ECE 4750 PSET 4

Tim Yao (ty252)

Nov 25, 2015

1 Tree Network Topologies

1.a Baseline I3L Microarchitecture

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
mul r1, r2, r3	F	D	I	Y0	Y1	Y2	Y3	W																					
mul r4, r1, r5		F	D	I	I	I	I	Y0	Y1	Y2	Y3	W																	
div r6, r7, r8			F	D	D	D	D	I	Z	Z	Z	Z	W																
div r9, r10, r11				F	F	F	F	D	I	I	I	I	Z	Z	Z	Z	W												
div r12, r13, r14								F	D	D	D	D	I	I	I	I	Z	Z	Z	Z	W								
mul r15, r12, r16									F	F	F	F	D	D	D	D	I	I	I	I	Y0	Y1	Y2	Y3	W				
mul r17, r15, r18													F	F	F	F	D	D	D	D	I	I	I	I	Y0	Y1	Y2	Y3	W

Figure 1: Pipeline Diagram for Baseline I3L Architecture

The total issue to commit cycle count is 27.

1.b Schedule Oldest Ready Instruction First on IO2L Microarchitecture

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
mul r1, r2, r3	I	Y0	Y1	Y2	Y3	W	C																	
mul r4, r1, r5					I	Y0	Y1	Y2	Y3	W	C													
div r6, r7, r8		I	Z	Z	Z	Z	W	r	r	r	r	C												
div r9, r10, r11						I	Z	Z	Z	Z	W	r	C											
div r12, r13, r14										I	Z	Z	Z	Z	W	C								
mul r15, r12, r16														I	Y0	Y1	Y2	Y3	W	C				
mul r17, r15, r18																		I	Y0	Y1	Y2	Y3	W	C

Figure 2: Pipeline Diagram for IO2L Architecture

The total issue to commit cycle count is 24.

1.c Optimal Scheduling on IO2L Microarchitecture

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul r1, r2, r3		I	Y0	Y1	Y2	Y3	W	C										
mul r4, r1, r5						I	Y0	Y1	Y2	Y3	W	C						
div r6, r7, r8					I	Z	Z	Z	Z	W	r	r	C					
div r9, r10, r11									I	Z	Z	Z	Z	W	C			
div r12, r13, r14	I	Z	Z	Z	Z	W	r	r	r	r	r	r	r	r	r	C		
mul r15, r12, r16							I	Y0	Y1	Y2	Y3	W	r	r	r	r	C	
mul r17, r15, r18											I	Y0	Y1	Y2	Y3	W	r	C

Figure 3: Pipeline Diagram for IO2L Architecture with Optimized Scheduling

The total issue to commit cycle count is 18.

1.d Scheduling Comparison

The optimized scheduler is able to achieve higher performance due to better exploitation of ILP. In the instruction sequence, there are two main sequences of data dependencies, so the optimal schedule will be to interleave these two sequences, along with the two independent div instructions, to maximize commits per cycle. Also because we have an un-pipelined divider, we will also want to start the divide instructions as early as possible (which is what the optimal schedule does). We might be able to implement this in hardware by using a mixed priority scheduling algorithm. The scheduler will have to place divide instructions at the highest priority (with the oldest div instruction issuing first). The next highest priority will be the oldest ready instruction that is not a divide. This will implicitly interleave instructions with data dependencies.

2 Register Renaming

2.a Architectural RAW, WAW, and WAR Dependencies

```
1 mul  r1, r2, r3
2 mul  r4, r1, r5
3 addu r6, r7, r8
4 mul  r1, r2, r5
5 addu r6, r6, r9
```

2.b Pipeline Diagram with Register Renaming

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
mul r1, r2, r3	F	D	I	Y0	Y1	Y2	Y3	W	C							
mul r4, r1, r5		F	D	i	i	i	I	Y0	Y1	Y2	Y3	W	C			
addu r6, r7, r8			F	D	I	X	W	r	r	r	r	r	r	C		
mul r1, r2, r5				F	D	I	Y0	Y1	Y2	Y3	W	r	r	r	C	
addu r6, r6, r9					F	D	i	I	X	W	r	r	r	r	r	C

