6.823 Computer System Architecture

Problem Set #4 Spring 2002

Students are encouraged to collaborate in groups of up to 3 people. A group needs to hand in only one copy of the solution to a problem set. Homework assignments are due at the beginning of class on the due date. To facilitate grading, each problem must be stapled separately and your latest group number must appear on each problem (If your group has changed please indicate this and we will assign you a new group number). Homework will not be accepted once solutions are handed out.

Problem 1: Out-of-order Scheduling

Ben Bitdiddle is adding a floating-point unit to the basic DLX pipeline. He has patterned the design after the IBM 360/91's floating-point unit. His design has one adder, one multiplier, and one load/store unit. The *adder* has a two-cycle latency and is fully pipelined. The *multiplier* has a three-cycle latency and is un-pipelined. The *load/store* unit has a very long latency and is fully-pipelined; it handles all memory requests in-order.

There are 4 floating-point registers, **`F0-F3`**. These are separate from the integer registers. There is a single write-back port to the floating-point register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions must spend one cycle in the write-back stage before its result can be used, as explained in lecture (refer to slide L11-30). However, integer results are available one cycle after issue as usual.

Ben is now deciding whether to go with (a) in-order issue using a scoreboard, (b) out-of-order issue, or (c) out-of-order issue with register renaming. His favorite benchmark is this inner loop (the last instruction is in a branch delay slot). Your job is to evaluate the three alternatives.

```
    loop:
I1        LF     F0, A(R2)
I2        MULTF  F3, F1, F2
I3        ADDF   F3, F3, F0
I4        SF     B(R2), F3
I5        LF     F1, C(R2)
I6        MULTF  F3, F0, F2
I7        ADDF   F3, F3, F1
I8        SF     D(R2), F3
I9        BNEZ   R2, loop
I10       ADDI   R2, R2, #-8
```

Problem 1.A — In-order using a scoreboard

Fill in the scoreboard in table A (refer to slide L11-30) to simulate the execution of one iteration of the loop. Assume that loads take **n** cycles (**n** very large), plus one cycle for write-back. Leave an empty row in the table to indicate the time spent waiting for the load result. Keep in mind that, in this scheme, no instruction is issued that has a WAW hazard with any previous instruction that has not written back (just like in the lecture slides).

In steady state, how many cycles does each iteration of the loop take, in terms of **n**? What is the bottleneck?

Problem 1.B — Out-of-order

Now consider a single-issue out-of-order implementation (refer to slide L12-3). In this scheme, the issue stage buffer holds multiple instructions waiting to issue. The decode stage can add up to one instruction per cycle to the issue buffer. The decode stage adds an instruction to the issue buffer if there is space and if the instruction does not have a WAR hazard with any previous instruction that has not issued or a WAW hazard with any previous instruction that has not written back. Assume you have an infinite issue buffer.

Table B represents the execution of one iteration of the loop ***in steady state***. Fill in the cycle numbers for the cycles at which each instruction issues and writes back. The first row has been filled out for you already; please complete the rest of the table. Note that the order of instructions listed is not necessarily the issue order. We define cycle 0 as the time at which instruction I_1 is issued. Assume again that loads take **n** cycles. As the load/store unit if fully-pipelined, you need not worry about the value of **n** possibly causing a structural hazard.

Draw arrows for the dependencies that are involved in the critical path of the loop in table B. Can the execution of multiple iterations be overlapped?

Problem 1.C — Register Renaming

The number of registers specified in an ISA limits the maximum number of instructions that can be in the pipeline. This question studies register renaming to solve this problem. In this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers.

Table C shows instructions from our benchmark for two iterations using the same format as in Table B. First, fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time you need to rename a register (T0, T1, T2, etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name. You may find it helpful to create your own rename table. Rename only floating-point instructions.

Next, fill in the cycle numbers for the cycles at which each instruction issues and writes back. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB). Assume that instruction I_2 was decoded in cycle 0, and cannot be issued until cycle 1. Also assume that loads take **n** cycles and that you have an infinite ROB.

For two iterations, at which cycle does the last-issued instruction get issued? At which cycle does the last-issued instruction get issued for 1000 iterations? What is the performance bottleneck?

