EXTENDED MODEL ANSWER

Comparative Architectures 2001 Paper 8 Question 10 (IAP)

Simple non-unrolled version for an ARM-like CPU (without fancy addressing modes):

```
; basic plan is:
; do 32bit add of four packed values
; see if there's been a carry over into the lsb of the next most sig byte
; if so, undo the carry and write 0xff into the byte.
; we optimise for the case where overflows are rare

Rn is the size of the array in bytes

Ra is ptr to src array1
Rb is ptr to src array2
Rd is ptr to dst array

load constant #0x01010100 -> Rc ; probably won't fit in immediate
sub.s Rn, #4 -> Rn              ; set flags
blt  finish_up                  ; branch if less than zero

top:
        ld [Ra] -> RA
        ld [Rb] -> RB

        add.s RA, RB -> Rs      ; set flags
        bcs fix_up              ; branch if carry set (carry in top byte)


        xor RA, RB -> Rt        ; compute bottom bits with xor
        xor Rs, Rt -> Ru        ; compare with the add

        and.s Ru, Rc -> Rv      ; set flags
                                ; look at just the lsb of each byte
                                ; is it as we expect. Otherwise, carry
                                ; between lower bytes must have occurred.

        bne fix_up_lower

        ; if we get here no overflow occurred

back:
        St [Rd] <- Rs           ; write to dest array
```

```
        add Ra, #4 -> Ra
        add Rb, #4 -> Rb
        add Rd, #4 -> Rd
        sub Rn, #4 -> Rn        ; book keeping

        bge top                 ; branch if >= 0

fix_up:
        xor RA, RB -> Rt
        xor Rs, Rt -> Ru

        and.s Ru, Rc -> Rv

        xor Rv, Rs -> Rs        ; correct the bits that were wrong

        or Rs, #0xff000000 -> Rs; may not fit in immediate (will on ARM)
                                ; clip to 255

        b fix_up_lower_a


fix_up_lower:

        xor Rv, Rs -> Rs        ; correct the bits that were wrong

fix_up_lower_a:

        and.s Ru, #0x01000000
        beq fix_up_lower2

        or Rs, #0x00ff0000 -> Rs ; clip to 255

fix_up_lower2:

        and.s Ru, #0x00010000
        beq fix_up_lower3

        or Rs, #0x0000ff00 -> Rs ; clip

fix_up_lower3:

        and.s Ru, #0x00000100
        beq back

        or Rs, #0x000000ff -> Rs ; clip

        b back
```

```
finish_up:
        ; deal with the <4 byte case
        ...
```

estimate number of cycles

Annotating the instruction inter-dependencies after a little instruction scheduling:

```
----
        ld [Ra] -> RA
        ld [Rb] -> RB

---- ; load to use delay may be a problem here. ignore.


        add Ra, #4 -> Ra
        add.s RA, RB -> Rs      ; set flags

----
        bcs fix_up              ; branch if carry set (carry in top byte)
        xor RA, RB -> Rt        ; compute bottom bits with xor
----
        xor Rs, Rt -> Ru        ; compare with the add
        add Rd, #4 -> Rd
----
        add Rb, #4 -> Rb
        and.s Ru, Rc -> Rv      ; set flags
                                ; look at just the lsb of each byte
                                ; is it as we expect? Otherwise, carry
                                ; between lower bytes must have occurred?
----

        bne fix_up_lower
        ; otherwise no overflow occurred
back:
        sub.s Rn, #4 -> Rn


----
        St [Rd-#4] <- Rs

        bge top                 ; branch if >= 0
----
```

Without unrolling the loop, we find that each iteration will take around 7 cycles (assuming no overflows occur, all branches are correctly predicted, all data accesses hit in the L1 cache with no load-to-use delay).

Thus: number of cycles = $N * 7/4$

with SIMD instructions

Assume we have a SIMD instruction that performs saturating addition on four 8 bit unsigned values packed into a register.

```
----
        ld [Ra] -> RA
        ld [Rb] -> RB

---- ; load to use delay may be a problem here. ignore.

        add Ra, #4 -> Ra
        add Rb, #4 -> Rb
----
        add Rd, #4 -> Rd
        add_8u_sat RA, RB -> Rs
----
        St [Rd] <- Rs
        sub.s Rn, #4 -> Rn
----
        bge top                 ; branch if >= 0
        ; unused slot :-(
----
```

Thus: number of cycles = $N * 5/4$

Ignoring the possible benefits of loop unrolling (which would be substantial), this inner loop will be 7/5 times faster than the old.

Speedup = Old run time / new run time

= (0.5 + 0.5) / (0.5 + (0.5*5/7))

= 1.17

17% faster using SIMD extentions.