Christopher Daffron CS 530 HW #4 11/11/14

4.1) Convert the following C code to MIPS and VMIPS

```
for (k = 0; i < seq_length; k++)
{
	clP[h++] = (tiPL[AA] * clL[A] + tiPL[AC]*clL[C] + tiPL[AG]*clL[G] + tiPL[AT]*clL[T]) *
(tiPR[AA]*clR[A] + tiPR[AC]*clR[C] + tiPR[AG] * clR[G] + tiPR[AT]*clR[T]);
	clP[h++] = (tiPL[CA] * clL[A] + tiPL[CC]*clL[C] + tiPL[CG]*clL[G] + tiPL[CT]*clL[T]) *
(tiPR[CA]*clR[A] + tiPR[CC]*clR[C] + tiPR[CG] * clR[G] + tiPR[CT]*clR[T]);
	clP[h++] = (tiPL[GA] * clL[A] + tiPL[GC]*clL[C] + tiPL[GG]*clL[G] + tiPL[GT]*clL[T]) *
(tiPR[GA]*clR[A] + tiPR[GC]*clR[C] + tiPR[GG] * clR[G] + tiPR[GT]*clR[T]);
	clP[h++] = (tiPL[TA] * clL[A] + tiPL[TC]*clL[C] + tiPL[TG]*clL[G] + tiPL[TT]*clL[T]) *
(tiPR[TA]*clR[A] + tiPR[TC]*clR[C] + tiPR[TG] * clR[G] + tiPR[TT]*clR[T]);

	clL += 4;
	clR += 4;
	tiPL += 16;
	tiPR += 16;
}
```

Assumptions for MIPS code:

