Simulated Study of In Order and Out of Order Superscalar Processors

Italo Borrelli—V00884840
CSc Undergrad, University of Victoria
British Columbia, Canada
iborrelli@uvic.ca

Connor Mcleod—V00725080
CSc Undergrad, University of Victoria
British Columbia, Canada
connormc@uvic.ca

***Abstract*—This project is a simulation of a processor running MIPS code. Current based on the PowerPC design, we explore the variation in cycle-time required to run code both in order and dynamically.**

I. Introduction

THIS project is a simulation of the PowerPC processor design using MIPS instructions, exploring the difference in execution time between inorder and dynamically ordered instruction processing. When running code through a processor, there can be times when the input of one instruction is being decided by an instruction currently in production. The inorder version forces us to wait for instructions to basically finish before we are able to execute the dependent instruction, but the dynamic alternative can allow us to run later instructions not dependant on earlier ones to limit the number of NOPs and hopefully speed-up overall program runtimes.

Our hypothesis on improvements in speed with an out of order processor Table I.

We believe, by turning our instruction queue into a pseudo-priority queue, we can prove faster the dynamically ordered version.

II. Simulation Code

To build our simulator, we take in a regular set of MIPS instructions and parse them into the instructions that would be run in the order they would be run according to the control flow that would be followed [1].

After parsing the input file and creating the instruction objects holding the registers they're using, we amend them to the "code" we will pass to our pipelines. Both pipelines fill up a buffer of the instructions that are coming next. We only need a subset of instructions prepared for "execution" because even in the dynamic pipeline we will only ever jump so far ahead in the code.

In the in order pipeline we simply check if the next instruction's execution unit is available. If it is, then we run the instruction. Either way the clock ticks onward. More detailed is the dynamic pipeline. After filling the buffer we take the index of the first instruction in the buffer and try to run it. If it can't run we check the next instruction and so on. Once we find one that can run we remove it from the buffer and put a new instruction in. The key here is the check: we check not only if the correct execution unit has finished with it's previous instruction, but also if the registers are ready for us to use, and if there's condition variables to take into account. Having all these ensures avoiding data hazards, therein ensuring the correctness of our simulation.

A. Correctness of Simulation

This model is correct in the most important ways. The instructions are run in the correct order for each variation, and preserve the integrity of the code going through the processor. In other words data hazards are avoided. This is the most important aspect of the simulator. This correctness gives the simulator the ability to meanfulling compare. Even if the number of cycles/instruction is wrong in terms of real world performance, It still allows for logically significant comparison of in order and dynamically ordered variations.

With this in mind, it is true that our instruction set is not currently true to life in terms of cycle-count. Also, only the registers are only considered in terms of data hazards and not the actual data itself. Because of this fact, calculations such as ADD/SUB or BEQ/BNE are not made, and therefore the status registers and conditional variables are not changed.

The instructions are taken into the buffer as they are ordered in the input file, so in order to do branch instructions correctly without data to manipulate, the test files are adjusted to run the instructions that would have been run. Through these adjustments, even if data is not held, loops still run the correct number of times or jumps to certain code blocks would still be effectively "jumped" to. This is solely based on the way a user decides to write the machine instruction files.

B. Possible Improvements

Given more time, there are a few improvements we would have liked to make to our simulation. By using the visitor design pattern we could switch out the PowerPC processor for most any other. By adjusting the number or types of execution units, we can simulate many different processor builds. By breaking execution units into functional units, we could do more granular simulations such as running ADD while a MULT is already underway. With more accurate cycle-times for instructions, we could even use this to test future processors before they are built. The number of cycles could be normally distributed and guessed for some operations, or instructions could have size of numbers in register and do some prediction for complex functions such as MUL and DIV could be extrapolated from there.

We also could implement branches to have probabilities for branch or number of time branching would take place. This would reduce the requirement to augment the input file code and make for a more flexible simulation. In the same vein, we could do ISA in a plaintext file that can be imported to make it easier to write instruction sets for new machines.