Problem 1.D — Tomasulo's Algorithm

Consider an out-of-order implementation using Tomasulo's algorithm (refer to slide L12-12). In this scheme, there is no separate floating-point instruction queue (ROB). Floating-point instructions go directly from instruction fetch to one of the reservation stations. If there is no room, the fetch stage stalls. If several instructions in a reservation station can be issued, the oldest instruction is issued.

Unlike the previous question, there are a fixed number of slots in each reservation station, which limits the number of registers that can be renamed. Assume that we have 3 slots for an adder reservation station, 2 slots for a multiplier station, 3 slots for stores, and 6 slots for loads.

How many iterations of the benchmark can be overlapped? We define two iterations as being overlapped if there are instructions from two different iterations in the same reservation station or in different reservation stations at the same time. Assume that **n** is very large (**n** > 100).

What is the minimum number of slots that we need for the multiplier reservation station to achieve the maximum possible performance? Assume that all other stations have infinite slots.

Problem 2: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the DLX ISA has no delay slots.**

Throughout this problem we will be working with the following program:

```
loop:
     LW   R4, 0(R3)
     ADDI R3, R3, #4
     SUBI R1, R1, #1
b1:
     BEQZ R4, b2
     ADDI R2, R2, #1
b2:
     BNEZ R1, loop
```

Assume the initial value of R1 is n (n>0).
Assume the initial value of R2 is 0 (R2 holds the result of the program).
Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

We will be using a 2-bit predictor state machine in this problem (refer to slide L13-11).

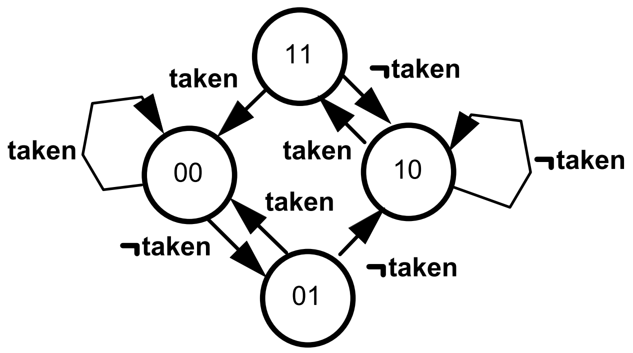

Figure 1: BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem 2.A Program

What does the program compute? That is, what does R2 contain when we exit the loop?

Problem 2.B — 2-bit branch prediction

Now we will investigate how well our standard 2-bit history branch predictor (refer to slide L13-12) performs. Assume the inputs to the program are n=8 and p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0,… etc.; i.e. the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table 1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table 1 contains an entry for every time a branch (either b1 or b2) is executed. The Branch Prediction (BP) bits in the table are the bits from the BHT. If b1 is being executed, then the b1 bits from the BHT are to be filled in. If b2 is being executed, then the b2 bits from the BHT are to be filled in.

Problem 2.C — Branch prediction with one global history bit

Now we add a global history bit to the branch predictor (refer to slide L13-13). Fill out Table 2, and again give the total number of mispredicts you get when running the program with the same inputs.

Problem 2.D — Branch prediction with two global history bits

Now we add a second global history bit (refer to slide L13-14). Fill out Table 3. Again, compute the number of mispredicts you get for the same input.

Problem 2.E — Analysis I

Compare your results from problems 2.B, 2.C, and 2.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For large n, what prediction scheme will work best? Explain briefly.

Problem 2.F — Analysis II

The input we worked with in this problem is quite regular. How would you expect things to change if the input were random (each array element were equally probable 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n?

What does this tell you about when additional history bits are useful and when they hurt you?

Problem 3: Register Renaming and Static versus Dynamic Scheduling

The following DLX code calculates the floating-point expression E = A * B + C – D, where the addresses of A, B, C, D, and E are stored in R1, R2, R3, R4, and R5, respectively:

```
LF      F0, 0(R1)
LF      F1, 0(R2)
MULTF   F0, F0, F1
LF      F2, 0(R3)
LF      F3, 0(R4)
SUBF    F2, F2, F3
ADDF    F0, F0, F2
SF      0(R5), F0
```

Problem 3.A Simple Pipeline

Calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive) on a simple in-order pipelined machine. Assume that the load latency is two cycles. Floating-point multiply has a four-cycle latency. Other floating-point arithmetic operations have two-cycle latencies. Write-back for floating-point registers takes one cycle. Also assume that all functional units are fully pipelined and ignore any write back conflicts. Give the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive.

