CS 530 Fall 2014
Computer Architecture

Homework 4 Solutions

1. Problem 4.1: *Show the MIPS and VMIPS code for the Mr. Bayes likelihood table computation.*

<u>MIPS Version</u>

Assume: R2 = seq_length and R3 = h.

```
      1) LI R1, #0                // k = 0
2) Loop:  SLT R0, R1, R2          // R0 = (k < seq_length) ? 1 : 0
      3) BEQZ R0, Exit            // exit loop if k = seq_length
      4) L.S F8, 0(RtiPL)         // load tiPL[0]
      5) L.S F0, 0(RclL)          // load clL[0]
      6) MUL.S F8, F8, F0         // F8 = tiPL[0]*clL[0]
      7) L.S F9, 4(RtiPL)         // load tiPL[1]
      8) L.S F1, 4(R6)            // load clL[1]
      9) MUL.S F9, F9, F1         // F9 = tiPL[1]*clL[1]
      10) ADD.S F8, F8, F9              // F8 = intermediate sum
      11) L.S F9, 8(RtiPL)        // load tiPL[2]
      12) L.S F2, 8(RclL)         // load clL[2]
      13) MUL.S F9, F9, F2        // F9 = tiPL[2]*clL[2]
      14) ADD.S F8, F8, F9        // F8 = intermediate sum
      15) L.S F9, 12(RtiPL)       // load tiPL[3]
      16) L.S F3, 12(RclL)        // load clL[3]
      17) MUL.S F9, F9, F3        // F9 = tiPL[3]*clL[3]
      18) ADD.S F8, F8, F9        // F8 = left sum
      19) L.S F10, 0(RtiPR)       // load tiPR[0]
      20) L.S F4, 0(RclR)         // load clR[0]
      21) MUL.S F10, F10, F4      // F10 = tiPR[0]*clR[0]
      22) L.S F9, 4(RtiPR)        // load tiPR[1]
      23) L.S F5, 4(RclR)         // load clR[1]
      24) MUL.S F9, F9, F5        // F9 = tiPR[1]*clR[1]
      25) ADD.S F10, F10, F9      // F10 = intermediate sum
      26) L.S F9, 8(RtiPR)        // load tiPR[2]
      27) L.S F6, 8(RclR)         // load clR[2]
      28) MUL.S F9, F9, F6        // F9 = tiPR[2]*clR[2]
      29) ADD.S F10, F10, F9      // F10 = intermediate sum
      30) L.S F9, 12(RtiPR)       // load tiPR[3]
      31) L.S F7, 12(RclR)        // load clR[3]
      32) MUL.S F9, F9, F7        // F9 = tiPR[3]*clR[3]
```

```
33) ADD.S F10, F10, F9      // F10 = right sum
34) MUL.S F9, F8, F10       // F9 = (left side)*(right side)
35) ADD R9, RclP, R3        // set offset of clP[h]
36) S.S F9, 0(R9)           // clP[h] = F9
37) ADDI R3, R3, #4         // h++
38) L.S F8, 16(RtiPL)       // load tiPL[4]
39) MUL.S F8, F8, F0        // F8 = tiPL[4]*clL[0]
40) L.S F9, 20(RtiPL)       // load tiPL[5]
41) MUL.S F9, F9, F1        // F9 = tiPL[5]*clL[1]
42) ADD.S F8, F8, F9        // F8 = intermediate sum
43) L.S F9, 24(RtiPL)       // load tiPL[6]
44) MUL.S F9, F9, F2        // F9 = tiPL[6]*clL[2]
45) ADD.S F8, F8, F9        // F8 = intermediate sum
46) L.S F9, 28(RtiPL)       // load tiPL[7]
47) MUL.S F9, F9, F3        // F9 = tiPL[7]*clL[3]
48) ADD.S F8, F8, F9        // F8 = left sum
49) L.S F10, 16(RtiPR)      // load tiPR[4]
50) MUL.S F10, F10, F4      // F10 = tiPR[4]*clR[0]
51) L.S F9, 20(RtiPR)       // load tiPR[5]
52) MUL.S F9, F9, F5        // F9 = tiPR[5]*clR[1]
53) ADD.S F10, F10, F9      // F10 = intermediate sum
54) L.S F9, 24(RtiPR)       // load tiPR[6]
55) MUL.S F9, F9, F6        // F9 = tiPR[6]*clR[2]
56) ADD.S F10, F10, F9      // F10 = intermediate sum
57) L.S F9, 28(RtiPR)       // load tiPR[7]
58) MUL.S F9, F9, F7        // F9 = tiPR[7]*clR[3]