In general longer a study of the CPU should be done to accurately get the number of cycles and a more concrete and accurate design.

III. Data Gathering

Simulated MIPS code was generated to be run with the simulator. The simulator gathers a variety of statistics, including the numbers of each type of instruction being run, the total clock cycles for the whole instruction set and the total number of cycles each instruction type takes in the queue and in the system in total after being put in the instruction buffer. These statistics are gathered for running the simulation instructions in order and out of order.

The simulated MIPS instructions were run. The number of each type of instruction for each instruction set can be found in Table I. The total number of cycles for each instruction in the buffer and in the pipeline are given in Table II and Table III respectively. Using this data, the averages for each instruction type were calculated and can be found for number of cycles in the instruction buffer and pipeline in Table IV and Table V respectively.

The percent improvement was calculated and the average taken. The values are shown in Table VI. It shows that dynamic ordering of instructions for a 16 instruction buffer is about a 30% reduction.

IV. Conclusion

We hypothesized that dynamically ordered pipelines would outperform in-order pipelines, and our data supports this claim. By limiting stalls in a way that preserves data integrity, we have reduced average cycles for each instruction type to about 70% in most cases.

Even though our project simulates the PowerPC design, with some simple adjustments we could simulate most any processor to get its statistics. We predict that a similar saving is achievable on other processors.

References

[1] D. Patterson, ‘Computer Organization and Design: The Hardware/Software Interface’, 5th ed., 2014.

APPENDIX

TABLE I: Number of Cycles per Instruction Type

Test	All Instructions	Floating Point	Fixed Point	Branch	Memory	Logical
1	219	0	66	23	71	59
2	28	0	4	0	12	12
3	71	0	26	23	11	11
4	25	0	6	2	9	7
5	33	0	10	2	13	7
6	107	0	50	23	23	11
7	34	0	6	0	15	13
8	111	0	34	23	29	25
9	199	0	72	27	60	38
10	225	0	72	48	58	46
11	264	0	96	50	68	48
12	597	0	204	123	152	114
13	574	50	162	121	138	100
14	612	0	206	121	161	121
15	3099	50	1014	586	820	612

TABLE II: Total Cycles Spent in Queue

			Total Number of Cycles in Queue					
Test	Execution	Number of Cycles	All Instructions	Floating Point	Fixed Point	Branch	Memory	Logical
1	In-order	114	1541	0	469	161	498	413
	Dynamic order	110	1477	0	457	161	470	389
2	In-order	15	155	0	26	0	63	66
	Dynamic order	14	140	0	23	0	57	60
3	In-order	48	640	0	261	244	67	68
	Dynamic order	36	441	0	199	143	47	52
4	In-order	28	367	0	86	21	121	121
	Dynamic order	19	129	0	25	6	31	49
5	In-order	32	423	0	145	23	141	94
	Dynamic order	21	183	0	52	8	51	52
6	In-order	54	693	0	342	161	136	54
	Dynamic order	54	693	0	342	161	136	54
7	In-order	20	229	0	39	0	94	96
	Dynamic order	17	182	0	32	0	76	74
8	In-order	72	984	0	336	248	214	186
	Dynamic order	56	721	0	259	144	170	148
9	In-order	134	1889	0	660	220	567	398
	Dynamic order	106	1347	0	505	190	342	266
10	In-order	161	2295	0	770	525	537	441
	Dynamic order	119	1529	0	542	310	358	297
11	In-order	177	2504	0	910	463	628	459
	Dynamic order	134	1824	0	738	345	371	356
12	In-order	424	6130	0	2106	1267	1491	1176
	Dynamic order	301	4163	0	1554	826	897	829
13	In-order	566	8421	953	2360	2241	1647	1154
	Dynamic order	378	5293	910	1774	1381	625	568
14	In-order	420	6051	0	2064	1238	1497	1186
	Dynamic order	308	4252	0	1556	797	1001	864
15	In-order	2264	33068	954	10702	6832	8041	6163
	Dynamic order	1636	23021	903	8157	4529	4843	4307

