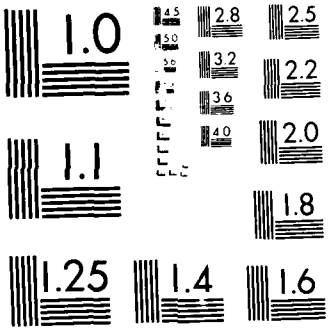
AD-A195 993 A MODEL FOR ARCHITECTURAL COMPARISON(U) WASHINGTON UNIV 1/1
SEATTLE DEPT OF COMPUTER SCIENCE S HO ET AL. APR 88
TR-88-04-01 MDA903-85-K-0072

UNCLASSIFIED F/G 12/6 NL



(2)

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 88-04-01	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Model for Architectural Comparison		5. TYPE OF REPORT & PERIOD COVERED Technical 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Sam Ho and Larry Snyder		8. CONTRACT OR GRANT NUMBER(s) MDA 903-85-K-0072 ARPA-4563, #2 code 5D30
9. PERFORMING ORGANIZATION NAME AND ADDRESS NW Laboratory for Integrated Systems Dept. of Computer Science, FR-35 University of Washington, Seattle, WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA - ISTO 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE April, 1988 13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR University of Washington 315 University District Building 1107NE 45th St., JD-16, Seattle, WA 98195		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Computer architecture, RISC, VAX, CISC, Crisp processor, Quarterhorse

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Recently, architectures for sequential computers have become a topic of much discussion and controversy. At the center of this storm is the Reduced Instruction Set Computer, or RISC, first described at Berkeley in 1980. While the merits of the RISC architecture cannot be ignored, its opponents have tried to do just that, while its proponents have expanded and frequently exaggerated them. This state of affairs has persisted to this day. This paper attempts not to settle the controversy, since there likely is no one answer, but to provide a quantitative framework for a discussion of the issues.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified.
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A Model for Architectural Comparison

Sam Ho and Larry Snyder

Department of Computer Science
University of Washington
Seattle, WA 98195

Technical Report 88-04-01

* Supported in part by DARPA under Contract MDA 903-85-K-0072 and a Minority Affairs Fellowship.

A Model for Architectural Comparison

Sam Ho and Larry Snyder [1]
University of Washington
Seattle, WA 98195

1 Light and Heat

Recently, architectures for sequential computers have become a topic of much discussion and controversy. At the center of this storm is the Reduced Instruction Set Computer, or RISC, first described at Berkeley in 1980. [P80] While the merits of the RISC architecture cannot be ignored, its opponents have tried to do just that, while its proponents have expanded and frequently exaggerated them. This state of affairs has persisted to this day. This paper attempts not to settle the controversy, since indeed there likely is no one answer, but to provide a quantitative framework for a rational discussion of the issues.

In this paper, we seek to shed some light on this topic. The model we present takes an architecture and a computation. It has the following features.

1. Quantitatively measures an architecture.
2. Examines an architecture working on a computation.
3. Separates the overall computation into logical pieces.
4. Determines from the architecture how long each piece takes.
5. Considers how much parallelism is available.
6. Compares the results with other architectures.

While the early Crisp processor [D87] and the IBM 801 [R82] embodied similarly small instruction sets, the controversy began with Patterson's paper, "The Case for a Reduced Instruction Set Computer." [PD80] In this paper, he extolled the virtues of a smaller, simpler instruction set, as opposed to the large and complex instruction set typified by the VAX architecture. Unfortunately, he also impugned the motives of the designers of such architectures, by suggesting marketing strategy to be the moving force behind the choice of instruction set. The designers of the VAX rose to the bait, and papers and articles of either persuasion began to pour out.

