FIRING-SQUAD PROCESSORS

FOR MULTIPLICATION-PLUS-ADDITION

March 1979

$x = (x_{n-1}...x_1x_0)$, $y = (y_{n-1}...y_1y_0)$ and $z = (z_{n-1}...z_1z_0)$ are n-bit two's-complement (or unsigned) binary numbers. xy+z will be computed.

The processor accepts the bits of x, y and z in ascending order at clock times 0, 1, ..., n-1 (bits x_i, y_i, z_i at clock time i) and replicas of the "sign bit" (or zero bits for unsigned) at clock times n, n+1, ..., 2n-1. It produces the bits of xy+z in ascending order at clock times 1, 2, ..., 2n.

So xy+z is computed in at most 2n+1 clock times; the result may begin to flow into a subsequent process with 1 clock time delay.

It follows that processors can be parallelled to compute scalar products of k-dimensional vectors in 2n+k clock times, and further parallelled to multiply-and-add k-dimensional matrices in the same time.

The processor is an array (a "squad") of machines ("soldiers"), having all the same structure. The next state of each machine depends on its current state and on that of its two neighbours. The leftmost machine (the "officer") takes the input and issues the output (the environment playing the role of the officer's "left neighbour").

The minimum number of soldiers in the squad depends on the binary width, n. (That is why the existence of such a processor does not contradict the fact that no linear-time multiplicator is known.)

Two versions will be given.

Straight Version

A machine has

- six "data" bits: p_0, q_0, p_1, q_1, p, q;

- a three-bit "accumulator": r, the bits of which will be denoted r_2, r_1, r_0;

- a two-bit "switch": s, its bits: s_1, s_0.

[n/2]+1 machines (at least) are needed to form a processor.

In addition to its own state, each machine "senses" the state of p, q, r_2, s of its left neighbour, denoted p^L, q^L, $r_2{}^L$, s^L, resp., and the state of r_0 of its right neighbour, denoted $r_0{}^R$.

The ***next state*** of a machine, indicated by asterisked letters, is produced this way.

if $s^L=3$: $s^* \leftarrow \min(s+1,3)$;

if s=0: $p_0{}^* \leftarrow p^L$, $q_0{}^* \leftarrow q^L$,
if s=1: $p_1{}^* \leftarrow p^L$, $q_1{}^* \leftarrow q^L$,
if s≥2: $p^* \leftarrow p^L$, $q^* \leftarrow q^L$;

if s=0: $r^* \leftarrow r_0{}^R + r_1 + r_2{}^L + p^Lq^L$,
if s=1: $r^* \leftarrow r_0{}^R + r_1 + r_2{}^L + p_0q^L + p^Lq_0$,
if s=2: $r^* \leftarrow r_0{}^R + r_1 + r_2{}^L + p_0q^L + p_1q_1 + p^Lq_0$,
if s=3: $r^* \leftarrow r_0{}^R + r_1 + r_2{}^L + p_0q^L + p_1q + pq_1 + p^Lq_0$.

[This can be simplified for hardware purposes in many ways. So, e.g., we add the transition $p^* \leftarrow p^L$ if s=1 (that is, if s≥1). Then we can write

if s=0: $r^* \leftarrow s_0s_1r_0{}^R + r_1 + r_2{}^L + p_0q^L + p^Lq_0 + p^Lq^L$,
if s>0: $r^* \leftarrow s_0s_1r_0{}^R + r_1 + r_2{}^L + p_0q^L + p^Lq_0 + p_1q + pq_1$.

Let P denote p^L for s=0 and p_0 for s>0. Then we can have

in all cases $r^* \leftarrow s_0s_1r_0{}^R + r_1 + r_2{}^L + Pq^L + p^Lq_0 + p_1q + pq_1$,

wherein the only switching is that for P. The first terms can be simplified to $s_1r_0{}^R$ if the rightmost machine senses 0 for $r_0{}^R$. - Obviously, the addition for r^* is the decisive operation as to clock time duration; this is to be cut down by clever circuitry.]

Before computation, a processor reset must be started, that is, the leftmost machine has to sense a 0 for s^L. (This reset will propagate from left to right; there is, however, no need to wait for that.) The reset can be given at the last ($2n^{th}$) clock time of a computation (so no extra time is used up between successive computations).

Now computation is done this way.

The leftmost machine shall sense 3 for s^L, and successively the bits of x, y, z, from the lowest bits on, followed by replicas of the sign bit (or zero), for p^L, q^L, $r_2{}^L$, resp. The output appears on r_0 of the leftmost machine, from the lowest bit on.

(Perhaps the simplest way to verify the procedure is this. First verify it for n=1,2. Then assume its validity by induction for the processor formed by the array of machines with the leftmost

machine left out, and ***join*** that processor to the leftmost machine.)

Queer Version

This is simpler, but (at least) n machines are needed.

Now s is one bit, and there are but two "data" bits: p_0 and q_0.

In addition, there is a two-bit "rail" (or "bus"): p, q, which is sensed by all machines simultaneously.

Note that signal propagation along the rail takes time, so there must be some trade-off between clock time duration and array length; the processor is ***not*** really linear-time, nor in fact a squad. (Still, for moderate values of n this can be practically neglected, and the processor performs well, if that is what is aimed at.)

Each machine senses the rail and s^L, r_2^L, r_0^R.

The ***next state*** is:

if $s^L=1$ and $s=0$: $s^* \leftarrow 1$, $p_0^* \leftarrow p$, $q_0^* \leftarrow q$, $r^* \leftarrow r_2^L + pq$,

if $s=1$: $r^* \leftarrow r_0^R + r_1 + r_2^L + p_0 q + p q_0$.

[Denote by P now p for $s=0$ and p_0 for $s>0$. Then we can compute in all cases $r^* \leftarrow s r_0^R + r_1 + r_2^L + Pq + p q_0$.]

Again, $s^L=0$ resets the processor.

Computation is done analogously to the straight version except that the leftmost machine must sense 1 for s^L, and the bits of x and y go to the rail.

(Verification is by simple induction.)

=====

Both versions can be modified to work with arbitrary-base (rather than binary) digits. E.g., with base 4 - two-bit - digits, the necessary number of machines and clock times is halved, at the cost of more storage and more complicated operations.

(It turns out that cell automata, even one-dimensional ones, ***may*** have practical applications.)

History

Investigation of "squads" for computational purposes was probably originated by F.C. Hennie (Iterative Arrays of Logical Circuits, MIT Press, 1961). The first to use them for multiplication (and presumably the first to multiply in linear time at all) was A.J. Atrubin (IEEE Transactions on Electronic Computers, **14** (1965), pp. 394-399). An improvement upon his method was given by D.E. Knuth (The Art of Computer Programming, Vol. 2, 4.3.3, Addison-Wesley, 1969, pp. 276-278); the improvement consists in that Knuth multiplies-and-adds (while Atrubin merely multiplies); and that with fewer states. The first version above is essentially the same as Knuth's. The second version is simplified by degradation from the first one (whereas it might make a very good chip).