- Assuming seq_length is stored in R1

```
		ADDI R2, R0, #0		# Initialize k to be 0
LOOP:		L.S F1, 0(RtiPL)	# Load tiPL[AA] into F1
		L.S F2, 0(RclL)		# Load clL[A] into F2
		MUL.S F1, F1, F2	# Store tiPL[AA] * ClL[A] in F1

		L.S F2, 4(RtiPL)	# Load tiPL[AC] into F2
		L.S F3, 4(RclL)		# Load clL[C] into F3
		MUL.S F2, F2, F3	# Store tiPL[AC] * clL[C] in F2

		L.S F3, 8(RtiPL)	# Load tiPL[AG] into F3
		L.S F4, 8(RclL)		# Load clL[G] into F4
		MUL.S F3, F3, F4	# Store tiPL[AG] * clL[G] in F3

		L.S F4, 12(RtiPL)	# Load tiPL[AT] into F4
		L.S F5, 12(RclL)	# Load clL[T] into F5
		MUL.S F4, F4, F5	# Store tiPL[AT] * clL[T] into F4

		ADD.S F1, F1, F2
		ADD.S F1, F1, F3
		ADD.S F1, F1, F4	# First half of equation

		L.S F2, 0(RtiPR)	# Load tiPR[AA] into F2
		L.S F3, 0(RclR)		# Load clR[A] into F3
		MUL.S F2, F2, F3	# Store tiPR[AA] * clR[A] into F2

		L.S F3, 4(RtiPR)	# Load tiPR[AC] into F3
		L.S F4, 4(RclR)		# Load clR[A] into F4
		MUL.S F3, F3, F4	# Store tiPR[AC] * clR[A] into F3

		L.S F4, 8(RtiPR)	# Load tiPR[AG] into F4
		L.S F5, 8(RclR)		# Load clR[G] into F5
		MUL.S F4, F4, F5	# Store tiPR[AG] * clR[G] into F4

		L.S F5, 12(RtiPR)	# Load tiPR[AT] into F5
		L.S R6, 12(RclR)	# Load clR[T] into F6
```

```
MUL.S F5, F5, F6       # Store tiPR[AT] * clR[T] into F5

ADD.S F2, F2, F3
ADD.S F2, F2, F4
ADD.S F2, F2, F5       # Second half of equation

MUL.S F1, F1, F2       # Multiply halves of equation

S.S F1, 0(RclP)        # Store into clP

ADDI RclP, RclP, #4    # Increment h

# Do the same thing 3 more times
L.S F1, 16(RtiPL)
L.S F2, 0(RclL)
MUL.S F1, F1, F2

L.S F2, 20(RtiPL)
L.S F3, 4(RclL)
MUL.S F2, F2, F3

L.S F3, 24(RtiPL)
L.S F4, 8(RclL)
MUL.S F3, F3, F4

L.S F4, 28(RtiPL)
L.S F5, 12(RclL)
MUL.S F4, F4, F5

ADD.S F1, F1, F2
ADD.S F1, F1, F3
ADD.S F1, F1, F4

L.S F2, 16(RtiPR)
L.S F3, 0(RclR)
MUL.S F2, F2, F3

L.S F3, 20(RtiPR)
L.S F4, 4(RclR)
MUL.S F3, F3, F4

L.S F4, 24(RtiPR)
L.S F5, 8(RclR)
MUL.S F4, F4, F5

L.S F5, 28(RtiPR)
L.S F6, 12(RclR)
MUL.S F5, F5, F6

ADD.S F2, F2, F3
ADD.S F2, F2, F4
ADD.S F2, F2, F5

MUL.S F1, F1, F2

S.S F1, 0(RclP)

ADDI RclP, RclP, #4

L.S F1, 32(RtiPL)
L.S F2, 0(RclL)
MUL.S F1, F1, F2
```

```
L.S F2, 36(RtiPL)
L.S F3, 4(RclL)
MUL.S F2, F2, F3

L.S F3, 40(RtiPL)
L.S F4, 8(RclL)
MUL.S F3, F3, F4

L.S F4, 44(RtiPL)
L.S F5, 12(RclL)
MUL.S F4, F4, F5

ADD.S F1, F1, F2
ADD.S F1, F1, F3
ADD.S F1, F1, F4

L.S F2, 32(RtiPR)
L.S F3, 0(RclR)
MUL.S F2, F2, F3

L.S F3, 36(RtiPR)
L.S F4, 4(RclR)
MUL.S F3, F3, F4

L.S F4, 40(RtiPR)
L.S F5, 8(RclR)
MUL.S F4, F4, F5

L.S F5, 44(RtiPR)
L.S F6, 12(RclR)
MUL.S F5, F5, F6

ADD.S F2, F2, F3
ADD.S F2, F2, F4
ADD.S F2, F2, F5

MUL.S F1, F1, F2

S.S F1, 0(RclP)

ADDI RclP, RclP, #4

L.S F1, 48(RtiPL)
L.S F2, 0(RclL)
MUL.S F1, F1, F2

L.S F2, 52(RtiPL)
L.S F3, 4(RclL)
MUL.S F2, F2, F3

L.S F3, 56(RtiPL)
L.S F4, 8(RclL)
MUL.S F3, F3, F4

L.S F4, 60(RtiPL)
L.S F5, 12(RclL)
MUL.S F4, F4, F5

ADD.S F1, F1, F2
ADD.S F1, F1, F3
ADD.S F1, F1, F4
```

```
                L.S F2, 48(RtiPR)
                L.S F3, 0(RclR)
                MUL.S F2, F2, F3

                L.S F3, 52(RtiPR)
                L.S F4, 4(RclR)
                MUL.S F3, F3, F4

                L.S F4, 56(RtiPR)
                L.S F5, 8(RclR)
                MUL.S F4, F4, F5

                L.S F5, 60(RtiPR)
                L.S F6, 12(RclR)
                MUL.S F5, F6, F6

                ADD.S F2, F2, F3
                ADD.S F2, F2, F4
                ADD.S F2, F2, F5

                MUL.S F1, F1, F2

                S.S F1, 0(RclP)

                ADDI RclP, RclP, 4

                # Increment clL, clR, tiPL, tiPR, and k
                ADDI RclL, RclL, #16
                ADDI RclR, RclR, #16
                ADDI RtiPL, RtiPL, #64
                ADDI RtiPR, RtiPR, #64
                ADDI R2, R2, #1
                BNE R2, R1, LOOP # Check if done
```