Figure 4: Pipeline Diagram with Register Renaming

2.c Register Renaming with Pointers in the IQ/ROB

	Stage				**RT**										
Cycle	**D**	**I**	**W**	**C**	**r1**	**r2**	**r3**	**r4**	**r5**	**r6**	**r7**	**r8**	**r9**	**Free List**	**IQ**
0					p0	p1	p2	p3	p4	p5	p6	p7	p8	p9,pA,pB,pC,pD	
1	1				:	:	:	:	:	:	:	:	:	p9,pA,pB,pC,pD	
2	2	1			p9*	:	:	:	:	:	:	:	:	pA,pB,pC,pD	p9/p1/p2
3	3				:	:	:	pA*	:	:	:	:	:	pB,pC,pD	pA/p9*/p4
4	4	3			:	:	:	:	:	pB*	:	:	:	pC,pD	pB/p6/p7
5	5	4			pC*	:	:	:	:	:	:	:	:	pD	pC/p1/p4
6		2	3		:	:	:	:	:	pD*	:	:	:		pD/pB*/p8
7		5	1		:	:	:	:	:	:	:	:	:		
8				1	:	:	:	:	:	:	:	:	:		
9			5		:	:	:	:	:	:	:	:	:	p0	
10			4		:	:	:	:	:	pD	:	:	:	p0	
11			2		pC	:	:	:	:	:	:	:	:	p0	
12				2	:	:	:	pA	:	:	:	:	:	p0	
13				3	:	:	:	:	:	:	:	:	:	p0,p3	
14				4	:	:	:	:	:	:	:	:	:	p0,p3,p5	
15				5	:	:	:	:	:	:	:	:	:	p0,p3,p5,p9	
16					:	:	:	:	:	:	:	:	:	p0,p3,p5,p9,pB	

Figure 5: Microarchitectural State (RT/FL/IQ) for Reg Renaming with Pointers in the IQ/ROB

	ROB				
Cycle	**0**	**1**	**2**	**3**	**4**
0					
1					
2	p9*/r1/p0				
3	p9*/r1/p0	pA*/r4/p3			
4	p9*/r1/p0	pA*/r4/p3	pB*/r6/p5		
5	p9*/r1/p0	pA*/r4/p3	pB*/r6/p5	pC*/r1/p9*	
6	p9*/r1/p0	pA*/r4/p3	pB*/r6/p5	pC*/r1/p9*	pD*/r6/pB*
7	p9*/r1/p0	pA*/r4/p3	pB/r6/p5	pC*/r1/p9*	pD*/r6/pB
8	p9/r1/p0	pA*/r4/p3	pB/r6/p5	pC*/r1/p9	pD*/r6/pB
9	p9/r1/p0	pA*/r4/p3	pB/r6/p5	pC*/r1/p9	pD*/r6/pB
10	p9/r1/p0	pA*/r4/p3	pB/r6/p5	pC*/r1/p9	pD/r6/pB
11	p9/r1/p0	pA*/r4/p3	pB/r6/p5	pC/r1/p9	pD/r6/pB
12	p9/r1/p0	pA/r4/p3	pB/r6/p5	pC/r1/p9	pD/r6/pB
13	p9/r1/p0	pA/r4/p3	•	pC/r1/p9	pD/r6/pB
14	p9/r1/p0	pA/r4/p3	•	•	pD/r6/pB
15	p9/r1/p0	pA/r4/p3	•	•	•

Figure 6: Microarchitectural State (ROB) for Reg Renaming with Pointers in the IQ/ROB

