Towards a Formal Theory of Computer Architecture [1]

S. M. Müller [2] and W. J. Paul [2]

February 1990

Abstract.

A theory for evaluating the performance of computer system architectures is proposed. Formally systems consist of a CPU architecture and a compiler. The model for CPU architecture is a generalization of switching circuits. In order to deal with constant factors of component cost and run time certain technology parameters are introduced, e.g. the relative size of gates in circuits and bits in ROM. Workload is modeled by benchmark programs, e.g. Dhrystone. Benchmark programs can be analyzed in such a way, that the effect of various changes in an architecture can be determined without too much effort. A typical theorem might be of the form: if ROM is very compact and fast, then CISC is better than RISC.

1. Introduction.

In this paper we try to make some steps toward the development of a formal theory of computer architecture. We would like to be able to prove theorems of the following nature: Under a workload with parameters W and a technology with parameters T architecture A1 is more cost-effective than architecture A2. The theorems should reflect reality in some sense. We feel that little has to be said about the desirability of such a theory. It would for instance be nice to compare RISC with CISC and VAX'es with 370's on a purely architectural level.

Given our objectives it is clear what we have to do. We have to formally define what we mean by the architecture of a computer system. We have to identify relevant parameters of technology and to define the cost and speed of architectures based on these parameters. We have to identify relevant parameters of workloads and to construct models of workloads with these parameters. We have to show that the evaluation of the performance of an architecture realized in a technology and under a model workload can be analyzed without extravagant effort. Finally we should give some first examples of theorems and we should compare the results with our intuition and reality.

---

1) Research partially funded by DFG, SFB 124

2) Institut for Computer Architecture, Computer Science Department, University of Saarland, D-6600 Saarbrücken 11, West Germany

2. A formal model of architecture.

We are interested in issues of price and performance. From the user's point of view a system performs well, when it lets him get done quickly. Getting done involves *both* writing programs and running them. High level languages support the process of writing programs. Good compilers and powerful hardware make programs run fast. We therefore consider compilers as integral parts of architectures.

We define an *architecture* as a 4-tuple (ML, H, L, Co) where ML is a machine language, H is a hardware which allows to execute programs from language ML, L is a set of high level languages and Co is a set of compilers which translates from the languages in L to ML. Because in this paper we are not proving lower bounds there is no need to refer to formal definitions of languages or compilers. Techniques for making such formal definitions are well known, e.g. using the concept of mathematical machines and simulation between such machines [2]. We just remark that such definitions can become somewhat tricky if we want say to consider pipelined execution of instructions. First we describe the more simple parts of an architecture, the high level languages, the corresponding compilers and the machine language. After that we proceed to develop a formal model for the hardware H.

2.1. An example of a high level language and a compiler.

C is used as high level language, but there are no string- or struct-assignments. Such operations are executed via procedure calls. A general description of the language is given in [1]. A C-compiler without any optimization is used. All expressions are determined on the stack, except simple assignments. That is not the best but the most simple way. There are two kinds of expressions:

arithmetical : a + b * c   structural : v. p → c [x]

Coding the different instructions of C, some special sequences are used very often:

load: constant value, global value, local value, global array element, local array element, parameter array element, stack element (pop)
store: constant value, global value, local value, global array element, local array element, parameter array element, stack element (push)
compute: on registers, on stack
call procedure/function, return from procedure/function

It suffices to generate code for those sequences and then to use them to put together the complete machine program. The stack-pointer and the frame-pointer are used frequently, therefore, they are stored in the so called scratch together with partial results. The scratch is a reserved part of the memory which can be addressed directly. Better compilers are of course possible. They are studied in a later and more detailed paper.