Problem 3.B Static Scheduling

Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence and give the number of cycles this sequence takes to execute on the simple in-order pipeline.

Problem 3.C Less Registers

Rewrite the code sequence, but now using only two floating-point registers. Optimize for minimum run-time. In general, when there are insufficient registers to perform a computation, the compiler may use temporary memory locations to hold intermediate values, a process known as register-spilling. In this case, you should not need to spill any registers. List the code sequence and give the number of cycles this takes to execute.

Problem 3.D Register renaming and dynamic scheduling

Now simulate the effect of running the original code on a single-issue machine with register renaming and out-of-order issue. Ignore structural hazards apart from the single instruction decode per cycle. Show how the code is executed and give the number of cycles required. How does this compare with the optimized code in Part 3.B?

Problem 4: Register Lifetimes

For this problem, we introduce a scheme whereby every source register specifier in an instruction has an extra *register lifetime* bit (set by the compiler) which indicates that the corresponding instruction is the last to need that particular register value, i.e., no following instruction will read the register before another instruction overwrites it with a new value.

You are now given the following DLX instruction sequence:

```
I1:   LW   R1, 0(R2)
I2:   ADDI R2, R5, #4
I3:   SUB  R4, R1, R5
I4:   ADD  R1, R2, R6
I5:   LW   R5, 0(R4)
I6:   ADD  R2, R1, R4
I7:   SW   0(R1), R4
I8:   LW   R4, 0(R5)
```

Problem 4.A — Adding RL Bits

Please create a table like the one below and place checks indicating which registers can have their register lifetime (RL) bits set for the code sequence given above. For instructions with only one source operand, use the Src1 column.

Instruction #	Src1 RL Bit	Src2 RL Bit
I_1		
I_2		
I_3		
I_4		
I_5		
I_6		
I_7		
I_8		

Given the register lifetime bits, we now attempt to optimize the register renaming techniques described in Lecture 12.

Problem 4.B — Freeing Physical Registers

In the physical register file renaming scheme described in Lecture 14, when can a physical register be safely freed (i.e. put on the free list and reused for new renamings)?

Problem 4.C — New Register Deallocation Policy

Assume now that we use the register lifetime bits to improve the performance of our machine. Explain concisely any change in the policy for when a physical register can be freed.

Problem 4.D — Benefits of RL Bits

In the following code sequences, an underlined operand denotes that the corresponding register lifetime bit is set. Circle any code sequences for which having register lifetime information could allow the microprocessor to rename additional instructions following I_3 sooner than if we just had the original register-renaming scheme. Assume that the `CPLX` instruction is a very long-latency operation that takes many cycles to complete. Explain your selections.

Sequence A

I_1: ADD R1, R3, R4
I_2: ADD R3, R1, R2
I_3: CPLX R1, $\underline{\text{R2}}$, R3

Sequence B

I_1: ADD R1, $\underline{\text{R2}}$, R3
I_2: CPLX R4, R5, R6
I_3: ADD R2, R1, R3

Sequence C

I_1: ADD R1, R2, R3
I_2: ADD R2, $\underline{\text{R2}}$, R5
I_3: CPLX R4, R5, R6

Sequence D

I_1: CPLX R1, R2, R3
I_2: ADD R3, $\underline{\text{R2}}$, R1
I_3: ADD R2, R1, R5

Problem 5: Synonyms & Homonyms

Consider adding a virtually-indexed, virtually-tagged, 4-way set-associative, write-back cache to a system running multiple processes. Assume the TLB and processor use ASIDs (Address Space Identifiers).

Problem 5.A

A *synonym* problem, also called aliasing, occurs when distinct virtual addresses refer to a same physical location (this can be within a single process or from multiple processes). Describe a *simple* scenario illustrating a serious problem caused by synonyms.

Problem 5.B

Can we avoid the synonym problem by changing the cache to be a write-through cache? Explain.

Problem 5.C

Can we avoid the synonym problem by changing the cache to be direct-mapped? For the direct-mapped case, does it matter if it is write-through or write-back? Explain

Problem 5.D

Will we still have a *synonym* problem if the cache is a virtually-indexed, physically-tagged cache? Explain.