The problem with these arguments was that they were not speaking of the same things. Each writer, quite naturally, chose examples that most supported his own view. The opposing writers, then, chose different examples with different results. On top of that, the RISC I chip from Berkeley contained an essentially unrelated piece of hardware, that of multiple overlapping register sets. The

[1] Supported in part by DARPA MDA903-85-K-0072 and Minority Affairs Fellowship

early papers on RISC often combined the effects of the register set and the instruction set with little regard for their relationship, which was tenuous, at best. When the RISC I chip turned out to have an error that caused it to run extremely slowly, it provided no vindication for the proponents of the CISC, since the problem had nothing to do with the complexity of the instruction set.

The model presented here is an attempt to provide a common quantitative basis for a discussion of this and other architectural questions. It is important to remember that a model will not, and is not intended to, settle a question once and for all. Instead, given a task, and several processors differing in some of their parameters, it provides a numerical basis to compare the results with the results from other possible sets of parameters. This dependence on the example cannot be ignored, and reflects the truth that the performance of any system depends greatly on what it is being used for, as compared to what it was designed for. It is unreasonable to expect a Lisp machine to perform matrix inversions efficiently, or conversely, to expect a matrix processor to execute Lisp well.

Another key point is to keep the comparison simple. If there are two or more changes between the two architectures being compared, the results from the model will reflect their combined effect. This is fine, if the changes are closely related, but it will lead only to confusion if the changes are not related.

2 A Model of Computation

In this model, we will define what a unit of computation is, and how long those units take to execute. To start with, we need a computer to examine. The definitions below hold true for a general computer, with an arbitrary instruction set and hardware capability.

In some of our examples, we will use the QuarterHorse, a 32 bit microprocessor designed at the University of Washington that bears close resemblance to the Berkeley RISC, except that it is microprogrammed.

2.1 The Processor

The *Functional* units are the basic blocks of the datapath of the processor. That is, they are such things as registers, shifters, arithmetic units, multipliers and the like. They act on the data being processed in the processor in some way. Registers do not, strictly speaking, act on the data, but the perform the "function" of making the data available to the other units. One way to change a processor is to change the functional units.

The *Control* portion of the processor determines what actions the functional units execute, and when they do so. On many commercial processors, this is a read-only memory containing microinstructions for a microprogram. On

other processors, particularly microprocessors, it is a collection of logic spread throughout the chip. The Berkeley RISC is one such processor.

Again, in any given comparison, only one of the functional units or the control should be changed. If there are several changes, it is difficult to separate the effects of the various changes. As an example, the RISC has a smaller instruction set than the VAX. In addition, the RISC has hardwired control, while the VAX is microprogrammed. On top of all that, the RISC has multiple register sets, but the VAX does not. With so many changes, it is not surprising that there is so much dispute on the merits of various aspects of the RISC.

2.2 The Calculation

Now that we have a computer, we need to give it a problem to work on. The problem (or family of problems) we choose will be called the *Calculation*. This choice is crucial to the evaluation we will perform, since the calculation determines which operations are likely to be executed, and worth optimizing for.

For example, computing a matrix product will bias the results towards processors which can multiply quickly, while running an operating system which does frequent block input will bias the results towards a processor with a block movement instruction. This bias is not inherently wrong. It is quite reasonable for any given processor to do well on some tasks but poorly on others. Indeed, it is a wise choice to design a processor with knowledge of its intended application, and to optimize its operation for a specific class of problems. A processor designed to do everything will likely not be outstanding in doing any one thing.

In the examples below, we will use a fairly generic arithmetic computation, that of computing the greatest common divisor of two integers, to illustrate the point.

From this calculation, a compiler generates a set of data dependencies, and the transformations of the data as they pass through a graph. That is, it generates a dataflow graph for the calculation, with each node of the graph corresponding to some action being performed on the data. How this graph is produced is more of a topic for writers of compilers, and will not be discussed in this model.

The computation of the greatest common divisor can be represented by a graph, where the nodes are such operations as subtractions and comparisons. Unfortunately, this example is somewhat too simple, as it does not allow any room for parallelism, in the form of branches in the graph.

