Develop dynamically scheduled processor model using ATOM

Dazhi Wang, Jing Zhang
Department of Computer Science
Duke University
wangdz, jzhang@cs.duke.edu

December 10, 2000

Abstract

Dynamically scheduled processor has dynamically scheduled pipeline, it can issue multiple instructions in one cycle and execute them out-of-order, thus achieving a better performance. In this paper we simulate an out-of-order superscalar model based on MIPS10000, and then evaluate the performance of this architecture as comparing to a simple pipeline model.

1. Introduction

Dynamic scheduling is a method in which the hardware determines which instructions to execute, as opposed to a statically scheduled machine, in which the compiler determines the order of execution. In essence, the processor is executing instructions out of order.

Dynamic scheduling is akin to a data flow machine, in which instructions don't execute based on the order in which they appear, but rather on the availability of the source operands. Dynamically scheduled machines can take advantage of parallelism, which would not be visible at compile time. They are also more versatile as code does not necessarily have to be recompiled to run efficiently since the hardware takes care of much of the scheduling. In a statically scheduled machine, code would have to be recompiled to take advantage of the machine's particular hardware. (All of this is assuming the machines use the same instruction set architecture. Of course, the code would have to be recompiled no matter what if the machines used different ISAs.)

2. Proposed solutions

2.1 Scoreboard

Scoreboard is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependencies; it is named after the CDC 6600 scoreboard, which developed this capability. The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they

do not depend on any active or stalled instruction. The scoreboard takes full responsibility for instruction issue and execution, including all hazard detection.

2.2 Tomasulo

Tomasulo's algorithm is another method of implementing dynamic scheduling. This scheme was invented by Robert Tomasulo, and was first used in the IBM 360/91. Tomasulo's algorithm differs from scoreboarding in that it uses register renaming to eliminate output and anti-dependences, i.e. WAW and WAR hazards. Output and anti-dependences are just name dependences; there is no actual data dependence.

Tomasulo's algorithm implements register renaming through the use of what are called reservation stations. Each reservation station corresponds to one instruction. Once all source operands are available, the instruction is sent for execution, provided a functional unit is also available. Once execution is complete, the result is buffered at the reservation station. Thus, unlike in scoreboarding where the functional unit would stall during a WAR hazard, the functional unit is free to execute another instruction. The reservation station then sends the result to the register file and any other reservation station that is waiting on that result. WAW hazards are handled since only the last instruction (in program order) actually writes to the registers. The other results are buffered in other reservation stations and are eventually sent to any instructions waiting for those results. WAR hazards are handled since reservation stations can get source operands from either the register file or other reservation stations (in other words, from another instruction).

2.3 VLIW: Very Long Instruction Word

Very Long Instruction Word (VLIW) is an increasingly popular approach to microprocessor design. This type of CPU chip uses long, fixed-length instructions made up of several shorter-length instructions that execute in parallel. VLIW differs in several important ways from older approaches such as CISC and RISC. Most new microprocessor architectures announced recently are based on VLIW principles.
It holds many instructions in fixed fields, scheduled by complier-know they can execute simultaneously. Its performance depends on success of compiler. In this machine all dependencies resolved, so hardware doesn’t have to check for them. So it makes instruction issue logic easy, also it fills in with NOPs if unable to find independent instruction. But so far, these machines not successful in the marketplace, because it can’t schedule current instructions.

Many of today's processors implement some form of out-of-order execution mechanism. The IBM PowerPC 604, the MIPS R10000, and the HP PA-8000 all include an out-of-order execution unit with in-order completion. Even Intel's P6 line (Pentium Pro, Pentium II, Celeron) implements out-of-order execution. In this paper we choose MIPS10000 as the superscalar processor model. And we use ATOM to simulate this architecture.

3. Methodology

We use the main idea of Tomasulo algorithm as well as the Mips R10000 model to simulate this dynamic scheduled processor. Mips R10000 is a dynamic, superscalar microprocessor that implements the 64-bit Mips 4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically issues them to five fully pipelined execution units. Instructions are decoded and graduate in order. Execution is out of order, except that memory access is sequential.

