Hardware Resources: A Generalizing View on Computer Architectures

Wolfgang Matthes, 1990

A manuscript in a tentative stage.
The finished paper has been published in
Computer Architecture News, March 1990.

Abstract
Computer hardware is seen as a collection of resources. Performance, functional capabilities, and quantity of these resources are decisive to overall system performance.
Starting from this paradigm, computer architecture design may be handled systematically in the future even by formalized methods.

1. Introduction

Computer architecture is mainly concerned with the interface between hardware and software. Most of the computer architecture work covers the instruction set, the principles of operation (e. g. instruction sequencing, addressing, interrupt handling etc.), and some related aspects of hardware structure, as the quantity of processing units, the software-accessible registers, the memory structure, and the data paths. Instruction set design and the corresponding principles of machine organization have found a great interest and have led to many controversial discussions, with the RISC-CISC debate being the most prominent example.
This paper demonstrates how a closer look on details of the hardware structure may lead to a more generalizing point of view which promises many problems discussed controversially to be handled in the context of a systematic approach by future work.

2. Computers as Collections of Hardware Resources

"Hardware resources" is a popular term in computer literature.[1] Reasoning about this term a little more in detail will yield a general paradigm which allows to handle important problems of computer architecture design in a considerably more systematic way.
Fig. 1 shows how each computer can be regarded as a collection of hardware resources, such as memory, processing means, data paths etc. These resources are program-controlled, i. e. the execution of a stored program determines how the resources are used in each of the

---

1 It had been used by F l y n n in his paper [2], for example.

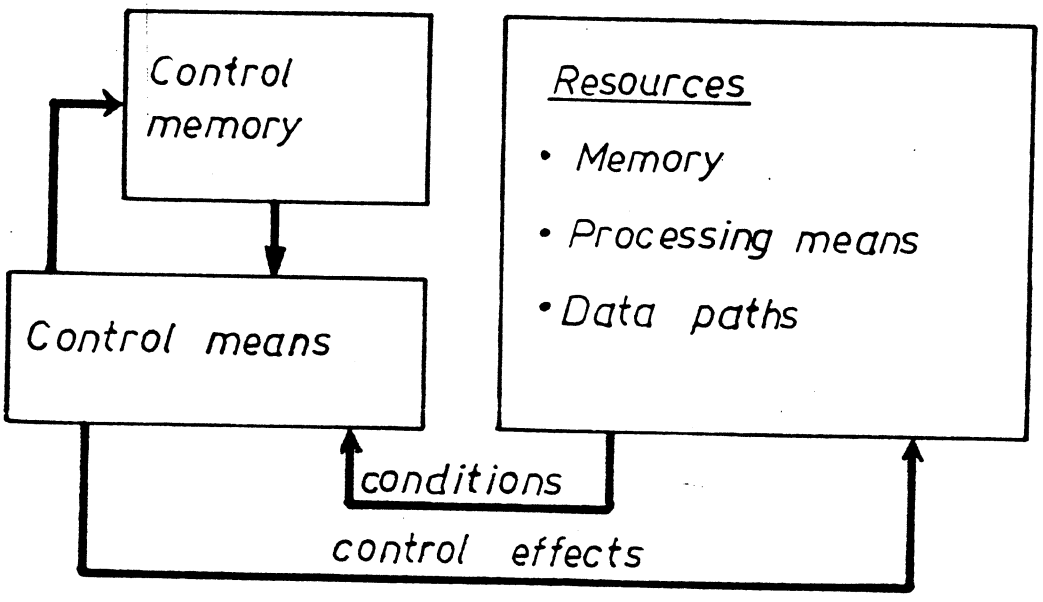


Fig. 1. Computer structure as a collection of resources.