2.2. An example of a machine language.

In general a machine language can be defined by means of a transition function over the set of configurations for the corresponding computer, but in most cases a simpler definition is sufficient: There are two classes of instructions. The first class contains all straight- forward operations. During such an opeartion the program counter is incremented and the values of CPU registers and of a memory cell may be changed. The most commen operations are :

computation: $OP_1 = OP_2 \text{ op } OP_3$ assignment: $OP_1 = OP_2$

OP_i might be a register of the CPU or a memory cell. Memory cells can be selected via many different addressing modes. For each machine one has to specify what addressing modes and what operations are allowed. Sometimes the modes depend on the instructions (e.g. for RISC Computers). The second class of operations are the branch instructions, like jumps and operations to handle procedures.

conditional jump: if condition then PC = S + C else PC ++
call: D = PC , PC = S + C
return: PC = D or PC = D + C

D : register   S : register or PC   C : constant

If S is the program counter, the operation is called relative; otherwise it is called absolute. Sometimes call and return are more powerful, the CPU registers are saved into the memory and restored afterwards. Often this is not necessary and if the CPU has many registers it takes a lot of time. A simple example of a machine language is given below.

jump	if ACC rel 0 then PC = a else PC ++ rel: $<, >, =, <=, >=, \neq, <=>$ (always true)	
load	dest := source dest : ACC, B, IR, PC	source : a, M(a), M(IR)
store	dest := ACC	dest : M(a), M(IR)
move	dest := ACC	dest : B, IR
compute	dest := ACC op B op : add, sub, nand, negsub (-ACC + B)	dest : ACC, B, IR

tab 1 : a simple machine language :
M(x) is the value of the memory cell with address x ; a is a constant, specified in the instruction; ACC, B are data registers, IR is an index register.

To determine the control logic, a coding schedule for the language has to be fixed. In our case the format of table 2 would be suitable. After specifying the machine language, code can be generated for the special compiler sequences. As an example the code for the load local sequence is given; the other codes look similar.

code for load local:

```
ACC := FP          (frame pointer: M[1])
B   := c           (constant)
IR  :=ACC add B
ACC := M[IR]
```

00000xxx	jump never	00100xxx	jump <	010x00xx	store M[a]
00001xxx	~ >	00101xxx	~ <>	010x01xx	store M[IR]
00010xxx	~ =	00110xxx	~ <=	01101xxx	move B
00011xxx	~ >=	00111xxx	~ always	01111xxx	move IR

yyddzzxx	yy:		dd:		zz:		
	10	compute	00	ACC	00	sub	/ M[a]
	11	load	01	PC*	01	negsub	/ M[IR]
			10	B	10	add	/ a
			11	IR	11	nand	/ a

tab. 2: opcode specification :
INR is the instruction register , INR<24:31> is the instruction word, called opcode and INR<0:23> is the constant a. *: is not defined for compute

2.3. The hardware.

Hardware consists of a CPU and a memory system M. To pictures of data paths of CPU's, as found in books on computer architecture, we attach exact meaning, cost measures and time measures. Second we add a model of control logic. Because CPU's can be controlled in very different ways the model has to be fairly general.

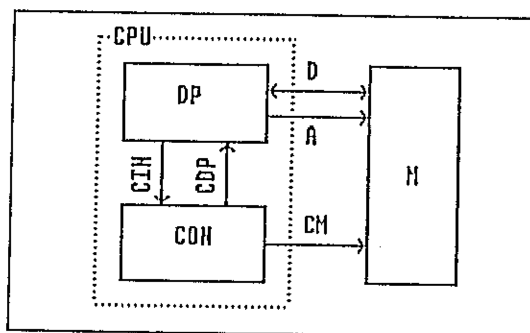

fig. 1: coarse structure of the hardware

The data paths DP of the CPU, the control logic CON and the memory M are interconnected as shown in figure 1. There is a bidirectional data bus D between DP and M, which is used to transport data and instructions between memory and CPU. The CPU provides addresses to the memory via address bus A. The control logic receives input from the data paths via bus CIN (typically from the instruction register and the condition codes). It controls the data paths via bus CDP and the memory via bus CM. Data paths, control logic and memory together are called the *hardware* H of the architecture.

2.3.1. Data paths.

Building blocks of data paths are: circuits, drivers, registers (we treat latches like registers), RAMs, ROMs (ROM is used in data paths for instance as lookup table for initial values of iteration algorithms) and internal busses. Symbols for the building blocks can be found in figure 2. We assume that we have gates for all symmetric boolean functions with two inputs and for negation. If S is a circuit with n inputs and m outputs we denote by $\varphi_S: \{0,1\}^n \to \{0,1\}^m$ the boolean function computed by S. We denote by c(S) the cost (= number of gates) of S and by d(S) the depth (= maximal number of gates on a path) of S. For details see [3, 5].