Based on the Mips R10000 model, we design our project scope as the following:
(1) Implements register renaming using map tables
(2) Implements a 32KB, 2-way set-associative, write-back L1 data cache
(3) Fetches and decodes 4 instructions per cycle, issues up to 2 integer instructions, 2 FP instructions and 1 load/store instruction per cycle
(4) Uses out-of-order execution and in-order graduation
(5) Assumes perfect branch prediction, no need for speculative execution

3.1 Composition of the Processor

The main functional units in our processor are:
(1) 64 **physical integer registers** and 64 **physical floating point registers**
(2) **Map table**: record the information that maps logical register into physical register
(3) **Register busy table**: record the physical registers' status. There are 3 statuses for a physical register: FREE means the register is not in use; BUSY means the register is in use, and contains invalid data; READY means the register contains valid data
(4) **Free integer register list and free FP register list**: record free physical integer registers and free physical floating point registers, respectively
(5) **Active list**: record all the instructions that have been decoded but haven't graduated
(6) **Integer queue, FP queue, and address queue**: record all integer instructions, FP instructions or load/store instructions that have been decoded but haven't been executed
(7) **Load buffer and store buffer**: record all the load/store instructions that have calculated the address but haven't finished memory access
(8) **Two integer ALUs, two floating point ALUs,** and **one address calculator**: All execution units are fully pipelined except the divider in the ALUs

Figure 3-1 shows the block diagram and pipeline-timing diagram for the superscalar processor based on Mips10000.

3.2 Pipeline in the Processor

We design our pipeline into 5 stages: Instruction Fetch, Instruction Decode, Instruction Issue, Instruction Execution, and Write Back. The following is how we implement each stage:

3.2.1 Instruction Fetch

It's easy to implement Instruction Fetch stage. We just count the number of the

instructions that has been fetched into the instruction buffer. As long as it reaches 4, or the instruction type is branch, jump or jump and link, we finish instruction fetch cycle.