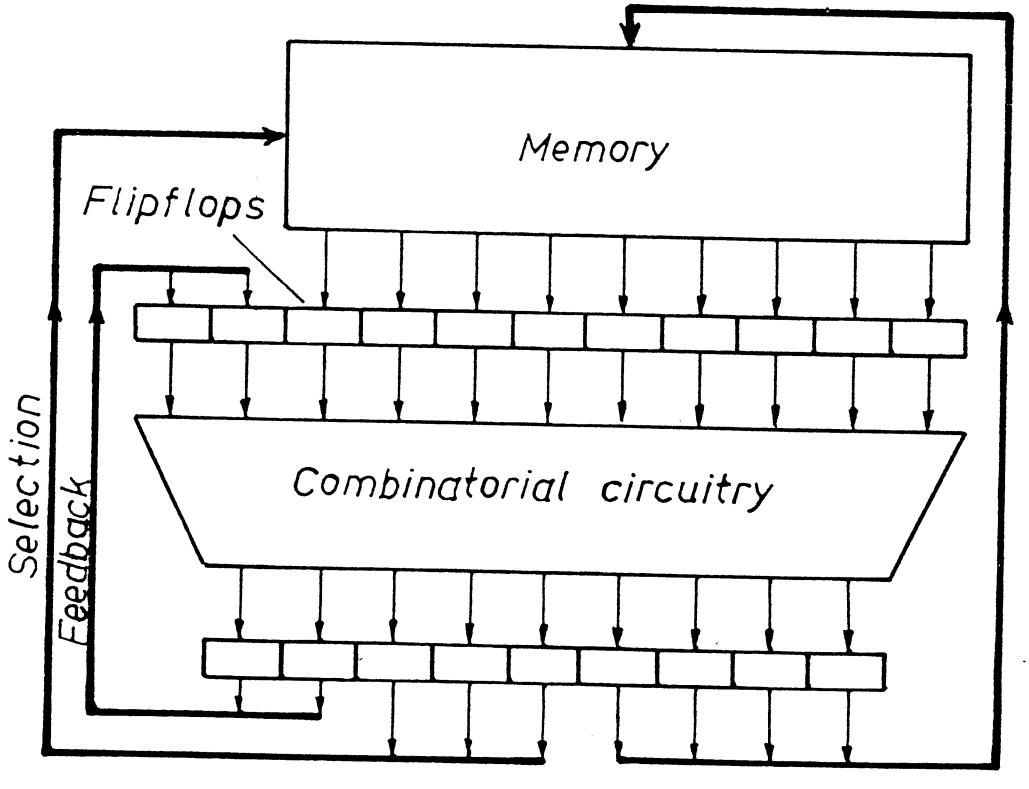


Fig. 2. In-detail view: the resources vector.

machine cycles. Thus the resources are to be completed by storage means for the program information (control storage) and by control means which select the appropriate control information for each machine cycle and which control the operation of the remaining resources. This paradigm covers two well-known principles:

1. The v. Neumann architecture. The control storage is accessed sequentially. The sequence is determined by the control means according to a hard-wired algorithm (instruction address counting or branching, respectively). In a genuine v. Neumamm machine, control storage and data storage are identical.

2. The dataflow architecture. The control storage is an associative storage which delivers the current resource control information according to resources available (i. e. "not busy") and the data which are ready to be processed.

The basic limitations of both approaches can be shown easily:

1. The v. Neumann machine is performance-limited, because the selection of control information is determined by the hard-wired instruction sequencing. Hence inherent parallelism during runtime cannot be used.

2. The genuine dataflow machine requires a control memory with completely associative access which cannot be built. Thus the dataflow principle is cost-limited; it can be implemented only with compromises between associative and sequential control storage access or between performance and cost, respectively.

3. Resources and Performance

1. Functional capabilities and sheer performance of the processing resources are decisive to the overall performance of the computer. Example: A computer with a hardware multiplication unit will perform multiplication faster than a machine which does multiplication by means of shift-add sequences.
If multiple processing resources are provided, performance will increase proportionally, provided that all these resources can be kept busy with useful work. The peak performance of an arbitrary computer structure can be calculated according to a simple formula:

$$\text{Peak\_Performance} = \frac{\text{Quantity\_of\_Appropriate\_Resources}}{\text{Duration\_of\_a\_Machine\_cycle}}$$

This is completely independent of a particular architectural principle.