Assumptions for VMIPS:

- seq_length is in R1

```
                ADDI VL, R0, #4         # Set the vector length to 4
                ADDI R2, R0, #0         # Initialize k to be 0
LOOP:           LV V1, 0(RtiPL)         # Load 4 elements of tiPL
                LV V2, 0(RclL)          # Load 4 elements of clL
                MULVV.S V1, V1, V2      # Multiply tiPL and clL
                SUMR.S F1, V1           # Summation

                LV V2, 0(RtiPR)         # Load 4 elements of tiPR
                LV V3, 0(RclR)          # Load 4 elements of clR
                MULVV.S V2, V2, V3      # Multiply tiPR and clR
                SUMR.S F2, V2           # Summation

                MUL.S F1, F1, F2        # Multiply summations

                S.S F1, 0(RclP)         # Store in clP

                ADDI RclP, RclP, #4     # Increment clP
                ADDI RtiPL, RtiPL, #16 # Increment tiPL by 4
                ADDI RtiPR, RtiPR, #16 # Increment tiPR by 4

                # Do the same thing 3 more times
                LV V1, 0(RtiPL)
                LV V2, 0(RclL)
```

```
MULVV.S V1, V1, V2
SUMR.S F1, V1

LV V2, 0(RtiPR)
LV V3, 0(RclR)
MULVV.S V2, V2, V3
SUMR.S F2, V2

MUL.S F1, F1, F2

S.S F1, 0(RclP)

ADDI RclP, RclP, #4
ADDI RtiPL, RtiPL, #16
ADDI RtiPR, RtiPR, #16

LV V1, 0(RtiPL)
LV V2, 0(RclL)
MULVV.S V1, V1, V2
SUMR.S F1, V1

LV V2, 0(RtiPR)
LV V3, 0(RclR)
MULVV.S V2, V2, V3
SUMR.S F2, V2

MUL.S F1, F1, F2

S.S F1, 0(RclP)

ADDI RclP, RclP, #4
ADDI RtiPL, RtiPL, #16
ADDI RtiPR, RtiPR, #16

LV V1, 0(RtiPL)
LV V2, 0(RclL)
MULVV.S V1, V1, V2
SUMR.S F1, V1

LV V2, 0(RtiPR)
LV V3, 0(RclR)
MULVV.S V2, V2, V3
SUMR.S F2, V2

MUL.S F1, F1, F2

S.S F1, 0(RclP)

ADDI RclP, RclP, #4

# Increment tiPL, tiPR, clL, clR
ADDI RtiPL, RtiPL, #16
ADDI RtiPR, RtiPR, #16
ADDI RclL, RclL, #16
ADDI RclR, RclR, #16

# Increment k and decide whether to branch
ADDI R2, R2, #1
BNE R2, R1, LOOP
```

4.2) Assuming seq_length = 500, what is the instruction count?
MIPS: 128 instructions per loop iteration + 1 initialization instructions

$$inst_{total} = 500(138) + 1 = 69{,}001$$

VMIPS: 56 instructions per loop iteration + 2 initialization instructions

$$inst_{total} = 500(56) + 2 = 28{,}002$$

4.9) C code:

```
for(i = 0; i < 300; i++)
{
        c_re[i] = a_re[i] * b_re[i] - a_im[i] * b_im[i]
        c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i]
}
```

Part a)

Per iteration counts:

- 6 FLOPS
- 6 MEM OPERATIONS
- 24 MEM BYTES

So the intensity is 6 FLOPS / 24 B = 0.25. This assumes that a_re, b_im, a_im, and b_re are already memory for the second statement due to the first statement.