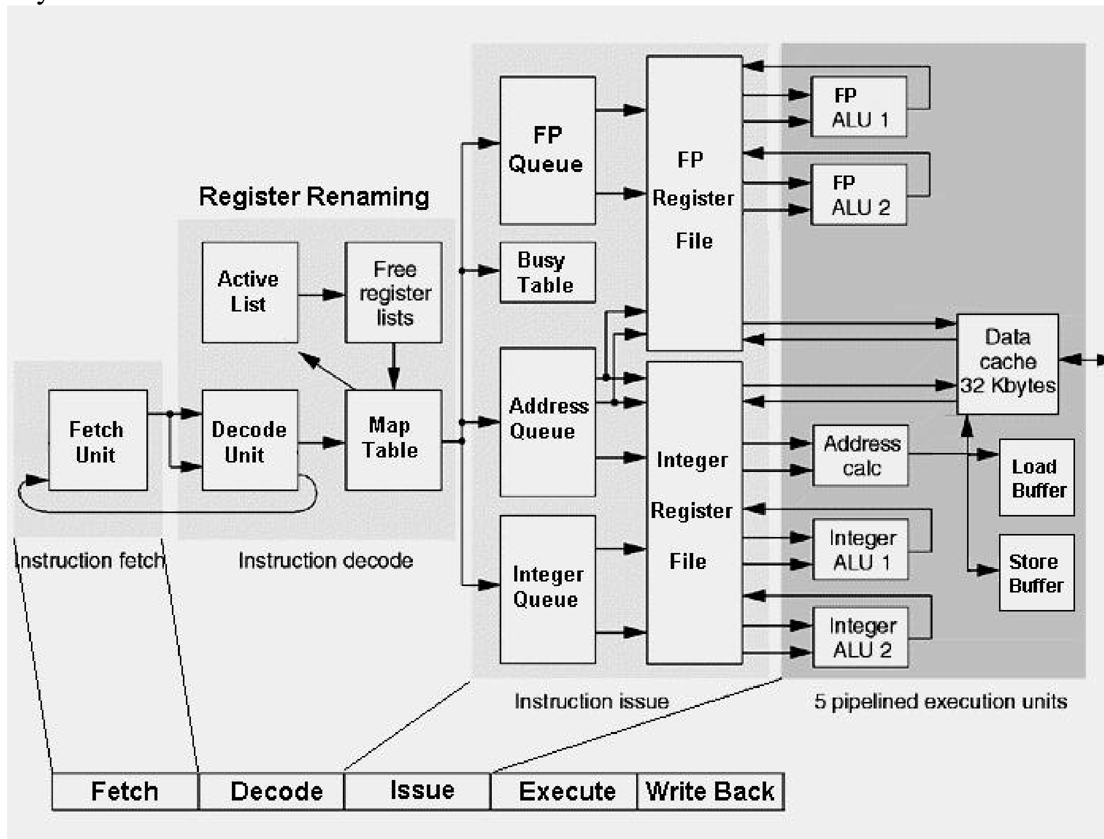

Figure 3-1 The Processor model based on Mips10000

3.2.2 Instruction Decode

Every cycle the processor decodes up to 4 instructions in order. The main task during decode stage is to rename logical registers into physical registers and put the decoded instructions into the active list and integer/FP/address queue. For each instruction, the map table is searched to get the source physical registers according to the source logical registers. To map the destination logical register *d* into a physical register, we do the following:

(1) Get a new physical register *pr* from the free register list

(2) Change the map table to map *d* into *pr,* and change the corresponding entry in register busy table to indicate *pr* is busy.

(3) Because the physical register which *d* is previously mapped into is not in the map table any more, we need to save the old mapping information into the decoded instruction.

After finishing register renaming, we put each decoded instruction (including the physical register operands, the old mapped physical register) into the active list in order; and also put it into integer queue, FP queue, or address queue according to its instruction type. As long as the free register list is empty, or active list is full, or the correspondent queue is full, instruction decoding is stalled.

3.2.3 Instruction Issue

Three functional units can issue instructions into execution units: Integer queue, FP queue, or address queue.

Integer queue: Every cycle we check all the entries in the integer queue to see if there are instructions whose source operands (correspond to physical registers) are ready. We do this by looking up the register busy table. If all the source operands for an instruction are ready, and the corresponding integer ALU is free, we issue the instruction to execution unit.

FP queue: The issue procedure is similar to integer queue.

Address queue: The address queue is different from integer/FP queue in that the instructions in it are issued in the original instruction order. In each cycle, only the head instruction in the queue can be issued into address calculator, while in integer/FP queues, instructions can be issued into execution units and executed out of order.

3.2.4 Instruction Execution

Integer ALUs: There are 2 integer ALUs, ALU2 is for multiplication and division, ALU1 is for all the other integer instructions. They are fully pipelined except for the divider in ALU2. During each cycle, the integer queue issue up to 2 instructions to integer ALUs. When an instruction finishes execution, and there is no contention for write back units, it can begin write back stage in the next cycle.

FP ALUs: It's similar to integer ALUs.

Address calculator: Each cycle the address calculator receives at most 1 instruction issued from the address queue and then calculates its address.

(1) If the instruction is **Load**: The store buffer is searched to see if there is an entry that has the same data address as the load instruction. If yes, we just pass that entry's source register value into the load instruction's destination register; otherwise we look for the data in the cache. Under a cache miss the load instruction is assigned a miss penalty and put into the load buffer.

(2) If the instruction is **Store**: Its data address is checked in the cache, if cache misses, we check the load buffer to find a load instruction whose data address is same as the store instruction. If there exists a match, we assign its execution cycles to be the miss penalty plus the cycles the load instruction need to finish, otherwise we assign its execution cycles to be the miss penalty. Then we put it into store buffer.

Those instructions that hit the cache can be written back after a cache read/write cycle, while the instructions in load/store buffers must wait for many cycles before write back because of the miss penalty.

3.2.5 Write Back

During this stage, the result is written into the destination register of the instruction, the corresponding entry in register busy table is set to READY, and we mark the instruction in active list as finished.

3.2.6 Graduation

Besides the 5 stages in the pipeline, in each cycle we also need to check the active list

to graduate instructions. We guarantee in-order graduation by only graduating the head instruction in the active list. If the head instruction is finished, we graduate it and release the old physical register saved during the decoding stage. Then we make the head point to the next instruction, and check it again. We stop graduation cycle when the head instruction hasn't finished.

3.3 Deal with Hazards in the Processor

3.3.1 Data hazard in registers:

RAW hazard: We don't issue the instruction into execution units until the source operands are ready, thus preventing RAW hazard.

WAW and WAR hazard: Because we use register renaming and each time we map the logical destination register to a different physical register, there is no WAW or WAR hazard.