2.d Register Renaming with Values in the IQ/ROB

	Stage				RT										ROB				
Cycle	**D**	**I**	**W**	**C**	**r1**	**r2**	**r3**	**r4**	**r5**	**r6**	**r7**	**r8**	**r9**	**IQ**	**0**	**1**	**2**	**3**	**4**
0																			
1	1																		
2	2	1			p0*									p0/r2/r3	p0*/r1				
3	3				p0*			p1*						p1/p0*/r5	p0*/r1	p1*/r4			
4	4	3			p0*			p1*		p2*				p2/r7/r8	p0*/r1	p1*/r4	p2*/r6		
5	5	4			p3*			p1*		p2*				p3/r2/r5	p0*/r1	p1*/r4	p2*/r6	p3*/r1	
6		2	3		p3*			p1*		p4*				p4/p2*/r9	p0*/r1	p1*/r4	p2*/r6	p3*/r1	p4*/r6
7		5	1		p3*			p1*		p4*					p0*/r1	p1*/r4	p2/r6	p3*/r1	p4*/r6
8				1	p3*			p1*		p4*					p0*/r1	p1*/r4	p2/r6	p3*/r1	p4*/r6
9			5		p3*			p1*		p4*						p1*/r4	p2/r6	p3*/r1	p4*/r6
10			4		p3*			p1*		p4						p1*/r4	p2/r6	p3*/r1	p4/r6
11			2		p3			p1*		p4						p1*/r4	p2/r6	p3*/r1	p4/r6
12				2				·		p4						p1/r4	p2/r6	p3*/r1	p4/r6
13				3						p4							p2/r6	p3*/r1	p4/r6
14				4	·					p4							•	p3*/r1	p4/r6
15				5						p4								•	p4/r6
16										·									•

Figure 7: Microarchitectural State for Reg Renaming with Values in the IQ/ROB

3 In-Order vs. Out-of-Order Superscalar Processors

Worked with Sacheth Hegde and Gautam Ramaswamy.

3.a Performance of In-Order Dual-Issue Processor

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
lw r1 , 0(r2)	F	D	I	L0	L1	W	C							
mul r3, r1, r4	F	D	I	I	I	Y0	Y1	Y2	Y3	W	C			
sw r3, 0(r5)		F	D	D	D	D	D	D	I	S	W	C		
addiu r2, r2, 4		F	D	D	D	D	D	D	I	A	W	C		
addiu r5, r5, 4			F	F	F	F	F	F	D	I	A	W	C	
addiu r6, r6, -1			F	F	F	F	F	F	D	I	B	W	C	
bne r6, r0, loop									F	D	I	A	W	C
opA									F	*	*	*	*	*

Figure 8: Pipeline Diagram for In-Order Dual-Issue Processor

As shown by the bold vertical lines, during steady state, each loop takes 9 cycles to execute. The W stage of the lw instruction is included because during looping, the W stages causes an extra cycle of "delay" between the last commit of the previous iteration and the first commit of the current iteration.

Therefore, the total number of cycles it takes to execute 64 iterations is 9*64 = 576 cycles.

3.b Performance of Out-of-Order Dual-Issue Processor

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
lw r1 , 0(r2)	F	D	I	L0	L1	W	C																
mul r3, r1, r4	F	D	i	i	I	Y0	Y1	Y2	Y3	W	C												
sw r3, 0(r5)		F	D	i	i	i	i	i	I	S	W	C											
addiu r2, r2, 4		F	D	I	A	W	r	r	r	r	r	C											
addiu r5, r5, 4			F	D	I	A	W	r	r	r	r	r	C										
addiu r6, r6, -1			F	D	i	I	B	W	r	r	r	r	C										
bne r6, r0, loop				F	D	i	I	A	W	r	r	r	r	C									
opA				F	*	*	*	*	*	*	*	*	*	*									
lw r1 , 0(r2)					F	D	I	L0	L1	W	r	r	r	C									
mul r3, r1, r4					F	D	i	i	I	Y0	Y1	Y2	Y3	W	C								
sw r3, 0(r5)						F	D	i	i	i	i	i	I	S	W	C							
addiu r2, r2, 4						F	D	i	i	I	A	W	r	r	r	C							
addiu r5, r5, 4							F	D	i	I	B	W	r	r	r	r	C						
addiu r6, r6, -1							F	D	i	i	I	A	W	r	r	r	C						
bne r6, r0, loop								F	D	i	i	I	A	W	r	r	r	C					
opA								F	*	*	*	*	*	*	*	*	*						
lw r1 , 0(r2)									F	D	i	I	L0	L1	W	r	r	C					
mul r3, r1, r4									F	D	i	i	i	I	Y0	Y1	Y2	Y3	W	C			
sw r3, 0(r5)										F	D	i	i	i	i	i	i	I	S	W	C		
addiu r2, r2, 4										F	D	i	i	I	A	W	r	r	r	r	C		
addiu r5, r5, 4											F	D	i	i	I	A	W	r	r	r	r	C	
addiu r6, r6, -1											F	D	i	i	I	B	W	r	r	r	r	C	
bne r6, r0, loop												F	D	i	i	I	A	W	r	r	r	r	C
opA												F	*	*	*	*	*	*	*	*	*	*	

Figure 9: Pipeline Diagram for Out-of-Order Dual-Issue Processor

As shown by the bold vertical lines, during steady state, it takes 9 cycles to execute 2 iterations of the loop. The first commit (by lw) of the second iteration is not included because the commit is part of previous iteration's last cycle

(this pattern is assumed to repeat; for example, the commit stage of the fourth iteration will be in the same cycle as the commit stage of the bne instruction at the end of the third iteration).

