Vectorization TD

For the following questions the x86_64 ISA (with SSE and AVX extensions) will be used, using the AT&T syntax (destination operand at last position).

Legacy instructions useful for loop control are:
- `ADD %r1,$n`: r1 = r1 + n
- `CMP %r1,%r2`: compare r1 with r2 (and set flags accordingly)
- `JNE <label>`: branch to <label> if r1 != r2 (Jump if Not Equal)
- `JLE <label>`: branch to <label> if r1 <= r2 (Jump if Less or Equal)
- `JL  <label>`: branch to <label> if r1 < r2 (Jump if Less)

Examples of legacy registers: RAX, RBX, RCX, RDX, RDI and RSI (total of 15 registers + RIP containing Instruction Pointer).

SSE instructions:
- For FP operations opcode is concatenation of:
  * operation (ADD, SUB, MUL, MOV...)
  * S/P: S for scalar (only one element processed) or P for packed (all elements processed)
  * S/D: S for single precision or D for double precision
- `MOV* (mem),%reg` will load a single element to   reg
- `MOV* %reg,(mem)` will store a single element from reg
- `ADD* %r1,%r2`: r2 = r1 + r2 (idem for SUB, MUL, DIV...)
- For packed MOV instructions no MOVPS/PD but MOVAPS/D and MOVUPS/D (A : Aligned and U : Unaligned)
- Examples: ADDPS (Packed addition on single precision FP elements), MOVSD (load/store of a single double precision FP element)

SSE registers are XMM0-15 (128 bits).

Memory operands typically follow the (base,index,scale) structure => element(s) will be read/write from/to byte (base + index x scale).
An offset can be added offset(...) => offset + (...)

Question 1

Write the scalar SSE code for the loop in the following C function:

```
void vadd (int n, float a[n], float b[n]) {
  int i;
  for (i=0; i<n; i++)
    a[i] = a[i] + b[i];
}
```

We will follow here the System V AMD64 ABI:
n in RDI, a in RSI, b in RDX

Question 2

Can this code be vectorized (is it legal to do so) ?

Question 3

Vectorize the loop defined in question 1 using SSE instructions assuming no memory alignment. To simplify exercise, it is assumed that n%4 == 0 (n is a multiple of 4) and n >= 4.
What do you think about efficiency of this code compared to scalar one ? For the CPU point of view, what is changing ?
What about the trip count of this new loop ? Its cost (cycles and instructions for one iteration) ?

Question 4

If alignment is assumed (with a pragma, a directive or a compiler flag), what changes have to be applied to previous code ?

Question 5

What if alignment not assumed and n%4 != 0 ? Which modifications do you need to apply to your assembly code ? You are not asked to write exact assembly code but just how to adapt it.

How to improve your solution knowing MOVAPS is faster than MOVUPS on aligned references which is itself faster than MOVUPS on unaligned references ?

Question 6

Most processors used today support the AVX instructions set. Search on the web in which extent it could make faster the vadd loop (compared to SSE).
Adapt the code written for question 3 to use AVX instructions and registers.

Question 7

For our instruction mix let us assume the following cost model taking into account pipeline filling in a superscalar processor:

- 1 iteration => 4.5x penalty
- 2 => 2.5x
- 3 => 1.9x
- 4 => 1.6x
- 5 => 1.4x
- 6 => 1.3x
- 7 => 1.2x
- 8 => 1.15x
- 9 => 1.10x
- 10+ => 1x (no penalty)

Each iteration of a loop (at assembly/binary level) has a normalized cost of 1.
For scalar (Q1) vs SSE (Q3) vs AVX (Q6) what is the total cost for n=3, 7, 8, 15 and 1000 ?
Give the related SSE/AVX speedup (compared to scalar).

Question 8

Discuss about vectorization profitability (scalar vs SSE vs AVX): in which cases (if any) a vectorized loop could be slower than its scalar version (or AVX slower than SSE) ?
Hints:

- think to 2D kernels (1D loop of 900 iterations becomes a 30x30 2D loop)
- CF question 5

Question 9

Let us now consider complex instead of float elements:

```
typedef struct {
  float re;
  float im;
} complex_t;

void vadd (int n, complex_t a[n], complex_t b[n]) {
  int i;
  for (i=0; i<n; i++) {
    a[i].re = a[i].re + b[i].re;
    a[i].im = a[i].im + b[i].im;
  }
}
```

What do you think about vectorization of this loop ? Legal, easy, efficient ?...

Question 10

Let us now consider a more complex example:

```
typedef struct {
  float x;
  double y;
  float z;
  double t;
} mystruct_t;

void vadd (int n, mystruct_t a[n], mystruct_t b[n]) {
  int i;
  for (i=0; i<n; i++) {
    a[i].x = a[i].x + b[i].x;
    a[i].y = a[i].y + b[i].y;
    a[i].z = a[i].z + b[i].z;
    a[i].t = a[i].t + b[i].t;
  }
}
```

What do you think about vectorization of this loop ? Legal, efficient ?... What is the main difficulty here and how to overcome it ?

Question 11

The following loop nest is not vectorized by some compilers:

```
void vadd (int n, float a[n][n], double b[n]) {
  int i, j;
  for (j=0; j<n; j++)
    for (i=0; i<n; i++)
      a[i][j] = a[i][j] + b[i];
}
```

Why (what makes it difficult/inefficient to vectorize) ? Is it efficiently vectorizable ? How could you help your compiler to vectorize ?