Faculty of Media Engineering and Technology
Dept. of Computer Science and Engineering
Dr. Milad Ghantous

CSEN 702: Microprocessors
Winter 2022

Practice assignment 5
Solution

Exercise 1

The following loop is the so-called DAXPY loop (double-precision aX + Y) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, Y = aX + Y, for a vector length 100.

Initially, R1 is set to the base address of array X and R2 is set to the base address of Y.

```
        DADDIU   R4,R1,#800   ; R1 = upper bound for X
foo:    L.D      F2,0(R1)     ; (F2) = X(i)
        MUL.D    F4,F2,F0     ; (F4) = a*X(i)
        L.D      F6,0(R2)     ; (F6) = Y(i)
        ADD.D    F6,F4,F6     ; (F6) = a*X(i) + Y(i)
        S.D      F6,0(R2)     ; Y(i) = a*X(i) + Y(i)
        DADDIU   R1,R1,#8     ; increment X index
        DADDIU   R2,R2,#8     ; increment Y index
        DSLTU    R3,R1,R4     ; test: continue loop?
        BNEZ     R3,foo       ; loop if needed
```

- Assume the functional unit latencies as shown in the table below.
- Assume a one cycle delayed branch that resolves in the ID stage. One cycle delayed branches have 1 stall inserted after them.
- Assume that results are fully bypassed. (The table already shows the latency needed when bypassing is applied)

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	5
FP add	FP ALU op	3
FP multiply	FP store	4
FP add	FP store	3
Integer operations and all loads	Any	1

1.1) Show the unscheduled loop and calculate the number of cycles it needs.
1.2) Show the scheduled loop and calculate number of cycles it needs. How much improvement over the unscheduled code?
1.3) Unroll the loop 3 times without any scheduling and compute the number of cycles needed per 1 iteration. Compare with the rest.
1.4) Unroll the loop 3 times with scheduling and compute the number of cycles needed per 1 iteration. Compare with the rest.

Solution

1.1) Unscheduled code:

```
DADDIU R4, R1, #800
foo: L.D F2,0(R1)
     stall
     MUL.D F4, F2, F0
     L.D F6,0(R2)
     Stall
     Stall
     Stall
     Stall // only 4 stalls not 5, because the L.D after the mult used 1 of the 5
     ADD.D F6, F4, F6
     Stall
     Stall
     Stall
     S.D F6,0(R2)
     DADDIU R1, R1, #8
     DADDIU R2, R2, #8
     DSLTU R3, R1, R4
     Stall // needed because branches are resolved in the decode. R3 not ready yet
     BNEZ R3, foo
     Stall //needed because of the delayed branch in the given.
```

Total: 19 cycles for the loop (we can safely exclude the first DADDIU before the loop starts)

1.2) Scheduled code:

```
DADDIU R4, R1, #800
FOO: L.D F2, 0(R1)
     L.D F6, 0(R2)
     MUL.D F4, F2, F0
     DADDIU R1, R1, #8
     DADDIU R2, R2, #8
     DSLTU R3, R1, R4
     Stall
     Stall  // 2 stalls needed because originally 5 were needed
     between MUL and ADD, but we inserted 3 instructions so only 2 stalls
     are needed now
     ADD.D F6, F4, F6
     Stall
     Stall  // we moved the store to the delayed branch slot but
     still the distance between add.d and s.d should be 3. The branch
     uses 1 of them, so we need 2 additional stalls.
     BNEZ R3, foo
     S.D F6, -8 (R2) // this takes the place of the delayed slot.
     Notice we need to adjust the address by replacing 0 with -8 because
     we incremented R2 prior to the store.
```

Total: 13 cycles. (Again ignoring the first DADDIU) Improvement =

19/13 = 1.46 times faster.

1.3) Unrolled 3 times without scheduling

```
DADDIU R4, R1, #800
foo: L.D F2,0(R1)
      1 stall
       MUL.D F4, F2, F0
       L.D F6,0(R2)
       4 stalls
       ADD.D F6, F4, F6
       3 stalls
       S.D F6,0(R2)
      L.D F2,8(R1)
      1 stall
       MUL.D F4, F2, F0
       L.D F6,8(R2)
       4 stalls
       ADD.D F6, F4, F6
       3 stalls
       S.D F6,8(R2)
      L.D F2,16(R1)
      1 stall
       MUL.D F4, F2, F0
       L.D F6,16(R2)
       4 stalls
       ADD.D F6, F4, F6
       3 stalls
       S.D F6,16(R2)
      DADDIU R1, R1, #24
       DADDIU R2, R2, #24
       DSLTU R3, R1, R4
       1 stall
       BNEZ R3, foo
       1 stall
```

Total: 45 cycles per 3 iterations => avg 15 cycles per iteration, which is (19/15)=1.26 times faster than normal loop, but the scheduled loop is still (15/13)= 1.15 times faster than the unrolled loop without scheduling.