59) ADD.S F10, F10, F9      // F10 = right sum
60) MUL.S F9, F8, F10       // F9 = (left side)*(right side)
61) ADD R9, RclP, R3        // set offset of clP[h]
62) S.S F9, 0(R9)           // clP[h] = F9
63) ADDI R3, R3, #4         // h++
64) L.S F8, 32(RtiPL)       // load tiPL[8]
65) MUL.S F8, F8, F0        // F8 = tiPL[8]*clL[0]
66) L.S F9, 36(RtiPL)       // load tiPL[9]
67) MUL.S F9, F9, F1        // F9 = tiPL[9]*clL[1]
68) ADD.S F8, F8, F9        // F8 = intermediate sum
69) L.S F9, 40(RtiPL)       // load tiPL[10]
70) MUL.S F9, F9, F2        // F9 = tiPL[10]*clL[2]
71) ADD.S F8, F8, F9        // F8 = intermediate sum
72) L.S F9, 44(RtiPL)       // load tiPL[11]
73) MUL.S F9, F9, F3        // F9 = tiPL[11]*clL[3]
74) ADD.S F8, F8, F9        // F8 = left sum
```

```
75) L.S F10, 32(RtiPR)         // load tiPR[8]
76) MUL.S F10, F10, F4         // F10 = tiPR[8]*clR[0]
77) L.S F9, 36(RtiPR)          // load tiPR[9]
78) MUL.S F9, F9, F5           // F9 = tiPR[9]*clR[1]
79) ADD.S F10, F10, F9         // F10 = intermediate sum
80) L.S F9, 40(RtiPR)          // load tiPR[10]
81) MUL.S F9, F9, F6           // F9 = tiPR[10]*clR[2]
82) ADD.S F10, F10, F9         // F10 = intermediate sum
83) L.S F9, 44(RtiPR)          // load tiPR[11]
84) MUL.S F9, F9, F7           // F9 = tiPR[11]*clR[3]
85) ADD.S F10, F10, F9         // F10 = right sum
86) MUL.S F9, F8, F10          // F9 = (left side)*(right side)
87) ADD R9, RclP, R3           // set offset of clP[h]
88) S.S F9, 0(R9)              // clP[h] = F9
89) ADDI R3, R3, #4            // h++
90) L.S F8, 48(RtiPL)          // load tiPL[12]
91) MUL.S F8, F8, F0           // F8 = tiPL[12]*clL[0]
92) L.S F9, 52(RtiPL)          // load tiPL[13]
93) MUL.S F9, F9, F1           // F9 = tiPL[13]*clL[1]
94) ADD.S F8, F8, F9           // F8 = intermediate sum
95) L.S F9, 56(RtiPL)          // load tiPL[14]
96) MUL.S F9, F9, F2           // F9 = tiPL[14]*clL[2]
97) ADD.S F8, F8, F9           // F8 = intermediate sum
98) L.S F9, 60(RtiPL)          // load tiPL[15]
99) MUL.S F9, F9, F3           // F9 = tiPL[15]*clL[3]
100) ADD.S F8, F8, F9          // F8 = left sum
101) L.S F10, 48(RtiPR)        // load tiPR[12]
102) MUL.S F10, F10, F4        // F10 = tiPR[12]*clR[0]
103) L.S F9, 52(RtiPR)         // load tiPR[13]
104) MUL.S F9, F9, F5          // F9 = tiPR[13]*clR[1]
105) ADD.S F10, F10, F9        // F10 = intermediate sum
106) L.S F9, 56(RtiPR)         // load tiPR[14]
107) MUL.S F9, F9, F6          // F9 = tiPR[14]*clR[2]
108) ADD.S F10, F10, F9        // F10 = intermediate sum
109) L.S F9, 60(RtiPR)         // load tiPR[15]
110) MUL.S F9, F9, F7          // F9 = tiPR[15]*clR[3]
111) ADD.S F10, F10, F9        // F10 = right sum
112) MUL.S F9, F8, F10         // F9 = (left side)*(right side)
113) ADD R9, RclP, R3          // set offset of clP[h]
114) S.S F9, 0(R9)             // clP[h] = F9
115) ADDI R3, R3, #4           // h++
116) ADDI RclL, RclL, #16      // clL += 4
```

```
    117) ADDI RclR, RclR, #16     // clR += 4
    118) ADDI RtiPL, RtiPL, #64   // tiPL += 16
    119) ADDI RtiPR, RtiPR, #64   // tiPR += 16
    120) ADDI R1, R1, #1          // k++
    121) J Loop                   // repeat loop
122) Exit:
```