Therefore, the total number of cycles it takes to execute 64 iterations is 9*32 = 288 cycles.

3.c Dual-Issue In-Order versus Dual-Issue Out-of-Order

To achieve proper branch prediction, we will need a branch predictor (such as a BHT) and especially a branch target buffer in the fetch stage. This will allow us to both predict the branch result and the branch target in the fetch stage, which enables fetching the speculative instructions on the next cycle.

In the instruction sequence, sw r3, 0(r5) and addiu r5, r5, 4 will take advantage of register renaming by renaming the destination register r5. Without renaming, the addiu instruction will cause a WAR with the sw instruction, because the addiu instruction completes before the sw instruction is issued.

All lw r1, 0(r2) are issued before the sw instruction is issued in the previous iteration. However, because they access different locations in memory (assuming r2 $\neq$ r5), no memory disambiguation is required. The only case in which memory disambiguation will be required is when r2 + 4 = r5 (this is because the lw instruction is always accessing r2 from the previous iteration plus 4).

We will never need to replay the load instruction that need to be replayed when issued speculatively. This is because the load instruction is never loading from the same location as the store instruction in the previous cycle (same reason as the previous question).

The out-of-order processor is significantly better than the in-order processor. This is mainly due to the ability to issue out-of-order, which allows the execution of the next iteration to start before the end of the current iteration. By overlapping the execution of successive iterations, we were able to double the performance (from 0.78 to 1.56 IPC) of the execution of this instruction sequence.

Optimizing the code will help the in-order processor, but not significantly. We can group the fetch blocks such that "addiu r2, r2, 4" fetches with the lw, "addiu r6, r6, -1" fetches with the mul, and "addiu r5, r5, 4" fetches with the sw. With this optimal scheduling, we are only saving a single cycle per iteration. Without out of order issue, we cannot start the next iteration before the current iteration finishes execution, which is how the out of order processor achieve most of the improvements from. Therefore, no, optimized scheduling for the in-order processor will not significantly help the performance.

4 Branch Prediction

4.a Two-Bit Saturating Counter Branch History Table

		Branch B0			Branch B1			Branch B2		
i	src[i]	**BHT**	**P**	**A**	**BHT**	**P**	**A**	**BHT**	**P**	**A**
0	0	WT	T	T	WT	T	T	WT	T	T
1	0	ST	T	T	ST	T	T	ST	T	T
2	12	ST	T	NT	ST	T	NT	ST	T	T
3	15	WT	T	NT	WT	T	NT	ST	T	T
4	0	WNT	NT	T	WNT	NT	T	ST	T	T
5	0	WT	T	T	WT	T	T	ST	T	T
6	11	ST	T	NT	ST	T	NT	ST	T	T
7	17	WT	T	NT	WT	T	NT	ST	T	T
8	0	WNT	NT	T	WNT	NT	T	ST	T	T
9	0	WT	T	T	WT	T	T	ST	T	T
10	11	ST	T	NT	ST	T	NT	ST	T	T
11	13	WT	T	NT	WT	T	NT	ST	T	T
12	9	WNT	NT	T	WNT	NT	NT	ST	T	T
13	0	WT	T	T	SNT	NT	T	ST	T	T
14	12	ST	T	NT	WNT	NT	NT	ST	T	T
15	15	WT	T	NT	SNT	NT	NT	ST	T	T
16	0	WNT	NT	T	SNT	NT	T	ST	T	T
17	8	WT	T	T	WNT	NT	NT	ST	T	T
18	12	ST	T	NT	SNT	NT	NT	ST	T	T
19	18	WT	T	NT	SNT	NT	NT	ST	T	NT

