competing for the rights to the usage of Helper RISC. But Helper RISC is a 2 issue SMT RISC, it can only consume at most 2 instructions per cycle. As a result, we need a scheduler to select 2 instructions from the requests[9].

There are 4 flags in the scheduler. Each one is assigned to exactly one thread. The usage of the flag is to determine whether the instruction coming from the Front-end is allowed to join the scheduling. If the flag is locked up, instruction belongs to that thread is excluded from scheduling. On the contrary, if the flag is unlocked, instruction would participate in scheduling.

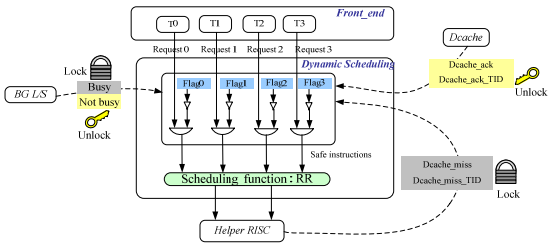

Figure 5: Block diagram of Dynamic Scheduling with lock and unlock event

The state of the flag is affected by two conditions. One is data cache miss, and the other is background load/store busy. Once data cache miss occurs, following instructions of that thread would be unsafe. We should lock the flag to prevent the following instructions from joining scheduling and dispatching to Helper RISC. The flag is locked until data cache miss is resolved.

As regards background load/store busy, we lock up the flag when background load/store unit is busy and another background load/store instruction is coming. For example, if background load/store is busy now, and a new background load/store instruction is coming from thread0, then we would lock up the flag of thread0 until background load/store unit is free.

Instructions whose flag are unlocked are safe instructions. Safe instructions would enter the scheduler to select two instructions, to dispatch to Helper RISC. Our scheduling function is Round-Robin. If thread 0 has the highest priority now, safe instruction of thread0 would be first serviced, and then thread1, thread2, and thread3 in turn. Last but no least, the priority register would promote to next thread each cycle to ensure fairness. Figure 5 shows the block diagram of Dynamic Scheduling.

3.3: SMT Helper RISC Architecture

The objective of hardware multithreading processor is dynamic and flexible sharing of data path and functional units between multiple threads. This approach increase application performance by improving throughput, the total amount of work done across multiple threads of execution. And the processor hides memory latency from one thread by executing instructions from other threads.

In our project, we provide a 4 hardware context multithreading processor with two functional engines. One is computation & program control engine, and the other is load\store & special move engine. Computation & program control engine: it is responsible of arithmetic and logic computation, program control, and processor configuration. Load\store & special move engine: it is responsible of load/store and special move operation. In our ISA, the two kinds of instructions have the ability to repeat several times with the help of repeat unit and repeat buffer.

The 4 hardware context multithreading processor is capable of issuing two instructions per cycle. The two instruction comes from two different threads could execute simultaneously to achieve SMT. Figure 3 shows the architecture of the SMT processor.

In both engines, it has two read ports and one write port to control register file and data register file. Control register file has four 16x32 registers banks, and each bank is assigned to a dedicated thread. Data register file has eight 32x64 register banks. They are shared between the threads. Each thread has two register files has two read ports and one write ports.

As long as the two engines execute instructions from different threads, it would have no conflict in access of register read, write ports. As a result, register port wouldn't become a burden in our SMT design.

3.3: Non-Blocking Load/Store

In multithreading processor, data path and function units are shared between all threads. A stall arise from data hazard or data cache miss of one thread may cause the whole machine to stall, then no instruction can promote to next stage of the pipeline. All threads must wait until the stall is resolved[8].

Although data hazard could have only one cycle delay, data cache miss would have hundreds of cycle instead. All of the threads must waste hundreds of cycle on waiting and that is harmful to the performance [4][2].

