Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

The Design And Implementation of An Object-Oriented Trace-Driven Simulator for Superscalar CISC Processors

C.C. Hsu*, D.C. Wang*, S.S. Shang*, S.E. Chang**, C.T. Chang*
Computer & Communication Research Laboratories *
Industrial Technology Research Institute
Chutung, Hsinchu, Taiwan, ROC
Information and Computer Engineering Department **
Chung-Yuan Christian University
Chung-Li, Taiwan, ROC

Abstract

Developing a superscalar complex-instruction-set-computing (CISC) processor requires trade-off of various design alternatives. In this paper, we present a trace-driven simulator to help us deal with that difficult, iterative task. The simulator was written in C++ and well-structured so that it can be easily reconfigured for evaluating various design trade-offs. The reconfigurable parts include numbers of pipeline stages and functional units, sizes of internal queues and buffers, latencies of each functional unit, and various scheduling and control schemes. We demonstrate its flexibility and versatility by simulating four DOS benchmark programs and reporting their results.

1. Introduction

In the initial design phase of a superscalar CISC processor [2, 3], there exist many design alternatives. One major problem in designing such a processor is to determine which microarchitecture to adopt and what level of performance it can achieve. It is hard to make various design trade-offs without thorough analysis. Hence, how to optimize cost and performance of an architecture becomes a very challenging task. For example, in such a processor how many and what functional units are needed, how these functional units are connected, and how large the sizes of internal queues are, are all important issues. To provide essential information to answer these questions, a processor simulator is usually imperative.

To provide a flexible simulation environment to evaluate various design alternatives, we develop an object-oriented simulator that is configurable by simply modifying processor design parameters. The simulator was written in C++ [4] and is well-structured so that each individual module can be easily replaced if any design is changed. Intel Pentium Pro [8, 9, 10], which supports speculative and out-of-order execution, in-order completion, and precise interrupts, is used as the baseline processor model in the simulator. In that processor, x86 instructions are first translated into fixed-length primitives, called μ-instructions, and then executed by an underlying reduced-instruction-set-processing (RISC) core. Design issues regarding a superscalar CISC processor can be found in [1].

Though the module for instruction decoding in the simulator is for x86 instruction set, it can be easily replaced for any other instruction set, either CISC or RISC. Furthermore, the simulator has various built-in configuration options, and supports a few types of system components and various system interconnection structures. Thanks to design modularity in the simulator, additional options and features can be easily added as needs arise. The flexibility and versatility of the simulator make it possible to evaluate various processor design alternatives rapidly.

The rest of the paper is organized as follows. Section 2 introduces the software simulation environment. Section 3 describes various modules in the simulator. Section 4 presents a case study of a superscalar CISC processor and reports some simulation results. Section 5 summarizes our contributions.

2. Simulation Environment

There are several approaches to evaluating system performance. One approach is to build real hardware that directly executes programs for measurement. Though hardware execution may obtain very accurate performance information, it takes much time and costs a lot, and is not suitable for a design requiring iterative fine-

tunings. Another approach is to use mathematical and analytical models. Though mathematical derivations take less time and cost little, they usually are not detailed enough and accuracy in results is often questionable. Another approach is to use a trace-driven simulator, which mimics cycle by cycle concurrent system activities. One advantage is that programs are not actually executed but processed, which achieves an acceptable accuracy in results with a reasonable amount of time in development.

Trace-driven simulation provides a balance between accuracy and development effort, and is adopted in our processor simulator. Fig. 1 shows the overall simulation environment used. Program binaries are first executed and trace-collected by a debugger. Next, a processor configuration file is provided to determine processor internal structures in the simulator. The processor simulator then processes traces and reports simulation results.

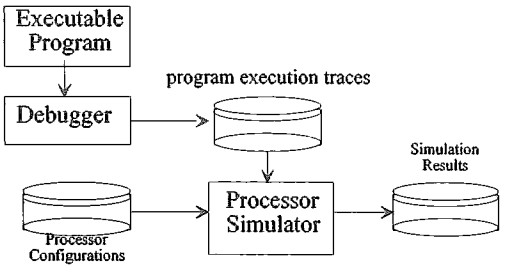

Figure 1 Simulation environment.

