Technical Report NCSU-ERL-94-18

Performance Prediction for Superscalar Processors

Sanjeev Banerjia Eric Schweitz Mark Vilas
Saurabh Misra

Department of Electrical and Computer Engineering
North Carolina State University
Box 7911
Raleigh, NC 27695-7911

1 Introduction

An effort is underway in the Electronics Research Laboratory at N.C. State University to gauge the advantages of multichip modules in building microprocessors. As the first step in this project, a single–chip superscalar processor is being designed and built. To explore the design space of microarchitectures for the design, we have run simulations on the proposed design for the ERL chip—named SAND, for **S**uperscalar **A**rchitecture **N**CSU **D**esign—as well as other contemporary superscalar processors to gauge what the expected performance would be. The metric we used to evaluate performance is cycles per instruction (CPI) as defined in a text on computer architecture [5]. We measured the performance of the processors as a whole, but did not perform detailed studies on the effects of a very important architectural parameter: caches. A second study was carried out to separataley analyze caches in detail.

The report is organized as follows. Section 2 details the software simulator used for architecture simulation. Sections 3 thru 6 present results of the simulations. Section 7 details lists modifications to the simulator to allow greater flexibility. Section 8 presents conclusions and directions for further research.

2 Simulation Framework

The motivation behind the processor subgroup's work was to aid in the development of SAND and also gain insight into the effects of architectural features on superscalar processor performance. We decided to model the performance of several current commercial chips, to have an idea where SAND would stand comparatively. We decided to simulate a total of four chips: SAND , PowerPC 601 from IBM/Apple/Motorola, POWER from IBM, and Alpha 21064 from DEC.

We decided to use a simulator written at Stanford University, ssim, which stands for *S*uperscalar *SIM*ulator; ssim was written primarily by Mike Johnson. ssim simulates a "dynamically scheduled superscalar processor", using the pixie program and a user program to be simulated. Details on ssim and using pixie can be found in documents available from Stanford University [2, 8]; many of the terms used in the description and operation of ssim are described in Mike Johnson's book on superscalar processors [1]. ssim makes use of two sources of input to determine the processor organization to be simulated: a machine file and command–line options. The machine file is an input file that contains a textual description of the processor, with details such as

System/Processor	CPI
Sun SPARCstation 2/scalar SPARC	1.22–1.93
Sun IPC/scalar SPARC	1.25–2.60
Sun SPARCserver 10/30/SuperSPARC	0.66–1.19
DECstation 5000/240/MIPS R3000	1.02–2.19
Bare SuperSPARC processor (assumes perfect memory)	0.625–0.71

Table 1: "Unofficial" CPIs for commercial processors

cache organization, instruction issue and decode width, number and types of functional units, latencies for functional units (which can vary depending on the precision i.e. single or double), number of reservation stations if distributed instruction issue is used, and other parameters. ssim's command–line options can overide certain machine file specifications and can specify additional parameters, such as the level of branch prediction to be used, number of register ports, whether or not to simulate cache miss effects, and many others. ssim assumes a four stage pipeline with fetch, decode, execute, and writeback stages. Examples of ssim machine files are the POWER mpe and POWER mps listings on pages 20 and 22 in Appendix B; a list of ssim command–line options is on page A in Appendix A.

ssim was developed to model the MIPS instruction set architecture (ISA). Although this is a shortcoming when faithfully modeling all aspects of a non–MIPS processor, our focus was on organization analysis, not ISA. Also, the MIPS ISA was chosen for SAND, so we weren't limited by ssim's MIPS bias.

In order to make our benchmarking as "real–world" as possible, we used several programs from the SPEC suite for integer performance evaluation; the SPEC floating–point programs were not run due to a lack of time.

We also attempted to gather performance information from the public domain about commercial processors, using CPI as the metric. This was difficult to obtain from "official" sources, as most processors use various popular benchmark–oriented numbers to illustrate their performance; we were unable to find any literature that quoted CPIs for contemporary superscalar processors. However, using a form of the public domain, the Usenet news system, we were able to procure some unofficial numbers. The information we gathered is summarized in Table 1. The SuperSPARC–based systems are the only superscalar chips for which we received information. SuperSPARC is a two–issue processor, which means its ideal CPI is 0.50. Its posted CPIs in the 0.625–1.19 range seem reasonable for such a chip.

3 SAND results

The architecture of SAND is detailed in a variety of documents that are available online at N.C. State University. Briefly, SAND is a dynamically scheduled superscalar processor: the hardware supports reordering of individual instructions. Instructions can be issued and completed out–of–order. The chip decodes 4 instructions per clock cycle, and can issue 4 instructions per cycle also. There are separate instruction and data caches, the sizes of which are not fixed at this time. A reorder buffer is used for detecting and handling hazards (via register renaming) and exception handling. A static branch prediction is employed (backward branches are predicted taken and forward branches are predicted not taken).

A modified version of ssim, esim (described in Section 7), was used for the simulations of SAND, as it allowed the simulations to closely match the architecture. Particularly useful were the instruction–to–functional unit assignments, and the Func_Unit definitions. However, neither esim nor ssim accurately model some aspects

esim options	# of BTB entries				
	No BTB	128	256	512	2048
Entries below simulate split 8 kb direct–mapped caches					
Simulate I– and D–caches	1.31				
Simulate I– and D–caches, predict branches with I–cache	0.92				
Simulate I– and D–caches, predict all branches taken	0.95				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.89	0.89	0.89	0.89
BTB associativity = 2		0.89	0.89	0.89	0.89
BTB associativity = 1		0.91	0.90	0.89	0.89
All entries below assume split 8 kb 2–way set associative caches					
Simulate I– and D–caches	1.30				
Simulate I– and D–caches, predict branches with I–cache	0.92				
Simulate I– and D–caches, predict all branches taken	0.95				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.88	0.88	0.88	0.88
BTB associativity = 2		0.88	0.88	0.88	0.87
BTB associativity = 1		0.90	0.89	0.88	0.88
Entries below simulate 16 kb direct–mapped caches					
Simulate I– and D–caches	1.31				
Simulate I– and D–caches, predict branches with I–cache	0.93				
Simulate I– and D–caches, predict all branches taken	0.96				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.88	0.87	0.87	0.87
BTB associativity = 2		0.88	0.87	0.87	0.87
BTB associativity = 1		0.89	0.88	0.87	0.87

Table 2: CPIs for the gcc SPEC benchmark on SAND

of the chip. SAND uses a unified reorder buffer; ssim/esim models separate reorder buffers for integer and floating–point operations. SAND uses one set of buses to pass operands to all functional units (both integer and floating–point), and one set of result buses also; ssim/esim models separate operand and result buses for integer and floating–point operations. SAND also has complex capabilities to pass operands for an instruction on successive clock cycles, which also isn't modeled by ssim/esim. ssim/esim doesn't model the static branch prediction scheme or the timing of branch prediction with respect to the pipeline stages when branch prediction is used.

