Hayden Miedema and Douglas Money

CIS 451

Professor Kurmas

Lab 7: Pipeline Optimizations

1. Write the straightforward MIPS assembly language code that implements the code snippet. This means that you should write all the assembly for line 1 (including any necessary sw instructions) before writing the assembly for line 2. Your code should not explicitly list any stalls. There should be 11 machine instructions in your code, not including the noops and trap #0 used to flush the pipeline at the end of the program.
   1. Offsets: Sum: 0 Item1:4 Item2: 8 Total: 12 Tax: 16 Sum2: 20 Item3: 24 Item4: 28

Assembly code:

lw r2, 4(r1)

lw r3, 8(r1)

add r4, r2, r3

sw 0(r1), r4

lw r2, 16(r1)

add r3, r4, r2

sw 12(r1), r3

lw r2, 24(r1)

lw r3, 28(r1)

add r4, r2, r3

sw 20(r1), r4

2. How many cycles does it take the "dlxview" machine to "execute" the code that you have written (not including the trap #0 at the end)? (Start counting with cycle 0, and stop counting when the final sw instruction is in the "WB" state. Note that the total number of cycles is one more than the number DLXview gives to the last cycle.)

The code takes 17 cycles to execute through.

3. Show the locations of the stalls and branch delays in your code from problem 1.

lw r2, 4(r1)

lw r3, 8(r1)

add r4, r2, r3

sw 0(r1), r4

nop

lw r2, 16(r1)

add r3, r4, r2

sw 12(r1), r3

nop

lw r2, 24(r1)

lw r3, 28(r1)

add r4, r2, r3

sw 20(r1), r4

4. Pretend you are an optimizing compiler: Rearrange (not rewrite) the instruction sequence to eliminate the data hazards (and the subsequent stalls).

Rearranging the instruction sequence with sum being called at the end gives the CPU enough time to store sum. While this is being done, more instructions are able to be pipelined that are independent of sum.

lw r2, 4(r1); load item1

lw r3, 8(r1); load item2

lw r5, 24(r1); load item3

lw r6, 28(r1); load item4

add r4, r2, r3; add item1+item2=sum

lw r2, 16(r1); load tax

add r7, r5, r6; add item3+item4=sum2

sw 0(r1), r4; store sum

add r3, r4, r2; add sum+tax=total

sw 20(r1), r7; store sum2

sw 12(r1), r3; store total

5. How many cycles does your "optimized" code take to execute?

This “optimized” code now takes only 14 cycles to execute because we got rid of the nops to push through the pipeline.

6. Run exampleCO-1 through dlxview then Identify *and explain* all data and control hazards that result in "wasted" cycles. Look carefully, there is a data hazard you probably don't expect!

The CPU inserts a stall after the first lw instruction is run (lw r6, 0x0(r2)). This is because the CPU has to wait for the lw to finish putting the value into r6 before running through the add r7, r5, r6. This data hazard is avoided because if the sw instruction were to run the value stored would be the incorrect r7 value.

A control hazard is apparent during the bnez r4, LOOP instruction. The CPU inserts a stall because it must wait for the previous pseudo subi r4, r4, 0x1 instruction to complete before being able to do a proper comparison in the bnez call.

7. How many cycles does it take to complete the program in exampleCO-1.s when MAX is 4?

When MAX = 4 the program takes 54 cycles to complete

8. Now, state the number of cycles as a function of MAX.

Cycles = 6 + 12 \* (MAX)

9. Re-arrange the code to eliminate any data hazards and branch delay slots.

addi r4, r0, 4; Set r4 (the loop counter) to MAX

lw r5, 0(r1); Load from a into r5

LOOP: lw r6, 0(r2); Load from b into r6

addi r2, r2, 4

add r7, r5, r6; r7 = r6 + r5

addi r1, r1, 4; increment pointers for arrays a, b, and c

sw 0(r3), r7; store result back in array c

subi r4, r4, 1; decrement loop counter

addi r3, r3, 4

bnez r4, LOOP; branch

lw r5, 0(r1); branch delay slot

sw 0(r3), r0; set c[0] to 0. (not important, just something to do after the loop.)

nop;

nop;

nop;

nop;

trap #0;