Fig. 2 illustrates our baseline processor model. It includes separate instruction and data caches (IC, DC); a branch target buffer (BTB); an instruction fetcher (IF); plural x86 instruction decoders; a μ-instruction queue; a reorder buffer (ROB); a reservation station (RS), whether centralized, distributed, or hybrid; a memory reorder buffer (MOB); associated control modules; and plural function units, such as an address generation unit (AGU), integer arithmetic logic units (ALU), a floating-point (FP) adder, and so on. This model covers a wide range of processor architectures, including scalar and superscalar ones.

In the baseline model the pipeline stages, according to their functions, are classified into seven tasks: instruction fetch, instruction decode, μ-instruction login, μ-instruction issue, μ-instruction execute, result complete, and retire. Each task may require one or several pipeline stages to implement and take a various number of cycles to realize and the associated information is defined in a configuration file. Table 1 describes the functions performed by each task .

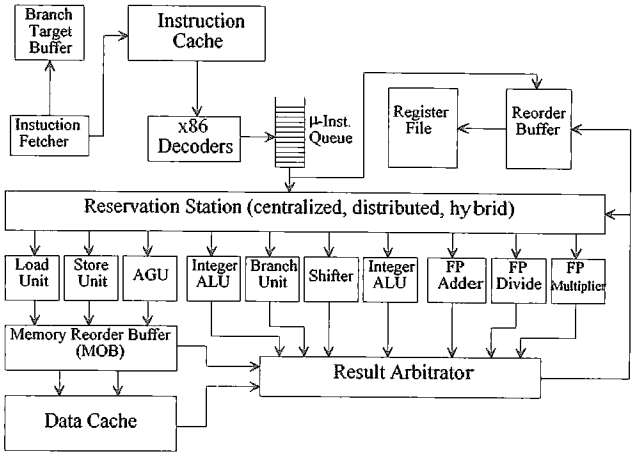

Figure 2 Baseline superscalar CISC processor model.

Task	Function Description
Fetch	Fetch instructions from the instruction cache.
Decode	Decode instructions, being translated and stored into the μ-instruction queue.
Login	Read μ-instructions, being register-renamed via the ROB and logged into the RS.
Issue	Dispatch ready μ-instructions to available functional units.
Execute	Execute μ-instructions and may require multiple cycles to complete.
Complete	Arbitrate results from functional units and store them back to the ROB.
Retire	Retire completely executed instructions and update processor state in order.

Table 1 Tasks performed in the baseline processor model.

```
WHILE (not finish) {
// Retire
      rob.Retire();          // Retire instructions at the ROB
// Result Complete
      Function_unit.Check_Result();
                  // Check results at functional units
// Execute
      Function_unit.Login(uinst);      // Execute μ-Inst.
// μ-Instruction Issue
      uinst = RS.Issue_Instruction();  // Issue μ-inst. at the RS
// μ-Instruction Login
      rob.Login(uinst);          // Rename μ-Inst. at the ROB
      RS.Login(uinst);           // Login μ-Inst. at the RS
// μ-Instruction Decode
      Decoder.Decode(inst);      // Decode instructions
      uinst = Decoder.Check();   // Translate into μ-inst.
// Instruction Fetch
      inst = Fetcher.Check_result(); // Get inst. from cache
      CYCLE++;
}
```

Figure 3 Flow for imitating concurrent pipeline activities in the simulator.

In order to simulate in software concurrent events in the processor model, queues are employed to isolate each task,

similar to what hardware registers are used to isolate pipeline stages in a pipelined processor. Each task has a queue to temporarily hold its execution status, and the depth of the queue determines the number of pipeline stages for that task. To simulate progress in pipelines, queues are repeatedly checked and processed in inverse order, i.e., retire, complete, execute, issue, login, decode, and fetch. As shown in Fig. 3, in each iteration status propagates from a task to the next, imitating pipelining in the processor model.

3. Simulator Design

The simulator comprises several modules, including the fetcher, instruction cache, branch target buffer, decoder, reorder buffer, functional units, reorder buffer, memory reorder buffer, data cache, and result arbitrator. Each module buffers appropriate information and controls various manipulations. They are described in details as follows.

3.1 Fetcher, Instruction Cache, and Branch Target Buffer

The fetcher, along with the instruction cache and branch target buffer, is responsible for instruction fetch, as depicted in Fig. 4. The fetcher has an internal queue, whose reconfigurable size determines the number of pipeline stages it has, and operates by sending an address to the instruction cache and branch target buffer to check whether a cache miss or branch instructions are detected. The instruction cache module records tags for recently used blocks. When receiving an address, the cache module searches its tag memory to verify if this address matches any tag. If a mismatch occurs, the fetcher stops to wait until pre-determined, reconfigurable cache miss cycles are expired.