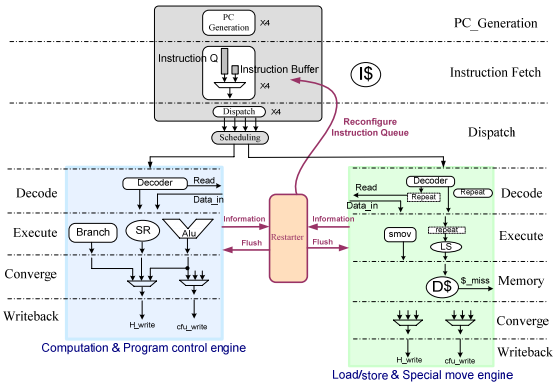

Figure 6: Architecture of non-blocking load/store

In our design, we avoid that by keeping data-path active in spite of data cache miss. Briefly, we flush instructions of cache miss thread to make data-path non-occupied, then other threads can promote smoothly and normally. The flushed instructions is reserved in Front-end at instruction buffer. After cache miss is resolved, the flushed instructions which are hold in instruction buffer after data cache miss is resolved. Figure 6 shows the architecture of non-blocking load/store.

In the following, we explain the idea describe above in detail in 3 steps:

step1: Instructions are issued from instruction queue. And they are reserved in instruction buffer after issue, as shown in Figure 7.

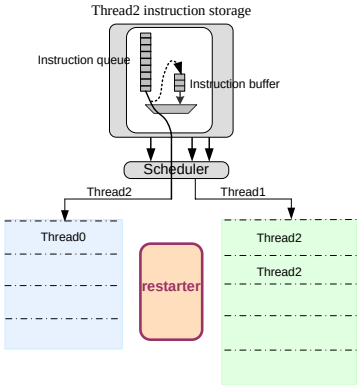

Figure 7: Instruction are issued and buffered

step2: When we encounter data cache miss:

(1) We use the information such as thread ID from various pipeline stages to decide which instruction should be flush.

(2) Restore the status flag or status register to return to previous state.

(3) Inform scheduler not to select that thread during data cache miss processing.

(4) Instruction buffer in Front-end must be reconfigure to guarantee that the instructions which are flushed previously would be issue again from instruction buffer and restart execution after data cache miss is resolved, as shown in Figure 8.

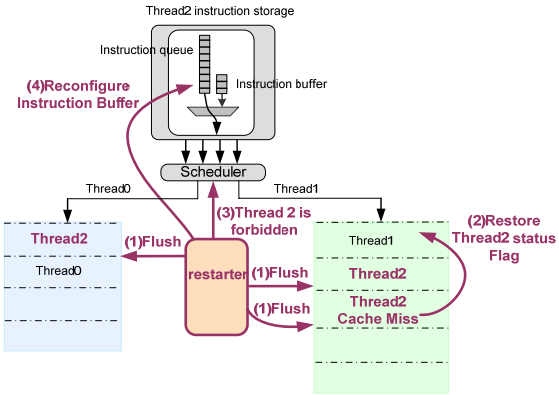

Figure 8: Four transactions

step3: After cache miss is resolved, the instructions which are flushed previously are re-issued from instruction buffer, as shown in Figure 9.

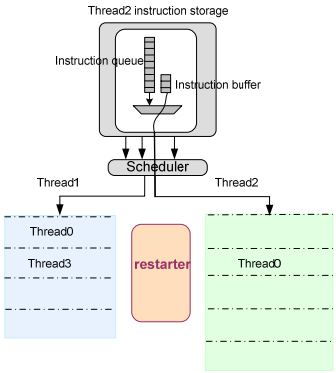

Figure 9: Flushed instructions are re-issued

As step3 described above has done, we return to step1, and instruction promote normally as usual. We can find out that under our architecture instruction progression is smooth, and is not disturbed due to data cache miss. Therefore, there is no cycle wasted, and throughput is keeping.

4: EXPERIMENT

We synthesize our design with Synopsis Design Compiler using 0.13 gm CMOS technology. We synthesize each module individually and record the synthesis reports of area, and timing requirement. Then we explore the impact when changes several thread parameters in the simulation.

4.1: Simulation Result

In this section, we measure the waiting time of execution threads. Waiting time means one thread must spend some time on waiting the access right of Helper RISC. This is because Helper RISC is shared between multiple threads. The more threads competing for the Helper RISC, the more waiting time each thread would incur.

