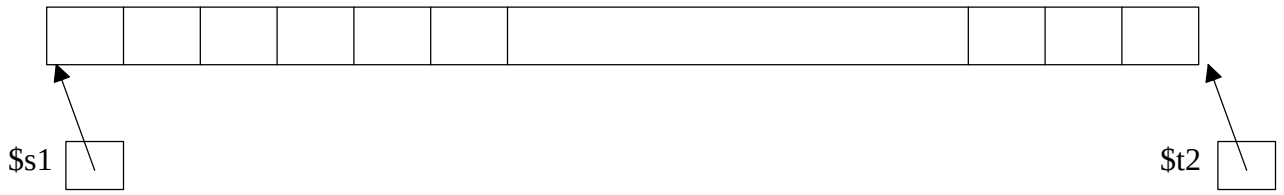
Team #:____ Return To:_____
Absent: 9-21-06

0. Consider the following MIPS loop to add a constant/scalar in $s2 to every element in an array. Suppose that $s1 contains the starting address of the array, and $t2 contains the address after the loop.



a) Complete the following timing diagram showing the stages. Assume data forwarding.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14
loop: lw $t0, 0($s1)	IF	ID	EX	MEM	WB									
add $t0, $t0, $s2		IF	ID	-	EX	MEM	WB							
sw $t0, 0($s1)			IF	-	ID	EX	MEM	WB						
addi $s1, $s1, 4					IF	ID	EX	MEM	WB					
bne $s1, $t2, loop						IF	-	ID	EX	MEM	WB			

b) Instead of stalling after the "lw" instruction, we can reorder the instructions by moving the "addi" after the "lw". What modifications to the "sw" instruction must be done?

c) Suppose we consider running this code on an two-issue superscalar MIPS-like machine with a functional unit to handle ALU and branch instruction, and a functional unit for data transfer instructions (lw and sw). The scheduled code on the functional units would be (blank spaces indicate NOOP instructions):

	ALU or Branch Instruction Functional Unit	Data Transfer Instruction Functional Unit	Cycle
loop:	addi $s1, $s1, 4	lw $t0, 0($s1)	1
			2
	add $t0, $t0, $s2		3
	bne $s1, $t2, loop	sw $t0, -4($s1)	4

Why can't the "add" instruction cannot be moved to cycle 2.

d) Loop unrolling (software pipelining) can be used to increase the amount of ILP is a loop (as well as decrease the amount of loop overhead). In loop unrolling multiple copies of the loop body are made and instructions from different loop iterations are scheduled together. The "addi" and "bne" instructions can be reduced to once per unrolled loop leaving the basic sequence of instructions as:

ALU or Branch Instruction Functional Unit	Data Transfer Instruction Functional Unit	Cycle
	lw $t0, 0($s1)	1
		2
add $t0, $t0, $s2		3
	sw $t0, -4($s1)	4

If we unroll the iterations of the loop (denoted by the subscripts) , then we get:

Team #:____ Return To:_____
Absent: 9-21-06

Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Cycle
lw_1						1
	lw_2					2
add_1		lw_3				3
sw_1	add_2		lw_4			4
	sw_2	add_3		lw_5		5
		sw_3	add_4		lw_6	6
			sw_4	add_5		7
				sw_5	add_6	8
					sw_6	9

After we get the "pipeline" full in cycles 4 or higher, we actually have too much ILP for our two-issue superscalar machine. If we only unroll the loop to a "depth" of 4 iterations, the we would have:

Iteration 1	Iteration 2	Iteration 3	Iteration 4	Cycle
lw_1				1
	lw_2			2
add_1		lw_3		3
sw_1	add_2		lw_4	4
	sw_2	add_3		5
		sw_3	add_4	6
			sw_4	7
				8

e) How can we fix, cycle 4 so that we don't have two data transfer instructions to perform?

f) If we unroll the loop to a depth of 4, how much should we move/walk register $s1 by each iteration of the unrolled loop?

g) During what cycle can we put the "addi" instruction?

h) During what cycle can we put the "bne" instruction?

i) Write the complete code after unrolling it to a depth of 4.

ALU or Branch Instruction Functional Unit	Data Transfer Instruction Functional Unit	Cycle
	lw $t0, 0($s1)	1
		2
add $t0, $t0, $s2		3
		4
		5
		6
		7
		8
		9

j) What effect does loop unrolling have on code size?

Team #:____ Return To:_____
Absent: 9-21-06

1. In the IA-64's (Itanium 2) *explicit parallel instruction computing (EPIC)* the compiler encodes multiple operations into a long instruction word so hardware can schedule these operations at run-time on multiple functional units without analysis, i.e., static multiple-issue. Why might the compiler be better able to find instructions that do not have dependencies than run-time hardware of a superscalar computer?

2. For a typical program on a traditional computer, more time is spent doing procedure/method calls than anything else. Why are procedure calls so time consuming?

3. What ways are parameters passed during a procedure call?

4. The Itanium assembly language can eliminate branch instructions by using predicate registers (e.g., p2, p3) as the following code:

if (R1 == R2)
  R3 = R3 + R1
else
  R3 = R3 - R1
end if
	cmp.eq p2, p3 = r1, r2
	(p2) add r3 = r3, r1
	(p3) sub r3 = r3, r1

However, why does this technique only improve performance if the "then" and "else" bodys are small sections of code?