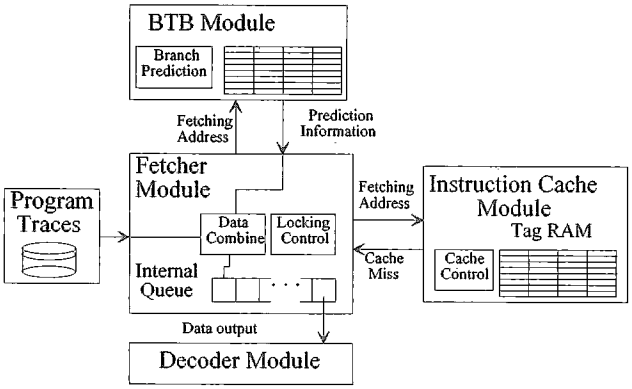

Figure 4 The interaction among the fetcher, BTB, and instruction cache modules.

The branch target buffer [5, 6] module is responsible for branch detection and prediction. Since traces contain only correct instruction execution path and exclude incorrect path following mispredicted branches, whether a branch will be taken or not has already been determined. To mimic branch misprediction penalty, a lock mechanism is used in the fetcher. If the execution path following a branch does not accord with the path predicted by the branch target buffer, the lock mechanism stops the fetcher from reading traces until the branch reaches and is processed by the branch unit. Since in the simulator only a branch instruction is processed when a branch misprediction occurs, this is not the same as the real situation where a few instructions following the wrongly predicted path are processed.

3.2 Decoder

The decoder module performs instruction decoding by table look-ups, as illustrated in Fig. 5. An instruction queue, comprising two buffers arranged in a circular ring, is used to match speed difference between the fetcher and decoder. When instructions in a buffer are being decoded, additional instructions from the fetcher are placed into the other buffer. If the instruction queue is full, the decoder will prevent the fetcher from sending.

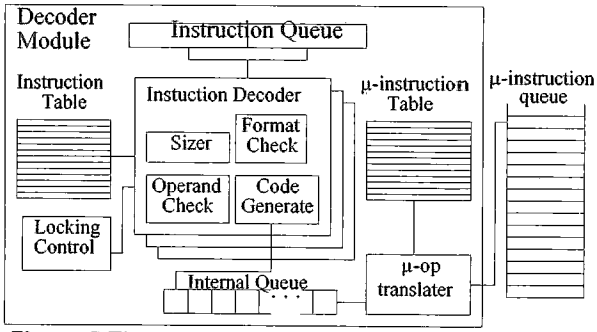

Figure 5 The structure of the decoder module.

The decoder decodes object code in the queue by looking up the instruction table, and then translates them into equivalent μ-instructions by looking up the μ-instruction table. Translated μ-instructions are placed into the μ-instruction queue and when it is full, the decoder must stop decoding. Like the fetcher, the decoder has an internal queue for storing concurrent status.

3.3 Reservation Station and Functional Units

The reservation station in the baseline model can be configured as centralized, distributed, or hybrid. To support these options, several functional unit modules are associated with a reservation station module, called a

hybrid module. Fig. 6 shows the structures for the functional unit, reservation station, and hybrid modules.

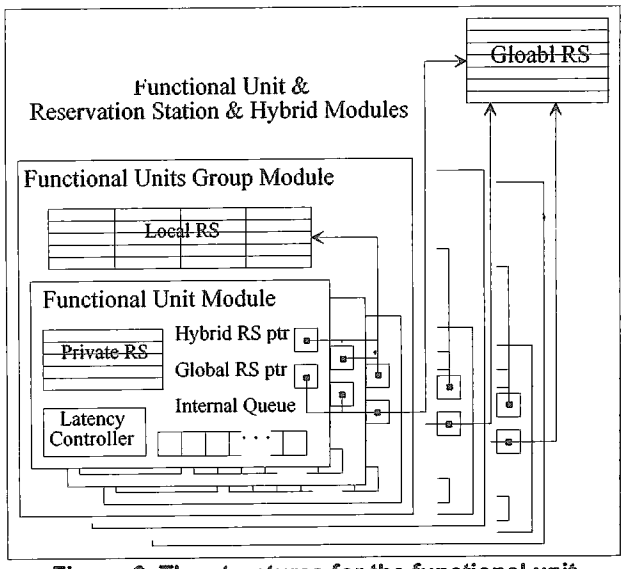

Figure 6 The structures for the functional unit, reservation station, and hybrid modules.

