4. A metrics of efficiency

The philosophy behind the ReAl API emanates from the concept of efficiency of implementation. Each processor, may it be general-purpose or highly specialized, may it consist of a singe core or be an ensemble of many cores, is basically a sequential state machine. It should do useful work. The task proper of a machine is not executing instructions but delivering output bit patterns according to the current input bit patterns. When an application problem is to be solved, intermediate variables, procedure calls and the like are essentially a waste of clock cycles or machine bandwidth (and, in consequence, power). Obviously, it would be better to have the job being done (for example, the rendering of a graphics presentation) than to push the EDX register onto the stack in order to get it free for multiplication and later to fetch it back again (this is an example of an instruction sequence which contributes nothing to solve the application problem, but is to be executed only to compensate for architectural quirks). Hence the metrics of raw performance should not be instructions per second (the ubiquitous MIPS), but effective application bits per second. To obtain an appropriate performance metrics PM, a certain interval of n machine cycles (cycle time = tc) is to be observed (n could be the number of machine cycles needed to execute a function or a complete application, for example). What is to be counted is the number of data bits B_i (operands and results) that have been moved over the data paths of the machine. However, only these data transfers will be considered that contribute to the solution of the application problem. Machines that waste bandwidth by fetching instructions and pushing register contents around may boast with impressive MIPS ratings, but their PM rating will be comparatively low.

$$\mathrm{PM} = \frac{1}{n \cdot t_c} \sum_{i=1}^{n} B_i$$

The utmost performance will be achieved when the following conditions are satisfied:

1. Each data bit belonging to the application problem is to be moved through the machine not more often but once.
2. In each machine cycle a partial result will be delivered.
3. The number of result bits delivered corresponds to the number of data lines (or processing width/machine word length, respectively).

Whether or not a machine can be built fulfilling these conditions, depends ultimately on the algorithm to be implemented. This property of the algorithm can be expressed by a characteristic value called efficiency of implementation e_i:

$$e_i = \frac{\sum_{i=1}^{n} \mathrm{CARDB}(A_i) + \sum_{j=1}^{m} \mathrm{CARDB}(R_j)}{z \cdot (\mathrm{ARG\_LINES} + \mathrm{RES\_LINES})}$$

- CARDB(A_i) and CARDB(R_j) designate the number of bit positions of the arguments and results.
- ARG_LINES and RES_LINES designate the number of signal lines available to move the argument and result data (processing width/machine word length). If there are fewer signal lines than bit positions (with respect to the particular data structure), the number of bit positions applies.
- z designates the number of machine cycles required to generate the total result.

Thus, the implementation efficiency is a dimensionless number in the interval between 0 and 1. If $e_i = 1$ then the most appropriate implementation is indeed feasible. Then each machine cycle will contribute to the final result by delivering partial results according to the number of available signal lines (or processing width, respectively). This is possible only when each partial result can be computed by combinational assignment from the argument words selected currently and, if necessary, by inclusion of state data determined in previous machine cycles.

The implementation efficiency is < 1 if:

1. The combinational assignment cannot be implemented (a question of cost and complexity).
2. Some result bits depend on argument bits within different argument words that require more than one machine cycle to be fetched.
3. Some argument values cause modifications of partial results already computed (that requires to fetch these result values again).

A situation according to the first condition is not always an insurmountable obstacle – it is ultimately a matter of discretion, what is considered too expensive or too complicated. On the other hand, the second and the third condition designates objective limits. Such algorithms can never be implemented with $e_i = 1$ (not even when cost does not matter). A plausible example is reordering of bits in a larger bit field (e.g., of pixels in a frame buffer) according to an index vector. Each argument bit is principally to be moved to any result bit position. In consequence, it may be necessary to fetch again partial results in order to insert new values. With regard to speed, the most desirable way to execute an algorithm is the immediate assignment. Obviously, this is not feasible for complex algorithms. Hence processing has to be done in steps (machine cycles). In each machine cycle, the number of bits to be processed depends on the width of the data paths. The determination of the number of data lines is a design decision. It depends on given goals of performance and limits of cost. For the most efficient algorithms (with regard to e_i), the number of signal transitions will not depend on the number of signal lines. E.g., when 64 bits are to be processed, there will be always 64 signal transitions, occurring simultaneously or consecutively according to the number of signal lines.

Efficiency of implementation and power consumption. It has been found that this metrics can be used to evaluate efficiency problems of power consumption, too. Today the primary problem is not transistor count, but saving power. According to a strict power saving philosophy, the universal computer is to be considered only a makeshift solution. With respect to an application problem, the true optimum solution would be a dedicated machine whose cycles are spent exclusively to compute the desired final results. In such a machine, neither clock cycles and memory bandwidth nor power would be wasted for fetching instructions, loading and storing intermediate values, calling functions and the like. ReAl machines should be true universal machines whose characteristics come as close to this ideal as possible. It is an old rule of thumbs that 30% of all memory access cycles in a conventional processor are used fetching instructions. As a rough estimate, 30% of all signal transitions could be related to the machine instructions (fetching, decoding, sequencing and the like). Consequently, a machine in which none of these activities takes place would draw 30% less power. This saving is related to full-speed operation (in contrast to the current practice of switching off idle functional units). There are two basic architectures without conventional machine instructions: the dataflow machine and the purely application-specific machine or accelerator. The ReAl API can be considered an approximation: conventional programs can be morphed partially into dataflow graphs, and resources can be interconnected to behave like application-specific machines.