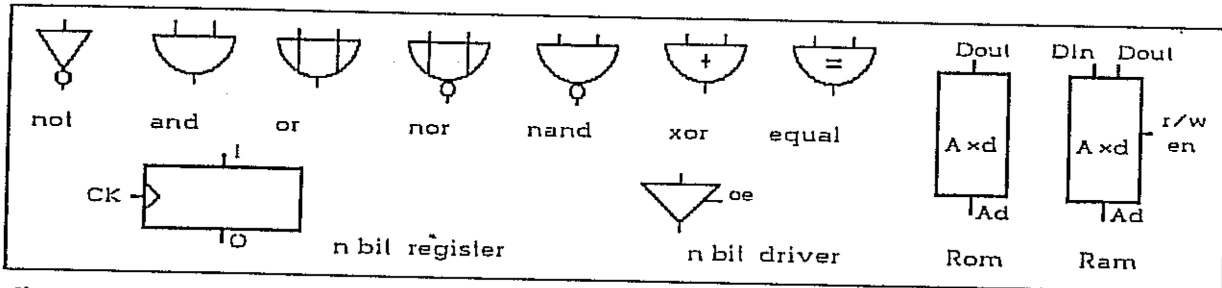

fig. 2 : symbols for building blocks

Each n-bit driver has n data inputs, n data outputs and one control input called oe (output enable, all our control signals are active high). Each n-bit register has n data inputs, n data outputs and one control input called ck (clock). Let a and d be natural numbers and let $A = 2^a$. An A×d-RAM has a address inputs Ad, d data inputs Din (data in), d data outputs Dout (data out) as well as two control inputs r and en (read and enable, r = 0 and en = 1 results in writing). An A×d-ROM has a address inputs Ad and d data outputs Dout. It behaves for most purposes like a gate with a inputs and d outputs. Each n-bit bus can have many sets of inputs and outputs, each n bits wide.

Building blocks of data paths cannot be combined in an arbitrary way. Table 3 specifies rules, by which inputs and outputs of building blocks can be combined. The rules need a little explanation. We cannot connect freely outputs of circuits or ROM with inputs of circuits or ROM because this could result in something like figure 3. Cycles containing neither a register nor a driver are forbidden.

out \ in	register	driver	bus	circuit	RxM-A	RAM-D
register	yes	yes	no	yes	yes	yes
driver	yes	yes (?)	yes	yes	yes	yes
bus	yes	yes	yes (?)	yes	yes	yes
circuit	yes	yes	no	yes (?)	yes	yes
RxM-D	yes	yes	yes	yes	yes	yes

tab. 3: rules for combination (RxM stands for RAM/ROM)

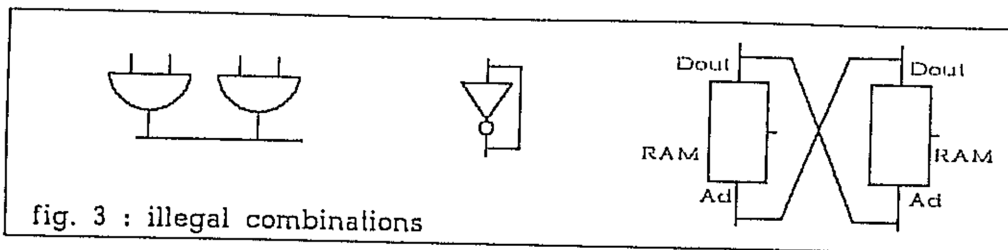

fig. 3 : illegal combinations

The control signals are connected to bus CDP. Inputs of building blocks are allowed to be connected to bus CDP. This makes particularly sense for certain inputs of circuits, e.g. select signals of MUXes or function signals of ALUs. Thus the control signals on bus CDP consist of the clock signals ck for the registers, the output enable signals oe for the drivers, the read and enable signals (r, en) for the RAMs and the open inputs f of the building blocks of the data paths. The control signals on bus CM consist of a read signal and an enable signal for the memory system. Outputs of building blocks of the data paths can be connected to CIN. Following backward any path from CIN one must reach a register before hitting a driver. Moreover no cycles are allowed on these paths. The data path of figure 4 allows to execute all instructions of the language ML, specified in 2.2. ALU and PC are complex circuits defined in figure 5. CON is the control logic, which is described later on. The carry-look-ahead principle [4] is used to construct the adder and the incrementer.