2. If the ensemble of processing resources has been selected, performance will be influenced by the connection structure together with the memory structure. These structures are decisive to feed the processing resources with data to be processed and to drain the results. Such activities should be done completely in parallel to processing or at least with a minimum of additional machine cycles.

3. In addition to these structures, instruction formats, addressing principles etc. must be designed with the main objective to keep most of the resources busy with useful workload during most of the time. Workload assignment can be done during compile time or during runtime. Assignment during runtime requires special circuitry to recognize segments of the dataflow out of the instruction stream. If the assignment can be done during compile time, such circuitry is not necessary. This makes higher performance possible (special circuitry will slow down the machine cycle or will occupy silicon area which would otherwise be available for additional processing resources). Furthermore a compiler can overlook the whole program, thus we can hope to detect more of the inherent parallelism than by watching the instruction stream during runtime (experience has shown that during runtime only the parallelism within short sequences of 5...10 instructions between branches can be used). However, objections against this approach (e. g. in [8]) should be taken into consideration, too. Hence appropriate compromises should be found (perhaps including hardware support for compile time dataflow analysis).

4. Resource Control

To reason about resource control and information flow organization requires to investigate hardware structures more in detail. Processing resources are made of flipflops or hardware registers and combinatorial circuitry. All flipflops or registers of the processing part of a computer can be regarded together as the resource vector. Parts of the resource vector need information from storage means, parts deliver selection information (i. e. addresses) to storage means, parts deliver data to storage means, and parts of the resource vector are fed back one upon another. This scheme is shown in fig. 2. It applies to all paradigms of digital (binary) information processing. Starting form this point of view, design alternatives can be investigated systematically (fig. 3):

a) Immediate assignment. All necessary information is fed immediately to the corresponding positions of the resource vector. This corresponds roughly to VLIW architectures or to the microprogram level of machines with "horizontal" microinstruction formats.

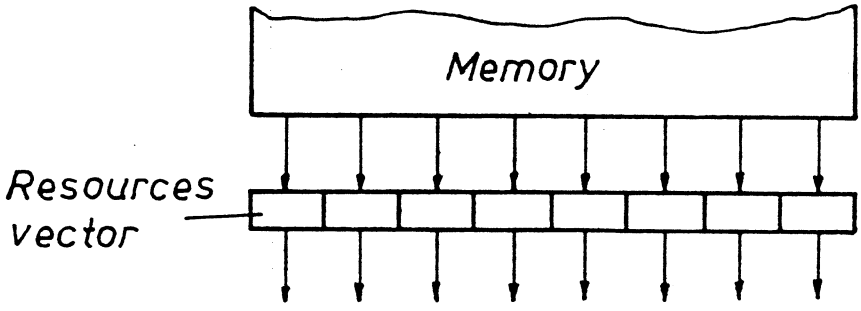
b) Group selection. In each cycle only a selected part of the resource vector is loaded. This part is coded explicitely within the information. This corresponds roughly to most of the RISC machines or to machines with "vertical" microinstruction formats.

c) Coded selection. The information is decoded by means of dedicated circuitry and affects parts of the resource vector selectively. This is the paradigm for most of the conventional architectures.