3.3.2 Data hazard in memory:

RAW hazard: We avoid the RAW hazard by checking the load instruction's address in the store buffer, if there is a match, we directly pass the data in store buffer to the destination register of the load instruction.

WAR hazard: We forbid the store instruction to execute until the load instruction finishes. So we can prevent WAR hazard.

WAW hazard: Because load/store instructions are executed in order, and the time for each store instruction to access memory is the same, so we avoid WAW hazard.

3.3.3 Structure hazard:

All structure hazards could be: free register list empty, active list full, load/store buffer full, integer/FP/address queue full, write back units contention. We considered all these structure hazards and stalled till the structure hazards disappear.

4 Experiment and Result

We use ATOM to simulate this dynamically scheduled processor model.

Number of physical registers	64 for integer 64 for floating point
Size of active list	32
Size of integer/FP/address queue	16
Size of load/store buffer	16
Cache	32KB, 2-way set-associative, write back
Cache missing penalty	20 cycles

Table 1 Parameters in dynamically scheduled Superscalar Processor

Integer Instructions		FP Instructions		Other Instructions	
Add/Sub	1	Add/Sub	2	Load	1
Multiply	5	Multiply	2	Store	1
Divide	9	Divide	12	Branch	1
Other	1	Other	1		

Table 2 Execution cycles for Different Types of Instruction

	Turb3d	Vortex	Tomcatv
Superscalar Processor	0.9167	0.7889	0.9219
Simple Pipeline	1.3966	1.5912	1.7937