1. *If the first number is larger than the second, exchange them.*
2. *Subtract the first number from the second.*
3. *If the difference is not zero, return to step one, using the first number and the difference. Otherwise, we have the answer.*

The steps of this algorithm correspond to the nodes of the graph. In each step, we perform one operation.

In the more general case, there would be steps that do not depend on each other. For these steps, it would not matter if one were executed first, or the other, or both simultaneously on a parallel processor.

We shall restrict ourselves to sequential processors, and so, where the calculation does not determine an ordering, we shall impose one. The linear arrangement of actions will be called the *Sequence*, denoted S. Construction of the sequence is also commonly done by the compiler.

Now that we have defined the sequence of actions to take in performing a calculation, we need to define these actions.

In the GCD example above, the natural set of actions would be subtraction, exchange, and conditional branch, along with the more housekeeping activities of operand and instruction fetches, decode, and operand storage. These are simple actions because this is a simple calculation.

Since our model concerns itself with how long these actions take to execute on various processors, we must choose the actions carefully. If the actions are too small, larger-scale optimizations will affect sequences of actions instead of single actions, and we will not be able to model them. On the other hand, actions that are too large mean that we will have an unmanageable number of actions, each of which is affected in the same way by the same change.

2.3 Action

We define the *Action* to be the fundamental unit of computation. What it is is restricted by the conditions above, and by the experiment we are examining. For example, if we are considering the effect of a multiplier on the processor, multiplication should certainly be an action. However, if we are considering the effect of overlapping register files, there is no compelling reason to make multiplication an action.

When we have decided on the set of actions we will refer to it as C.

In the example above, we describe the second half of the set of actions as housekeeping. This distinction is worth noting, since we do not want to have the processor spend all its time on such ancillary activities, but rather on actions with some bearing on the calculation.

The *Overhead* actions, which we will denote as C_O, are just those actions which do not contribute to the calculation, but need to be executed anyway. These are usually instruction fetches, decodes and the like.

The remaining actions are *Computational*. These actions, denoted C_C, are actually part of the sequence associated with the calculation. In general, arithmetic and comparisons will fall into this category.

In addition, even among computational actions some of them are *Wasted*. These actions are computations that do not do anything towards the overall calculation, but are executed anyway, for lack of anything better to do, or

because a limitation in the instruction set requires its execution in order to perform some other activity.

In RISCs, such problems are generally caused by a limited overall clock structure which allocates time for an arithmetic operation whether it is wanted or not, and having the ALU to serve as the only channel between the input and output ports of the registers. An example is the RISC I, which insists on an addition when a register to register move is desired.

The remainder, and hopefully majority of the actions are *Useful*. These are the actions that do contribute to the calculation.

2.4 Time

And now, we need to see how long it takes for a processor to execute these actions. We measure time in basic units, which are *Cycles* of a master clock. This virtual clock does not necessarily correspond to the actual system clock of the processor, because some processors, particularly those with hardwired control, divide the incoming clock into many parts for different actions, while other processors, to prevent race conditions, use various arrangements of several clocks in what is really one cycle.

On microprogrammed processors, it is easier to determine what a cycle is, since the clock rate of the microprogram is generally the right measure for a cycle. The QuarterHorse falls into this category.

The key to the model is the mapping of each action to the number of cycles it needs to execute. We assign to each action a fixed cost which is the time, in cycles, needed to execute it. Generally, this will be a small integer, such as one or two, but it could be quite large, for more complex actions, such as multiplication, or of medium size, for actions of intermediate complexity. It is also quite common to have the carry chain for additions and subtractions take a slightly larger number of cycles than logical operation which do not involve a carry.

Finally, we can multiply the cycle by the *Clock Rate*, the time needed for one cycle, to get the time. In most cases, we can ignore this step, since the architectures in the experiment will have the same clock rate, but sometimes the clock rate is noticeably affected by the architecture. As an example, proponents of the RISC frequently claim that adding instructions to an architecture will slow it down. While the model cannot verify this claim, if it is true and measurable, we can take it into account.