Each functional unit module includes a private reservation station, an internal queue, and a latency control mechanism. Private reservation stations are used to support the distributed reservation station option. Each hybrid module includes several functional unit modules and a local reservation module, which is shared by all of them within. Local reservation stations are used to support the hybrid reservation station option. Only one global reservation station exists, which is shared by all functional unit modules, and is used to support the centralized reservation option. Two pointers are provided for each functional unit, facilitating the accesses to a local and the global reservation stations, respectively. Depending on what reservation station option is chosen, each functional unit will access an appropriate reservation station, be it private, local, or global.

The reservation station is responsible for scheduling and dispatching μ-instructions. The reservation station locates μ-instructions that are free of data dependency and dispatches them to idle functional units for execution. Ideally, the reservation station could schedule any μ-instruction out of order, subject only to data dependency. However, certain restriction imposes upon scheduling rules because of requiring a sequential memory access order. That is, stores must be issued with respect to their program order. Loads between two consecutive stores, in contrast, can be issued out of order. Other than that, loads must be issued in order.

Every functional unit does not actually execute μ-instructions. Instead, they are held in a queue for several cycles, determined by the time needed for executing them. When the latency for executing a μ-instruction expires, the functional unit sends a completion signal to the result arbitrator. The result arbitrator handles concurrent complete results and determines sequences of reporting them back to the ROB, subject to a predetermined, reconfigurable structure of internal interconnection buses. Details will be discussed later in this section.

We need to consider hardware resources usage when the distributed reservation station option is chosen. In many cases, only one of the functional units in the same hybrid module is able to receive a μ-instruction per cycle, because of their sharing the same dispatching bus. In some cases, functional units, such as load, store and AGU, each have their respective dispatching buses, and each are capable of receiving a μ-instruction per cycle. This is because memory accesses occur very frequently in a superscalar processor, therefore to improve memory access capability extra hardware is mostly justifiable.

3.4 Reorder Buffer

The reorder buffer records the order of μ-instructions and performs register renaming and speculative execution. Fig. 7 shows the structure of the reorder buffer. Since there is no real instruction execution in the simulator, we chose not to implement register renaming for the sake of simplicity. However, we do support dependency checking, which determines if instructions cannot be issued due to dependencies with previous instructions. For some instruction sets, X86 instruction set in particular, instructions use flags or partial registers as sources. In these cases, reorder buffer also needs to support flag dependency checking and partial register dependency checking.

The retirement control in the reorder buffer checks the last several μ-instructions in the retirement window. However, different instructions may be translated into different numbers of μ-instructions, and the reorder buffer must retire μ-instructions in program order and handle precise interrupts, therefore the μ-instructions translated from the same instruction as well as in the retirement window must all be completed before they can be safely retired. Consequently, the reorder buffer checks if the μ-instructions in the retirement window are completed or not. Furthermore, if the last μ-instruction in the retirement window is not the last μ-instruction of an instruction, the reorder buffer waits till all the translated μ-instructions have been completed.

A retire scheme may require the reorder buffer to wait for all instructions in the retire window to finish, and hence can become a bottleneck for system performance. So, we designed another retirement scheme that allows

partial retirement, i.e., the reorder buffer can retire those μ-instructions in the retire window which were translated from the first instruction and have been completed.

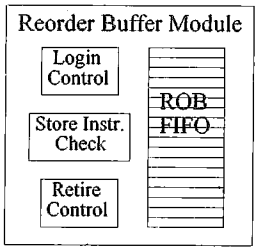

Figure 7 The structure of the reorder buffer module.

3.5 Memory Reorder Buffer and Result Arbitrator

The memory reorder buffer contains storage for facilitating the simulation of load, store and load forward operations. The structure for the memory reorder buffer is depicted in Fig. 8. It controls memory reference operations and maintains data access in instruction order to support precise interrupt and data store to memory in program order. When μ-instructions pass through the load, store and address generation unit, they are stored. But before loads are stored, they must be checked against the stores addresses for possible load forwarding.