Table 3 CPIs for the SuperScalar Processor and 5-stage Simple Pipeline

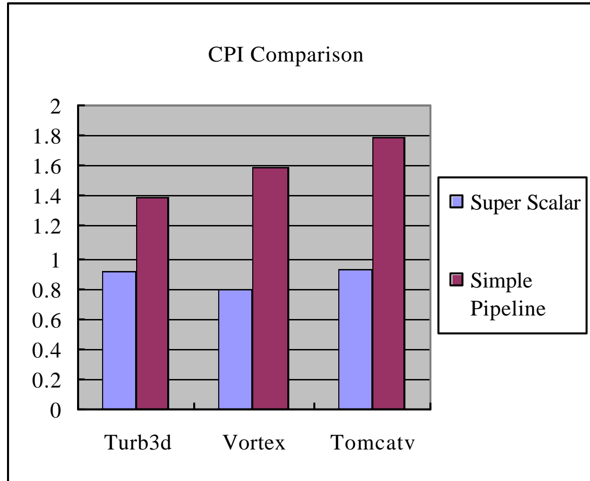

Figure 1

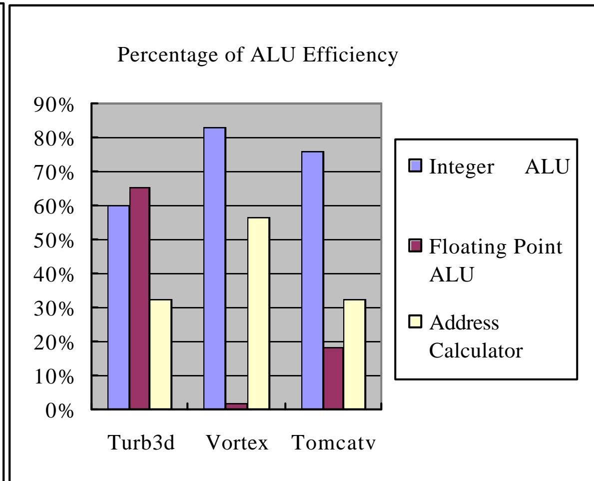

Figure 2

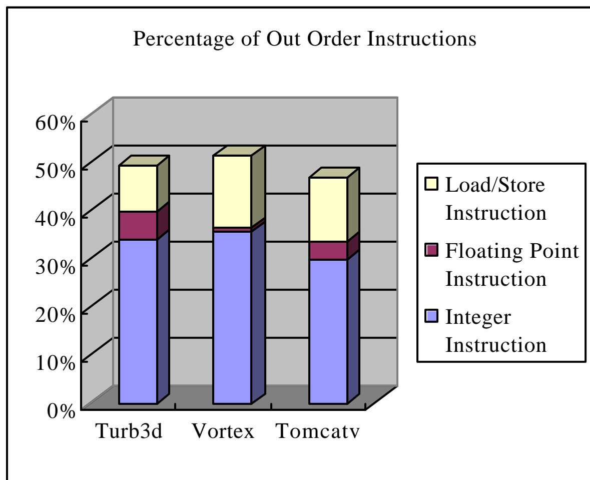

Figure 3

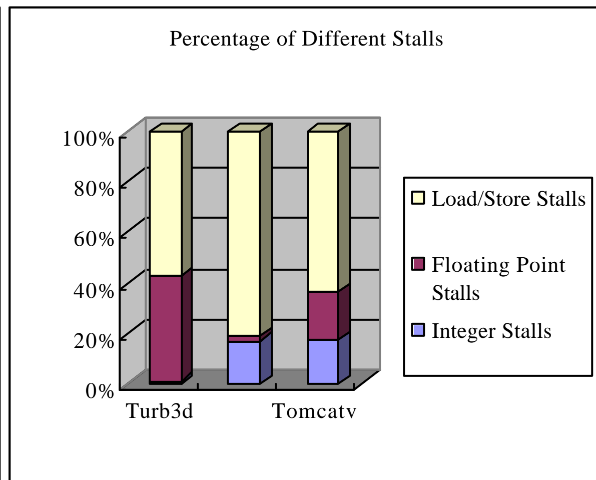

Figure 4

Three benchmarks are used in our simulation: Vortex, Turb3d and Tomcatv.

Table 3 and Figure 1 compares the CPI of this dynamically scheduled processor to the CPI of 5-stage simple pipeline using different benchmarks. The 5 stages in the simple pipeline are: Fetch, Decode, Execute, Memory and Write back. There is also a cache in the simple pipeline processor; we consider the structural hazard and data hazard, but no control hazard (same as the superscalar processor). We can see that the CPI of this dynamically scheduled processor is much smaller than the CPI of the simple pipeline. In these 3 benchmarks, the average CPI of the superscalar processor is about 0.9, while the CPI for the simple pipeline is about 1.5.

Figure 2 is the Integer/FP/Address ALU efficiency for different benchmarks. That is, the percentage of time that the ALU is in use. The sum of the three kinds of ALU efficiency is larger than 100% because of overlapping usage in the same cycle. The integer ALU efficiency is very high because there are lots of integer instructions in each benchmark. The floating point ALU efficiency is low in Vortex because it contains very few FP instructions.

Figure 3 shows how many integer/FP/address instructions are executed out of order. We can see that more than 40% of the instructions are executed out of order for each benchmark.

Figure 4 illustrates the stalls caused by different types of instructions. Load/store instructions cause more than half of the stalls in each of the three benchmarks. So the cache-hit rate has a great impact on CPU performance.

5 Conclusions and Future Works

From our simulation result we can see that this dynamically scheduled processor greatly reduces the CPI by executing multiple instructions in one cycle and by out-of-order execution. It also avoids name dependences using register-renaming technique. And our future work will be:

(1) Compare various techniques to implement branch prediction and speculative execution.

(2) Implement different cache configurations to see their impacts to CPI.

(3) Find more efficient register renaming techniques. In our processor model the old physical register is not released until the subsequent instruction graduates. If we can release it as long as it is no longer in use, we can reduce the hardware cost by using less physical registers.

Reference

[1] ANDERSON, D. W., F. J. SPARACIO, AND R.M.TOMASULO [1967], "The IBM 360 Model 91:Processor philosophy and instruction handling," *IBM J. Research and Development* 11:1 (January), 8-24

[2] COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND P. K. RODMAN [1987], "A VLIW architecture for a trace scheduling compiler," *Proc. Second Conf. On Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (March), Palo Alto, Calif, 180-192

[3] JOHNSON, M. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.

[4] JOUPPI, N. P. AND D. W. WALL [1989] "Available instruction-level parallelism for superscalar and superpipelined processors," *Proc. Third Conf. On Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272-282

[5] KENNETH, C. YEAGE, "The MIPS R10000 Superscalar Microprocessor", *IEEE Vol.* 16, No. 2: April 1996, pp. 28-40

[6] SMITH, J. E. [1989], "Dynamic instruction scheduling and the Astronautics ZS-1," *Computer* 22:7 (July), 21-35

[7] TOMASULO, R. M. [1967] "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Research and Development* 11:1 (January), 25-33