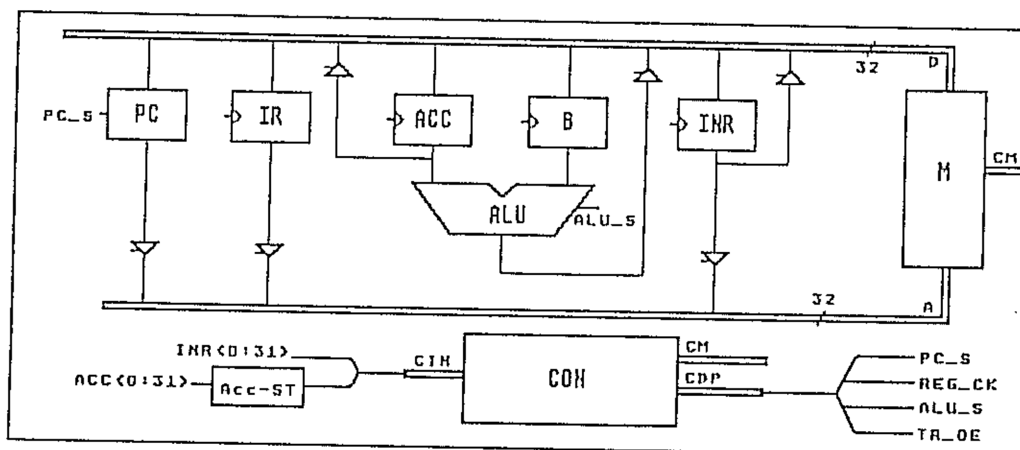

fig. 4: data path of the model architecture

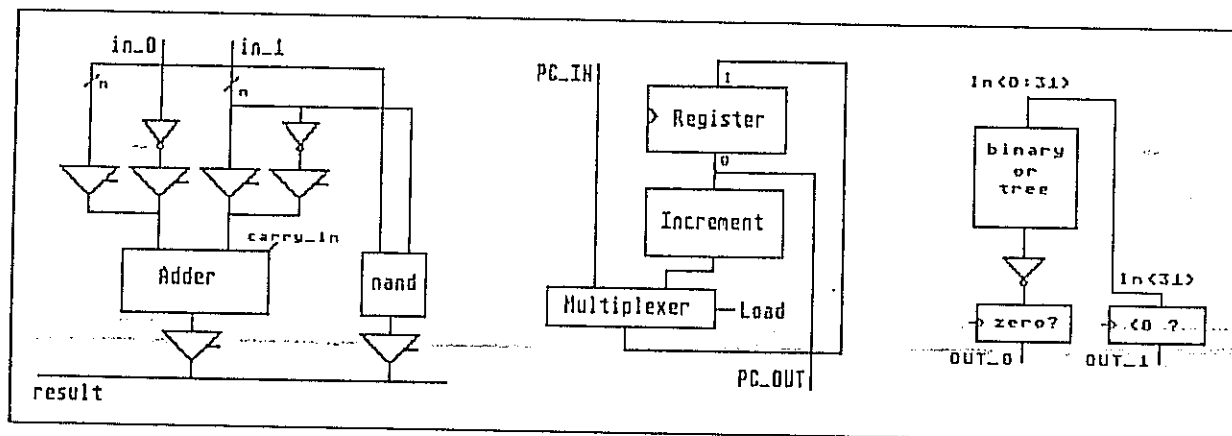

fig. 5: data path of ALU, PC and ACC-Status

2.3.2. Control Logic.

We treat the control logic as a finite automaton with input signals CIN and output signals CDP and CM. It consists of registers, circuits and ROM interconnected according to the rules of table 3. Open inputs of circuits and ROM must be connected to CIN. Open outputs of circuits and ROM must be connected to CDP or CM. Inputs and outputs of registers must be connected to circuits or ROM of the control logic. In the case of our model architecture the control logic looks like figure 4. We will always update all registers of the control simultaneously, thus we could treat them like a single register. The content of the registers of the control logic is called the *micro state* of the CPU. Figures 6 and 7 show two common examples of control logic.

