ATG Press Appetizer

Volume 1, Number 1, December 2000

Globe CPU

Farnad Laleh
Aria Technology Group

ABSTRACT

Current microprocessor architecture suffers from employing some invented traditions in the 60s. Adhering those traditions has originated many unjustifiable design-level complexities till now. Many approaches that are based on those traditions have come to dead end in recent years. Among them, pipelining and superscalar techniques are notable. In this paper, we show that a powerful microprocessor does not need pipelining and superscalar techniques. The natural result of such an approach, which we call it *globe scale architecture* is the removal of branch prediction, speculative execution, and very long instruction words. Finally, we conclude that a typical CPU based on the proposed architecture dose not need more than the one-tenth of the transistors placed in a typical year-2000 microprocessor. In addition, it is shown that the proposed architecture achieves the best running speed for a sequential computer program.

1. INTRODUCTION

The Beginning of the 1960s was the start of inventing some new methods in computer architecture. Major inventions were made in IBM's STRETCH project. The base of employed methods in today's CPU architecture returns to that time and mainly the IBM 360/91 architecture. Pipelining, out-of-order execution and imprecise interrupt are among them. For that time transistor-based computers, those methods granted high boost to the computation performance. The performance gain motivated current microprocessor architects to employ the same methods to raise their designs performance. However, it was a misleading. The proposed methods had been designed for the technologies of the 60s, and employing them at the beginning of the 90s, which had enabled manufactures to place multi-million transistors on a single chip, was a real misleading. The negative impact of this misleading in today's CPUs is quite evident. Limiting the clock speed for a bug-free pipelining, and an ALU that operates in a double clock rate of the CPU core, are among the results of the misleading. In fact, the methods that were the heroes of the 60s architectures, are the devils of today's.

In this paper, we establish a new computer architecture approach based on today's available resources, rather than the traditions invented by pioneers in the 50s and 60s. In section 2, we discuss about the available resources for computer architects in the 21st century. In section 3, we present the proposed approach. Finally, in section 4, we draw a roadmap for CPU developers in the 21st century.

2. RESOURCES FOR CPU ARCHITECTS

Generally, we have two types of resources for CPU architects: physical and logical. Physical resources derive from advantages and limitations of the physics, e.g., 0.18-micron IC process, and high bandwidth of buses. Logical resources derive from the advantages and constraints of the mathematical rules and theorems in computer science, e.g., the Church-Turing thesis. In this paper, we do not use any new resource type and do not quit from the traditional computer science framework. We just organize the proposed resources in a new fashion.

The number of transistors we can settle on a single die is proportional to the number of functional units on a chip (CPU). However, it is not necessarily proportional to the computational power of a chip. It depends on the size of the transistors and the selected placement and routing strategies for their alignment on the die. Today, we can place up to 50 million transistors on a single die. We do not attempt to do it; rather we want to use the die space for an extended routing strategy. This is our first resource.

The second resource is several buses of 64-bit width or more. We can attach two or more instructions to each other and load them once to the CPU via the proposed buses. It is not for the purposes like superscalar computing or speculative execution.

Conventional CPUs use some functional ALUs to perform their calculating operations. These ALUs are functional units, i.e., their current operation is a function of the executing CPU's instruction. Logically, it is possible to build some ALUs that are not functional and always perform a few fixed operations. This type of ALU that is called *fixed arithmetic logic unit* (FALU) is our third resource (a logical resource).

In the next section, we explain how we can organize these three resources to establish our proposed CPU architecture. This type of architecture is applicable to any type of CPU, e.g., a DSP. In this paper, we focus on microprocessors, and by the term "CPU," we mean a typical general-purpose microprocessor.

3. GLOBE SCALE ARCHITECTURE

In traditional computer science, a CPU or *random access machine*, is a device with a set of instructions that are supposed to be arranged in a desired form for making a desired program. In a regular CPU (in this paper, we call RISC CPUs as regular CPUs because their reduced instruction set is simple for comparison purpose. However, what we explain in this paper, is extensible to CISC and EPIC CPUs as well) we have three major groups of instructions as the following:

1. Data transfer
2. Control
3. Arithmetic and logic

What we call computation is the third group of instructions and the first and the second groups are for arranging the third group in a desired manner. So we have a concept named *arrangement* that employs the first and the second groups. We want to replace the proposed concept with a simple and new concept, which we call it *selection.* The new concept eliminates the third group of instructions and relaxes the first and second groups. To introduce the new concept, consider the following simple example. Suppose you go to a restaurant and see the list of available foods in the menu and order to the servant (CPU) your desired appetizer, meal, and desserts. The servant goes and prepares your proposed foods (arithmetic results) in the order you have requested him. In fact, you *arrange* the works (arithmetic operations) he should do. Now consider you go to a fast food restaurant, which has prepared all of the listed foods before your arrival and demand. What you need to do is to *select* your desired foods (arithmetic results). This is the concept of selection in a CPU, that is, preparing all possible arithmetic results before the program demand, and selecting the proper one on the program demand.