1.4) unroll 3 times with scheduling

```
DADDIU R4, R1,#800
Foo: L.D F2,0(R1)          F2= x[i]
     L.D F6,0(R2)          F6= y[i]
     MUL.D F4,F2,F0      F4= a.x[i]
     L.D F2,8(R1)          F2= x[i+1]
     L.D F10,8(R2)         F10= y[i+1]
     MUL.D F8,F2,F0     F8 = a.x[i+1]
     L.D F2,16(R1)       F2= x[i+2]
     L.D F14,16(R2)      F14= y[i+2]
     MUL.D F12,F2,F0    F12 = a.x[i+2]
     ADD.D F6,F4,F6        F6 = a.x[i] + y[i]
     DADDIU R1,R1,#24      fix address R1 by + 24 (3 iterations)
     ADD.D F10,F8,F10      F10 = a.x[i+1] + y[i+1]
     DADDIU R2,R2,#24      fix address R2 by + 24 (3 iterations)
     DSLTU R3,R1,R4        test loop finishing
     ADD.D F14,F12,F14     F14 = a.x[i+2] + y[i+2]
     S.D F6,-24(R2)        save y[i] to memory
     S.D F10,-16(R2)       save y[i+1] to memory
     BNEZ R3,foo
     S.D F14,-8(R2)        save y[i+2] to memory
```

Zero stalls are needed.
Total number of cycles = 19 cycles per 3 iterations, so on average we need 6.33 cycles per iteration.
This is faster than:

- The normal loop by 3 times.
- The scheduled loop by 2 times.
- The unrolled-unscheduled loop by 2.3 times.

Exercise 2

The loop iterations N might not be divisible by the unrolling factor K (the number of iterations we enroll the loop). Suggest a scheme to organize that.

Solution

For an unrolling factor k, we must iterate Integer [(N/K)] times.
Example: for N = 13 and unrolling factor K =3, we iterate 13/3 = 4 times. (Each iteration is the loop unrolled 3 times, totaling 12 iterations)
The hanging iterations can be computed using [N MOD K] = 13 mod 3 = 1. So we need to execute the loop 1 more iteration.

Exercise 3

Consider the following code:

```
for (i=0; i<100; i=i+1)
{
  A[i+1] = A[i] + C[i];    /* S1 */
  B[i+1] = B[i] + A[i+1];  /* S2 */
}
```

Is there any loop-carried dependency? Can we get rid of it?

Solution

In S1, A[i+1] requires A[i] that is computed in the previous iteration
In S2, B[i+1] also required B[i] from previous iteration

Example:

```
A[1]= A[0]+C[0];
B[1]= B[0]+A[1];

A[2]= A[1]+C[1];
B[2]= B[1]+A[2]];

A[3]= A[2]+C[2];
B[3]= B[2]+A[3]];
```

And so on

It cannot be eliminated in this case since both statements depend on previous iterations.
Look below
A[1]= A[0]+C[0];

Iteration 1:
B[1]= B[0]+A[1];
A[2]= A[1]+C[1];

Iteration 2
B[2]= B[1]+A[2]];
A[3]= A[2]+C[2];

And so on.

Exercise 4

Consider the following code:

```
for(i=0; i<100; i++) {

 A[i] = B[i]++;
 C[i] = A[i]*5;
 B[i] = B[i]/5;
A[i] = D[i];

}
for(i=0; i<100; i++) {
 B[i]=1;
}
```

Mention all the dependencies types found in the code and suggest a renaming scheme to solve/avoid them to the best of your ability.

Solution

For the 1st for loop:

- True data dependence between 1st and 2nd line on A[i].
- Output dependence between 1st and 4th line also on A[i].
- Anti-dependence between 1st and 3rd line on B[i].

- Anti-dependence between 2nd and 4th line on A[i]. There are no dependencies in the 2nd for loop.

New code after renaming.

```
for(i=0; i<100; i++) {

 P[i] = B[i]++; // rename A here to P to remove output dependence
 C[i] = P[i]*5; // rename A to P because there's a true dependence
 Q[i] = B[i]/5; // rename B to Q to remove anti-dependence with 1st line
 A[i] = D[i];  // keep A the same as this is what we want after the loop

}
for(i=0; i<100; i++) {
 Q[i]=1;
// since we changed B to Q, this must be changed in all the following code.
}
```