POUGHKEEPSIE
Dept. 271 - Bldg. 946
Extension 221M

July 28, 1961

Memorandum to: Dr. S. G. Campbell

Subject: Progressive Indexing

The following is a discussion of a new type of progressive indexing scheme. It is believed to be an improvement over the present 7030 scheme in that (1) it is more consistent with usual programming practices and is thus easier to master by new programmers, (2) it is simplier to implement by hardware, (3) it is potentially faster, and (4) it does not tend to slow down the I-box processing of other full word instructions.

Extension of the new scheme beyond the present framework is also discussed.

1. 7030 Progressive Indexing. One of the unique powerful 7030 features, progressive indexing provides for continuous updating of index registers as they are being used in the VFL instructions. The debugging of one-shot VFL programs is vastly simplified by its use, and in the hands of experienced programmers, it can also help in the creation of optimum programs.

   However, as implemented on the 7030, progressive indexing entails the following disadvantages.

   a) The effective address generation scheme deviates from standard practice. In progressive indexing the index value field is used as the effective address, the numeric address field is used to modify the index value field in anticipation of future needs. This action differs from the standard indexing practice where the effective address is obtained by addition between the address field and the index value field. The coexistence in the machine of more than one scheme

leads to bewilderment and confusion among new 7030 programmers, to the extent that many of them strive to avoid progressive indexing and rely solely on regular indexing techniques.

A minor inconvenience is that the index updating is always ahead of schedule. When the schedule, so to speak, is changed by an unexpected branch, the index register modification frequently has to be undone by programming.

b) The present scheme requires additional hardware and special hardware sequencing mechanisms. The leading half-word of a full word instruction contains no information on how the specified index register is to interact with the numeric address field in the effective address creation. As a result, the second half-word has to be decoded first.

During the Y-Z transfer within the I-box, either unusual action is taken to fetch, index and decode the second half-word first ("full-word straight" case), or the first half-word and the specified index register contents are fetched first, but nothing can be done unless and until the second half-word has arrived in the Y' register. In this second ("full-word not straight") case, the P field of the second half-word is decoded to enable the indexing of the first half-word to proceed.

If progressive indexing is specified, action is taken to ensure the address of the first half-word remains or is regenerated in Y, while the index value field is placed in ZL as the effective address. The purpose is to allow the execution of a pseudo V ± I(CR) instruction after the second half-word has been indexed, decoded and lookahead-loaded. During the V ± I(CR) pseudo instruction execution, the pertinent index register is fetched, although it has been fetched and used once before.

The unusual sequencing and the need to provide for alternatives in information routing naturally entails a hardware cost and maintenance cost.

c) The present scheme is not fast. I-box time is lost in the "full-word not straight" case if the arrival of the second half-word is delayed by memory traffic jams and/or logical interlocks.

Further, because the first half-word remains in Y until the second half-word is used up, the buffering in Y and Y' loses its effectiveness, affecting the I-box processing of future instructions.

Still another delay is due to the execution of the pseudo V + I(CR) instruction which has to duplicate some of the earlier index actions, such as the redundant index refetch. The pseudo instruction also loads an index-recovery level into the lookahead, influencing look-ahead and E-box timing. This last item, however, cannot be helped without a complete re-examination of the recovery problem.

On direct measurement on the LASL 7030, the excess time due to progressive indexing varies between 0 and 3 us, as seen in the following tables for external operands with left half indexed: (time in us.)

No. Bytes	+	+(PX)	Diff	%	ST	ST(PX)	Diff	%
1	4.82	7.52	2.70	56	6.92	9.33	2.41	35
2	4.96	7.52	2.56	51	7.23	9.33	2.10	29
4	6.16	7.52	1.36	22	7.84	9.63	1.79	23
6	7.36	7.52	0.16	2	9.03	9.64	0.61	7
8	8.56	8.56	0	0	9.62	10.22	0.60	6
16	13.50	13.49	(-0.01)	0	14.41	14.99	0.58	4

It should be noted that the higher excess times occur for the critical areas of small number of bytes, and that known techniques of improving VFL sequencing in I-box and LA will make the percentage excess much more critical.

d) The present scheme tends to penalize other full-word instructions. Because of the possibility of occurrence of progressive indexing, all "full word not straight" cases of full word instructions cannot proceed until the second half-word is available in Y. This places a penalty on standard VFL instructions, I/O instructions, SIC-branches and branch on bit instructions. This waiting delay does not seem to be present for "full word straight" situations. More indirect time losses due to increased logical complexity is hard to pin down.

2. A New Progressive Indexing Scheme. Much can be done to improve the progressive indexing sequencing within the I-box to save time and hardware. But sweeping simplification can be made by always using the sum between the index value field and the address field as the effective address. This is, of course, the standard indexing technique. The new progressive indexing scheme differs from regular indexing essentially only in that the effective address replaces the index value field in question.