The result arbitrator receives the output from the functional units, MOB and data cache and controls the sequences of writing results produced by the functional units back to the ROB. Because there is a limited number of result buses connected to the ROB, if the number of results produced by the functional units is larger than what ROB can handle, some information may be lost. Thus, we need the result arbitrator to control this writeback sequence. On the other hand, to avoid some results waiting too long to be written back to ROB, the result arbitrator adopts the round-robin policy for the writeback.

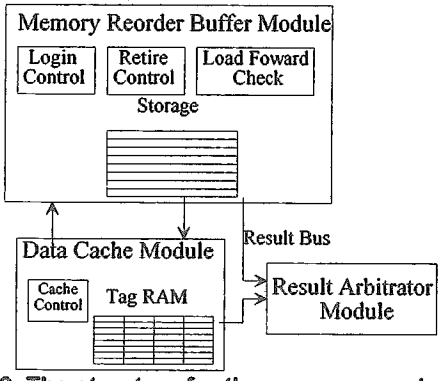

Figure 8 The structure for the memory reorder buffer module.

4. X86 Superscalar Processor Example

In this section, we show how to use the previously described simulator to simulate an x86 superscalar architecture, such as the Intel Pentium Pro. The pipeline stages of the baseline processor are shown in Fig. 9, while the system configuration is depicted in Fig. 10. A configuration file for this processor is shown in the Appendix. In this simulator, we assume that the instruction and data caches are prefect, i.e., no cache misses. The branch target buffer is perfect as well so that there is no miss for the branch prediction. We use this simulator to examine two things: how flag dependency degrades system performance, and the trade-off between full retirement scheme and partial retirement scheme. DOS™ debugger was used to collect program trace. Table 2 shows the profiles of benchmark programs that we collected trace for.

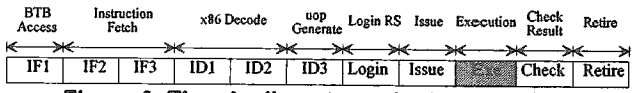

Figure 9 The pipeline stages in the baseline processor model.

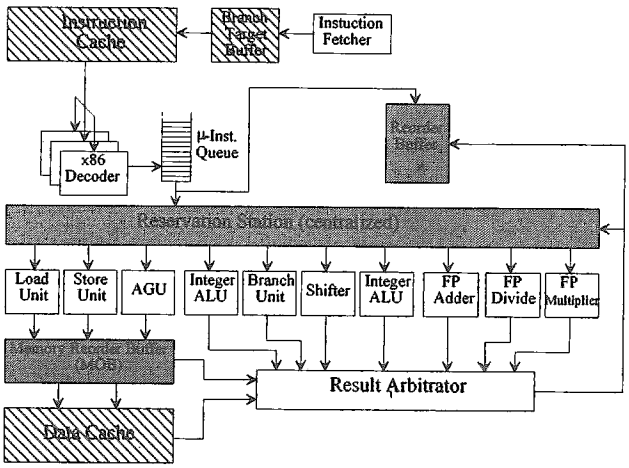

Figure 10 System configuration for simulation.

These benchmarks include a CPU performance test benchmark, a C compiler, an application program and a graphic display program. They were run on the simulator for the following 4 cases, and the results are shown in Table 3.

Case 1. The ROB uses full retirement scheme with flag dependency check.

Case 2. The ROB uses full retirement scheme without flag dependency check.

Case 3. The ROB uses partial retirement scheme with flag dependency check.

Case 4. The ROB uses partial retirement scheme without flag dependency check.

From Table 3 changing the retirement scheme can improve performance up to 1.71%-7.66%. On the other hand, not checking the flag dependency can only improve performance up to 0.91%-2.72%. Hence, we conclude that changing the retirement scheme can better improve system performance than without checking flag dependency. And this improvement due to allowing partial retirement accounts for about 4%.

