MVI Code Examples

Assumptions:

- NT macro-code
- Incoming argument registers are a0, a1,,,answer into reg v0

MIN-MAX

To do an unsigned saturating subtract on 8 packed bytes; a0 - a1

```
minub8  a0,    a1,    a1     # get minimum at each pos.
subq    a0,    a1,    v0     # no borrow's
```

The same idea would apply to 4 packed words, using minuw4. And, indeed, thru the MIN-MAX examples, what we do with bytes can be equivalently done with words.

Note, the MVI instructions have a latency of 2, so that the above sequence stalls. For better performance, the two instructions would want to be separated by other work, so as to not stall. This applies to the other sequences shown, as well.

Note, the MVI instructions issue from integer pipe 1. Thus, they cannot co-issue with stores, shifts, extracts, other MVI's. PCA56 can dual-issue integer instructions. For performance code, you want to dual-issue wherever possible.

To do an unsigned saturating add on 8 packed bytes: a0 + a1

```
eqv          a1,    zero,        v0     # complement a1
minub8       a0,    v0,          a0     # don't add too much!
addq         a0,    a1,          v0     # no carry's
```

At the completion of arithmetic operations, you may want to clamp the data at a high or low value other than 255 or 0.

To clamp a0, to a maximum of 251:

```
minub8       a0,    a1,          v0     # a1 has 8 bytes of value 251!
```

Yes, I said maximum, and issued a minub8. The minub8 keeps the minimum of what it sees.

To clamp a0, to a minimum of 4:

```
maxub8       a0,    a1,          v0     # a1 has 8 bytes of value 4!
```

PACK-UNPACK

These instructions are primarily to move data between the representations of 8-in-a-register, and 4-in-a-register. The data are stored 8-in-a-register. Some computations requiring more precision are carried out 4-in-a-register. The answers are clamped and then returned to 8-in-a-register format.

Incoming unsigned (low-endian) bytes HGFEDCBA in v0

```
unpkbw      v0,    a0            # a0 gets 0D0C0B0A
srl         v0,    32,    v0     # shift down others
                                  # the 0's are really 0's!
unpkbw      v0,    v0            # v0 gets 0H0G0F0E
xx                                # do whatever

minuw4      v0,    a1     v0     # a1, literal for clamp
minuw4      a0,    a1,    a0     # has 255 in each word

pkbw        a0,    a0            # 0000DCBA
pkbw        v0,    v0            # 0000HGFE

sll   v0,    32,    v0     # HGFE0000
bis   v0,    a0,    v0     # HGFEDCBA
```

As before, this code has significant stalls. There are, of course, other usages available for the creative programmer. For instance, in video, you may receive (little endian) VYUYVYUY in a quadword from an incoming DMA. You want to separate this into separate buffers of Y, U, V.

```
pkwb        v0,    a0            # 4 y's in a0
stl         a0,    (a1)          # put 4 y's into buffer

srl         v0,    8,     v0     # target u's and v's
lda         a1,    4(a1)         # move pointer

pkwb        v0,    v0            # now have VUVU
pkwb        v0,    a0            # now have UU

srl         v0,    8,     v0     # VUV
stw         a0     (a2)          # put 2 u's in buffer

pkwb        v0,    v0            # now have VV
lda         a2,    2(a2)         # move pointer

stw   v0,    (a3)          # put 2 v's in buffer
lda   a3,    2(a3)         # move pointer
```

As before, this code has stalls. More importantly, it is inefficient to talk to the cache system in such small fits and starts. If performance is required, the code should be unrolled so that a whole 32 bytes is written out consecutively (ideally on a d_Cache boundary!) to one stream, and the loads should be prefetch covered.

PERR

PERR is a specialized instruction for pattern match/search operations.

If you

```
perr  a0,    a1,    v0
```

on incoming unsigned data, the corresponding bytes in a0, and a1 are subtracted. Then each of these 8 intermediate answers has its absolute value taken. Then these 8 intermediate answers are summed,

the result being right justified in v0.

If your pattern match/search operation is simply the sum of absolute differences, your code is composed of perr's and addq's (or subq's or addl's or subl's...). There have to be supporting loads, and a final compare. This leads to very fast search code.

If your pattern match/search involves a more complex evaluation, you can still utilize that power of perr! Consider that the primary task of search is to throw away losing candidates. So, prior to a final evaluation loop, a fast perr-based loop can be run to reject the vast majority of candidates. The more detailed, slower, code is then run on a final short list of candidates.

Do an 8x8 compare.

a0 and a1 are the pointers (to quadword-aligned data)
a2 is the counter, containing 8
v0 has the reference value, from which we subtract this 8x8

```
Loop:
        ldq     a0,     (a4)            # fetch two eights
        ldq     a1,     (a5)

        perr    a0,     a1,     a0      # sum 8 abs differences
        lda     a4,     8(a4)           # move pointers

        lda     a5,     8(a5)
        subq    a2,     1,      a2      # count loop

        subq    v0,     a0,     v0      # subtract diff from ref
        bgt     a2,     loop
```

This loop intends to show the instruction usage. Its performance is mediocre. There is no latency after the loads before the perr. The loop isn't unrolled; of the 8 issues, only 2, the perr and the subq, are doing work. Yes, some loads are unavoidable, but a reference pattern could reside in the registers for some amount of time.

If data was coming from off-chip, prefetches are going to help the data bandwidth.