VMIPS Version

Assume: R2 = seq_length and R3 = h.

```
    1) LI R1, #0                  // k = 0
    2) LI VL, #4                  // VL = 4
3) Loop:  SLT R0, R1, R2          // R0 = (k < seq_length) ? 1 : 0
    4) BEQZ R0, Exit              // exit loop if k = seq_length
    5) LV V1, RtiPL               // load tiPL[0] – tiPL[3]
    6) LV V2, RclL                // load clL[0] – clL[3]
    7) MULVV.S V3, V1, V2         // V3[i] = tiPL[i]*clL[i]
    8) SUMR.S F0, V3              // F0 = left sum
    9) LV V4, RtiPR               // load tiPR[0] – tiPR[3]
    10) LV V5, RclR               // load clR[0] – clR[3]
    11) MULVV.S V3, V4, V5        // V3[i] = tiPR[i]*clR[i]
    12) SUMR.S F1, V3             // F1 = right sum
    13) MUL.S F2, F0, F1          // F2 = (left side)*(right side)
    14) ADD R9, RclP, R3          // set offset of clP[h]
    15) S.S F2, 0(R9)             // clP[h] = F2
    16) ADDI R3, R3, #4           // h++
    17) ADDI RtiPL, RtiPL, #16    // RtiPL += 4
    18) ADDI RtiPR, RtiPR, #16    // RtiPR += 4
    19) LV V1, RtiPL              // load tiPL[3] – tiPL[7]
    20) MULVV.S V3, V1, V2        // V3[i] = tiPL[i]*clL[i]
    21) SUMR.S F0, V3             // F0 = left sum
    22) LV V4, RtiPR              // load tiPR[3] – tiPR[7]
    23) MULVV.S V3, V4, V5        // V3[i] = tiPR[i]*clR[i]
    24) SUMR.S F1, V3             // F1 = right sum
    25) MUL.S F2, F0, F1          // F2 = (left side)*(right side)
    26) ADD R9, RclP, R3          // set offset of clP[h]
    27) S.S F2, 0(R9)             // clP[h] = F2
    28) ADDI R3, R3, #4           // h++
    29) ADDI RtiPL, RtiPL, #16    // RtiPL += 4
    30) ADDI RtiPR, RtiPR, #16    // RtiPR += 4
```

```
    31) LV V1, RtiPL                  // load tiPL[8] – tiPL[11]
    32) MULVV.S V3, V1, V2            // V3[i] = tiPL[i]*clL[i]
    33) SUMR.S F0, V3                 // F0 = left sum
    34) LV V4, RtiPR                  // load tiPR[8] – tiPR[11]
    35) MULVV.S V3, V4, V5            // V3[i] = tiPR[i]*clR[i]
    36) SUMR.S F1, V3                 // F1 = right sum
    37) MUL.S F2, F0, F1              // F2 = (left side)*(right side)
    38) ADD R9, RclP, R3              // set offset of clP[h]
    39) S.S F2, 0(R9)                 // clP[h] = F2
    40) ADDI R3, R3, #4               // h++
    41) ADDI RtiPL, RtiPL, #16        // RtiPL += 4
    42) ADDI RtiPR, RtiPR, #16        // RtiPR += 4
    43) LV V1, RtiPL                  // load tiPL[12] – tiPL[15]
    44) MULVV.S V3, V1, V2            // V3[i] = tiPL[i]*clL[i]
    45) SUMR.S F0, V3                 // F0 = left sum
    46) LV V4, RtiPR                  // load tiPR[12] – tiPR[15]
    47) MULVV.S V3, V4, V5            // V3[i] = tiPR[i]*clR[i]
    48) SUMR.S F1, V3                 // F1 = right sum
    49) MUL.S F2, F0, F1              // F2 = (left side)*(right side)
    50) ADD R9, RclP, R3              // set offset of clP[h]
    51) S.S F2, 0(R9)                 // clP[h] = F2
    52) ADDI R3, R3, #4               // h++
    53) ADDI RtiPL, RtiPL, #16        // RtiPL += 4
    54) ADDI RtiPR, RtiPR, #16        // RtiPR += 4
    55) ADDI RclL, RclL, #16          // clL += 4
    56) ADDI RclR, RclR, #16          // clR += 4
    57) ADDI R1, R1, #1               // k++
    58) J Loop                        // repeat loop
59) Exit:
```