10. How many cycles does it take to complete your optimized program when MAX = 4?

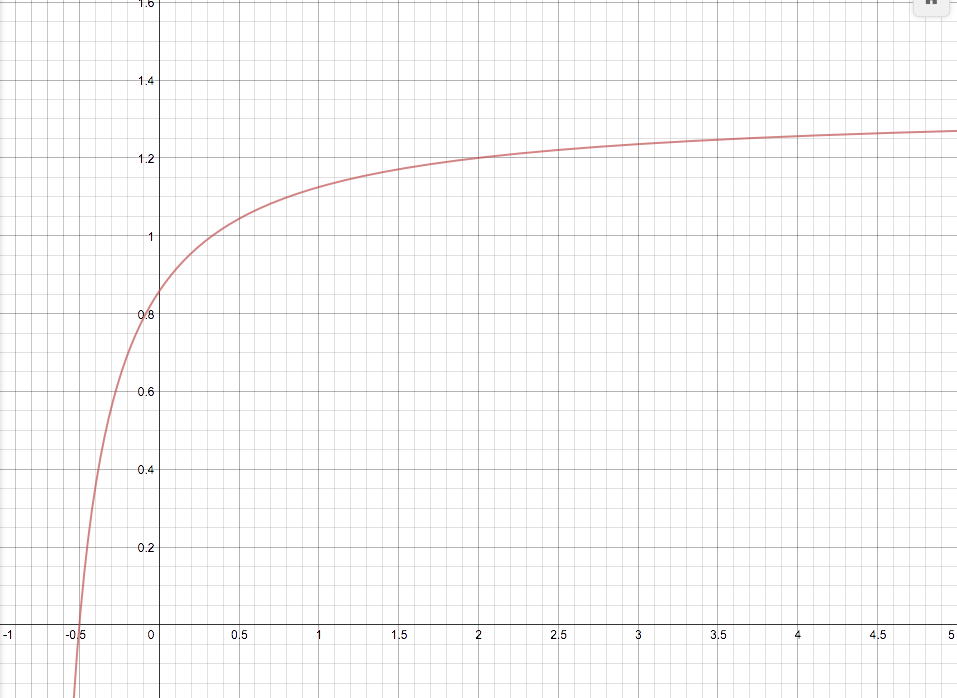
The new optimized program takes 43 cycles to finish

11. State the number of cycles of your re-arranged program as a function in terms of MAX. (**Important:** At this point, have the instructor double-check your answers to make sure you're on the right track.)

Cycles = 6 + 9\*(MAX)

12. What is the speedup of your optimized code when MAX = 4?

The speed up is the difference between the two run times. The first took 54 cycles while the second took 43. Therefore, the speedup is by a factor of 54/43 = 1.256.

13. Use your favorite plotting tool (I like gnuplot) to plot the speedup of your optimized code as a function of MAX. Attach the graph to your lab submission.

14. As MAX gets large, the speedup approaches an upper bound. What is this bound?

This upper bound is 4/3, or ~ 1.33333333.

15. Assume that the value of MAX is known to be 4 at the time the program is compiled. "Unroll" the optimized loop from problem 9 and optimize the unrolled loop to use as few cycles as possible. Where possible, modify the offset of lw instructions in order to remove addi instructions. (Because MAX is known at compile time, you are effectively removing the loop. Your optimized code should have 17 instructions and no stalls.)

lw r5, 0(r1); load A[0]

lw r6, 0(r2); load B[0]

lw r8, 4(r1); load A[1]

lw r9, 4(r2); load B[1]

add r7, r5, r6; add A[0]+B[0]

lw r10, 8(r1); load A[2]

add r4, r8, r9; add A[1]+B[1]

lw r11, 8(r1); load B[2]

lw r14, 12(r2); load B[3]

lw r13, 12(r1); load A[3]

add r12, r10, r11; add A[2]

sw 8(r3), r12; store A[2]+[2]

add r15, r13, r14; add A[3]+B[3]

sw 4(r3), r4 ; store A[1]+B[1]

sw 0(r3), r7; store A[0]+B[0]

sw 12(r3), r15; store A[3]+A[3]

nop;

nop;

nop;

nop;

trap #0;

16. How many cycles does your "unrolled" assembly take?

The unrolled loop program takes 20 cycles.