Another item we wanted to investigate on the simulator was the parallelism of the benchmark programs. To obtain the program parallelism, we maximize the machine parallelism by letting each functional unit have one bus connected to the reservation station, i.e., each functional unit has its own group and each functional unit can receive one μ-instruction from the reservation station. The simulation results are shown in Table 4 (assume ROB entry number is 40). As the requirement of full retirement and flag dependency check being relaxed, the issue rates for the four benchmark programs tend to increase with one exception that the issue rate for benchmark bcc in case 3 is larger than that in case 4. After analyzing their statistics in issuing μ-instructions, we found that though enforcing flag dependency check stalls certain μ-instructions, a certain number of subsequent μ-instructions in benchmark bcc were able to bypass, which accounts for an increase in its issue rate. This indicates that flag dependency in benchmark bcc is less tightly-coupled as compared to the other three benchmarks.

benchmarks	x86 inst.	μ-inst.	μ-inst./x86 inst.
bcc	100000	233847	2.3385
bytem16	100000	231801	2.3180
jpg2gif	100000	330555	3.3056
gifshow	100000	233014	2.3301

Table 4 Average number of μ-instructions per x86 instruction for each benchmark.

From the issue rate history, we found that the program parallelism falls between 1 and 4 μ-instructions, which implies that program execution time is limited by the program parallelism. At most 4 μ-instructions can be issued in a cycle. From Table 5 we see that on an average an x86 instruction is translated into 2.3 to 3.3 μ-instructions. Therefore, the program parallelism is 1.2 to 1.7 x86 instructions. This explains why allowing partial retirement in ROB can just improve performance up to 7%.

5. Conclusions

In this paper, we presented an object-oriented trace-driven simulator to help designers determine various system parameters encountered in a superscalar CISC processor design. Its behavior is configurable through configuration files to accommodate different design considerations and trade-offs. To demonstrate its flexibility and versatility we simulated four DOS benchmark programs on the simulator to investigate the performance impact of partial retirement scheme and flag dependency checking.

As far as the four benchmark programs were concerned, we found that flag dependency in these benchmarks is not serious, and thus checking the flag dependency just slightly lowers system performance. Second, allowing partial retirement improves system performance up to 7%, with an average of 4%. Finally, the program parallelism of these benchmarks was found to be small, roughly 1.2-1.7 x86 instructions.

Trace-driven simulation is very efficient in that it only concentrates on simulation control rather than execution. We believe that our simulator has successfully served the purpose of helping designers explore different alternatives of the design issues facing them, and estimate the prospective performance if they chose to use a particular implementation.

References

[1] S.S. Shang, et al., "Design Issues in Developing A Superscalar CISC Processor," Technical Report, ITRI/CCL, 1996.

[2] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[3] J.L. Hennesy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publisher Inc., 1990.

[4] Allen I. Holub, *C+C++ Programming with Objects in C and C++*, McGraw-Hill Inc., 1992.

[5] Johnny K. F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, Jan 1984, pp.6-22.

[6] C.H. Perleberg and A.J. Smith, "Branch Target Buffer Design and Optimization," IEEE Trans. on Computers, Vol. 42, No. 4, April 1993, pp. 396-412.

[7] Tom R. Halfhil, "Intel's P6," Byte Magazine, April 1995, pp.42-58.

[8] Sebastian Rupley and John Clyman, "P6: The Next Step?," PC Magazine, Sep. 1995, pp. 102-137.

[9] Linley Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," Microprocessor Report, Feb 16, 1995, pp. 9-15.

Benchmark	Total Inst.	Branch Inst.	Uncond. Branch	Cond. Branch	Call/Return	Taken	Not Taken	Static Branch
bcc	5643843	687440	143	541060	58504/87733	0.6807	0.3193	150
bytem16	3667451	451595	27369	298100	90100/36026	0.5192	0.4808	364
jpg2gif	1396510	300528	25156	275372	0/0	0.6098	0.3902	53
gifshow	5665958	676854	64477	445882	78324/88171	0.5084	0.4916	594

Benchmarks	Description	Command Line
bcc	Borland C++ command line compiler	bcc bgidemo.c
bytem16	Byte magazine CPU performance test benchmark	bytem16
jpg2gif	JPG to GIF transfer program	jpg2gif picture.gif
gifshow	GIF display program	gifshow picture.gif

Table 2 Benchmark statistics and descriptions.