2.5 The Interesting subset

Since the differences between the processors in any given experiment should be small, to avoid the mixing of effects warned about earlier, the majority of the actions will have identical numbers of cycles in their implementations in the two processors. To reduce the difficulty of computing the effects of the change, and

to see how much of the calculation is affected at all by the modifications, we can separate the actions into two categories.

The *Interesting* actions are those which are in some way different between the processors. The remaining, unchanged, actions are called *Common*. Then, the fraction of the cycles attributed to interesting actions is a measure of how much the modification affects the processor. Furthermore, the change in the overall time of the calculation is equal to the fractional change for the interesting actions multiplied by the share the interesting actions have in the overall computation.

2.6 Parallelism

While we are discussing sequential processors, we cannot entirely ignore parallelism. In particular, at the level of the action, even sequential processors allow some parallelism. This is, for example, why many processors have two data buses, allowing two operands to be simultaneously fetched, or even three buses, allowing yet another operand to be stored at the same time. Alternatively, actions that do not depend on each other can also proceed in parallel. This type of parallelism typically occurs between the instruction stream and the execution stream, in the form of prefetch buffers.

The *Maximum Parallel Set* of a processor is the set, P, of all the actions it can perform simultaneously. More strictly, $P = C_1 \times \ldots \times C_n$, the Cartesian product of several sets of actions. This notation means that in any given cycle, each one of the C_i can have an action in progress. The *Maximum Parallelism* is n, the largest number of actions that could conceivably be executing at one time. Sometimes, we will also use this cross product notation to denote cases where we want to point out that one specific group of actions that may execute in parallel with the rest. In such cases, the C_i may themselves have further structure. When that is the case, the maximum parallelism is the dimension of the maximal parallel set, and not just the visible portion with which we concern ourselves.

3 Derived Numbers

3.1 Derived Numbers in Time

Now that we have a model of what the processor is doing, we can combine this with the calculation the processor is performing to get quantities that reflect on the time the processor is taking.

The *Length* of the calculation is the number of cycles needed for the processor to perform the calculation. This is not quite adding up the actions performed in the calculation and multiplying by the appropriate number of cycles for each action, because we have actions that can proceed in parallel. During such times, we can, in the time of one cycle, be working on more than one action.

The *Time* of the calculation is the length, which is expressed in cycles, of the calculation multiplied by the cycle rate, giving a value in units of time. Clearly, the less time it takes for the calculation, the faster the processor. In most cases, we will deal with the length, rather than the time, of a calculation, and assume that whatever the clock rate is, it is the same for the processors we examine.

The *Efficiency* of the processor is the ratio of the length of the calculation to the number of actions in the sequence of the calculation. That is, it is the number of cycles needed to execute, on average, one useful action. This number is somewhat misleading, since it depends on what we chose as the actions, but among processors within one comparison, where the set of actions is the same, it provides a measure of how well the processor is doing, which is normalized for the size of the various calculations within one family.

3.2 Derived Classes of Processors

Within the general framework of processors defined previously, we can point out several interesting types of processors.

Any description of the controversy between the RISC and the CISC would be incomplete without an attempt at defining the differences between them. These differences, however, do not all fit within the realm of this model. In particular, the issue of the complexity of the control is difficult to quantify. Thus, the question of whether design time and chip area would be better spent on some other optimization is left unanswered. Also, we cannot determine how much, if at all, faster the clock rate of a RISC would be, compared to the CISC implemented in equivalent technology, although if we accept the claims of some other person as to what this difference is, we could incorporate it into the model.

3.2.1 RISC and CISC

A *RISC* is an architecture with few actions, either overhead or computational, in each instruction, and a small number of total instructions. To some extent, whether a processor can be considered to be a RISC depends on the calculation, since the calculation determines the choice of actions. As an example, an instruction with floating point support in its instruction set would be a RISC if the calculation performed floating point computation, making the floating point operations a basic action. If the calculation did not perform such operations, the processor would then be carrying considerable excess baggage, and would be harder to justify as a RISC.