To implement the proposed concept, we replace the so-called *general-purpose registers* with a new type of registers called *wild registers.* The only possible operation that a typical computer program can do with these registers is to write (store) on them. In fact, there is no need to use any other operation for them. They are connected to several FALUs in a combinatorial fashion. This is for producing all possible arithmetic results in parallel. Consequently, the outputs of the FALUs are stored in a new type of registers called *mass registers.* For a computer program, these are read-only registers. Only the FALUs can write on them. The following example shows a better view of the proposed concept.

Suppose we have four wild registers. Similar to conventional CPUs, our arithmetic operations have one or two operand(s). Therefore, the number of possible combinations is C(4, 2) = 6 and we need six FALUs. As mentioned before, FALU is not a functional unit and it performs all of its defined operations on its two inputs, in parallel. For a regular CPU, the defined operations are add, subtract, multiply, divide, and, or, xor, shift logical L/R, shift arithmetic, increment, decrement, and compare.

Therefore, every FALU has twenty output lines and the output of each line is stored on a unique mass register. Consequently in a CPU with four integer wild registers, we need $6 \times 20 = 120$ integer mass registers. The same stuff could be considered for floating point wild registers. In a floating point FALU, we do not have logical operations; instead, we have single and double precision operations. The numbers of mass registers for the integer and floating point operations are almost equal.

Wild registers can feed from both of memory and mass registers. Mass registers can send their data to both of memory and wild registers. In the proposed architecture, when we load our desired data into wild registers, all of possible results are prepared in mass registers. Only by selecting the corresponding mass register, we reach our required result. This is the replacement of arrangement by selection. As can be seen, the third group of instructions has been eliminated. Figure 1, shows the concept of wild and mass registers, schematically.

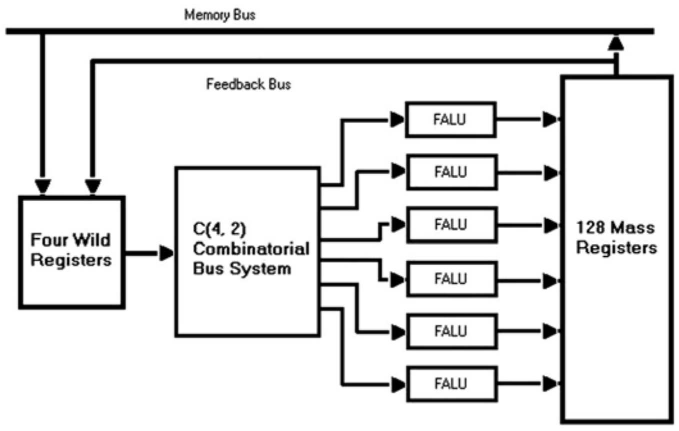

Fig. 1. *Wild and mass registers interconnections*

Currently, we have only two groups of instructions: data transfer and control. Since we use multiple FALUs and each FALU performs several arithmetic operations in parallel, we need an extended number of flag registers. In addition, since the required clock cycles for different arithmetic operations (especially for floating point operations) are different, we need some condition registers to indicate the situations of the performing and terminated arithmetic operations. Since in control instructions, we load/store an address to/from memory pointer registers, we can consider them as data transfer instructions. The only difference between them is the conditional operation. Adding a condition bit to the instruction word, see figure 2, solves the problem. This method is similar to the Predication in EIPC architecture. When the condition bit is on, the instruction is performed conditionally. Using this method, we can unify control and data transfer instructions. Currently, we have only one group of instructions, which is called data transfer.

Let's have a look to the advantages of the elimination of arithmetic instructions and the unification of control and data transfer instructions.

1. Since we have only one type of instructions, we can fix them on a 32-bit instruction word in a decoded mode. So we do not need any decoding process in the CPU cycles.
2. We use combinatorial buses and FALUs. Therefore, we reach the maximum logically available parallelism. In addition, we do not have a separate execution cycle as we have in conventional CPUs. The proposed CPU fetches and executes, independently.