It might be said that this cannot possibly handle the (V - I), (V - IC) and (V - ICR) options, available in 7030 progressive indexing. The answer is simple: use the positive counterparts with complement addresses to produce the same effect. Also by adopting the convention that a zero refill field means "do not refill", the (V+IC) option can again be dropped, its function being adequately covered by the (V+ICR) option. As a result, only 2 bits are needed to specify the four truly useful secondary operations: regular indexing, immediate addressing, (V+I), and (V+ICR).

A comparison with the present progressive indexing scheme shows that the new scheme can do everything the present scheme can do, though in a slightly altered manner. The present scheme uses the index value field as a pointer for operand address, then advances the pointer in anticipation for the next operation. The new scheme uses the index value field as a pointer for the previous task, and advance only when necessary. In a certain sense, the index value field is always behind, and the name post-gressive indexing seems adequate.

Clearly post-gressive indexing is quite consistent with conventional indexing practices, and programming education will be greatly enhanced by its adoption. By staying behind rather than anticipating the next operation, the new scheme avoids some of the slight inconveniences (such as undoing the anticipation because of a conditional branch) and is even easier to code and debug.

Much of the unusual I-box sequencing of full-word instructions can now be removed, or replaced by much simpler ones due to the removal of unneeded alternative information routes.

There is no longer need to process the second half-word ahead of schedule. It is expected that the new scheme will lead to large savings in hardware and servicing cost.

The time delay in the "full-word not straight" case will be removed, for the treatment of the leading half of a full word instruction will now be the same, independent of the second half. There is probably no longer need to "save" the left half-word in the Y register. The execution of the pseudo V+I(CR) instruction can, however, be done ahead of the operations dictated by the second half-word, freeing the Y register at a much earlier time. Even the pseudo instruction itself can be speeded up, as the V+I part is already performed during effective address creation, and the redundant index fetch is probably no longer required.

The penalty on other full-word instructions will be reduced to zero, as the effective address creation will be standard.

3. A Simple Example. Suppose one wants to add 301 binary unsigned 8-bit fields with lead bits 100 bit positions apart. If in $5 the value field contains the address of the leading bit of the left-most field, the count field contains 100, and the refill field contains zero, the two types of progressive indexing can be contrasted below:

a) Progressive Indexing

```
      L(BU, 8)(V+I), 100($5)
PX    +(BU, 8)(V+I), 100($5)
      +(BU, 8)(V+I), 100($5)
      +(BU, 8)(V+I), 100($5)
      CB, $5, PX
```

b) Post-gressive Indexing

```
      L(BU, 8), 0($5)
PGX   +(BU, 8)(V+I), 100($5)
      +(BU, 8)(V+I), 100($5)
      +(BU, 8)(V+I), 100($5)
      CB, $5, PGX
```

c) Progressive Indexing, faster version

```
      L(BU, 8)(V+I), 100($5)
PXF   +(BU, 8)(V+IC), 300($5)
      +(BU, 8), -200($5)
      +(BU, 8), -100($5)
      BZXCZ, PXF
```

d) Post-gressive Indexing, faster version

```
      L(BU, 8), 0($5)
PGXF  +(BU, 8), 100($5)
      +(BU, 8), 200($5)
      +(BU, 8)(V+ICR), 300($5)
      BZXCZ, PGXF
```

It is seen that the post-gressive indexing code is more natural and is certainly no less efficient.

4. Extensions. It is natural to inquire whether one can avail other instructions with the powerful progressive indexing feature. In the 7030 when three bits are needed to specify it, there is no room in half-word instructions. The slower speed is also an impediment.

With post-gressive indexing, the operations are expected to be faster, particularly if the I-box sequencing is re-examined, and if somehow index recovery can be made fast, or non-existent. The fact that only 2 bits are really needed instead of 3 will also made a difference.

It is even possible to let the index registers to carry the secondary operation bits. There are already two unused bits in the index word, and the installment of this feature leads to no logical difficulties.

If optimum flexibility is demanded, it is possible to install one extra bit per instruction which means "carry out" (1) or "ignore" (0) the post-gressive secondary operation specified in the index value field.

The impact of full post-gressive indexability on most instructions in a machine is likely to be profound. For I-box instructions LX, SX will behave like automatic filing operations, removing much of the need for the slow RNX (rename) instruction. The entire class of immediate indexing arithmetic instructions can now be deleted, their function being automatically provided by secondary options on direct indexing instructions. Floating vector and matrix multiply programs will be even easier to write and debug. A new point of view will be added to systematic mesh calculations. The possibility of performing floating arithmetic without referring to memory will entail savings in programming efficiency and speed. The number of memory interlock occurrences will also decrease.

5. Acknowledgements. Discussions with C. T. Apple, J. L. Garrity, C. L. Gerberich, D. H. Gibson (Kingston), N. Hardy, G. R. Hira, and R. L. Rockefeller (Kingston) have clarified much of the issues involved in the new post-gressive indexing scheme. The constructive criticism from C. T. Apple, D. H. Gibson and N. Hardy is particularly appreciated.

T. C. Chen

Tien Chi Chen
Special Studies

TCC/img