Appendix 1 Pipeline Data for Tomcatv

Simple Pipeline	Dynamically Scheduled Processor
cycles: 2354681678 insts: 1312765812 raw stalls: 1057127014 waw stalls: 0 struct stalls: 23772619 int stalls: 669588112 fp stalls: 38955518 mem stalls: 348583384 Load Instructions: 276001723 Store Instructions: 113568690 Read from Memory: 14461526 Write to Memory: 10987917 Load Misses: 14461526 Store Misses: 3543361	Total Instructions: 1315089149 Total Cycles: 1212359300 Int add insts: 191931822 Int multi insts: 193 Int divide insts: 0 Int other insts: 428514517 Branch insts: 212012924 Fp add insts: 47514319 Fp multi insts: 31319223 Fp divide insts: 1830919 Fp other insts: 11767972 Load insts: 263566219 Store insts: 113713246 Int ALU1 Usage: 922527517 Int ALU2 Usage: 1158 Fp ALU1 Usage: 107322202 Fp ALU2 Usage: 115158536 Address Adder Usage: 390197260 Out of Order Int Insts: 390339589 Out of Order Fp Insts: 48806731 Out of Order Mem Insts: 177117398 Int Stalls: 30052090 Fp Stalls: 33008195 Mem Stalls: 110657459 Direct read from store buffer: 12917795 Read from Memory: 14411405 Write to Memory: 10978187 Load Misses: 14411405 Store Misses: 3538526

Appendix 2 Pipeline Data for Turb3d

Simple Pipeline	Dynamically Scheduled Processor
cycles: 13351740671 insts: 9559849778 raw stalls: 2799972680 waw stalls: 0 struct stalls: 1040754013 int stalls: 326752035 fp stalls: 845574419 mem stalls: 1627646226 Load Instructions: 1546343349 Store Instructions: 1266648260 Read from Memory: 142710995 Write to Memory: 136916906 Load Misses: 142710995 Store Misses: 86259792	Total Instructions: 9559889351 Total Cycles: 8763566156 Int add insts: 1209028143 Int multi insts: 6495156 Int divide insts: 0 Int other insts: 2615440418 Branch insts: 611940105 Fp add insts: 1179654585 Fp multi insts: 1072343526 Fp divide insts: 5045828 Fp other insts: 46936464 Load insts: 1536578763 Store insts: 1266651042 Int ALU1 Usage: 5238870264 Int ALU2 Usage: 38970936 Fp ALU1 Usage: 2495628479 Fp ALU2 Usage: 3228238342 Address Adder Usage: 2813015654 Out of Order Int Insts: 3258388324 Out of Order Fp Insts: 560061780 Out of Order Mem Insts: 905460955 Int Stalls: 6882329 Fp Stalls: 999995533 Mem Stalls: 1388430903 Direct read from store buffer: 9775321 Read from Memory: 142691601 Write to Memory: 136863648 Load Misses: 142691601 Store Misses: 86237495

Appendix 3 Pipeline Data for Vortex

Simple Pipeline	Dynamically Scheduled Processor
cycles: 5188572448 insts: 3260760323 raw stalls: 2011616493 waw stalls: 0 struct stalls: 3009386 int stalls: 1273768704 fp stalls: 1637 mem stalls: 737846152 Load Instructions: 935350276 Store Instructions: 528604673 Read from Memory: 8690528 Write to Memory: 8207389 Load Misses: 8690528 Store Misses: 3823094	Total Instructions: 3260760329 Total Cycles: 2572314333 Int add insts: 333055123 Int multi insts: 1572625 Int divide insts: 0 Int other insts: 1107374763 Branch insts: 354778492 Fp add insts: 0 Fp multi insts: 291 Fp divide insts: 0 Fp other insts: 24085 Load insts: 932621622 Store insts: 528604673 Int ALU1 Usage: 2119440137 Int ALU2 Usage: 9435750 Fp ALU1 Usage: 48056 Fp ALU2 Usage: 873 Address Adder Usage: 1463954950 Out of Order Int Insts: 1156881285 Out of Order Fp Insts: 23700 Out of Order Mem Insts: 482002217 Int Stalls: 60496611 Fp Stalls: 364 Mem Stalls: 287522655 Direct read from store buffer: 2728655 Read from Memory: 8690528 Write to Memory: 8208516 Load Misses: 8690528 Store Misses: 3823094