Butler Lampson

Currently a senior engineer at the Systems Research Center of Digital Equipment Corporation in Palo Alto, California, Butler Lampson was an associate professor of computer science at the University of California, Berkeley, a founder of the Berkeley Computer Corporation, and a senior research fellow at Xerox PARC's Computer Science Laboratory. Lampson's many accomplishments in so many areas of computer design and research make him one of the most highly regarded professionals in the field. He has worked on hardware systems, such as the Ethernet local network and Alto and the Dorado personal computers; operating systems, such as the SDS 940 and Alto; programming languages, such as LISP and Mesa; application programs, such as the Bravo editor and the Star office system; and network servers, such as the Dover printer and the Grapevine mail system.

I met Butler Lampson in Palo Alto at the offices of Digital Equipment Corporation where he works one week out of a six-week cycle; the other five weeks he works in Philadelphia. He is what he calls a "tele-commuter," doing much of his work via telecommunication lines. Unlike so many others in this fast-paced, quickly growing industry, Butler Lampson doesn't exhibit many entrepreneurial interests. His focus is singular: He is concerned with the successful design of a computer system, whether it be hardware, software applications, languages, or networks. Lampson writes very little source code today, he is a system designer, the person with the vision and expertise who lays the groundwork for a complex system. And he is undoubtedly one of the best.

INTERVIEWER: What attracts you to computers?

LAMPSON: I think the computer is the world's greatest toy. You can invent wonderful things and actually make them happen.

INTERVIEWER: But you can do that in other fields too ….

LAMPSON: It's much, much harder to make things happen in other fields. If you're a physicist, you have to live with what nature has provided. But with computer science, you invent whatever you want. It's like mathematics, except what you invent is tangible.

INTERVIEWER: Is your background in math and physics?

LAMPSON: I studied physics at Harvard. Toward the end of my studies I did quite a bit of programming for a physics professor who wanted to analyze spark-chamber photographs on a PDP-1. When I went to Berkeley to continue studying physics, a very interesting computer research project was going on, but it was well concealed. I found out about it from a friend at a computer conference I attended in San Francisco. He asked me how this project was doing. When I said I'd never heard of it, he told me which unmarked door to go through to find it.

INTERVIEWER: And what lurked behind this door?

LAMPSON: It was the development of one of the first commercial timesharing systems, the SDS 940.

INTERVIEWER: Did you get involved with the project?

LAMPSON: Deeply. I eventually gave up physics, because I found computer work a lot more interesting. That was fortunate because, if I had stayed in physics, I would have gotten my Ph.D. right around the time of the great crash for physics Ph.D.s.

INTERVIEWER: So you walked through an unmarked door into a concealed computer project and changed from physicist to computer scientist? Apart from programming on the PDP-1, did you have any earlier contact with computers?

LAMPSON: Yes, a friend and I did some hacking on an IBM 650 while I was still in high school. The 650, which was the first business computer, was nearing the end of its useful life, so there wasn't much demand for time on it.

INTERVIEWER: You've studied a range of sciences. Do you see an overlap between physics, mathematics, and computer science?

LAMPSON: Only in the sense that physics and mathematics, like other respectable disciplines, require that you think clearly to succeed in them. That's why many successful computer people come from these fields. It's harder to do what people do nowadays-start in computer science and stay in it because it's a very shallow discipline. It doesn't really force you to exercise your intellectual capabilities enough.

INTERVIEWER: Do you think the field is shallow because it's so primitive and young?

LAMPSON: That's the main reason. There seems to be some evidence of it becoming less shallow, but it's a slow process.

INTERVIEWER: Do you consider computer science to be on the same level as physics and mathematics?

LAMPSON: I used to think that undergraduate computer-science education was bad, and that it should be outlawed. Recently I realized that position isn't reasonable. An undergraduate degree in computer science is a perfectly respectable professional degree, just like electrical engineering or business administration. But I do think it's a serious mistake to take an undergraduate degree in computer science if you intend to study it in graduate school.

INTERVIEWER: Why?