The SPEC programs gcc, li, eqntott, and li were simulated for SAND. Though the cache organization is not fixed at this time, the cache design group is examining three organizations: 8 kb direct–mapped, split I– and D–caches; 8 kb 2–way set associative, split I– and D–caches; and 16 kb direct–mapped, split I– and D–caches. These three alternatives were examined in the simulation runs. Other architectural features were also examined. Various levels of branch prediction were modeled, in order to interpolate/estimate what level is reasonable to expect, since the exact prediction scheme isn't modeled. Use of a branch target buffer (BTB) for dynamic branch prediction was also modeled. A sample machine file for SAND is listed on page B in

	# of BTB entries				
esim options	No BTB	128	256	512	2048
Entries below simulate 8 kb direct–mapped caches					
Simulate I– and D–caches	1.57				
Simulate I– and D–caches, predict branches with I–cache	1.42				
Simulate I– and D–caches, predict all branches taken	1.47				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		1.28	1.26	1.25	1.25
BTB associativity = 2		1.29	1.26	1.25	1.25
BTB associativity = 1		1.29	1.26	1.25	1.25
Entries below simulate 8 kb 2–way set associative caches					
Simulate I– and D–caches	1.38				
Simulate I– and D–caches, predict branches with I–cache	1.24				
Simulate I– and D–caches, predict all branches taken	1.30				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		1.20	1.19	1.18	1.18
BTB associativity = 2		1.22	1.19	1.18	1.18
BTB associativity = 1		1.26	1.23	1.20	1.19
Entries below simulate 16 kb direct–mapped caches					
Simulate I– and D–caches	1.41				
Simulate I– and D–caches, predict branches with I–cache	1.27				
Simulate I– and D–caches, predict all branches taken	1.32				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		1.14	1.12	1.11	1.11
BTB associativity = 2		1.15	1.12	1.12	1.12
BTB associativity = 1		1.19	1.17	1.13	1.12

Table 3: CPIs for the li SPEC benchmark on SAND

	# of BTB entries				
esim options	No BTB	128	256	512	2048
Entries below assume 8 kb direct–mapped caches					
Simulate I– and D–caches	1.42				
Simulate I– and D–caches, predict branches with I–cache	1.05				
Simulate I– and D–caches, predict all branches taken	1.04				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		1.04	1.04	1.04	1.04
BTB associativity = 2		1.05	1.04	1.04	1.04
BTB associativity = 1		1.05	1.05	1.05	1.04
Entries below simulate 8 kb 2–way set associative caches					
Simulate I– and D–caches	1.34				
Simulate I– and D–caches, predict branches with I–cache	0.97				
Simulate I– and D–caches, predict all branches taken	0.99				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.96	0.96	0.96	0.96
BTB associativity = 2		0.97	0.96	0.96	0.96
BTB associativity = 1		0.97	0.97	0.97	0.96
Entries below simulate 16 kb direct–mapped caches					
Simulate I– and D–caches	1.38				
Simulate I– and D–caches, predict branches with I–cache	1.01				
Simulate I– and D–caches, predict all branches taken	1.00				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		1.01	1.01	1.01	1.01
BTB associativity = 2		1.01	1.01	1.01	1.01
BTB associativity = 1		1.01	1.01	0.98	0.97

Table 4: CPIs for the eqntott SPEC benchmark on SAND

	# of BTB entries				
esim options	No BTB	128	256	512	2048
Entries below simulate 8 kb direct–mapped caches					
Simulate I– and D–caches	1.02				
Simulate I– and D–caches, predict branches with I–cache	0.98				
Simulate I– and D–caches, predict all branches taken	1.04				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.98	0.98	0.98	0.98
BTB associativity = 2		0.98	0.98	0.98	0.98
BTB associativity = 1		0.98	0.98	0.98	0.98
Entries below simulate 8 kb 2–way set associative caches					
Simulate I– and D–caches	1.02				
Simulate I– and D–caches, predict branches with I–cache	0.98				
Simulate I– and D–caches, predict all branches taken	1.04				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.98	0.98	0.98	0.98
BTB associativity = 2		0.97	0.98	0.98	0.98
Entries below assume 16 kb direct–mapped caches					
Simulate I– and D–caches	0.74				
Simulate I– and D–caches, predict branches with I–cache	0.68				
Simulate I– and D–caches, predict all branches taken	073				
Entries below simulate D–cache and use the BTB for branch prediction					
BTB associativity = 4		0.68	0.68	0.68	0.68
BTB associativity = 2		0.68	0.68	0.68	0.68
BTB associativity = 1		0.68	0.68	0.68	0.68

Table 5: CPIs for the compress SPEC benchmark on SAND

Appendix B.

The results of the simulations are presented in Tables 2–5. The largest performance gain for all benchmarks was accomplished through the use of some sort of branch prediction. For gcc, using ssim's static scheme yielded about a 27% reduction in CPI and using the I–cache for dynamic prediction [1, pp.71–76] yielded a 29–30% reduction, irrespective of cache size. Use of a BTB reduced the CPI by 33–34%. With gcc, cache size and BTB associativity did not have a large influence on accurately predicting branches and therefore reducing CPI (that is, although CPI was reduced through the use of hardware branch prediction, using *additional* hardware didn't reduce CPI substantially). The opposite was true for the li benchmark. Use of a 2048 entry, 4–way set associative BTB in a 16 kb direct–mapped cache versus a 128 entry, direct–mapped BTB in a 8 kb direct–mapped cache yielded a 14% reduction in CPI. The eqntott benchmark exhibited similar behavior, but was not as sensitive to cache and BTB size. Compress showed relatively little improvement when using branch prediction or a larger cache, except when the largest cache size was used, which yielded a 25–31% improvement.