TABLE III: Total Cycles Spent in Pipeline

			Total Number of Cycles in Pipeline					
Test	Execution	Number of Cycles	All Instructions	Floating Point	Fixed Point	Branch	Memory	Logical
1	In-order	114	1760	0	535	184	569	472
	Dynamic order	110	1696	0	523	184	541	448
2	In-order	15	183	0	30	0	75	78
	Dynamic order	14	168	0	27	0	69	72
3	In-order	48	711	0	287	267	78	79
	Dynamic order	36	512	0	225	166	58	63
4	In-order	28	406	0	106	23	130	128
	Dynamic order	19	168	0	45	8	40	56
5	In-order	32	470	0	169	25	154	101
	Dynamic order	21	230	0	76	10	64	59
6	In-order	54	800	0	392	184	159	65
	Dynamic order	54	800	0	392	184	159	65
7	In-order	20	263	0	45	0	109	109
	Dynamic order	17	216	0	38	0	91	87
8	In-order	72	1095	0	370	271	243	211
	Dynamic order	56	832	0	293	167	199	173
9	In-order	134	2116	0	760	247	627	436
	Dynamic order	106	1574	0	605	217	402	304
10	In-order	161	2534	0	856	573	595	487
	Dynamic order	119	1768	0	628	358	416	343
11	In-order	177	2796	0	1034	513	696	507
	Dynamic order	134	2116	0	862	395	439	404
12	In-order	424	6783	0	2366	1390	1643	1290
	Dynamic order	301	4816	0	1814	949	1049	943
13	In-order	566	9233	1199	2564	2362	1785	1254
	Dynamic order	378	6105	1156	1978	1502	763	668
14	In-order	420	6705	0	2312	1359	1658	1307
	Dynamic order	308	4906	0	1804	918	1162	985
15	In-order	2264	36601	1200	11954	7418	8861	6775
	Dynamic order	1636	26554	1149	9409	5115	5663	4919

TABLE IV: Average Cycles Spent in Queue

			Average Number of Cycles in Queue					
Test	Execution	Number of Cycles	All Instructions	Floating Point	Fixed Point	Branch	Memory	Logical
1	In-order	114	7.04	0.00	7.11	7.00	7.01	7.00
	Dynamic order	110	6.74	0.00	6.92	7.00	6.62	6.59
2	In-order	15	5.54	0.00	6.50	0.00	5.25	5.50
	Dynamic order	14	5.00	0.00	5.75	0.00	4.75	5.00
3	In-order	48	9.01	0.00	10.04	10.61	6.09	6.18
	Dynamic order	36	6.21	0.00	7.65	6.22	4.27	4.73
4	In-order	28	14.68	0.00	14.33	10.50	13.44	17.29
	Dynamic order	19	5.16	0.00	4.17	3.00	3.44	7.00
5	In-order	32	12.82	0.00	14.50	11.50	10.85	13.43
	Dynamic order	21	5.55	0.00	5.20	4.00	3.92	7.43
6	In-order	54	6.48	0.00	6.84	7.00	5.91	4.91
	Dynamic order	54	6.48	0.00	6.84	7.00	5.91	4.91
7	In-order	20	6.74	0.00	6.50	0.00	6.27	7.38
	Dynamic order	17	5.35	0.00	5.33	0.00	5.07	5.69
8	In-order	72	8.86	0.00	9.88	10.78	7.38	7.44
	Dynamic order	56	6.50	0.00	7.62	6.26	5.86	5.92
9	In-order	134	9.49	0.00	9.17	8.15	9.45	10.47
	Dynamic order	106	6.77	0.00	7.01	7.04	5.70	7.00
10	In-order	161	10.20	0.00	10.69	10.94	9.26	9.59
	Dynamic order	119	6.80	0.00	7.53	6.46	6.17	6.46
11	In-order	177	9.48	0.00	9.48	9.26	9.24	9.56
	Dynamic order	134	6.91	0.00	7.69	6.90	5.46	7.42
12	In-order	424	10.27	0.00	10.32	10.30	9.81	10.32
	Dynamic order	301	6.97	0.00	7.62	6.72	5.90	7.27
13	In-order	566	14.67	19.06	14.57	18.52	11.93	11.54
	Dynamic order	378	9.22	18.20	10.95	11.41	4.53	5.68
14	In-order	420	9.89	0.00	10.02	10.23	9.30	9.80
	Dynamic order	308	6.95	0.00	7.55	6.59	6.22	7.14
15	In-order	2264	10.67	19.08	10.55	11.66	9.81	10.07
	Dynamic order	1636	7.43	18.06	8.04	7.73	5.91	7.04