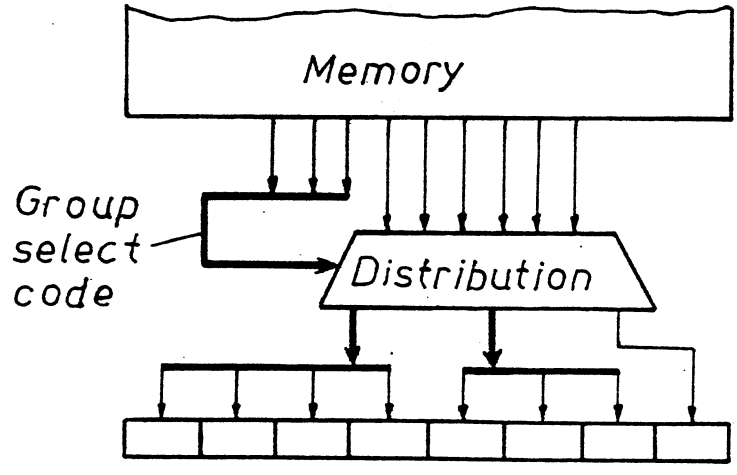
More in detail, the information fed to the resource vector must provide for:

1. the selection of the operations to be performed
2. the arguments of these operations
3. the destination control of the results produced
4. the selection of the successor control information
5. a coded description of the own format, if necessary.

a) Immediate assignment



b) Group selection



c) Coded selection

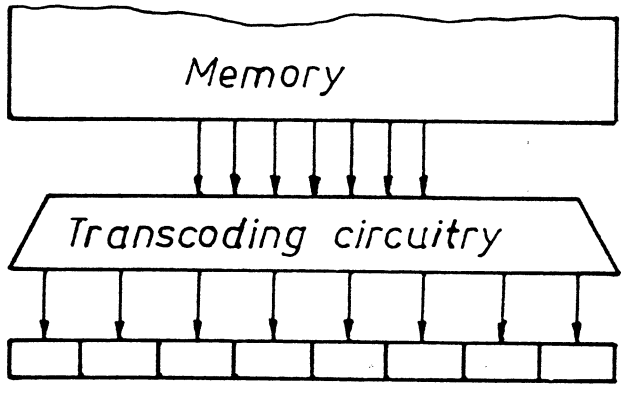


Fig. 3. Alternatives: feeding the resources vector from memory.

Operation codes are ordinal numbers into the set of all operations which are provided in the hardware. Conventional machine instructions have only one operation code. VLIW instructions contain multiple operation codes. In microinstructions the operation control information is coded within various control fields or even single bits.

Arguments can be delivered as immediate values, i. e. the proper values accompany the operation codes and the other control information. (An extreme solution is to provide all arguments of an operation as immediate values accompanying the control information. This is just another view of the well-known dataflow concept.) However, in most cases the values are not available immediately, but must be fetched according to selection information (i. e. memory addresses). In conventional architectures an address specification comprises an ordinal number into the set of addressing principles (absolute, base + displacement, indexed etc.) and some immediate values (address of the base register, offset etc.). For destination control, similar addressing principles are provided. In addition, it is not necessary to store each result in memory. Instead it could be kept within the resource vector, e. g. in an accumulator or top-of-stack register.

Control information sequencing is usually implemented by hard-wired counting. If the sequence is to be modified, appropriate selection information (branch addresses) must be delivered together with an ordinal number into the set of branch conditions.

If there is no 1:1-correspondence between the control information and the bit positions of the resource vector, the control information must describe its own format. These descriptive codes control the selection or transcoding circuitry shown in fig. 3 b) or c), respectively.

The control information is used according to the following scheme:

Step 1: The control information is loaded into the resource vector.
Step 2: The arguments are fetched.
Step 3: The results are calculated.
Step 4: The results are assigned (i. e. moved to the memory).
Step 5: The successor control information is selected.

Not all the steps 1 - 5 can be executed simultaneously. The time interval to calculate the results (step 3) is determined by the design of the corresponding resources. Given this, only the following steps may be accelerated, executed in parallel, or omitted:

- Argument fetch (step 2). In many cases it can be overlapped with the preceding result calculation ("look ahead"); it can be accelerated by providing extremely fast argument memories (data caches); or it can be avoided by feeding arguments immediately accompanying the control information (dataflow principle).

- Result assignment (step 4). Cache memories can accelerate result assignment if designed properly ("non-store-thru"- principle). Result assignment should be avoided or postponed whenever possible. Compilers should try to keep results within the resource vector, and hardware architecture should be designed with this objective in mind.