A *CISC*, by contrast is the opposite of a RISC. This processor has more instructions, and each of which performs more actions. The usual example of this is the VAX, which has over two hundred instructions, performing such varied tasks as manipulation of doubly-linked lists, and about a dozen addressing modes, which may involve up to three memory references and two additions and

shifts for each operand fetched. This means that each instruction invokes more computational, but also more overhead, actions.

We earlier defined the classes of possible parallelism. Now, we point out some common types of parallelism within a processor. These are, by no means, the only ways in which parallelism is possible, but they are the ones most frequently used in what are basically sequential processors.

3.2.2 Instruction Prefetch

Instruction Prefetch is the technique of fetching the next instruction to be executed while the current instruction is still executing. Symbolically, if we call C_{fetch} the set containing the action or actions necessary for an instruction fetch, and C_{other} the set containing the remainder of the possible actions, then the maximum parallel set $P = C_{fetch} \times C_{other}$. In such a processor, we can effectively discount the time necessary for instruction fetches from the overall time of the calculation. The RISC processor fetches one instruction ahead, while more complex processors, such as the VAX, typically fetch several bytes ahead, and provide them as needed to the instruction decoder.

3.2.3 Pipelining

Pipelining is the technique of sequentially partitioning the actions in an instruction into several classes, each of which can execute independently, but in order. Each step requires the same number of cycles to execute, so that several instructions in various stages of execution can be simultaneously processed.

The MIPS processor is pipelined. This processor divides its instruction processing into three stages. The first is instruction fetch and decode, the second operand decode, execution and store, and the third operand load. The processor is arranged so that one action from each of these classes can be executing at any given cycle. That is, if we call the first class C_{fetch}, the second C_{ex}, and the third C_{load}, the parallel set $P = C_{fetch} \times C_{ex} \times C_{load}$.

In the more general case, there can be many stages in the pipeline, and many instructions can be simultaneously in their respective states of execution.

4 Some example results

4.1 Block movement

A *Block Move* is the copying of a significant quantity of data from one location to another, with minimal change. In applications such as operating systems, block moves frequently result from the transfer of information from a buffered device to the user's address space. For such devices as disk and tape drives, this can mean moving a thousand bytes of data at a time. In applications such

as these, where such block movement is common, having such an instruction available will save a considerable amount of time.

In Clark and Levy's paper, [CL82] where the example instruction load consisted of an interactive operating system, such moves occurred frequently. Here, (Multiuser/All modes) the highest ranked, by time, instruction was the MOVC3 instruction, which is the block move instruction, consuming 13 percent of the total time. By frequency of occurrence, however, it was less than one percent of the instructions. In fact, in this benchmark, the average block move was of 20 words.

To analyze the costs of implementing the move in various ways, let us assume that each instruction executed incurs one overhead action for the decode, and each word moved incurs one computational action. Furthermore, subtraction and branch are also each computational actions. We assume that prefetching hides the cost of instruction fetch, except after a branch. Further, let all of these actions require one cycle each to execute.

In the architecture with such a block move instruction, we see that a block move of 20 words is implemented with a single instruction. This instruction incurs one decode and twenty moves, for a total of 21 cycles.

$$Time = (Decode + 20 * Move) * 1\frac{Cycle}{Action} = 21$$

Now let us consider the architecture without a block move. If the equivalent operation were implemented by unrolling the instruction into a number of individual moves, we then have twenty decode actions as well as twenty move actions. In a strictly sequential machine, except for the instruction prefetch, we would then require 40 cycles to do the same thing.

$$Time = 20 * (Decode + Move) * 1\frac{Cycle}{Action} = 40$$

However, the decode of one instruction can go in parallel with the move of the previous word. In this case, we have the pipeline $(Prefetch \rightarrow) Decode \rightarrow Execute$. Here, except for the first word, on each of the twenty following cycles, we complete one move at the same time as the decode for the next move, for a time of 21 cycles. In this case, the pipelining allows the block move to run as fast as it would if there were a special instruction for it. Even so, there is still a twenty-to-one space penalty for the unrolled instructions versus the single instruction.

