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.