We promote from multithreading Helper RISC to SMT Helper RISC to increase issue bandwidth and reduce waiting time. As Figure 10 shows, the outcome is unapparent when there are only two threads due to the short of instruction selection. On the other hand, SMT can improve the most case to increase architecture performance. So it is getting better when the number of threads grows. Therefore we do not only improve performance but also reduce execution time to get the goal of our accelerative.

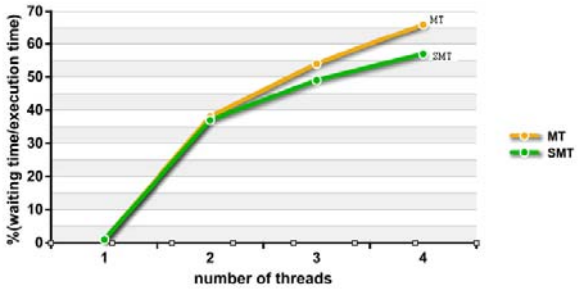

Figure 10: Waiting time of different #thread on different model

4.2: Hardware Synthesis

Table 2 shows the synthesis report of multithreading Helper RISC. Table 3 shows the synthesis report of SMT Helper RISC with non-blocking load\store. As we can see, only a few area increment is incurred as we upgrade from multithreading Helper RISC to SMT Helper RISC with non-blocking load\store. And timing critical delay is not worsened, either. However, Front-end has a little improvement instead, because we refine the coding style especially.

Table 2: Synthesis report of multithreading Helper RISC

Module	Data arrival time (ns)	Area (gate count)	Timing constraint (ns)
Helper RISC	3.26	35824	4
Front-end	1.61	45128	2
HDYS	0.62	820	2

Table 3: Synthesis report of SMT Helper RISC with non-blocking load/store

Module	Data arrival time (ns)	Area (gate count)	Timing constraint (ns)
Helper RISC	3.26	51145	4
Front-end	1.62	43520	2
HDYS	1.73	1835	2

5: CONCLUTION

In this paper, we propose a SMT RISC processor with non-blocking load/store. With its multithreading ability, our VisoMT architecture can create multiple accelerative

threads which are executed concurrently on the multithreading RISC. It is fitting especially in the multimedia application due to the inherent parallelism. Besides, we upgrade the multithreading RISC processor to SMT RISC processor incorporating non-blocking load/store with little overhead. It would increase the throughput of the system and decrease the competition pressure of the multithreading RISC processor among the threads. Thus we conclude that when we propose simultaneous multithreading RISC processor to increase overall system performance by enabling processor to share unused CPU resource across effective multiple threads of parallelism.

References

[1] Michael Bekerman, Avi Mendelson, and Gad Sheaffer. Performance and Hardware complexity tradeoffs in designing multithreaded

[2] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), volume 27, pages 51-61, New York, NY, 1992. ACM Press.

[3] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithread sparc processor. IEEE Micro, 25(2):21-29.2-005

[4] K. Oner and M. Dubois. Effects of memory latencies on non-blocking processor/cache architectures. Supercomputing, 1993.

[5] Bailey D.W.Bell S.L. Biro L.L. Bowhill W.J.Dever D.E.Felix S.Gammack R.GerminiV.Growan M.K. Gronowski P.Jackson D.B. Mehta S.Morton S.V.Pickholtz J.D.Reilly N.H. Smith M.J. Preston R.P., Badeau R.W. Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading. In ISSCC2002: In Proceedings of the International Solid State Circuits Conference, 2002.

[6] Steve E. Raasch and Steven K. Reinhardt. The impact of resource partitioning on smt processors. pact, 00:15, 2003

[7] Behnam Robatmili, Nasser Yazdani, Somayeh Sardashti, and Mehrdad Nourani. Thread-sensitive instruction issue for smt processors. IEEE Comput. Archit. Lett., 3(1):5, 2004.

[8] Dean M. Tullsen and Jeffery A. Brown. Handling long-latency loads in a simultaneous multithreading processor. micro, 00:318, 2001.

[9] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In ISCA, pages 191-202, 1996.

[10] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In ISCA'95: Proceedings of the 22nd annual international symposium on Computer architecture, pages 392-403, New York, NY, USA, 1995.ACM Press.

[11] J. Seng, D. Tullsen, and G. Cai. Power- sensitive multithreaded architecture. pages 199-208.