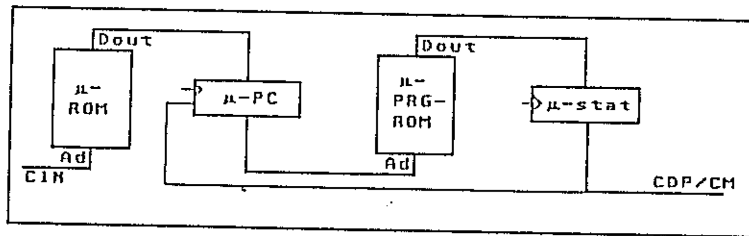

Fig. 6 : microcoded control logic.
This kind of logic is usually used in CISC-computers

The second kind is the hardwired logic. Each control signal is specified by a boolean function being written as a boolean polynom. The data of the CIN bus and the registers of the control logic are used as input signals. To keep the description of the matched circuits well defined, a fixed hardware realization for the polynomials is defined. Only the gates for the binary functions 'and' and 'or' and for the unary function 'not' are allowed. First the inverse of all variables and then the monomials are computed. After that the polynomial is realized by using the realization of the monomials and some or-gates. To keep the depth of the circuit small, balanced binary trees are used. Figure 7 shows an example.

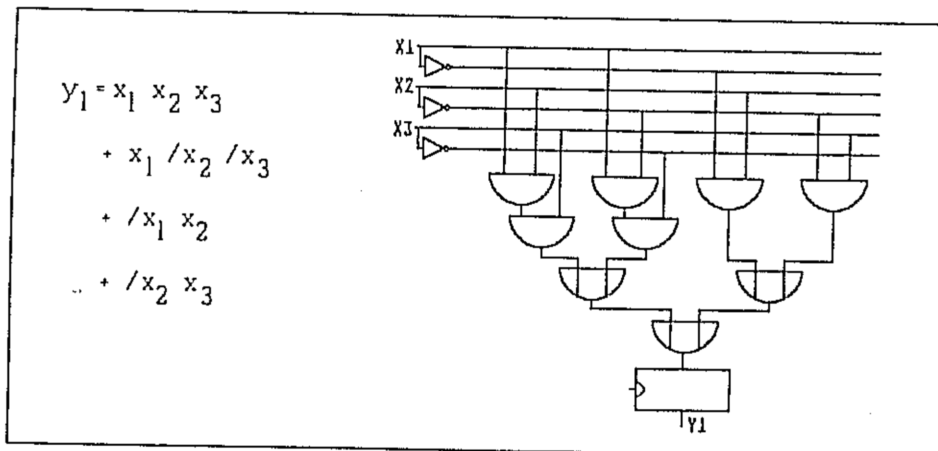

fig. 7 : The realization of a simple boolean polynomial

For the data path of figure 4 we have decided to use the hardwired version. Using our model we could show later that this version is more cost-effective than the microcoded

one. Doing so the polynomials look like the two following ones. Ex is a 1-bit register of the control logic, whose value alternates between 0 and 1. This signal is used to decide, whether an instruction has to be fetched or executed.

```
ACCCK    = INR <31> */INR <29> */INR <28> *EX   ; load ACC & compute ACC
INRATOE  = INR <30> *  INR <27> */INR <26> *EX   ; store M[a] & load M[a]
```

2.3.3. Register transfer.

A *configuration* for hardware with s RAMs M1, ..., Ms and memory system M0 is defined as a (s+3)-tuple $C = (\mu, \rho, \varphi_{M0}, \varphi_{M1}, \dots, \varphi_{Ms})$. Here μ is the micro state, ρ is the actual content of the registers of the data paths. If bus A is a bits wide and bus D is d bits wide, then $\varphi_{M0}: \{0,1\}^a \rightarrow \{0,1\}^d$ is a boolean function which maps each address α to the actual content of the cell of the memory system with address α. If the CPU reads from the memory system, then the memory system behaves like a circuit which computes function φ_{M0}. For $i \geq 1$ the functions φ_{Mi} are defined analogously for the RAMs Mi.

We treat the memory system often like one of the RAMs of the data paths namely RAM M0. The hardware which we have defined works in *CPU-cycles* in the following way: all control signals are determined from μ and ρ by the circuits and ROMs which connect the registers of the data paths and control logic to busses CDP and MC. If we treat this combination of circuits and ROMs like a single circuit S, then we have for the actual value $\zeta \in \{0,1\}^*$ of the control signals: $\zeta = \varphi_S(\mu,\rho)$. Value ζ defines a *modified hardware* $H(\zeta)$ in the following way.