$$Time = (Decode + 19 * (Decode \wedge Move) + Move) = 21$$

If, instead, the move were implemented as a tight loop, the space penalty would be minimal, but there would be a time penalty. In this case, let us examine the replacement sequence

- Move word

- Decrement counter
- Jump if not zero

In this case, even assuming that the branch at the end disrupts the pipeline only for the loop exit, we see that each time through the loop requires three cycles, and outside the loop we have a cycle for decode at the beginning, and two cycles for fetching and decoding the instruction after the loop, to repair the pipeline disruption. This makes a total time of 63 cycles. Here, pipelining cannot make up for the loss of the special instruction.

$$Loop = Move + Decrement + Jump = 3.$$

$$Time = Decode + 20 * Loop + 2 * Repair = 63.$$

4.2 Operand Address Modes

One of the characteristics of a RISC is the small number of address modes. Most RISCs have a load-store architecture, in which fetching data from external memory is separate from the computation involving that data. These are instruction sequences of the form *Load A* followed by *Add A (from register)*. The alternative is called memory-to-memory. Here, the computational instruction is allowed to reference data from memory directly: *AddM A (from memory)*. If we make the assumption that the two instructions *Load* and *Add* require one more cycle of overhead than the single instruction *AddM*, we discover that the load-store machine requires one extra cycle each time on operand is referenced from memory. If we let L_{MM} be the length of the calculation on the memory-to-memory machine, and L_{LS} be the length of the same calculation on the load-store machine, and N_S be the number of operands fetched from memory, then we find that $L_{LS} = L_{MM} + N_S$.

In Wiecek's paper, [W82] just over 55 percent of the total operand references are to immediate or register data, and thus do not reference memory. Almost all the rest refer to memory once, with virtually none referring to memory twice. Since this same paper also indicates that 1.74 operands are referenced in the average instruction, multiplying this by the 45 percent of operands that reference memory produces 0.96 data memory references in each instruction. If, as we stated above, each such reference costs an additional cycle in the load-store machine, this means that having these memory reference modes saves almost one cycle for each instruction.

4.3 Floating point

Another case where a few instructions can make a major difference in the computation is that of floating point operations. Floating point operations require a large amount of time to execute, so many processors have special hardware

to assist in this computation. In such processors, the control of the processor has little bearing on this portion of the calculation, since we have a functional unit to do all of the work. If we give up this functional unit, we will need to emulate floating point operations with an equivalent series of additions, shifts and other operations available to us with the functional units we chose. Typically, just these actions will require on the order of ten times as much time as a well-designed floating point coprocessor.

In the system described by Clark and Levy, [CL82] which has a floating point processor, the time spent in floating point multiplication, T_{FP} is 3.5 percent of the total time used in the workload. The remaining instructions, then, account for the other 96.5 percent T_{other} of the time.

$$Time = T_{FP} + T_{other} = 3.5 + 96.5 = 100.$$

If we say that deleting the floating point hardware assistance will increase this time by a factor of ten, that increases the time fraction of the instruction from 3.5 percent to 35 percent of the original computation. The other instructions, of course, are unchanged.

$$Time = T_{FP} + T_{other} = (10 * 3.5) + 96.5 = 131.5$$

If we take our previous assumption that an unpipelined RISC will break each action into a separate instruction, each with one cycle of overhead, as well as the cycle of computation, we now have ten cycles for overhead, as well as ten cycles of computation for the equivalent floating point operation.

$$Time = T_{FP} + T_{other} = (20 * 3.5) + 96.5 = 166.5$$

With these assumptions, using a RISC processor for an unpipelined machine without hardware floating point support more than doubles the cost of not having such support, as compared with a machine with single instructions for floating point operations.