17. What is the speedup of this unrolled code over the optimized program from problem 9?

The speedup is 43/20, or by 2.15 units.

18. The tradeoff of unrolling loops is that your source code grows in length. By what percentage did your source code grow when you unrolled the loop (as compared to the source code you wrote for problem 9)? Do not count the trailing nops and trap #0.

The first program contains 12 instructions while the second contains 16. Our code grew by 1-(16/12) = .33. The unrolled loop style program will have 33% more instructions.

19. Assume that the value of MAX is not known at compile time, but is known at run-time before the loop begins. Unroll the optimized loop from problem 9 so that each iteration of the unrolled loop performs the work of 4 original loops. (In other words, iteration 1 of the new loop computes c[0] through c[3]; iteration 2 computes c[4] through c[7], etc.) For simplicity, assume that MAX is a multiple of 4.

addi r16, r0, 16; Set r16 (the loop counter) to MAX

lw r5, 0(r1); load A[0]

LOOP:lw r6, 0(r2); load B[0]

lw r8, 4(r1); load A[1]

lw r9, 4(r2); load B[1]

add r7, r5, r6; add A[0]+B[0]

lw r10, 8(r1); load A[2]

add r4, r8, r9; add A[1]+B[1]

lw r11, 8(r1); load B[2]

lw r14, 12(r2); load B[3]

lw r13, 12(r1); load A[3]

add r12, r10, r11; add A[2]

sw 8(r3), r12; store A[2]+[2]

add r15, r13, r14; add A[3]+B[3]

sw 4(r3), r4 ; store A[1]+B[1]

sw 0(r3), r7; store A[0]+B[0]

sw 12(r3), r15; store A[3]+A[3]

subi r16, r16, 4; decrement loop counter by 4 to loop correctly

addi r1, r1, 16; increment address

addi r2, r2, 16; increment address

addi r3, r3, 16; increment address

bnez r16, LOOP; branch

lw r5, 0(r1); branch delay slot

sw 0(r3), r0; set c[0] to 0. (not important, just something to do after the loop.)

nop;

nop;

nop;

nop;

trap #0;

20. How could you modify your code so that it could handle any value of MAX (as opposed to multiples of 4 only)?

We could make a single loop at the end to calculate the numbers after the 4x loops run. For example: the loop should run 14 times. The 4x loop would iterate 3 times, and then the 1x loop at the end would iterate 2 times.

21. Write a function that describes the number of cycles taken by your optimized code (from problem 19) in terms of MAX. *(Verify your answer with the instructor before continuing.)*

Cycles = 7+(21\*MAX) / 4

22. When MAX is large, what is the speedup of the unrolled version as compared to the optimized code from problem 9?

When MAX = 400 7+(9400)/7+(21400)/4 = 1.7119

The speedup of the new instruction is 1.7119 when MAX is 400

23. How much bigger (in terms of percentage) is the unrolled source code than the version in problem 9?

1-25/14=0.7857

The new program 78.57% larger than the version is problem 9.

24. Write a function that describes the number of cycles taken by your unrolled loop in terms of both MAX, and I (the number of iterations per unrolled loop.)

7+(5+(4\*i))(MAX/i)

25. Write a function that describes the speedup of your unrolled loop (as compared to the optimized code in problem 9) in terms of both MAX, and I.

7+(9\*MAX)/7+(5+(4\*i))(MAX/i)

26. As MAX grows large, the effects of the constant terms (i.e., the "+6") become negligible. If you ignore these terms, MAX should cancel, leaving a formula in terms of I. What is it?

(9\*MAX)/(4\*i)(MAX/i) = 9i/4i

27. What is the limit of this speedup as I grows? (Hint: There is a finite limit. If you think that the answer is "there is no limit", then you made a mistake somewhere.)

The limit is infinity

28. Calculate the size of your unrolled loop as a function of I.

When MAX = 16 7+(5+(4\*i))(16/i)

29. What is the limit of this function as I grows?

The limit of this function as I grows is 71

30. What would you consider the optimal number of iterations per unrolled loop? Why? (This is an opinion question. There isn't a right answer, you just have to say something thoughtful.)

After i = 9 the asymptote begins to slow down exponentially. The returns of the unlooping are not easy to be seen after 9.