Simulations were run with gcc to model the effects of modifying the SAND architecture, for performance improvements. For example, with gcc, 82% of the instructions issued to the Shifter unit are delayed for 2 or more cycles. We added an extra shifter to see if that would increase the throughput, since the delays were not caused by full reservation stations. This did not have an impact on CPI, though, so it seems as though there is a limitation in the gcc code of how much instruction level parallelism (ILP) can be achieved, at least with respect to instructions assigned to the shifter. Additional Load/Store units were also added, without any positive effect.

4 POWER results

POWER, as implemented in the RS/6000 chipset, does not fit into Mike Johnson's superscalar architecture model very neatly at all. First, POWER's ISA is not a "typical" RISC ISA. It has some aspects of the RISC philosophy, but has complex instructions such as string operations and the often referred to multiply–add instruction. Secondly, the chipset borrows some ideas from the DAE architecture, in that the fetch and decode unit is running ahead of the functional operation units and feeding these unit's input queues with instructions [7]. The general idea is to buffer enough instructions in the functional units reservation stations, so that the functional unit is kept busy while the decoder resolves branches and other instruction stream discontinuities. With some scheduling, this technique can often eliminate branch penalties. Finally, POWER uses register renaming in its floating point unit (FPU), but does not use it in its fixed point (integer) unit (FXU). This does not fit ssim easily, as there is no way to turn dependency checking on for one functional unit and off for another.

POWER was simulated for the eqntott, compress, gcc, li, and espresso benchmark programs. Both the ssim and esim simulators were used as well as two flavors of ssim machine file. We chose to use two different machine files because different information sources indicated different parameter values. These two machine files are known as the machine.power.saurabh (mps) and machine.power.eric (mpe), listed on pages B and B, respectively, in Appendix B.

The principle benchmarks used were compress and eqntott. ssim was run to completion on compress for both mps and mpe machine files without options. This simulation process took over 10 hours per simulation. Because the simulation time was so long the group decided to use the -s option on the simulator to terminate the simulation after a few million cycles. Compress was run again with the stop value set to 10,000,000 instructions and eqntott was run with stop set to 5,000,000 instructions. The 5,000,000 instruction simulation did not take as long to complete, yet seemed to provide reasonable results (Johnson also stated that after 5M instructions, he found most of the transients had died out of the simulation). All subsequent simulations used a stop value of 5 million instructions.

	Machine file		
ssim/esim options	mps	mpe	mpee
No options	0.78	0.98	
Check for output and anti–dependencies and issue in order	0.99	1.37	
Simulate I– and D–caches			1.27
Simulate I– and D–caches, predict all branches taken			1.32
Simulate I– and D–caches, predict branches with I–cache			1.18
Simulate I– and D–caches, predict branches with I–cache, two integer writeback buses			1.09
Simulate I– and D–caches, predict all branches taken, two integer writeback buses			1.25
Simulate I– and D–caches, predict all branches taken, 16k fully–associative caches		1.05	

Table 6: CPIs for the compress SPEC benchmark on POWER

To determine the effects of dependency checking the eqntott and compress simulations were run with to -da, -do, and -issue_in_order options for mps and mpe. These options enforce anti-dependency checking, output dependency checking, and reservation station issue respectively. Performance decreased by 15–30the first runs.

In order to see the effects of the simulated cache, the mps and mpe files caches parameters were modified such that their size was 16,384 bytes, words per line was 4, the associativity was 128, and the miss penalty was kept the same at 12. Three more simulations were then run with the -icache -dcache and -pred_take command line options. Eqntott with the original mps and mpe files was run with the -icache and -dcache options for the sake of comparison. The larger cache did show some improvement.

Next the mpe file was modified for use on esim and several of the default functional units were merged. This modified mpe file is the machine.power.eric.esim file (mpee), listed on page B in Appendix B. The mpee file was used to simulate eqntott, compress, gcc, espresso (with 4 different input files), and also li. Each benchmark was run three times to determine the performance of the caches and branch prediction strategies. The first run used the -icache and -dcache flags only, the second used the -icache, -dcache, and -pred flags, and the final run used -icache, -dcache, and -pred_take. The -pred flag enabled the BTB and icache branch prediction and -pred_take predicted all branches as taken in the simulation. While -pred and -pred_take increased the number of instructions per cycle, the results between the two options were varied. This suggests that neither prediction method was superior in general, but could be superior given a specific program.

While running simulations, it was observed that performance could be improved by increasing the result bus width, and thereby increasing the number of results which could be written to the reorder buffer in a single clock. Intuitively, this makes sense in that allowing both the load/store unit and the integer unit to write results concurrently would eliminate latencies introduced by bus arbitration. Again, esim was used to simulate mpee with first with the -icache, -dcache, -pred, and -resbus_int2 switches and again with -icache, -dcache, -pred_take, and -resbus_int2. The results are shown in the table. While performance was improved, it was only by 3–7%.

	Machine file		
ssim/esim options	mps	mpe	mpee
Check for output and anti–dependencies and issue in order	1.04	1.39	
Simulate I– and D–caches	1.09	1.37	1.37
Simulate I– and D–caches, predict all branches taken			1.18
Simulate I– and D–caches, predict branches with I–cache			1.11
Simulate I– and D–caches, predict branches with I–cache, two integer writeback buses			1.09
Simulate I– and D–caches, predict all branches taken, two integer writeback buses			1.12
Simulate I– and D–caches, predict all branches taken, 16k fully–associative caches	1.04	1.27	

Table 7: CPIs for the eqntott SPEC benchmark on POWER

ssim/esim options	**mpee machine file**	
	gcc	li
Simulate I– and D–caches	1.56	1.61
Simulate I– and D–caches, predict all branches taken	1.18	1.49
Simulate I– and D–caches, predict branches with I–cache	1.41	1.52