LAMPSON: Because most of what you learn won't have any long-term significance. You won't learn new ways of using your mind, which does you more good than learning the details of how to write a compiler, which is what you're likely to get from undergraduate computer science. I think the world would be much better off if all the graduate computer-science departments would get together and agree not to accept anybody with a bachelor's degree in computer science. Those people should be required to take a remedial year to learn something respectable like mathematics or history, before going on to graduate-level computer science. However, I don't see that happening.

INTERVIEWER: What kind of training or type of thought leads to the greatest productivity in the computer field?

LAMPSON: From mathematics, you learn logical reasoning. You also learn what it means to prove something, as well as how to handle abstract essentials. From an experimental science such as physics, or from the humanities, you learn how to make connections in the real world by applying these abstractions.

INTERVIEWER: Like many influential programmers, in the early seventies you spent time at the Xerox PARC think tank, surrounded by great minds. Was that an inspiring time for you ?

LAMPSON: It was great. We felt as though we were conquering the world.

INTERVIEWER: Did any of your colleagues from that time influence your thinking?

LAMPSON: We all influenced each other. There was a lot of give and take. Bob Taylor's influence was very important. It was a combination of how he ran the laboratory and his consistent view of the ways in which computers are important.

INTERVIEWER: Even though Xerox created a think tank of computer experts, they failed to implement and bring to market many of the ideas. Did that disappoint you; did you think the world wasn't ready for those products?

LAMPSON: It's always hard to know what people are ready for. Were we aware of the outside world? Yes, we knew that it existed. Did we understand the whole situation perfectly? Probably not. Were we surprised when Xerox was unable to sell Stars? No, not really.

My view of the whole enterprise at Xerox PARC was that we couldn’t expect something like that to last forever. It lasted almost 15 years, so that was pretty good.

The purpose of PARC was to learn. You owe something to the company that’s paying you to learn, and we felt we should do what we could, within reasonable bounds, to benefit Xerox. But it wasn’t critical that Xerox develop those ideas. Their failure was not really surprising, because they were trying to get into a new business that nobody knew much about. There were many ways for things to go wrong. Some things went wrong in marketing–the quality of the technical people was high, but they never got the quality of marketing people the company needed.

For example, Bob Sproull and I spent a lot of time designing this project called Interpress, which was a printing standard. I put a lot of energy into that, and I would have really liked Xerox to take it and make everybody adopt it. Instead, they completely screwed it up. As a result, some of the people who worked on it went off and started Adobe Systems, and invented a similar product called PostScript, which is clearly a standard everyone will now adopt. That sort of thing is annoying, but the main product of a research laboratory is ideas.

INTERVIEWER: Are there any research labs devoted to ideas today?

LAMPSON: I have my own biases on that. The best research places are Digital Equipment Company, where I am now, and Bell Labs.

INTERVIEWER: Do you feel that research has a practical limit?

LAMPSON: I think it is unlikely that expert systems are going to work. In the early seventies, it was clear that the projects we were involved in then could be made to work. At least, I couldn’t see any fundamental reasons why they should not work, whereas now I can see a lot of fundamental reasons why artificial-intelligence systems are unlikely to work. It seems that some people have taken a very small number of experiments, which often have very ambiguous outcomes, and have made generalizations about those results in a totally crazy way.

INTERVIEWER: Why do you think people are so fascinated by the idea of artificial intelligence?

LAMPSON: Well, I’m not sure. Part of it is the basic computer fallacy that states the computer is a universal engine that can do anything. If there is no obvious way to show that something is impossible, then some people assume it must be possible. A lot of people don’t understand what the consequences of complexity are, and without that understanding they are likely to get burned. If they are not willing to take the word of someone who has gotten burned, then the only way they are going to find out is to try it and get burned themselves. Very few of the people who are excited about artificial intelligence have tried it.