Figure 10: Two-Bit Saturating Counter BHT Execution

4.b Two-Level Adaptive Branch Predictor to Exploit Temporal Correlation

		Branch B0				Branch B1				Branch B2			
i	src[i]	**BHSRT**	**BHT**	**P**	**A**	**BHSRT**	**BHT**	**P**	**A**	**BHSRT**	**BHT**	**P**	**A**
0	0	000	WT	T	T	000	WT	T	T	000	WT	T	T
1	0	001	WT	T	T	001	WT	T	T	001	WT	T	T
2	12	011	WT	T	NT	011	WT	T	NT	011	WT	T	T
3	15	110	WT	T	NT	110	WT	T	NT	111	WT	T	T
4	0	100	WT	T	T	100	WT	T	T	111	ST	T	T
5	0	101	WT	T	T	101	WT	T	T	111	ST	T	T
6	11	011	WNT	NT	NT	011	WNT	NT	NT	111	ST	T	T
7	17	110	WNT	NT	NT	110	WNT	NT	NT	111	ST	T	T
8	0	100	ST	T	T	100	ST	T	T	111	ST	T	T
9	0	101	ST	T	T	101	ST	T	T	111	ST	T	T
10	11	011	SNT	NT	NT	011	SNT	NT	NT	111	ST	T	T
11	13	110	SNT	NT	NT	110	SNT	NT	NT	111	ST	T	T
12	9	100	ST	T	T	100	ST	T	NT	111	ST	T	T
13	0	101	ST	T	T	000	ST	T	T	111	ST	T	T
14	12	011	SNT	NT	NT	001	ST	T	NT	111	ST	T	T
15	15	110	SNT	NT	NT	010	WT	T	NT	111	ST	T	T
16	0	100	ST	T	T	100	WT	T	T	111	ST	T	T
17	8	101	ST	T	T	001	WT	T	NT	111	ST	T	T
18	12	011	SNT	NT	NT	010	WNT	NT	NT	111	ST	T	T
19	18	110	SNT	NT	NT	100	ST	T	NT	111	ST	T	NT

Figure 11: Two-Level BHT for Temporal Correlation Execution

4.c Two-Level Adaptive Branch Predictor to Exploit Spatial Correlation

		Branch B0				Branch B1				Branch B2			
i	src[i]	**BHSR**	**BHT**	**P**	**A**	**BHSR**	**BHT**	**P**	**A**	**BHSR**	**BHT**	**P**	**A**
0	0	0	WT	T	T	1	WT	T	T	1	WT	T	T
1	0	1	WT	T	T	1	ST	T	T	1	ST	T	T
2	12	1	ST	T	NT	0	WT	T	NT	0	WT	T	T
3	15	1	WT	T	NT	0	WNT	NT	NT	0	ST	T	T
4	0	1	WNT	NT	T	1	ST	T	T	1	ST	T	T
5	0	1	WT	T	T	1	ST	T	T	1	ST	T	T
6	11	1	ST	T	NT	0	SNT	NT	NT	0	ST	T	T
7	17	1	WT	T	NT	0	SNT	NT	NT	0	ST	T	T
8	0	1	WNT	NT	T	1	ST	T	T	1	ST	T	T
9	0	1	WT	T	T	1	ST	T	T	1	ST	T	T
10	11	1	ST	T	NT	0	SNT	NT	NT	0	ST	T	T
11	13	1	WT	T	NT	0	SNT	NT	NT	0	ST	T	T
12	9	1	WNT	NT	T	1	ST	T	NT	0	ST	T	T
13	0	1	WT	T	T	1	WT	T	T	1	ST	T	T
14	12	1	ST	T	NT	0	SNT	NT	NT	0	ST	T	T
15	15	1	WT	T	NT	0	SNT	NT	NT	0	ST	T	T
16	0	1	WNT	NT	T	1	ST	T	T	1	ST	T	T
17	8	1	WT	T	T	1	ST	T	NT	0	ST	T	T
18	12	1	ST	T	NT	0	SNT	NT	NT	0	ST	T	T
19	18	1	WT	T	NT	0	SNT	NT	NT	0	ST	T	NT

Figure 12: Two-Level BHT for Spatial Correlation Execution

4.d Branch Predictor Comparison

	Two-Bit FSM Accuracy	**Two-Level Temporal Accuracy**	**Two-Level Spatial Accuracy**
Branch B0	30%	90%	30%
Branch B1	50%	65%	85%
Branch B2	95%	95%	95%
All Branches	58%	83%	70%