Table 8: CPIs for the gcc and li SPEC benchmarks on POWER

ssim/esim options	**mpee machine file**			
	espresso input file			
	bca	cps	ti	tial
Simulate I– and D–caches	1.33	1.27	1.35	1.41
Simulate I– and D–caches, predict all branches taken	1.22	1.12	1.18	1.30
Simulate I– and D–caches, predict branches with I–cache	1.23	0.98	1.22	1.33

Table 9: CPIs for the espresso SPEC benchmark on POWER

Entries assume perfect caches and no branch prediction			
	# of BTB entries		
	128	512	2048
BTB associativity = 4	1.826	1.781	1.778
BTB associativity = 2	1.837	1.785	1.778
BTB associativity = 1	1.870	1.801	1.790

Comparison between # of reservation stations (Reservation station distribution is uniform among all FUs)			
# of instructions issued & decoded/cycle	**# of reservation stations**		
	2	4	8
2	1.988	1.988	1.988
3	1.982	1.982	1.982
4	1.978	1.978	1.978

Table 10: CPIs for the li SPEC benchmark on the Alpha 21064

Results of the simulations are shown in Tables 6–9.

5 Alpha 21064 results

The Alpha 21064 is the first chip in a family of chips from DEC [3, 6]. In terms of raw benchmark performance, it is one of the fastest cpu's available. One observation about the implementation is that it is mainly optimized for clock speed, as opposed to CPI. There are 150 and 200 Mhz versions available, which are two to three times faster than most other current generation microprocessors. The hardware on–chip reflects this design approach, as most features are a bit less complex than its counterparts. For example, it is a two–issue superscalar machine, but, for the most part, instructions begin and complete in order. There is no register renaming to eliminate anti– and output–dependencies. This can be seen in the reservation station number comparisons. CPI does not rise at all with an increased number of reservation stations, because the rigid issue and completion requirements do not require instructions to be stored at the functional units.

Certain options were used with esim in order to better simulate the Alpha architecture. These options allowed esim to:

- add 2 additional stages to the decoder,
- allocate an entry in the result buffer for every instruction,
- issue and complete instructions in–order,
- check for output and anti–dependencies,
- issue a max of 2 instructions per cycle,
- disallow a load to bypass a store,
- prearbitrate results buses during decode, and
- limit the store buffer to 4 entries.

The 21064 has two different branch prediction strategies. It has a 2k by one bit branch history buffer in the instruction cache, which is similar to a branch target buffer, and it can also use a static prediction scheme. In my simulations, I found that dynamic branch prediction performed only marginally better, 2.9%. I also found that out–of–order issue and completion with register renaming increased performance by 7.6%. Implementing a four instruction decoder barely increased performance at all. Johnson [1] suggests that using only one or two of his recommendations (out- of–order issue/completion, register renaming, dynamic branch prediction, and four issue decoder) without the rest can hamper the effectiveness of those that are implemented. These simulations support that suggestion, in that the biggest increase is seen when two are used together: out–of–order issue and completion, and register renaming.

Simulation results for Alpha are shown in Table 10.

6 PowerPC 601 results

The ssim superscalar simulator allows the user to modify several architectural features to observe effects on performance. The ultimate indicator of performance is the CPI, which can be calculated using the output provided by ssim at the end of a simulation. The machine file is the interface used by ssim to define a particular architecture. Among the parameters that the user can define in the machine file are the cache configuration, the BTB, the instruction bus width, number of functional units for integer, branch, load and store units and latencies for each functional unit. The correct usage of ssim would involve using output statistics only to compare two different configurations of the same architecture. If, for example, we wish to improve branch prediction then we can change the BTB configuration and the number of branch processing units (BPUs) to observe how each affects the branch prediction and then make a decision based on improvement observed and cost of new configuration. The machine file does also limit the accuracy with which one can specify an architecture for simulation. For example, if we have a split cache, we cannot specify different sizes for the instruction and dataa caches. There are more limitations in accurately describing an architecture that are inherent in any structured interface format such as the machine file.

The PowerPC 601 [4] implementation of the PowerPC architecture specifies up to 3 instruction issues per cycle, one each to the integer unit, the floating point unit, and the branch prediction unit. The unified cache is 32K, and is 8–way associative with 8 words per block. The miss penalty is 9 cycles. The three execution units can execute instructions of only one type each. The integer unit has a 32 bit data width, and places results into any one of the writeback buses. The results are then stored in the register files. Enough writeback buses are present so as to support the peak issue rate of 3 instructions per cycle.

Three simulations with different machine files were performed to compare the relative effects on performance that the number of execution units, the depth of the result buffers, and branch prediction mechanism provide.

To observe the effects of different architecture modifications of the PowerPC, I used four of the SPECint benchmarks to collect CPI figures using different hardware configurations of the PowerPC. I used ssim to simulate the effects of varying the number of integer execution units, the number of integer result buses, the instruction cache size, the number of instructions decoded and issued per cycle, the Branch Target Buffer entry size and the BTB associativity. Simulations results are listed in Table 11. I noticed that for the most part CPI did not change dramatically between different variations of the same parameter. For example, in the gcc benchmark, with 3 integer execution units provided CPI remains that same as that of 1 integer execution unit. With the compress benchmark, which has a high data locality the effects of changing the instruction cache did not make any difference in CPI. This is attributed to the fact that the data cache was assumed infinite, and the compress benchmark uses a small instruction alogrithm to compress, which is stored in less than 32K.

gcc			
Perfect data caches assumed			
# of integer units	**# of instructions decoded & issued/cycle**		
	1	2	3
1	1.27	1.02	1.08
2	1.27	1.02	1.08
3	1.27	1.02	1.08

eqntott			
Simulate data cache effects			
BTB associativity	**# of BTB entries**		
	512	1024	2048
2	0.98	0.98	0.98
4	0.98	0.98	0.98
8	0.98	0.98	0.98

li			
Assume perfect data cache, decode/issue 3 instr/cycle			
# of integer units	**# of integer results buses**		
	1	2	3
1	0.97	0.92	0.92
2	0.98	0.91	0.91
3	0.98	0.91	0.91

compress			
Assume perfect data cache, decode/issue 3 instr/cycle			
# of integer units	**Unified cache size**		
	32 kb	64 kb	128 kb
1	0.88	0.88	0.88
2	0.88	0.88	0.88
3	0.88	0.88	0.88

Table 11: CPIs for PowerPC simulations

7 SSIM modifications: ESIM

As is apparent from the machine file listings in Appendix B and command–line options in Appendix A, ssim is highly configurable and very flexible. However, ssim's model of superscalar architectures is very different from the commercial CPUs that we were attempting to simulate. For example, ssim assumes that there are nine distinct functional units and two distinct reorder buffers. Unfortunately, none of the machines we were interested in modeling had reorder buffers of any kind, nor did they have so many functional units. The typical case was to have some combination of an integer unit, a fp unit, a combination load/store unit, and a branch unit. ssim also lacked the ability to add new functional units which might not have fit neatly into ssim's model.