I see really extreme versions of this. For instance, the Defense Advanced Research Projects Agency is funding a program that is supposed to use all of these wonderful expert systems and AI techniques, as well as parallel computing, to produce truly wonderful military devices, such as robot tanks. They published a ten-year plan that included a foldout showing lines of development and milestones when breakthroughs would be made. It's all nonsense because no one knows how to do these things. Some of the problems may be solved within the next ten years—but to have a schedule! The world doesn't work that way. If you don't know the answers to the problems, you can't schedule when you're going to finish the project.

INTERVIEWER: You seem to scorn complexity. When you design a system, do you strive for simplicity?

LAMPSON: Right. Everything should be made as simple as possible. But to do that you have to master complexity.

INTERVIEWER: In practical terms, how do you achieve that?

LAMPSON: There are some basic techniques to control complexity. Fundamentally, I divide and conquer, break things down, and try to write reasonably precise descriptions of what each piece is supposed to do. That becomes a sketch of how to proceed. When you can't figure out how to write a spec, it's because you don't understand what's going on. Then you have two choices: Either back off to some other problem you do understand, or think harder.

Also, the description of the system shouldn't be too big. You may have to think about a big system in smaller pieces. It's somewhat like solving problems in mathematics: You can write books that are full of useful hints, but you can't give an *algorithm*.

INTERVIEWER: I know your experience with computers is broad. You've developed computers, operating systems, and applications. Does each of these require a different discipline?

LAMPSON: Obviously, if you want to develop applications, you need a fair amount of sensitivity about user interface, which is not as important in developing a piece of hardware. When you design hardware, typically you're much more concerned with the arbitrary constraints imposed by the particular technology that you are working with. Nobody knows how to build really complicated hardware systems, so designing hardware tends to be simpler. Software is much more complicated.

INTERVIEWER: Do you still program?

LAMPSON: Only in an abstract sense. I no longer have the time to write actual programs. But I've spent the first half of this year writing a twenty-five page abstract program for a name server. To give you a sense of the relationship between an abstract program and a real program, the abstract has been translated into about seven thousand lines of Modula-2 code. So I cheat. I haven't done hall-time programming for about six or seven years. I did a lot of it before that.

INTERVIEWER: What are some of the more important programs that you wrote?

LAMPSON: I wrote sizeable chunks of the 940 operating system and I wrote two or three compilers for an interactive language for scientific and engineering calculations. I wrote a SNOBOL compiler. In addition, Peter Deutsch and I designed a programming language that was one of the predecessors to C and wrote a compiler for it. I've written design-automation programs, and another operating system that I did in the early seventies at Xerox.

INTERVIEWER: Is it easier for you to develop programs now than it was ten or fifteen years ago?

LAMPSON: Designing programs is probably harder now because the aspiration level is so much greater. But the actual programming is a lot easier now than it used to be. The machines have much more memory so you don't have to squeeze as hard. You can concentrate more on getting the job done and not worry about getting the most out of limited resources. That helps a lot. Also, the programming tools are somewhat better now. But, since I don't write code anymore, development is *much* easier for me.

INTERVIEWER: What sort of processes do you go through when you design or develop a program?

LAMPSON: Most of the time, a new program is a refinement, extension, generalization, or improvement of an existing program. It's really unusual to do something that's completely new. Usually I put into a new program an extension or improvement on something that's already a model in an existing program or user interface. For example, Bravo was the result of two ideas I had. One was about how to represent the text that was being edited inside the computer model and the other was how to efficiently update the screen. But the basic idea for Bravo came from a system called NLS that was done in the late sixties by Doug Engelbart at SRI. NLS provided a mouse-oriented, full-screen display of structured text. The confluence of those ideas led to the development of Bravo.

In the old days, I would scribble notes down on a piece of paper and start hacking, or get somebody else enthralled and start them hacking. Nowadays, I try to write the crucial parts of the idea in fairly precise, but abstract, language. Typically, there's a lot of iteration at that stage.

For example, I worked on a name-server project with Andrew Birrell and Mike Schroeder. At Xerox, they had built a system called Grapevine, which was a distributed name server. Grapevine could handle a few thousand names, and it started to break down as it grew toward the few-thousand limit. After we did that project, we had some idea of how a name server should hang together, but we wanted to handle a few billion names, which posed some significant design problems. I decided that its basic elements should be local databases, and each one would essentially implement a fairly standard tree-structured name scheme. And all of these databases would fit together in a loosely coupled way.