bcc

	Exec Time	Speedup	Speedup	Speedup
case 1	96744	1.0000		
case 2	94888	1.0196	1.0000	
case 3	93021	1.0400		1.0000
case 4	93289	1.0370	1.0171	0.9971

bytem16

	Exec Time	Speedup	Speedup	Speedup
case 1	106108	1.0000		
case 2	103300	1.0272	1.0000	
case 3	98556	1.0766		1.0000
case 4	97083	1.0930	1.0640	1.0152

gifshow

	Exec Time	Speedup	Speedup	Speedup
case 1	118772	1.0000		
case 2	117706	1.0091	1.0000	
case 3	114912	1.0336		1.0000
case 4	113591	1.0456	1.0362	1.0116

jpg2gif

	Exec Time	Speedup	Speedup	Speedup
case 1	205917	1.0000		
case 2	202986	1.0144	1.0000	
case 3	201613	1.0213		1.0000
case 4	198220	1.0388	1.0240	1.0171

Table 3 The simulated execution cycles of each benchmark with a 40-entry ROB.

case1

benchmark	Exec. Cycles	Average Issue Rate
bcc	94467	2.48
bytem16	103730	2.23
jpg2gif	197819	1.67
gifshow	116846	1.99

case2

benchmark	Exec. Cycles	Average Issue Rate
bcc	94393	2.48
bytem16	102526	2.26
jpg2gif	195891	1.69
gifshow	116685	2.00

case3

bcc	92452	2.53
bytem16	97277	2.38
jpg2gif	193212	1.71
gifshow	111679	2.09

case4

bcc	94033	2.49
bytem16	96635	2.40
jpg2gif	191164	1.73
gifshow	111455	2.09

Table 5. Average issuing rate for each case.

Appendix. A configuration sample file for Pentium Pro class processor.

```
// The Machine Configuration File
Instruction_Cache
    size_bytes          16384
    bytes_per_block     32
    associativity       4
    miss_penalty        4
Data_Cache
    size_bytes          16384
    bytes_per_block     32
    associativity       4
    miss_penalty        4
Branch_Target_Buffer
    Two_Bit_Counter     //prediction mechanism, e.g. GAg, etc
    entries             128
    associativity       4
Central_Window          40    // reservation station type
Fetcher
    pipe_stage          3
    buffer_size         16
    buffer_entries      2
Decoder
    instance            3
    pipe_statge         2
Reorder_Buffer
    entries             40
    login_no            4
    retire_no           4
// Functional Unit Define
// Function name/ mnemonic name
Integer IU
    function_group      0           // binding group
    instance            1           // no. of instances
    issue_latency       [1, 1, 1]
    result_latency      [1, 1, 1]
    reservation_station_depth   4   // distributed RS no.
Integer_Mul I_MULT
    function_group      0
    instance            1
    issue_latency       [10, 10, 10]
    result_latency      [10, 10, 10]
    reservation_station_depth   4
Integer_Div I_DIV
    function_group      0
    instance            1
    issue_latency       [20, 32, 45]
    result_latency      [20, 32, 45]
    reservation_station_depth   4
Shifter
    function_group      0
    instance            1
    issue_latency       [1, 2, 4]
    result_latency      [1, 2, 4]
    reservation_station_depth   2
Integer IU
    function_group      1
    instance            1
    issue_latency       [1, 1, 1]
    result_latency      [1, 1, 1]
    reservation_station_depth   4
Branch BU
    function_group      1
    instance            1
    issue_latency       [1, 1, 1]
    result_latency      [1, 1, 1]
    reservation_station_depth   4
Address_Gen AGU
    function_group      2
    instance            1
    issue_latency       [1, 1, 1]
    result_latency      [1, 1, 1]
    reservation_station_depth   4
Load  LOAD
    function_group      2
    instance            1
    issue_latency       [1, 2, 4]
    result_latency      [2, 3, 5]
    reservation_station_depth   4
Store  STORE
    function_group      2
    instance            1
    issue_latency       [1, 2, 4]
    result_latency      [2, 3, 5]
    reservation_station_depth   4
Float_Add
    function_group      3
    instance            1
    issue_latency       [1, 1, 4]
    result_latency      [2, 2, 6]
    reservation_station_depth   2
Float_Mul
    function_group      3
    instance            1
    issue_latency       [1, 1, 4]
    result_latency      [4, 5, 6]
    reservation_station_depth   2
Float_Div
    function_group      3
    instance            1
    issue_latency       [12, 19, 32]
    result_latency      [12, 19, 32]
    reservation_station_depth   2
Float_Conv
    function_group      3
    instance            1
    issue_latency       [1, 1, 4]
    result_latency      [2, 2, 4]
    reservation_station_depth   2
```