Therefore it became apparent that ssim needed to be altered to meet our needs more directly. (Our initial attempts to set the number_available field for a functional unit to 0 caused ssim to crash.) ssim was modified to add and delete functional units from the simulator to more accurately reflect the hardware being simulated. The result of this effort was *E*ric's superscalar *SIM*ulator (esim), so named after the author of the changes Eric Schweitz.

In simple terms, Johnson's model is that of a instruction fetch/decode unit which is responsible for determining an instruction's type and then passing it on to the correct functional unit's reservation station. At that point, the functional unit will execute the instruction when its operands are made available. Thus, certain instruction types are sent to specific functional units in a pre–arranged, hard–coded relationship. Using the `-c`*N* command–line option, it is possible in ssim to combine all the reservation stations into a single central instruction window. However, this would still allow multiple instructions to be issued to several functional units which may be the same physical unit in the system being modeled. Using the FIFO central window would prevent multiple issue in cases where it may in fact be desired. Thus it seemed that the ability to control the number and type of functional units would be helpful in simulating the commercial processors we were interested in studying.

The new capability was implemented by making modifications to the machine file language used to specify the organization of the machine being simulated. Only three changes were necessary to the machine file language to support the idea of adding and deleting functional units from the machine description. The word "Func_Unit" was added to the vocabulary so that new functional units could be specified and added to the machine. The "is" verb was added to remap an existing functional unit to another existing functional unit. The final change was to add the "Instructions" word to the vocabulary. The purpose of this section of the machine file is to allow the user to map MIPS instructions directly to existing functional units. The "is" and "Instructions" mappings modify the same internal table in esim, but with a different level of user control. This is important to note, because mapping instructions through "is" can be redone (or undone) by using the "Instructions" command, possibly crashing the simulator. It is best to lay out the machine file with existing Func_Unit's first (both ones that are built-in and additional ones), map MIPS instructions to the newly added functional units (the pre–existing functional units are mapped by default), and then remove the unnecessary built-in functional units through the "is" mechanism.

Appendix B contains listings of two machine files—SAND and the POWER mpee files—that use esim's features. In the SAND listing, two new functional units are defined, Int_Mul and Int_Div, and the integer multiply/divide instructions–MULT, MULTU, DIV, DIVU–are assigned to these new units. This overides the default instruction–to–functional– unit mappings, whereby those instructions map to the Float_Div unit. Several move and shift instructions are also reassigned from their default ssim mappings, from the Integer unit to the Shifter unit. The Store unit has been removed by mapping its functionality to the Load unit; this is common in many processors, which use a combined load/store unit rather than separate units. Appendix C lists all of the MIPS instructions and their default mappings, which are customizable through the "Instructions" command in the machine file.

The code changes to implement the new features were mostly to the esim.c module in the initialization of the

program. Obviously, the routine to read and interpret the machine file was changed. Decode_Instruction() in esim.c was modified to use the new user–customizable instruction–to–functional– unit mapping table rather than hard–coded mappings. (Decode_Instruction() is the function that converts MIPS instructions into the simulator's internal instruction record format.) The file instname.h was added and includes the mapping table and some other declarations used for the new features. Only minimal changes were necessary to fdunit.c and exunit.c, and consisted mostly of making per functional unit calculations compute on newly added functional units as well as the built-in units. Of course the output routines were enhanced to print the statistics for user-added functional units as well as the built-in functional units.

8 Conclusion and Future Work

Simulation of the various architectures proved to be an interesting and beneficial task. We were able to determine the benefits, if any, of adding extra hardware to a given organization. The simulations of SAND gave a rough idea of how this processor stacks up against current commercial processors, and also highlighted potential areas for design improvement (for example, the largest cache size was not always a big win).

Future work in this area should concentrate on modifying the ssim/esim source code so that a wider variety of processor organizations can be modeled. In particular, to accurately model SAND, changes need to be made to how the reorder buffers, results buses, operand buses, and branch prediction strategy are modeled. Also, benchmarking on the SPECfp suite could be run to optimize floating–point performance.

References

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

[2] Mike Johnson and Michael D. Smith. ssim: A Superscalar Simulator. Available via ftp from velox.stanford.edu in pub/ssim/manual.ps.

[3] Edward McLellan. The Alpha AXP Architecture and 21064 Processor. *IEEE Micro*, 13(3), June 1993.

[4] Motorola. *PowerPC 601 – Risc Microprocessor User's Manual.* Motorola, 1993.

[5] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantative Approach.* Morgan–Kaufmann, 1990.

[6] Richard L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4):19–34, 1993.

[7] J.E. Smith. Decoupled access/execute architectures. *ACM Transactions on Computers Systems*, 2(4):298–308, November 1984.

[8] Michael D. Smith. Tracing with pixie. Available via ftp from velox.stanford.edu in pub/pixie_doc/manual.ps.