The operations were defined and programs were written on a fairly high level, without too much concern for detail. We ended up with a twenty-five or thirty-page combination of program and specifications. This served as a detailed design from which a summer student turned out the seven thousand lines of prototype implementation.

That's usually the process I follow. How long it takes depends on how difficult the problem is. Sometimes it takes years.

INTERVIEWER: Did the languages you developed take as much work?

LAMPSON: Sometimes it's quick: Peter Deutsch and I had the first version of one language going in about two months. On the other hand, I've been working with Rod Burstall on a kernel programming language that we work on two or three times a year. That project's been going on for about five years now and we still don't have an implementation. We keep changing our minds.

INTERVIEWER: Do you think there are certain techniques that lead to a good program or a good system?

LAMPSON: Yes. The most important goal is to define as precisely as possible the interfaces between the system and the rest of the world, as well as the interfaces between the major parts of the system itself. The biggest change in my design style over the years has been to put more and more emphasis on the problem to be solved and on finding techniques to define the interfaces precisely. That pays off tremendously, both in a better understanding of what the program really does, and in identification of the crucial parts. It also helps people understand how the system is put together. That's really the most important aspect of designing a program.

Designing a program is completely different from inventing algorithms. With algorithms, the game is to get the whole plan in your head and then shuffle all the pieces around until you settle on the best way to accomplish it. The trick here is to sufficiently define the algorithm so that you can get it completely in your head.

INTERVIEWER: When you design a system or a program, what makes you certain that it can be implemented?

LAMPSON: One possibility is to carry the design down to a level of programming where two things are true. First, I already know about the primitives I write the programming in — they have been implemented many times before, so I have a high degree of confidence that the program will work. Second, I understand the primitives well enough to estimate within a factor of two or three how much memory they are going to cost. I can then design my program in a sensible way. So I can be fairly confident that it's possible to implement a function, and I can roughly estimate its performance. Of course, it's possible to do that and overlook something very important. There's no way around that.

The other possibility is to very carefully and formally write down what properties the program should have and then convince yourself that it has them. You always do that in an informal way, but as you do it more formally, it gets to be a lot more work, and the chance of missing something important gets smaller. At some point, the formal analysis becomes counterproductive. But if you do too little of it, the risk of missing something important increases. If you don't find out about the missing element until the program's all coded, then a tremendous amount of work has been thrown away.

Of course, what usually happens is that you don't throw that work away; you try to patch it up. That has very bad consequences. First of all, you don't solve the problem. You just convert it into another problem, which isn't the one you should really be solving. And then you patch the program until it solves the other problem. That approach is very unsatisfactory.

Sometimes I think that the goals people are trying to reach are just too much to ask for. Programmers often lose sight of the fact that the problems in building software systems arise because what they are trying to do is just too hard. They believe the computer is a universal engine that can do anything. It's very easy to be seduced into the proposition that a group of one or five or ten or fifty or a thousand programmers can make the computer do anything. That's clearly not right.

INTERVIEWER: What qualities does a programmer need to write a successful program?

LAMPSON: The most important quality is the ability to organize the solution to the problem into a manageable structure, with each component specified in a simple way. Sometimes successful programmers can do that but can't explain what they did, because they can't see the structure. Some people are good programmers because they can handle many more details than most people. But there are a lot of disadvantages in selecting programmers for that reason–it can result in programs that no one else can maintain. That doesn't seem to happen so much anymore, because now it's more fashionable to have one person or group design the programs and then hire someone else to write all the code.

INTERVIEWER: Do you see a hazard in that approach?

LAMPSON: The hazard is that eventually the designer can lose touch with reality, and that leads to designs that can't be implemented.

INTERVIEWER: And have you ever designed programs that could not be implemented?