For each driver or RAM whose output enable signal is 0 delete the driver resp. the RAM. Each n-bit driver D whose output enable signal equals 1 is replaced by the gate G(D), computing the identity function on $\{0,1\}^n$. Each $2^a \times d$ RAM Mi with en = r = 1 will behave like a gate R(Mi) which presently computes φ_{Mi}. Thus we replace it by the gate from figure 8. If en=1 and r=0 we replace Mi with the combination of a gate W(Mi) and a register L(Mi) from figure 8. This replacement is only made in order to simplify later the definition of the timing model. For each i fix the function computed by gate W(Mi) in an arbitrary way.

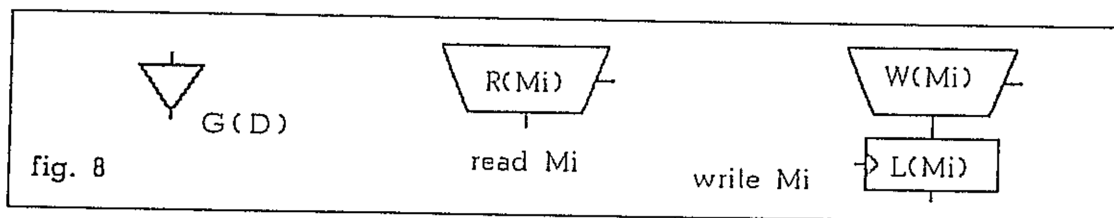

fig. 8

We call ζ *admissible* if the modified hardware $H(\zeta)$, consisting of registers interconnected by combinations of circuits, ROM and busses, contains no cycles. Moreover the only open inputs left are the ck inputs of the (artificially introduced) registers L(Mi) and inputs, dominated by control signals; e.g. an input of an and-gate is allowed to be open, if the other input is connected to a control signal, whose value equals 0.

Treating combinations of circuits, ROM, busses and drivers, which contains no cycles, as circuits T the actual value ν of each group of signals can be expressed as $\nu = \varphi_T(C)$.

We say that a register L is *clocked* during the actual CPU-cycle if L belongs to the control logic or L belongs to the data paths and the actual value of the ck signal for L (which is a bit of ζ) equals 1 or $L = L(Mi)$ for some i.

We finally say how to update the configuration. For each register L which is clocked the actual value of the data inputs of L becomes the new content of L. For each x and y and each RAM i with en = 1 and r = 0 such that the address inputs of W(Mi) have actual value x and the data inputs of W(Mi) have actual value y the value $\varphi_{Mi}(x)$ is changed to y.

2.3.4. Cost.

We measure the cost of CPU's in gate equivalents. Each building block has its basic costs, as figured in the tabel below. These values are scaled by three different densities. The total costs are taken as the weighted sum.

∂_a arithmetic , ∂_m memory, ∂_s single building blocks

The densities as well as the parameters α and γ depend on the technology. Different values will make different designs of CPU's cost-effective. The costs for the generation of timing signals and for layout have been abstracted away in this model.

building block	basic costs	gate delay
and, or, nand, nor	1	1
not	0.5	1
exor	3	2
1 bit driver	3	1
1 bit register	6	4
Ram/Rom $2^a \times d$	$6\alpha 2^a \times d$	γ a

tab. 4 : basic costs and gate delays

2.3.5. Time.

We measure the length of CPU-cycles in equivalents of gate delays. Let ζ be any admissible value ζ of the control signals. We call a configuration C a *ζ-configuration*, if during the CPU cycle beginning with C the control signals assume value ζ. Such a cycle will be called a *ζ-cycle*. Suppose the previous CPU cycle was a ζ'-cycle which ended at time t, the set of registers S was clocked and a ζ-configuration C was produced. We model how the outputs of some registers and gates become unstable and then stabilize again.

The outputs of all registers become stable at time t. Their value is determined by μ resp. ρ. If a gate G lies in $H(\zeta')$ and in $H(\zeta)$ and all paths into G lie in $H(\zeta')$ and in $H(\zeta)$ and

if all these paths come from registers not in S and if the outputs of G were stable at time t then the outputs of G remain stable and keep their old value. The outputs of all other gates in $H(\zeta)$ are considered unstable.