3. Looking to the above advantages, one can find that there is no need for pipelining and therefore branch-prediction and speculative execution. In addition, employing the FALU enables us to achieve maximum parallelism and to dismiss superscalar computing and instruction level parallelism. What we need are high memory bandwidth and low memory latency to load more data to wild registers and read more from mass registers.
4. The proposed CPU has no decode and dynamic execution units. It uses fixed length instructions. Therefore, it employs a reduced number of transistors down to the one-tenth of today's CPUs. However, because of employing combinatorial bus system, it uses an extended on-chip routing. This type of architecture is attractive for multiprocessing on a single chip.
5. With a 32-bit instruction word, we are able to take over most functions. There is no need to use a *very long instruction word.*

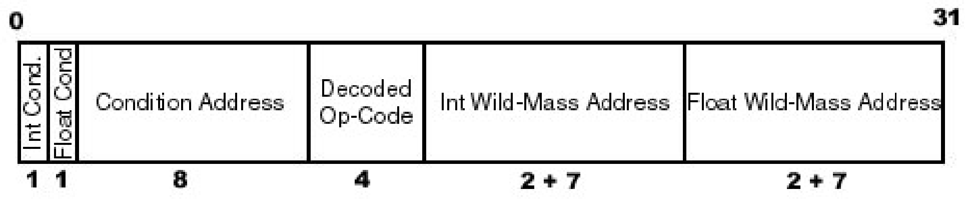

Fig. 2. *A typical GSA instruction word*

Figure 2, shows a typical globe scale architecture (GSA) instruction word in a CPU with four wild registers for each of the integer and floating point operations. The first bit of the word is *integer condition bit.* Whenever this bit is on, the CPU goes to the address of the indicated condition register which is designated by an 8-bit condition address. It is to see if the proposed condition is true or not. If true, the CPU executes instruction. If integer condition bit is off, the CPU directly executes the instruction. The second bit is *floating point condition bit*, which does the same thing for the floating point conditions. This feature of the instruction word makes it very suitable for *multiprocessor computations*. There is also a 4-bit op-code for indicating memory transfer modes and a few conventional instructions, e.g., no-op. The remaining bits are for indicating the integer and floating point wild and mass registers. Memory pointer registers are considered as a part of mass registers' address space. However, their I/O has some subtle differences with mass registers. Since we use 64-bit data bus, the remaining 32-bit word would be an immediate operand or a memory address.

In GSA, there are different ways that we can define the structure of an instruction word. What we have explained was just a simple example. In fact, logical scalability of the proposed architecture is quite simple. For ambitious powerful CPUs, we can extend the 32-bit instruction word to a 64-bit one. We call it *internal scalability.* From a different point of view, the proposed CPU is similar to a memory chip. Its instruction word, which contains the addresses of condition, wild, and mass registers, looks like the address bus of a memory chip and its 32-bit operand is similar to memory data. This property grants the ability of multiprocessing to the proposed CPU, in a new manner. It could be done in a similar way that is used to increase the capacity of a memory module by connecting several memory chips to a single bus. We call this property as *external scalability*. It seems that enough reasons have been presented to show why the proposed architecture is globe scale.

4. ROADMAP FOR CPU ARCHITECTS

Although the GSA removes many performance barriers from CPU architecture, memory latency is still a major obstacle. Since the only instruction type in GSA is data transfer, it exposes the ugly feature of this obstacle more than before. Current memory technologies enable us to achieve a high bandwidth, but not a low latency. In conventional architectures, we have two major methods to decrease the latency: caching and reordering. It is quite evident that the reordering has no meaning in GSA, so the only applicable approach would be caching.

Today's common approach in caching is to bring L1 and L2 caches on CPU die. It is good, but some better methods could be used. One approach would be the addition of a programmable cache memory to the CPU die. Another approach is to add a small fully associative cache memory to DRAM modules. Using this trick, a DRAM module is seen like an SRAM from the memory controller side. In fact, memory controller can read and write from/to the proposed cache memory. Afterward, the data are transferred between the proposed cache and DRAM chips using today's memory signaling system in a high transfer rate. Combining the two recent approaches is a good way for saving a GSA-based CPU from the memory latency disadvantages.

An important issue in GSA is the degree of compatibility with the former approaches like CISC, RISC, and EPIC. The RISC approach because of its reduced instruction set has the most compatibility with the GSA. It is possible to compile RISC assembly codes for a GSA-based CPU, easily. However, it is required more works for doing the same thing for CISC assembly codes, and please forget EPIC codes!

Finally, we should mention that the GSA is not only an advanced method for parallel processing or computational power boosting, but also it has an interesting impact on signal and image processing, AI, and sensor fusion applications.

5. COMMENTS FOR THE SECOND EDITION

In the new edition of this paper, we made no change to the content of the first edition. It is due to the passing of a very cool period (2000-2004) in the field of CPU architecture. Four years after the publication of the first edition, the poor presence of EPIC CPUs in the market can be seen apparently, which was predicted by the author. After the first publication of this paper, we received some criticism (some of them from a pioneer computer architect) about our presentation style, including the reference-less nature of this paper. In this edition, we eased some of the previous complicated sentences and also corrected the second figure.

REFERENCES

It should be mentioned that most papers of our Appetizer series are reference-less. As our Appetizer papers are more a critique of current industrial trends, we eliminate the references for not pointing to a few names in the industry as devil. We also suppose that the reader of these series is among the industrial activists.