LAMPSON: No, I don't think so. The closest I've come to that was in the mid-seventies, at Xerox PARC. With several other people, I designed a file system that provided *transactions*. That's the ability to make several changes to stored data as an *atomic* operation: Either all the changes get made or none of them do. An example of a transaction is the transfer of money from an account at one bank to an account at a different bank. You don't want to leave the system in a state where the money was removed from one account but not added to the other. I worked a lot on an early design of that system, and it was built. It did work, but was generally judged to be quite unsatisfactory.

INTERVIEWER: Can you describe a beautiful program?

LAMPSON: I don't know if it's possible to answer that question. I can't tell you what a beautiful painting is, or a beautiful piece of music. Perhaps it's easier to describe a program because it is composed of diverse qualities; it is an engineering object as well as art.

A beautiful program is like a beautiful theorem: It does the job elegantly. It has a simple and perspicuous structure; people say, "Oh, yes. I see that's the way to do it."

INTERVIEWER: If you could draw a parallel between programming and another art, such as painting, writing, composing music, or sculpture, which one would it be?

LAMPSON: It would be better to choose architecture, which encompasses major engineering considerations. Very few programs can be regarded as pure art. A program is expected to do something and art is part of it. But when I say programming is an art, it's in the same way that mathematics is: People don't normally classify mathematics as an art.

INTERVIEWER: In what sense is computer science a science?

_L_AMPSON: The easiest answer is that it's a science to the same extent that mathematics is a science. A computer program can be viewed as a mathematical object. One way to understand such an object is to make abstract statements about it that you can prove. The only other way of understanding the object is trial and error, which doesn't work very well. If it is the least bit complicated, you can try for a long time and never find out what an object's properties are.

INTERVIEWER: Do you think the way that systems are designed and developed will radically change?

LAMPSON: Well, yes and no. What really makes design better is the development of higher levels of abstractions to work with. For instance, you can program a VisiCalc spreadsheet more easily than you can write a BASIC program to solve a certain type of problem.

On the other hand, our aspirations are constantly increasing, so the development of better abstractions doesn't make the task of programming much easier: It means we can do more elaborate things. We can do more because the primitives that we are using are much more powerful.

INTERVIEWER: Do you think the microcomputer and personal computer are part of a phase that will pass into something else?

_L_AMPSON: Of course not! I give a talk to general audiences titled The Computer Revolution Hasn't Happened Yet. Its basic theme is that major changes in the world take a long time. Look at the industrial revolution. It was at least sixty years or more from the time it started until it had a major impact on people's lives. That was true for the second industrial revolution as well. Telephones and electric lights were invented around 1880 but it wasn't until the twenties that their use became widespread.

I think this is true for the computer revolution. People like to think that we're moving much faster because we go from idea to finished product in six months. That's nonsense. Computing is just beginning to become a significant part of the economy and just starting to affect people's lives. Technical trends are going to increase at a furious rate, and the natural form for those trends is the so-called personal computer.

You can look at the trends in the base technology and see where it's going. It takes silicon, a rotating magnetic medium, a keyboard, and a display to make a computer. Each of those components is evolving so you can get more and more of what you want in a small box at a modest price. There's no indication any of that is going to change. I'm not saying there won't be any big computers, but there is an overwhelming trend toward personal computers.

INTERVIEWER: Where do you see the industry in five to ten years?

LAMPSON: The computer industry will increase at a good clip for at least another twenty years. There are innumerable other things that computers eventually will do that they're not doing currently because they're not quite cheap enough, fast enough, or people simply do not understand a problem well enough.

INTERVIEWER: Do you think the status of computer science or the computer industry in our society will change? Will computer scientists become like the physicists in the early twentieth century, making great breakthroughs that significantly change society?

LAMPSON: No. It will be nothing like the status of physicists in the early twentieth century. Most people didn't even know physicists existed until 1945, when they became well known because of the atom bomb. I don't think computers are dramatic enough to compare with that. But the saying, "The computer revolution hasn't yet begun," means that in twenty years there will be a computer on every fingernail. They will be pervasive in ways we can foresee, and in many ways which we can't. This will result in a tremendous change in the way the world works, just as the automobile resulted in a tremendous change. But it will also take a long time, just as the automobile took a long time to change society. In 1920 when the automobile was first introduced, it was very hard to predict the consequences. Some were obvious, but not all of them. This will be even more true with the computer because the changes it creates will be much more profound.