TABLE V: Average Cycles Spent in Pipeline

			Average Number of Cycles in Pipeline					
Test	Execution	Number of Cycles	All Instructions	Floating Point	Fixed Point	Branch	Memory	Logical
1	In-order	114	8.04	0.00	8.11	8.00	8.01	8.00
	Dynamic order	110	7.74	0.00	7.92	8.00	7.62	7.59
2	In-order	15	6.54	0.00	7.50	0.00	6.25	6.50
	Dynamic order	14	6.00	0.00	6.75	0.00	5.75	6.00
3	In-order	48	10.01	0.00	11.04	11.61	7.09	7.18
	Dynamic order	36	7.21	0.00	8.65	7.22	5.27	5.73
4	In-order	28	16.24	0.00	17.67	11.50	14.44	18.29
	Dynamic order	19	6.72	0.00	7.50	4.00	4.44	8.00
5	In-order	32	14.24	0.00	16.90	12.50	11.85	14.43
	Dynamic order	21	6.97	0.00	7.60	5.00	4.92	8.43
6	In-order	54	7.48	0.00	7.84	8.00	6.91	5.91
	Dynamic order	54	7.48	0.00	7.84	8.00	6.91	5.91
7	In-order	20	7.74	0.00	7.50	0.00	7.27	8.38
	Dynamic order	17	6.35	0.00	6.33	0.00	6.07	6.69
8	In-order	72	9.86	0.00	10.88	11.78	8.38	8.44
	Dynamic order	56	7.50	0.00	8.62	7.26	6.86	6.92
9	In-order	134	10.63	0.00	10.56	9.15	10.45	11.47
	Dynamic order	106	7.91	0.00	8.40	8.04	6.70	8.00
10	In-order	161	11.26	0.00	11.89	11.94	10.26	10.59
	Dynamic order	119	7.86	0.00	8.72	7.46	7.17	7.46
11	In-order	177	10.59	0.00	10.77	10.26	10.24	10.56
	Dynamic order	134	8.02	0.00	8.98	7.90	6.46	8.42
12	In-order	424	11.36	0.00	11.60	11.30	10.81	11.32
	Dynamic order	301	8.07	0.00	8.89	7.72	6.90	8.27
13	In-order	566	16.09	23.98	15.83	19.52	12.93	12.54
	Dynamic order	378	10.64	23.12	12.21	12.41	5.53	6.68
14	In-order	420	10.96	0.00	11.22	11.23	10.30	10.80
	Dynamic order	308	8.02	0.00	8.76	7.59	7.22	8.14
15	In-order	2264	11.81	24.00	11.79	12.66	10.81	11.07
	Dynamic order	1636	8.57	22.98	9.28	8.73	6.91	8.04

TABLE VI: Average Improvement for Dynamic Processing

	All types	Floating Point	Fixed Point	Branch	Memory	Logical
Queue	0.711775746	0.9507101045	0.7431892594	0.6595669714	0.6588990149	0.7254838796
Pipeline	0.7408551387	0.9608183903	0.7760658178	0.6879047078	0.691223941	0.7500813102