Let G be some ordinary gate in $H(\zeta)$. Consider the smallest $t' \geq t$ such that at time t' all inputs of G are stable or such that one input is stable and has a value which determines the value of the output of G independently of the other input, then the output of G becomes stable at time t' + delay (G).

Similarly let D be a driver with G(D) in $H(\zeta)$. Consider the smallest $t' \geq t$ such that at time t' all inputs of D are stable. Then the outputs of G(D) become stable at time t' + delay (D).

Finally we treat ROM resp. RAM Mi. Again consider the smallest $t' \geq t$ such that at time t' all inputs to the ROM resp. gate R(Mi) resp. W(Mi) are stable, then the outputs become stable at time t' + delay (Mi).

Let $\tau(\zeta)$ be the maximum value $t'-t$ taken over all ζ-configurations C such that at time t' the inputs of all registers which are clocked in ζ-cycles are stable and the previous cycle has finished at time t. Then the length of a ζ-cycle is definied to be $\tau(\zeta) + \delta$, where δ is a parameter of the technology. The parameter δ models setup and hold times of registers as well as time lost due to the fact, that clock edges are hardly ever where they are needed. It is tempting to abstract δ away resp. to set $\delta = 0$. However this allows to run pipelines with registers in transparent mode as fast as the corresponding circuit without registers inbetween. In this paper $\delta = 1 +$ delay (register) $= 5$.

A simpler model for timing would have been possible by simply considering a refined version of circuit depth. The data path of the PC (fig. 5) gives one example why we have chosen a more detailed model. The time of the "goto" instruction, PC =INR⟨0 : 23⟩, would be determined by the incrementer, although the outputs of the multiplexer stabilize much earlier. Many more examples exist.

2.3.6. Reset.

At time 0 all registers are clocked and assume value 0. $\varphi_{Mi}(x) = 0...0$ for all $i \geq 1$ and for all x. The outputs of all gates are considered unstable.

3. Modelling workload.

We do not have to invent anything. Models for workload exist and are called benchmarks. To model the workload we use here Dhrystone 1.0 [6]. The program has to be hand coded and analyzed carefully. This is very tedious. To keep the compilation and analysis more manageable, the special compiler sequences, as mentioned above, are used.

If one changes the architecture, one often has only to adapt these sequences. The results of such an analysis are given in table 5. The compiler does not use any optimization; the expressions are evaluated on a stack in M.

The benchmarks have one strong point in their favour: they are reasonably short and they are completely specified. In order to defend our choice of model we have to address the drawbacks - perceived or real - of benchmarks: i) there are as many benchmarks as there are buyers of machines and ii) benchmarking is an (obscure?) art. Clearly we believe that different users can produce very different workloads. If this is so, then these workloads *should* have different models. In real life we solve different problems on very different machines. But certain benchmarks, e.g. Livermoore Loops or Dhrystone are considered to be much more important than others. Benchmarks come in two flavors: *inner loops* and *synthetic*.

If a small number of lines of code eat up most of the CPU time of a user he may use these lines as a benchmark. This is the inner loops type. If the same lines or small variations thereof (what ever small means) are used by many users, then such a benchmark can gain importance. Such importance is inherently temporary and relative to the state of the art of architecture and of the algorithms.

For the construction of a syntetic benchmark one proceeds in two steps. First one gathers statistics on certain program parameters of programs written in a high level language. Second one constructs in a reasonably straight forward way a program which behaves in an average way with respect to the underlying parameters. In most environments except the labs of large manufactories gathering statistics makes serious difficulties of mainly two kinds: First, users often do not own the source code of the programs which they are running. Second, there are legal issues. It is by no means obvious that the following is legal for someone operating a computing center: for 1 day and 1 high level language copy all programs compiled and run that day, gather statistics by an automatic procedure then erase the copies.

	const.	global	scratch	local	stack	lo[A]	par[A]	indirect
load	354	15	139	446	178	2	36	72
store	%	5	144	64	272	0	36	161

jump	106
Op register	202
Op stack	88

add	sub	mul	div	&&	!	==
193	85	7	1	1	2	1

tab. 5 : amounts of the sequences in Dhrystone

4. The quality and a first theorem.

To evaluate and compare different architectures the *quality* of an architecture has to be defined. Depending on the point of view, the definitions differ very much. If performance is demanded at any price, quality is defined as the invers of the run time of the underlying benchmark. In most cases performance per cost are more important. The costs of an architecture are determined according to the cost function, introduced in an earlier chapter.

quality $Q \equiv$ performance / cost = 1 / cost * run time

time-depending cost function $TDC \equiv$ cost * run time = 1 / quality

A first theorem.