Part b) Assumptions:

- a_re, b_re, a_im, and b_im are arrays in memory
- N is stored in R1
- The starting address of a_re is in R2
- The starting address of b_re is in R3
- The starting address of a_im is in R4
- The starting address of b_im is in R5
- The starting address of c_re is in R6
- The starting address of c_im is in R7
- All address registers can be modified freely

```
        ANDI R8, R1, 63        # N mod 64
        MTC1 VLR, R8
LOOP:   LV V1, R2              # load elements of a_re
        LV V2, R3              # load elements of b_re
        LV V3, R4              # load elements of a_im
        LV V4, R5              # load elements of b_im
        MULVV.S V5, V1, V2     # V5 = a_re * b_re
        MULVV.S V6, V3, V4     # V6 = a_im * b_im
        SUBVV.S V5, V5, V6     # V5 = a_re * b_re - a_im * b_im
        SV R6, V5              # store c_re

        MULVV.S V5, V1, V4     # V5 = a_re * b_im
        MULVV.S V6, V3, V2     # V6 = a_im * b_re
        ADDVV.S V5, V5, V6     # V5 = a_re * b_im + a_im * b_re
        SV R7, V5              # store c_im
```

```
        ADDI R9, R0, 4                  # R9 = 4
        MUL R9, R9, R8                  # R9 = Length * 4 (bytes to advance pointer)
        ADD R2, R2, R9                  # advance a_re
        ADD R3, R3, R9                  # advance b_re
        ADD R4, R4, R9                  # advance a_im
        ADD R5, R5, R9                  # advance b_im
        ADD R6, R6, R9                  # advance c_re
        ADD R7, R7, R9                  # advance c_im
        SUB R1, R1, R8                  # decrease N by vector length

        LI R8, #64                      # set R8 to 64
        MTC1 VLR, R8
        BGTZ R1, LOOP
```

4.11a)

```
int vect_length = 64;
while( vect_length != 1 )
{
        vect_length = vect_length / 2;
        for(i = 0; i < vect_length; i++)
        {
                dot[i] = dot[i] + dot[i + vect_length];
        }
}
```

4.14a) Does the following loop have a loop-carried dependency?

```
for(i = 0; i < 100; i++)
{
        A[i] = B[2*i+4];
        B[4*i+5] = A[i];
}
```

The statement does not have a loop-carried dependency because the assigned values of B are always odd indices and the values of B used in the first statement are always even indices. The loop also passes the GCD test since GCD(2,4) is 2 and 2 does not divide 1.

4.14b) Loop code:

```
for(i = 0; i < 100; i++)
{
        A[i] = A[i] * B[i]; /* S1 */
        B[i] = A[i] + c; /* S2 */
        A[i] = C[i] * c; /* S3 */
        C[i] = D[i] * A[i]; /* S4 */
}
```

True Dependencies:

- S1 to S2: A[i]
- S3 to S4: A[i]

Antidependencies:

- S1 to S2: B[i]
- S1 to S3: A[i]
- S2 to S3: A[i]

- S3 to S4: C[i]

Output dependencies:

- S1 to S3: A[i]

Removed code via renaming:

```
for(i = 0; i < 100; i++)
{
        Z[i] = Z[i] * B[i];
        B1[i] = Z[i] + c;
        C1[i] = C[i]
        A[i] = C1[i] * c;
        C[i] = D[i] * A[i];
}
```

4.16) The peak single precision floating point throughput for this GPU is 384 GFLOP/sec as shown below:

$$(1.5\ GHz)(16\ processors)\left(16\ \frac{SP\ FPU}{processor}\right) = 384\ GFLOP/sec$$

The peak throughput is shown below.

$$\left(384\frac{GFLOP}{sec}\right)\left(4\frac{bytes}{SP\ FP}\right)(3\ operands) = 4608\ {}^{GB}/_{S} = 4.6\ {}^{TB}/_{S}$$