INTERVIEWER: Do you see any problems with a computer on every fingernail in twenty years?

LAMPSON: I don't see anything wrong with that. Certainly on every wrist. Computers are there to help and I hope they will do that in a positive way.

I recently read an interesting article about how encryption techniques could give individuals more control over how their personal data is disseminated. It is always hard to predict how these things will actually come out because a lot of political issues are involved as well as technical ones, but that article is an interesting illustration of how computer technology can give individuals more control over their lives. Even in situations where big organizations like banks are intimately and unavoidably involved in our lives, we can still find ways to give individuals far more control by using machines cleverly. Ten years ago nobody could have imagined that computers would give us more individual control. In fact, everybody thought the exact opposite would happen.

INTERVIEWER: What do you see as problem areas for the personal computers that exist today?

LAMPSON: Personal computers are fairly junky. I don't define that as a problem. They're new and people are learning about them, and they're getting better rapidly. Alan Kay made a great comment about the Mac—it was the first computer good enough to criticize. It makes sense for people who are building the next generation of computers or programs to think about what's wrong with the current ones, in order to make the next ones better.

That's why I think the idea of computer literacy is such a rotten one. By computer literacy I mean learning to use the current generation of BASIC and word-processing programs. That has nothing to do with reality. It's true that a lot of jobs now require BASIC programming, but the notion that BASIC is going to be fundamental to your ability to function in the information-processing society of the twenty-first century is complete balderdash. There probably won't be any BASIC in the twenty-first century.

INTERVIEWER: So how should we prepare ourselves for the future?

LAMPSON: To hell with computer literacy. It's absolutely ridiculous. Study mathematics. Learn to think. Read. Write. These things are of more enduring value. Learn how to prove theorems: A lot of evidence has accumulated over the centuries that suggests this skill is transferable to many other things. To study only BASIC programming is absurd.

INTERVIEWER: Is the industry being overrun by BASIC programmers?

LAMPSON: No, and I don't think there's anything particularly harmful about programming in BASIC. What is bad is that people get very worried and feel that their children won't have a future if they don't learn to program in BASIC. There's no reason for them to worry.

INTERVIEWER: But nobody knows for certain what skills will be required.

LAMPSON: Well, there's some truth to that, but we have some idea of the direction in which computer science is evolving. You can look at the systems that are built in research laboratories and get a feeling for it. You could have visited Xerox in 1975 and gotten a very good sense of what the high end of the personal computer world was going to be like in 1985.