A Command line options for ssim

```
Command line options to 'ssim'
==============================

Options are:
        <funit_descriptor_file_name>
        -add_pipeN      add N pipeline stages to decoder
        -align          align instructions in decoder
        -arb            allocate result buffer for every instruction
        -cN             centralized window of N locations
        -compl_in_order issue and complete instructions in sequential order (default no)
        -cs_add         use carry-save adders (can issue dependent adds)
        -d[a,o]         check for [a]:anti-dependencies
                                  [o]:output dependencies
                        (true dependencies always checked)
        -dcache         simulate data cache effects (default inf)
        -dcache_portsN  N ports to data cache (default one)
        -decodeN        decode N instructions per cycle (default 2)
        -depbitsN       use N bits of address for memory dependency checking (default 32)
        -exact_pred     expect exact branch prediction (no don't care for not taken)
        -fifo_wind      manage central window as fifo
        -flsh_mispred   flush result buffer on misprediction
        -icache         simulate instruction cache effects (default inf)
        -issue_in_order issue instructions in sequential order (default no)
        -lbrnpredN      limit outstanding branch predictions to N
        -ldep[tN,aN,oN] stall decoder after N[t]:true dependencies
                                            [a]:anti-dependencies
                                            [o]:output dependencies
        -ldbrnN         limit decode to N branches per cycle
        -lamemN         limit load/store address logic to N addresses per cycle
        -ldmemN         limit decode to N loads/stores per cycle
        -liN            limit issue to N instructions per cycle
        -liop           limit issue to N operands per cycle
        -merge          merge instruction runs if predicted OK
        -multi_brn      execute multiple branches per cycle
        -multi_path     fetch multiple branch paths if possible
        -nobyp_load     loads do not bypass stores
        -nofwd_store    do not forward store data from buffer
        -perf           perfect branch prediction
        -pre_addrN      limit pre-address buffer to N locations
        -prearb         pre-arbitrate result buses during decode (default no)
        -pred           predict branches with BTB (or icache if enabled)
        -pred_take              predict all branches take
        -pred_rtns      predict branches and procedure returns
        -pN             print current statistics every N cycles
                        (default print at end only)
        -rb_intN        use integer result buffer of N entries
                        (overrides machine configuration)
        -rb_floatN      use float result buffer of N entries
                        (overrides machine configuration)
        -regsN          use N register read ports (default 2*decode width)
        -resbus_intN    use N integer result buses
        -resbus_floatN  use N floating-point result buses
        -rstnN          use uniform reservation stations of N entries each
                        (overrides machine configuration)
        -sN             stop at first opportunity after N instructions
        -sched_dmiss    schedule around dcache misses (default no)
        -strbufN        limit store buffer to N locations
        -sum<fname>     append results to summary file fname
        -t              set test mode (print all instructions traced)
        -verbose        set verbose mode (more detailed statistics)
```

```
Remainder of line after -prog keyword is the progam
specification of the pixified program, including
arguments.  Input and output files for this program
are specified after '<' and '>' tokens, respectively.
```

B Machine files used for simulations

Machine file for SAND

```
Caches
    size_bytes 16384
    words_per_block 4
    associativity 1
    miss_penalty 12

Instruction_Bus_Width 4

Integer_Result_Buffer_Depth 16

Float_Result_Buffer_Depth 8

Integer_Result_Buffer_Width 2

Integer_Writeback_Width 2

Float_Result_Buffer_Width 2

Float_Writeback_Width 2

Func_Unit Integer
    number_available 2
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 4

Func_Unit Branch
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 4

Func_Unit Int_Mul
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 4

Func_Unit Int_Div
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 4

Func_Unit Shifter
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 2
```

```
Func_Unit Load
    number_available 1
    issue_latency
        single   1
        double   2
        extended 4
    result_latency
        single   2
        double   3
        extended 5
    reservation_station_depth 8

Func_Unit Float_Add
    number_available 1
    issue_latency
        single   1
        double   1
        extended 4
    result_latency
        single   2
        double   2
        extended 6
    reservation_station_depth 2

Func_Unit Float_Mul
    number_available 1
    issue_latency
        single   1
        double   1
        extended 4
    result_latency
        single   4
        double   5
    reservation_station_depth 2

Func_Unit Float_Div
    number_available 1
    issue_latency
        single   12
        double   19
        extended 32
    result_latency
        single   12
        double   19
        extended 32
    reservation_station_depth 2

Func_Unit Float_Conv
    number_available 1
    issue_latency
        single   1
        double   1
        extended 4
    result_latency
        single   2
        double   2
        extended 4
    reservation_station_depth 2

Instructions
    MULT    INT_MUL
    MULTU   INT_MUL
    DIV     INT_DIV
```

```
    DIVU    INT_DIV
    MFHI    SHIFTER
    MFLO    SHIFTER
    MTHI    SHIFTER
    MTLO    SHIFTER
    SLT     SHIFTER
    SLTU    SHIFTER

Func_Unit Store
    is Load
```

Machine file for POWER: mpe

```
Caches
    size_bytes 8192
    words_per_block 16
    associativity 2
    miss_penalty 12

Branch_Target_Buffer
    entries 128
    associativity 2

Instruction_Bus_Width 4

Integer_Result_Buffer_Depth 11

Float_Result_Buffer_Depth 18

Integer_Result_Buffer_Width 1

Integer_Writeback_Width 1

Float_Result_Buffer_Width 1

Float_Writeback_Width 1

Integer
    number_available 1
    issue_latency
        single   1
    result_latency
        single   2
    reservation_station_depth 8

Shifter
    number_available 1
    issue_latency
        single   1
    result_latency
        single   2
    reservation_station_depth 8

Branch
    number_available 1
    issue_latency
        single   1
    result_latency
        single   3
    reservation_station_depth 4

Load
    number_available 1
    issue_latency
        single   1
        double   1
        extended 2
    result_latency
        single   3
        double   3
        extended 4
    reservation_station_depth 6

Store
    number_available 1
```

```
    issue_latency
        single   1
        double   1
        extended 2
    result_latency
        single   3
        double   3
        extended 4
    reservation_station_depth 4

Float_Add
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   5
        double   5
        extended 5
    reservation_station_depth 10

Float_Mul
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   5
        double   5
        extended 5
    reservation_station_depth 10

Float_Div
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   20
        double   20
        extended 20
    reservation_station_depth 10

Float_Conv
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   5
        double   5
        extended 5
    reservation_station_depth 10
```