Problem 5.E

A *homonym* problem can happen when two processes use the same virtual address to access different physical locations. One way to avoid this problem is to flush the cache during each context-switch. Alyssa P. Hacker suggests that by adding a small amount of information to each cache line, the homonym problem can be eliminated without needed to flush the cache. Explain briefly how this can be done. Will this also solve the *synonym* problem? (Assume the original virtually-indexed, virtually-tagged cache is being used.)

Problem 5.F

Can we solve the *homonym* problem by instead using a virtual-index, physical-tag cache? Explain.

Problem 5.G

Ben Bitdiddle claims that both the synonym and homonym problems for the virtually-indexed, virtually-tagged cache can be avoided by making the cache physically-indexed, physically-tagged. Does Ben's idea completely solve the homonym and synonym problems? What are the drawbacks of this scheme over virtually-indexed caches? Explain briefly.

Table A – problem 1.A

Instr. Issued	Time (cycles)	Functional Unit Status							Registers Reserved for Writes
		Store (n)	Adder (2)		Multiplier(3)			WB	
I_1	0	F0							F0

Table B – problem 1.B

	Time		Op	Dest	Src1	Src2
	Issue	WB				
I_1	0	n	`LF`	F0	R2	▨
I_2			`MULTF`	F3	F1	F2
I_3			`ADDF`	F3	F3	F0
I_4		▨	`SF`	▨	R2	F3
I_5			`LF`	F1	R2	▨
I_6			`MULTF`	F3	F0	F2
I_7			`ADDF`	F3	F3	F1
I_8		▨	`SF`	▨	R2	F3
I_9		▨	`BNEZ`	▨	R2	▨
I_{10}		▨	`ADDI`	R2	R2	▨

Table C – problem 1.C

	Time		Op	Dest	Src1	Src2
	Issue	WB				
I_1	0	n	`LF`	t0	R2	▨
I_2			`MULTF`	t1	F1	F2
I_3			`ADDF`	t2	t1	t0
I_4		▨	`SF`	▨		
I_5			`LF`	t3		▨
I_6			`MULTF`			
I_7			`ADDF`			
I_8		▨	`SF`	▨		
I_9		▨	`BNEZ`	▨		▨
I_{10}		▨	`ADDI`			▨
I_1			`LF`			▨
I_2			`MULTF`			
I_3			`ADDF`			
I_4		▨	`SF`	▨		
I_5			`LF`			▨
I_6			`MULTF`			
I_7			`ADDF`			
I_8		▨	`SF`	▨		
I_9		▨	`BNEZ`	▨		▨
I_{10}		▨	`ADDI`			▨

System State		Branch Predictor	Behavior		Updated Values
PC	R3/R4	BP bits	Predicted Behavior	Actual Behavior	New BP bits
b1	4/1	**10**	**N**	**N**	**10**
b2	4/1	**10**	**N**	**T**	**11**
b1	8/0	**10**	**N**	**T**	**11**
b2	8/0	**11**	**N**	**T**	**00**
b1	12/1				
b2	12/1				
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					

Table 1: Behavior of branch prediction (Problem 2.B)

System State			Branch Predictor		Behavior		Updated Values		
PC	R3/R4	history bit	BP bits		Predicted Behavior	Actual Behavior	New BP bits		New history
			set 0	set 1			set 0	set 1	
b1	4/1	**1**	**10**	**10**	**N**	**N**	**10**	**10**	**0**
b2	4/1	**0**	**10**	**10**	**N**	**T**	**11**	**10**	**1**
b1	8/0								
b2	8/0								
b1	12/1								
b2	12/1								
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									

Table 2: Behavior of branch prediction with one history bit (Problem 2.C)

System State			Branch Predictor				Behavior		Updated Values				
PC	R3/R4	history bits	BP bits				Predicted Behavior	Actual Behavior	New BP bits				New Hist.
			set 0	set 1	set 2	set 3			set 0	set 1	set 2	set 3	
b1	4/1	**11**	**10**	**10**	**10**	**10**	**N**	**N**	**10**	**10**	**10**	**10**	**01**
b2	4/1	**01**	**10**	**10**	**10**	**10**	**N**	**T**	**10**	**11**	**10**	**10**	**10**
b1	8/0												
b2	8/0												
b1	12/1												
b2	12/1												
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													

Table 3: Behavior of branch prediction with two history bits (Problem 2.D)

Note: history bits = 10 maps to BP set 2.