• Selection of and access to successor control information (step 5 and step 1 of next cycle). This should be overlapped with result calculation ("instruction look ahead"). Branches can be overlapped by means of principles borrowed from RISC architectures or from micro-programming (e. g. "late" multiway branching). Besides overlapping, acceleration is possible, too (instruction caches; dense coding of instructions which allows to fetch multiple instructions (i. e. more meaningful control information) in each instruction access).

5. Outlook

Each resource is characterized by the corresponding information structures and by the operations it can perform. Hence it can be represented by means of algebraic structures. By adequate formalization, a resource algebra or a set of algebraic structures for different purposes (e. g. performance evaluation[1]) could be built. In the future this approach could replace many of the "gut feel"[2] decisions in current architecture design by at least approximative calculations. A computer architecture could be developed according to the following scenario:

1. The objective of the architecture design is to provide a powerful, cost-effective, feasible, and versatile collection of resources.

2. The designer has to tackle the following problems:

a) Select the resources with respect to functional capabilities, performance, and cost. If cost and performance of each candidate resource (e. g. adders, multipliers, address generators, register files etc.) are known, also the statistical profile of application requirements, it may be a problem of linear programming to choose an appropriate mix of resources within given cost constraints.[3]

b) Provide sufficient connection and memory structures in order that all resources can be kept busy with useful work during most of the time.

c) Provide an efficient encoding of the control information (i. e. an appropriate instruction set).

d) Provide compilers which can make optimum use of all resources.

A sufficiently enriched resource algebra could be used as an intermediate language. Compilers could translate all application programs into this language which describes the resources needed, their interaction, and the inherent parallelism. In a second pass, a machine program for the real computer could be compiled from the intermediate language (i. e. a mapping from the abstract to a concrete collection of resources).

---

1 [4] may serve as an example.
2 The term was used by L i n c o l n in one of his remarks on supercomputer architecture design ([3]).
3 C o l w e l l mentioned such an approach in his thesis [1], but regarded it to be infeasible due to lacking foundations.

Such work is yet to be done. However, a brief outlook can be given how architectures designed according to this paradigm could appear:

- They shall not only being based on the best of CISC and RISC principles (e. g. as the Intel 80860 and 80486 or the Motorola 64040), but also on the best of VLIW, microprogramming, and dataflow principles.

- They shall contain more resources than conventional architectures, at least multiple processing resources so that inherent parallelism could be used.

- The resources shall be more appropriate to application requirements, and they shall be optimized internally.

6. References

[1] Colwell, R.P. The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems. Department of Computer Science Carnegie-Mellon University, CMU-CS-85-159, Pittsburgh, Pa., 1985.

[2] Flynn. M. J. Some Computer Organizations and Their Effectiveness. IEEE TC-21: 948-960, September, 1972.

[3] Lincoln, N. R. It's really not as much fun building a supercomputer as it is simply inventing one. In: Kuck, Lawrie, Samek (eds.) High Speed Computer and Algorithm Organization, 3-11, Academic Press, 1971.

[4] Müller-Wichards, D. An Algebraic Approach to Performance Analysis. In: Lecture Notes in Computer Sciences 295, 159-185, Springer, 1988.

[5] Wilson, R. 80860 CPU positions Intel to take on minisupercomputers. Computer Design Vol. 28 No. 7, 20-23, April 1, 1989.

[6] Wilson, R. Intel 80486 carries complex instruction set to RISC speeds. Computer Design Vol. 28 No. 9, 18-20, May 1, 1989.

[7] Wilson, R. 68040 moves toward RISC camp with redesigned pipelines, caches. Computer Design Vol. 28 No. 9, 22, May 1, 1989.

[8] Wirth, N. Hardware Architectures for Programming Languages and Programming Languages for Hardware Architectures. Operating Systems Review Vol. 21 No. 4, 2-8, October, 1987.