Machine file for POWER: mps

```
Caches
    size_bytes 65536
    words_per_block 4
    associativity 2
    miss_penalty 12

Branch_Target_Buffer
    entries 32
    associativity 2

Instruction_Bus_Width 4

Integer_Result_Buffer_Depth 16

Float_Result_Buffer_Depth 8

Integer_Result_Buffer_Width 2

Integer_Writeback_Width 2

Float_Result_Buffer_Width 2

Float_Writeback_Width 2

Integer
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 5

Shifter
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 2

Branch
    number_available 1
    issue_latency
        single   1
    result_latency
        single   2
    reservation_station_depth 4

Load
    number_available 1
    issue_latency
        single   1
        double   2
        extended 4
    result_latency
        single   2
        double   3
        extended 5
    reservation_station_depth 8

Store
    number_available 1
```

```
    issue_latency
        single   1
        double   2
        extended 4
    result_latency
        single   2
        double   3
        extended 5
    reservation_station_depth 8

Float_Add
    number_available 1
    issue_latency
        single   2
        double   1
        extended 4
    result_latency
        single   2
        double   2
        extended 6
    reservation_station_depth 5

Float_Mul
    number_available 1
    issue_latency
        single   1
        double   1
        extended 4
    result_latency
        single   4
        double   5
        extended 6
    reservation_station_depth 5

Float_Div
    number_available 1
    issue_latency
        single   16
        double   19
        extended 38
    result_latency
        single   16
        double   19
        extended 38
    reservation_station_depth 5

Float_Conv
    number_available 1
    issue_latency
        single   1
        double   1
        extended 4
    result_latency
        single   2
        double   2
        extended 4
    reservation_station_depth 5
```

Machine file for POWER: mpee

```
Caches
    size_bytes 8192
    words_per_block 16
    associativity 2
    miss_penalty 12

Branch_Target_Buffer
    entries 128
    associativity 2

Instruction_Bus_Width 4

Integer_Result_Buffer_Depth 11

Float_Result_Buffer_Depth 18

Integer_Result_Buffer_Width 1

Integer_Writeback_Width 1

Float_Result_Buffer_Width 1

Float_Writeback_Width 1

Func_Unit Integer
    number_available 1
    issue_latency
        single   1
    result_latency
        single   2
    reservation_station_depth 8

Func_Unit Shifter
    is Integer

Func_Unit Branch
    number_available 1
    issue_latency
        single   1
    result_latency
        single   3
    reservation_station_depth 4

Func_Unit Load
    number_available 1
    issue_latency
        single   1
        double   1
        extended 2
    result_latency
        single   3
        double   3
        extended 4
    reservation_station_depth 6

Func_Unit Store
    is Load

Func_Unit Float_Add
    number_available 1
    issue_latency
        single   1
```

```
        double   1
        extended 1
    result_latency
        single   5
        double   5
        extended 5
    reservation_station_depth 10

Func_Unit Float_Mul
    is Float_Add

Func_Unit Float_Div
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   20
        double   20
        extended 20
    reservation_station_depth 10

Func_Unit Float_Conv
    is Float_Add

Instructions
    MULT   Integer
    MULTU  Integer
```

Machine file for PowerPC

```
Caches
    size_bytes 32768
    words_per_block 8
    associativity 8
    miss_penalty 9

Instruction_Bus_Width 8

Integer_Result_Buffer_Depth 1

Float_Result_Buffer_Depth 1

Integer_Result_Buffer_Width 1

Integer_Writeback_Width 1

Float_Result_Buffer_Width 1

Float_Writeback_Width 1


Integer
    number_available 2
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 4

Shifter
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 1

Branch
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 1

Load
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single    1
        double    1
        extended  1
    reservation_station_depth 1

Store
    number_available 1
    issue_latency
        single    1
        double    1
```

```
        extended  1
    result_latency
        single    1
        double    1
        extended  1
    reservation_station_depth 1

Float_Add
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single    1
        double    1
        extended  1
    reservation_station_depth 1

Float_Mul
    number_available 1
    issue_latency
        single   1
        double   2
        extended 2
    result_latency
        single   1
        double   2
        extended 2
    reservation_station_depth 1

Float_Div
    number_available 1
    issue_latency
        single   4
        double   4
        extended 4
    result_latency
        single   17
        double   31
        extended 31
    reservation_station_depth 1

Float_Conv
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   1
        double   1
        extended 1
    reservation_station_depth 1
```