Let the technology parameters be fixed as: $\partial_a = 0.25$, $\partial_m = 0.1$, $\partial_s = 1$, α *and* γ *variable. For the architecture, fixed in earlier paragraphs, and the workload defined by Dhrystone it is better to execute multiplications per software than to use an additional Wallace-tree multiply unit.*

To prove the theorem we have to determine the costs and the run times for both architectures. There is no memory in the control logic or in the data path. The costs of the memory system M are not taken into account. The architecture A1 without any multiplier costs 2090 gate equivalents and the architecture A2 costs 4166,5 gate equivalents. The Wallace-tree multiplier is very expensive ($8210 * \partial_a$) and some more control logic is necessary. It takes three steps to deternime the run time. The execution-time has to be determined first for the machine instructions and then for the special code sequences. In most cases these times are identical for both architectures, only the multiplication takes different times. After that the run time is computed as a scalar product:

$$\text{runtime}(A) = \sum_{s:\text{special sequence}} a(s) \cdot r(s)$$

a(s) : amount of s in Dhrystone
r(s) : run time s

fetch	: ts + 7	store	: max(ts, 13) + 13
jump	: 26	move	: 26
add	: 36	load const	: 26
sub, negsub	: 37	load memory PC	: ts + 15
nand	: 26	load memory ACC, IR, B	: max(ts, 13) + 13
mul (A2)	: 51	mul (A 1)	: 372 ts + 218 max(ts, 13) + 1088

tab. 6: run times for the machine instructions ($ts = \gamma a$)

	load	store
global/scratch	ls + max(ls, 13) + 20	ls + max(ls, 13) + 20
constant	ls + 33	not defined
local	4 ls + 2 max(ls, 13) + 118	6 ls + 4 max(ls, 13) + 158
stack	6 ls + 3 max(ls, 13) + 152	8 ls + 5 max(ls, 13) + 209
local array	11 ls + 5 max(ls, 13) + 309	13 ls + 7 max(ls, 13) + 349
parameter array	14 ls + 8 max(ls, 13) + 369	16 ls + 10 max(ls, 13) + 411

compute on stack (without CPU-compute-cycle) : 22 ls + 13 max(ls, 13) + 549
&& : 5 ls + 221 == : 3 ls + 136 ! : 2 ls + 92 (CPU operations)

tab. 7 : run times for the special sequences

The results of the time-depending cost function show that the architecture A1 is the better one independent of the memory access time. The Wallace-tree multiply unit is very expensive and is used rarely under the given workload. Therefore it is plausible that this multiplier is not cost-effective. In reality some more registers and drivers would be appended rather than a very fast and expensive multiply unit.

TDC(A1) = 26,593,160 ls + 14,889,160 max(ls, 13) + 716,706,900
TDC(A2) = 40,615,042 ls + 22,415,770 max(ls, 13) + 1,094,231,229

Further theorems should deal with the question, how to make architecture A1 better, with respect to hardware as well as to compilers. As a first compiler optimization a push-operation could be eliminated if it is followed by a pop-operation. Registers for various purposes could be added. Also it might be useful to have a direct path to the ALU and the D bus from each register of the data path. This and some more aspects will be discussed in a later article.

5. References.

[1] Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice - Hall, Inc. 1977

[2] Loeckx, J., Mehlhorn, K., Wilhelm, R.: Grundlagen der Programmiersprachen. Teubner - Verlag. 1986

[3] Savage, J.E.: The Complexity of Computing. John Wiley & Sons, Inc. 1976

[4] Spaniol, O.: Arithmetik in Rechenanlagen. Teubner - Verlag. 1976, pp. 40-45

[5] Wegener, I.: The Complexity of Boolean Functions. Wiley - Teubner, Series in Computer Science 1987

[6] Weicker, R. P.: Dhrystone: A Synthetic Systems Programming Benchmark. Communications of the ACM 27, 10 (Oct. 1984), pp. 1013-1030