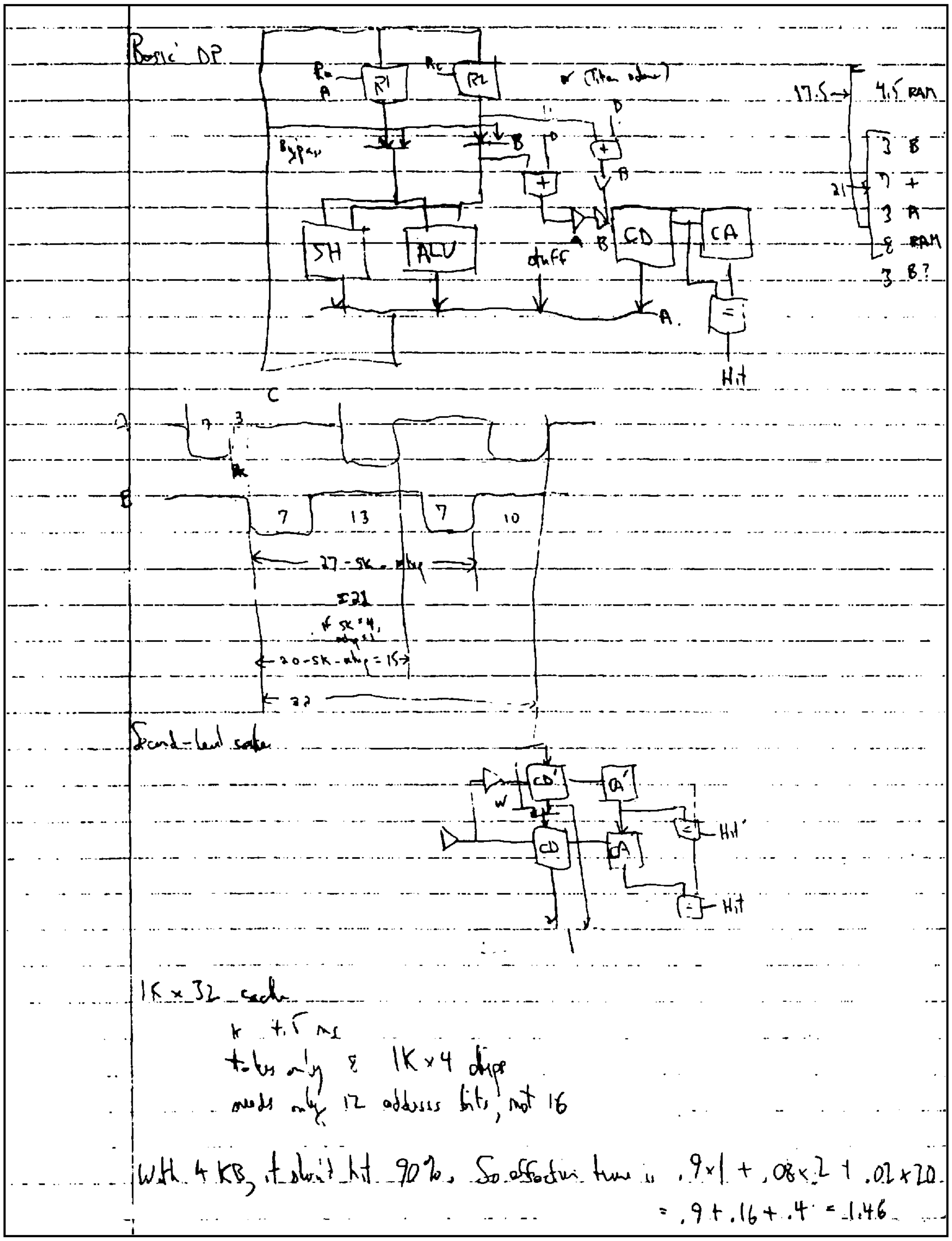
INTERVIEWER: Do you think we'll see the day when everyone will write their own programs?

LAMPSON: There are different degrees of programming. When the Smalltalk system was built, Alan Kay said one of his visions was that children would be able to use it and write interesting programs. But it didn't work that way. He said Smalltalk was like providing children with a lot of bricks: Children can build certain kinds of structures from bricks, but it's a very rare child that will be able to invent the arch.

If programming just means giving the computer instructions, I think everybody will do that at some level. Most business people operate a spreadsheet and that's programming in some sense. I think you'll see more of that. Creative programming is another matter.

INTERVIEWER: As a programmer, did you find that your work became an all-consuming part of your life?

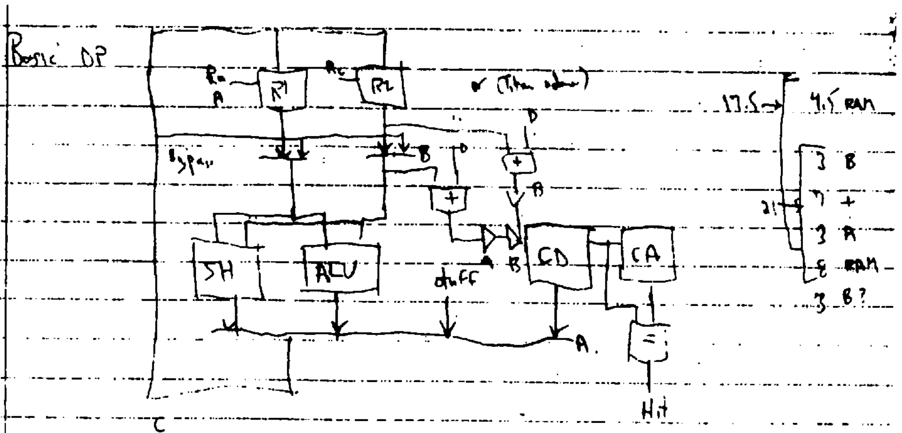
LAMPSON*:* I went through that stage earlier in life, but not anymore. I’m too old for that.