4.4 The CRISP processor: an example architecture

The Crisp processor was designed at Bell Laboratories for the efficient execution of C programs. It embodies many of the same ideas the RISC processor from Berkeley. However, it includes some special features, which will will describe briefly below.

The first idea is the stack cache. This cache allows dynamic choice of which variable to keep locally and which to keep in external memory. The idea is that the most frequently referenced variables will be at the top of the stack, and thus in fast registers, while less often used material is in external storage

Here, based on numbers from the Crisp paper "Register Allocation for Free", are numbers for the register activity on a VAX. The basic numbers are 0.77

memory references actions per instruction for the standard VAX, 1.34 for a VAX without registers, and 0.24 for a VAX with Stack Cache registers. These then are divided by the ratio of memory reference time to overall time to produce a number for the time saved. That is, if we let N be the number of instructions, and N_M be the number of memory references, for the standard VAX, $N_M = 0.77N$, for a VAX making no use of registers, $N_M = 1.34N$, and for the Stack Cache version of the processor, $N_M = 0.24N$.

By contrast, the Wiecek paper gives 1.18 memory references per instruction and 1.8 register references per instruction. This results in total data references of 3.0 per instruction. Note here that the register usage count includes references to FP and AP, the frame and argument pointers. Another count, that of operands per instruction, gives a total of 1.8 operands per instruction, some of which, reference both memory and registers one or more times. These are the displacement operands, which reference both register and memory. Also, the displacement deferred operands refer to registers once, and memory twice. Adding the indexed modifier adds yet another reference for each of registers and memory, for a possible maximum of two register references and three memory references in a single instruction.

These differences can, in part, be attributed to differences in what is being measured. In fact, an accurate evaluation of branch folding requires the the measurement of memory usage be reasonable. Clearly, from these data, the instruction mix can greatly affect, here by a factor of two, the number of memory references in a processor.

In addition, even with the same instruction mix, determining which instructions can eliminate memory references by using a Stack cache can be difficult. The VAX does not distinguish in its instruction set which operands are stackable, and so, we must guess at which operands these are. In particular, the displacement operands are frequently used for all of stack, global, argument, local, and indirect operation. Sorting these apart is a major task.

Another aspect of the Crisp processor is branch folding. This optimization, which presupposes accurate branch prediction, allows branching in the preferred direction to be eliminated during the prefetch stage.

Here, this allows taken branches to be optimized from a typical one cycle to zero. Since branches occur every four instructions or so, in Wiecek, this can be a major saving of computation time.

Evaluating Branch folding is somewhat easier. In the simplest case, without using branch spreading, there is a saving of a cycle, or the entire fetch and execute pair in the pipeline, when a branch goes in the preferred direction. Thus we need only to count the number of branches executed and the fraction in the correct direction to get an estimate of the savings.

By applying these numbers to the one-cycle saving for each correctly predicted branch, we have another test for the value of folded, but unspread, branches.

The effect of branch spreading is similar to that of the earlier technique of

delayed branch. Both allow additional cycles to be saved if the branch and the comparison on which it relies can be separated by a few instructions to be executed unconditionally. Finding such instructions is the task of the compiler, and, as such, depends both on the details of the optimization in the compiler and the problem being examined. The RISC group at Berkeley has also published claims about the value of the delayed branch.

4.5 The RISC and the CISC

As we well know, this dichotomy has attracted much debate recently. This is an attempt to consider the claims of each side in the framework of the model. As such, it measures only those things which are within the view of the model, and not other effects we have seen claimed. Here we use the same simple model of the RISC we have used above. In this model, the RISC has exactly as many cycles of overhead as of computation, and they alternate. Furthermore, each instruction has exactly one computational action in it. In this case, the RISC will use two cycles for each useful action, since every computation is useful, and is accompanied by a cycle of overhead. Numerically, if N_U is the number of useful actions in the calculation, the number of wasted actions $N_W = 0$, so the total number of computational actions $N_C = N_U$. The number of overhead actions $N_O = N_C$, giving a total of $2N_U$ actions. We assumed that each action take one cycle, so there are $2N_U$ cycles to do N_U useful actions. This leaves the efficiency $E = 1/2$.