Figure 13: Summary of Branch Predictor Accuracies

For branch B0, two-level temporal achieved the best accuracy while two-bit FSM and two-level spatial performed equally worse. This is because the branch pattern repeats in time (as shown by the BHSRT and the actual branch result; the pattern is T,T,NT,NT,T,T,NT,NT,etc). This means that the BHT of each BHSRT element repeats in time, which allows for high accuracy with a temporal two-level predictor. The two-bit FSM performed very badly due to it lagging behind for each prediction. Because the branch result switches every two iterations, the branch predictor has to swing back and forth between the strong states of both sides. This is very expensive and causes only 1/4 correct prediction for every T,T,NT,NT pattern. The two-level spatial also performed very badly because it behaves the same as the two-bit FSM (it in fact is functionally the same). This is because the branch B2 is always taken, therefore fixing the BHSR to 1 during the execution of branch B0 (thereby forcing to act as a two-bit FSM).

For branch B1, two-level spatial achieved the best accuracy while two-bit FSM and two-level temporal performed similarly worse. This is because whenever branch B0 chooses NT, branch B1 will also be NT (when a value is greater than saturation, it will always also be greater than 0). Therefore, there is a spatial correlation between branch B0 and branch B1. The two-level temporal achieved worse accuracy because there isn't a fully repetitive pattern in the input data, therefore, there is no definitive temporal repetition in the branching pattern. The two-level FSM again performed the worst due to the same reason as state in branch B0.

For branch B2, all predictors performed equally well. This is because the branch pattern is very simple (loop). The branching pattern is always taken except for the last cycle. Because all of the predictors start on weakly taken state, they will always converge to a strongly taken state and always guess taken.

5 Connections to Classic Architectures

5.a IBM System/360 Model 91 with the Tomasulo Algorithm

The IBM System/360 Model 91 supports out of order writeback/completion as shown by the example instruction sequence on pg.171 of Hennessy and Patterson. Register renaming was added to eliminate WAW and WAR hazards. Furthermore, System/360 also has out of order issue (in the textbook, this is stated as the execute stage; pg. 174 of Hennessy and Patterson). Instructions are issued in order from the issue queue, but stalled in the functional units until their source operands are available. This is functionally the same as an out of order issue stage that we have in the IO2E and IO2L microarchitectures. Finally, the writeback stage is most similar to a late commit system. As stated on pg. 172, if multiple instructions write to the same destination register, only the latest instruction is used to update the register file. Therefore, the IBM System/360 Model 91 is most similar to an IO2L architecture.

5.b Register Renaming in the MIPS R10K and the Intel P6 Microarchitectures

The MIP R10000 microarchitecture uses a pointer-based register renaming scheme. As stated on page 33 of the report (see source), the register mapping system contains register map tables, free lists, active list, and busy-bit tables. The register map tables takes on the duty of the rename table in our scheme. The register map table keeps track of all current mappings from the PRF to the ARF. The active list then acts as the ROB, keeping track of all current instructions that are active in the processor. The active list holds the ARF register and the previous PRF register. The free list acts exactly like the FL in our pointer-based scheme. The busy-bit tables simply act as the list of valid bits in the ROB table. When combined, these units function as a pointer-based register renaming system.
Source: The MIPS R10000 Superscalar Microprocessor by Kenneth C. Yeager
(http://people.cs.pitt.edu/ cho/cs2410/papers/yeager-micromag96.pdf)

The Intel P-6 microarchitecture uses a register renaming scheme that is similar to a value-based scheme. As stated on page 2 of the tech report (see source), the RAT (register alias table), which is similar to our rename table, determines whether an operand should be taken from the RRF (real register file), which is similar to our ARF, or from the ROB. If a source operand is currently pending, then the reservation station (similar to our issue queue) is given a pointer to a location in ROB instead and the instruction waits until that source operand is no longer pending. This means that all values are stored in either the ROB or the RRF, which is similar to our value-based scheme, where all values are stored in either the ROB or ARF.
Source: Intel's P6 Uses Decoupled Superscalar Design by Linley Gwennap
(http://www.cs.cmu.edu/afs/cs/academic/class/15213-f01/docs/mpr-p6.pdf)