Machine file for Alpha

```
Caches
    size_bytes 8192
    words_per_block 4
    associativity 1
    miss_penalty 12

Instruction_Bus_Width 2

Integer_Result_Buffer_Depth 3

Float_Result_Buffer_Depth 6

Integer_Result_Buffer_Width 1

Integer_Writeback_Width 1

Float_Result_Buffer_Width 1

Float_Writeback_Width 1

Func_Unit Integer
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 1

Func_Unit Branch
    number_available 1
    issue_latency
        single   1
    result_latency
        single   1
    reservation_station_depth 1

Func_Unit Load
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   3
        double   3
        extended 3
    reservation_station_depth 4

Func_Unit Float_Add
    number_available 1
    issue_latency
        single   1
        double   1
        extended 1
    result_latency
        single   6
        double   6
        extended 6
    reservation_station_depth 1

Func_Unit Float_Div
    number_available 1
```

```
    issue_latency
        single   6
        double   6
        extended 6
    result_latency
        single   6
        double   6
        extended 6
    reservation_station_depth 1

Func_Unit Int_Mult
    number_available 1
    issue_latency
        single   2
    result_latency
        single   2
    reservation_station_depth 1

Instructions
    SLL     SHIFTER
    SRL     SHIFTER
    SRA     SHIFTER
    SLLV    SHIFTER
    SRLV    SHIFTER
    SRAV    SHIFTER
    SYSCALL BRANCH
    BREAK   BRANCH
    VCALL   BRANCH

    MULT    INT_MULT
    MULTU   INT_MULT

    DIV     FLOAT_DIV
    DIVU    FLOAT_DIV
    MFHI  INTEGER
    MFLO  INTEGER
    MTHI  INTEGER
    MTLO  INTEGER
    JR    BRANCH
    JALR  BRANCH
    ADD   INTEGER
    ADDU  INTEGER
    AND   INTEGER
    OR    INTEGER
    XOR   INTEGER
    NOR   INTEGER
    SUB   INTEGER
    SUBU  INTEGER
    SLT   INTEGER
    SLTU  INTEGER
    J     BRANCH
    JAL   BRANCH
    BCOND BRANCH
    BEQ   BRANCH
    BNE   BRANCH
    BLEZ  BRANCH
    BGTZ  BRANCH
    ADDI  INTEGER
    ADDIU INTEGER
    SLTI  INTEGER
    SLTIU INTEGER
    ORI   INTEGER
    XORI  INTEGER
    ANDI  INTEGER
```

```
LUI   INTEGER
LWC1  LOAD
LDC1  LOAD
LB    LOAD
LH    LOAD
LW    LOAD
LBU   LOAD
LHU   LOAD
LWL   LOAD
LWR   LOAD
LD    LOAD
SWC1  STORE
SDC1  STORE
SB    STORE
SH    STORE
SW    STORE
SWL   STORE
SWR   STORE
SD    STORE
BC  BRANCH
MTC FLOAT_CONV
MFC FLOAT_CONV
CTC FLOAT_CONV
CFC FLOAT_CONV
FADD_S  FLOAT_ADD
FSUB_S  FLOAT_ADD
FMPY_S  FLOAT_MUL
FDIV_S  FLOAT_DIV
FABS_S  FLOAT_ADD
FMOV_S  FLOAT_ADD
FNEG_S  FLOAT_ADD
FSQRT_S FLOAT_DIV
FCVTW_S FLOAT_CONV
FCVTS_S FLOAT_CONV
FCVTD_S FLOAT_CONV
FCVTE_S FLOAT_CONV
FCMP_S  FLOAT_ADD
FADD_D  FLOAT_ADD
FSUB_D  FLOAT_ADD
FMPY_D  FLOAT_MUL
FDIV_D  FLOAT_DIV
FABS_D  FLOAT_ADD
FMOV_D  FLOAT_ADD
FNEG_D  FLOAT_ADD
FSQRT_D FLOAT_DIV
FCVTW_D FLOAT_CONV
FCVTS_D FLOAT_CONV
FCVTD_D FLOAT_CONV
FCVTE_D FLOAT_CONV
FCMP_D  FLOAT_ADD
FADD_E  FLOAT_ADD
FSUB_E  FLOAT_ADD
FMPY_E  FLOAT_MUL
FDIV_E  FLOAT_DIV
FABS_E  FLOAT_ADD
FMOV_E  FLOAT_ADD
FNEG_E  FLOAT_ADD
FSQRT_E FLOAT_DIV
FCVTW_E FLOAT_CONV
FCVTS_E FLOAT_CONV
FCVTD_E FLOAT_CONV
FCVTE_E FLOAT_CONV
FCMP_E  FLOAT_ADD
```

```
Func_Unit Shifter
    is Integer

Func_Unit Store
    is Load

Func_Unit Float_Mul
    is Float_Add

Func_Unit Float_Conv
    is Float_Add
```

C MIPS instruction–to–functional–unit mappings

Instruction Name	Built-in Functional Unit
SLL	SHIFTER
SRL	SHIFTER
SRA	SHIFTER
SLLV	SHIFTER
SRLV	SHIFTER
SRAV	SHIFTER
SYSCALL	BRANCH
BREAK	BRANCH
VCALL	BRANCH
J	BRANCH
JAL	BRANCH
BCOND	BRANCH
BEQ	BRANCH
BNE	BRANCH
BLEZ	BRANCH
BGTZ	BRANCH
JR	BRANCH
JALR	BRANCH
BC	BRANCH
MFHI	INTEGER
MFLO	INTEGER
MTHI	INTEGER
MTLO	INTEGER
ADD	INTEGER
ADDU	INTEGER
AND	INTEGER
OR	INTEGER
XOR	INTEGER
NOR	INTEGER
SUB	INTEGER
SUBU	INTEGER
SLT	INTEGER
SLTU	INTEGER
ADDI	INTEGER
ADDIU	INTEGER
SLTI	INTEGER
SLTIU	INTEGER
ORI	INTEGER
XORI	INTEGER
ANDI	INTEGER
LUI	INTEGER
LWC1	LOAD
LDC1	LOAD
LB	LOAD
LH	LOAD
LW	LOAD
LBU	LOAD
LHU	LOAD
LWL	LOAD
LWR	LOAD
LD	LOAD
SWC1	STORE
SDC1	STORE
SB	STORE
SH	STORE
SW	STORE
SWL	STORE
SWR	STORE
SD	STORE

MULT	FLOAT_DIV
MULTU	FLOAT_DIV
DIV	FLOAT_DIV
DIVU	FLOAT_DIV
FSQRT_S	FLOAT_DIV
FDIV_S	FLOAT_DIV
FDIV_D	FLOAT_DIV
FSQRT_D	FLOAT_DIV
FDIV_E	FLOAT_DIV
FSQRT_E	FLOAT_DIV
FADD_S	FLOAT_ADD
FSUB_S	FLOAT_ADD
FCMP_S	FLOAT_ADD
FADD_D	FLOAT_ADD
FSUB_D	FLOAT_ADD
FABS_D	FLOAT_ADD
FMOV_D	FLOAT_ADD
FNEG_D	FLOAT_ADD
FABS_S	FLOAT_ADD
FMOV_S	FLOAT_ADD
FNEG_S	FLOAT_ADD
FCMP_D	FLOAT_ADD
FADD_E	FLOAT_ADD
FSUB_E	FLOAT_ADD
FABS_E	FLOAT_ADD
FMOV_E	FLOAT_ADD
FNEG_E	FLOAT_ADD
FCMP_E	FLOAT_ADD
FMPY_S	FLOAT_MUL
FMPY_E	FLOAT_MUL
FMPY_D	FLOAT_MUL
MTC	FLOAT_CONV
MFC	FLOAT_CONV
CTC	FLOAT_CONV
CFC	FLOAT_CONV
FCVTW_S	FLOAT_CONV
FCVTS_S	FLOAT_CONV
FCVTD_S	FLOAT_CONV
FCVTE_S	FLOAT_CONV
FCVTW_D	FLOAT_CONV
FCVTS_D	FLOAT_CONV
FCVTD_D	FLOAT_CONV
FCVTE_D	FLOAT_CONV
FCVTW_E	FLOAT_CONV
FCVTS_E	FLOAT_CONV
FCVTD_E	FLOAT_CONV
FCVTE_E	FLOAT_CONV