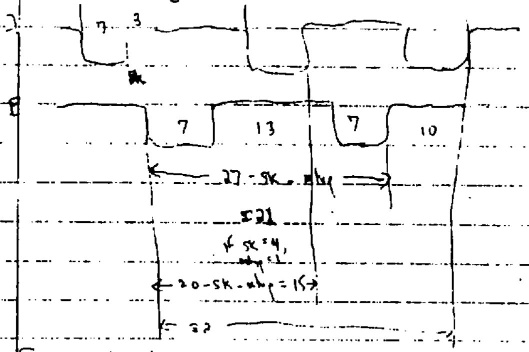


1K × 32 cache
is 4.5 ns
takes only 8 1K×4 chips
needs only 12 address bits, not 16

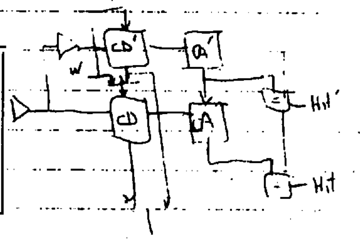
With 4 KB, it should hit 90%. So effective time is .9×1 + .08×2 + .02×20
= .9 + .16 + .4 = 1.46

Sketch by Butler Lampson of a design for a fast CPU with a two-level cache. See the Appendix (pages 373-382) for additional specifications for this CPU and other programs.





Second-level cache



This sketch from Butler Lampson shows his preliminary design for a fast CPU with a two-level cache along with performance calculations. The timing of the pipeline is worked out at the bottom of the second page.

1K × 32 cache

k 4.5 ns

takes only 8 1K×4 chips

needs only 12 address bits, not 16

With 4 KB, it should hit 90%. So effective time is $.9\times1 + .08\times2 + .02\times20$

$= .9 + .16 + .4 = 1.46$

Conclusion — 20 is much too much. We need another level.

Try $.9 \times 1 + .095 \times 4 + .005 \times 20$

$= .9 + .38 + .1 = 1.38$

Not much better

$.9 \times 1 + .08 \times 2 + .015 \times 5 + .005 \times 20$

$= .9 + .16 + .08 + .1 = 1.24$

Maybe the first scheme isn't so bad. Compare

$.8 \times 1 + .18 \times 2 + .02 \times 20$

$= .8 + .36 + .4 = 1.56$

Not much worse. The .8 is conservative from 3Cs 750 measurements (4K8 DM)

So ~ .5 cycles / cache ref will go into miss stalls.
On VAX we do 2/3 read and /3 write, roughly. So there are also per instruction.

For secondary cache. 1 MByte = 64K × 32 × 4. So 4 banks of 64b. If a 4ns clock is feasible, we can get a word every 12 ns from 128 chips. Shifting out the 4 words at 100 MHz would be a perfect rate for 128-bit ~~[illegible]~~ word. Humble

The pipeline is

R inst	A	8-10 ns		
R inst	B			
R reg	A	4.5 RAM, 3 bypass + latch		7.5/8
ALU / cache	B	4 add/latch 4.5 RAM	}	12.5/16 to allow 1 clock on [illegible]
ALU / cache	A	4 = /latch	}	There's lots more time for data
W reg	B	this is where we can stop it easily		

On a stall, the B clock gets stopped, so the cache address (latched by B after adding) holds its stalled value. The instruction register holds at the next instruction. The write from the stalled instruction is stopped. This all seems fine. I'm not clear on what would happen if we tried to delay the write another cycle, doing more bypassing, in order to [illegible] more time for the stall.