On the other hand, the CISC needs to have more cycles of overhead, but if it can also get more cycles of computation. Furthermore, some of the actions in the CISC may be wasted, if the instruction packages more actions than are necessary. Here, the efficiency is equal to the number of useful actions divided by the total number of cycles. This means that if all actions take one cycle, the total number of cycles L is equal to twice the number of computational cycles N_C, which is equal to the wasted actions N_W plus the useful actions N_U. $L = 2N_W + 2N_U$. The efficiency E is $\frac{L}{N_U}$.

5 Conclusion

This model is not an answer for all the arguments that have gone by in architectural discussion. Instead, it is a basis for fair and reasoned discussion. We have seen how to:

- Describe an architecture quantitatively.
- Identify where architectures differ.
- Remove irrelevant or unrelated factors.
- Separate logical actions from the implementation.

- Determine how much time each action consumes.

While it is true that honest differences of opinion will always remain, and all bias can never be removed, a clear separation of the parts of an architecture and their effects on overall performance will help prevent one aspect of a processor for erroneously receiving credit for what properly must be ascribed to some other part or effect. When there is a quantitative basis for evaluating architectures, is easier for discussions to shed more light and less heat.

6 References

1. Berenbaum, Ditzel, McLellan. The Hardware Architecture of the CRISP Microprocessor. *14th Symposium on Computer Architecture* June 1987.

2. Clark, D., Levy, H. Measurement and Analysis of Instruction Use in the VAX 11/780. *9th Symposium on Computer Architecture* April, 1982.

3. Colwell, Hitchcock, Jensen, Sprunt, Kollar. Computers, Complexity and Controversy. *Computer* Sep. 1985, p.8

4. Ditzel, D., McLellan, H. Register Allocation for Free: The C Machine Stack Cache. *Symposium on Architectural Support for Programming Languages and Operating Systems*. p.48 1982

5. Ditzel, D., McLellan, H. Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. *14th Symposium on Computer Architecture* June 1987.

6. Fitzpatrick, Foderaro, Katevenis, Landman, Patterson, Peek, Peshkess, Sequin, Sherburne, VanDyke. A RISCy approach to VLSI. *VLSI Design*. Fourth Quarter 1981.

7. Ho, Jinks, Knight, Schaad, Snyder, Tyagi, Yang. The QuarterHorse: A Case Study in Rapid Prototyping of a 32-bit Microprocessor Chip. *IEEE International Conference on Computer Design: Very Large Scale Integration*, p.161, 1985.

8. Katevenis, Sherburne, Patterson, Sequin. The RISC II Micro-Architecture. *VLSI 83*.

9. Patterson, D. A RISCy Approach to Computer Design. *COMPCON Spring*. 1982

10. Patterson, D. Reduced Instruction Set Computers. *Communications of the ACM* 28,1. 1985.

11. Patterson, D., Ditzel, D. The Case for the Reduced Instruction Set Computer. *Computer Architecture News* 9,3 p.25 1980

12. Patterson, D., Sequin, C. RISC I: A Reduced Instruction Set VLSI Computer. *8th Symposium on Computer Architecture*, p.443 1981

13. Przybylski, Gross, Hennessy, Jouppi, Rowen. Organization and VLSI Implementation of MIPS. *Journal of VLSI and Computer Systems* 1,2. 1984

14. Radin, G. The 801 Minicomputer. *Symposium on Architectural Support for Programming Languages and Operating Systems*. p.39 1982

15. Wiecek, C. A Case Study of VAX-11 Instruction Set Usage for Compiler Execution. *Symposium on Architectural Support for Programming Languages and Operating Systems*. March 1982.

END

DATE

FILMED

9-88

DTIC