2. Problem 4.2: *Assuming seq_length = 500, what is the dynamic instruction count for both implementations?*

<u>MIPS Version</u>

1 instruction for initialization, 120 instructions for single loop, 2 instructions to exit loop

$$Dynamic\ instruction\ count\ =\ 1\ +\ (120)(500)+\ 2 = \mathbf{60,003}$$

<u>VMIPS Version</u>

2 instructions for initialization, 56 instructions for single loop, 2 instructions to exit loop

$$Dynamic\ instruction\ count\ =\ 2\ +\ (56)(500)+\ 2 = \mathbf{28,004}$$

3. Problem 4.9: *Multiplication of vectors with single-precision complex values. Processor runs at 700 MHz and maximum vector length of 64.*

   a. *What is the arithmetic intensity of this kernel? Justify your answer.*

      Each loop has 6 single-precision memory accesses (i.e., 24 bytes total) and 6 FLOPs.

$$Arithmetic\ intensity = \frac{6}{24} = \mathbf{0.25}$$

   b. *Convert this loop into VMIPS assembly code using strip mining.*

      Strip mine remainder of loop: 300 mod 64 = 44

```
    1)  LI VL, #44                 // first 44 operations
    2)  LI R1, #0                  // i = 0
3)  Loop: LV V1, a_re + R1         // load a_re
    4)  LV V3, b_re + R1           // load b_re
    5)  MULVV.S V5, V1, V3         // a_re*b_re
    6)  LV V2, a_im + R1           // load a_im
    7)  LV V4, b_im + R1           // load b_im
    8)  MULVV.S V6, V2, V4         // a_im*b_im
    9)  SUBVV.S V5, V5, V6         // top subtraction
    10) SV V5, c_re + R1           // store c_re
    11) MULVV.S V5, V1, V4         // a_re*b_im
    12) MULVV.S V6, V2, V3         // a_im*b_re
    13) ADDVV.S V5, V5, V6         // bottom addition
    14) SV V5, c_im + R1           // store c_im
    15) BNE R1, #0, Else           // Check if first iteration
    16) ADDI R1, R1, #44           // first iteration, i += 44
    17) J Loop                     // repeat loop
18) Else: ADDI R1, R1, #256        // not first iteration, I += 256
19) Skip: BLT R1, #1200, Loop      // Check if next iteration
```

4. Problem 4.11 (a): *Reduction and scalar expansion. Show how the C code will look for executing the second loop using recurrence doubling.*

```c
for (int i = 64/2; i > 0; i /= 2)
        for (int j = 0; j < i; j ++)
                dot[j] = dot[j] + dot[j + i];
```

5. Problem 4.14: *Analyze loop potential for parallelization.*

   a. *Does the following loop have a loop-carried dependency?*

      Apply the greatest common divisor test to check for loop-carried dependency. In this example, a = 4, b = 5, c =2, and d = 4, then GCD(2, 4) = 2 and d – b = -1. Since 2 does not divide -1, no dependence is possible.

   b. *Find all the true dependences, output dependences, and antidependences. Eliminate the output dependences and antidependences by renaming.*

      <u>True dependences (RAW)</u>

      1. A[i] between S1 and S2
      2. A[i] between S3 and S4

      <u>Output dependences (WAW)</u>

      1. A[i] between S1 and S3

      <u>Antidependences (WAR)</u>

      1. B[i] between S1 and S2
      2. A[i] between S2 and S3
      3. C[i] between S3 and S4

```
for (i = 0; i < 100; i++) {
        A[i] = A[i] * B[i];
        X[i] = A[i] + c;            // B to X, remove antidep. #1
        Y[i] = C[i] * c;            // A to Y, remove output dep. #1 and antidep. #2
        Z[i] = D[i] * Y[i];         // C to Z, remove antidep. #3
}
```

6. Problem 4.16: *Assume a hypothetical GPU with 1.5 GHz, 16 SIMD processors each with 16 SP FPU, and 100 GB/sec off-chip memory bandwidth. Without considering memory bandwidth, what is the peak single-precision floating-point throughput for this GPU in GFLOP/sec? Is this throughput sustainable given the memory bandwidth limitation?*

$$Peak\ FP\ throughput = 1.5\,G\ \times 16 \times 16 = \mathbf{384\ GFLOP/sec}$$

The peak FP throughput performs 384 GFLOP/sec each requiring 2 SP FP values, or equivalently 8 bytes. The bandwidth required to sustain the computation needs to be at least,

$$384\,\frac{GFLOP}{sec} \times 8\ bytes = 3{,}072\ GB/sec$$

Therefore, having only 100 GB/sec the computation becomes memory bound.