Chapter 1

Introduction, Focus, Overview

1.1 Introduction, Focus, Overview

Welcome to the world of computer architecture and the xxx pages that we hope to use to introduce you to its fundamentals. We chose the title for the book "Computer Architecture: Fundamentals, Tradeoffs, Challenges" to reflect three essential elements of Computer Architecture that transcend the field.

"Fundamentals" because we will focus on fundamentals in the belief that if you master the fundamentals, there is no limit to how high you can soar, given your intelligence and the energy you wish to devote to the task.

"Tradeoffs" because computer architecture is an engineering discipline, and engineering is all about tradeoffs. Throughout the book we will examine choices. It keeps coming up: Should we do A or should we do B. A is more expensive but gives us a better result. B's result is not as good, but costs a lot less. What is the cost and what is the benefit depends on what choice we are examining. For example, cost may be the amount of logic required, and benefit may be the time it takes the computer to carry out the task. Or, cost may be the number of hours spent on solving the problem and benefit may be the resulting execution time required, once the design has been completed. What is relevant is that computer engineering problems almost always come with tradeoffs, stated simply as what is the cost, what is the benefit, and the engineer is tasked with deciding how to handle the tradeoffs.

"Challenges" because computer architecture is a live, vibrant, continually evolving field which is always pointing ahead, creating solutions for new problems whose solutions will reflect the new resulting state of the field.

1.2 Organization of the Book

The book is broken down into 14 chapters, most dealing with one essential subdiscipline of computer architecture. This section introduces those 14 chapters, although we do not expect you to master them yet. Our objective here is to give you a map of the road we plan to travel, so you will have a better appreciation of where each chapter fits in the scheme of things when we get to that chapter.

1.2.1 Chapter 1. Introduction, Focus, and Overview.

In this chapter, we take a short look at each of the chapters. The objective is to give you a sense of what computer architecture is all about. Mastery will come later.

1.2.2 Chapter 2. The ISA

Chapter 2 deals with the Instruction Set Architecture (ISA), the interface between the hardware and software of a computer. The ISA is a specification which both the hardware and software understand. It allows the software to specify what it needs the hardware to do, and lets the hardware know what it

must do to carry out the work specified by the software. There are many ISAs, with many tradeoffs in their specifications. We will specify a few ISAs, and deal with the tradeoffs they incur.

1.2.3 Chapter 3,4. Microarchitecture

Chapter 3 introduces the most basic of microarchitectures of a computer, the logic structures required to carry out the work of the ISA in dealing with the requirements specified by the software. Here, too, are tradeoffs for us to deal with. Chapter 3 also introduces pipelining, which provides an opportunity for higher performance of the resulting computer (i.e., computer programs take less time to execute). Unfortunately, that benefit comes with a cost that needs to be solved to obtain that higher performance. We lump that cost into two problems, one caused by conditional branches, the other caused by the sequential order of execution of instructions in a program. Chapter 4 deals with those costs.

1.2.4 Chapter 5. Process

Chapter 5 introduces the notion of a process, the atomic unit of activity as seen by the operating system. Each process has its own priority (how urgent it is that the process needs to complete its work) and privilege (what the process has the right to do). Chapter 5 uses these notions to deal with interrupts and exceptions.

1.2.5 Chapters 6,7,8. Storage

Chapters 6,7,8 deal with storage, the actual physical memory locations that store instructions and data (Chapter 6), the virtual memory location each process has (Chapter 7), and the on-chip storage called Cache Memory (Chapter 8).

1.2.6 Chapter 9. Fixed Point Arithmetic

Chapter 9 deals with fixed point arithmetic, usually integers, but not always. The important point is that the binary point is fixed for every representable number. If the binary point is to the right of the least significant bit, each value is an integer. If the binary point is to the left of the most significant bit, every value is a fraction having value less than 1. If the binary point is somewhere in between these two extremes, each value has an integer part and a fractional part. What those values are depends on exactly where the binary point is. The chapter deals with addition and multiplication of values expressed as above. Long integers, Kogge-Stone adders, BCD Arithmetic, Booth's Algorithm, and Arithmetic with Residue Numbers are all included.

1.2.7 Chapter 10. Floating Point Arithmetic

Chapter 10 deals with floating point arithmetic of values expressed in what is often referred to as scientific notation. IEEE Arithmetic is discussed in detail. Rounding modes, excess codes for exponents, subnormal numbers, infinities, use of Not-a-Numbers (NaN), and floating point exceptions are all studied.

1.2.8 Chapter 11. Input/Output

Chapter 11 deals with the basic notions of I/O, and importantly the use of Asynchronous I/O. The basic notions include the time difference of executing an I/O instruction compared to an instruction carried out in the processor, the use of loads and stores to handle input and output rather than special I/O instructions, bus arbitration, and bus transactions.

1.2.9 Chapter 12. Single Thread Parallelism

Chapter 12 deals with parallelism while processing a single thread. Topics include pipelining, SIMD - both array processors and vector processors, two ways to do SIMD, Decoupled Access/Execute, HPS, and compute Data Flow.

1.2.10 Chapter 13. Multiple Thread Parallelism

Chapter 13 deals with parallelism with multiple threads. Topics covered include Amdahl's Law, Tightly and Loosely coupled multiprocessors, interconnection networks, cache coherence, and memory consistency.

1.2.11 Chapter 14. Pot Pourri (Other Stuff)

Chapter 14 is a catch-all chapter to cover other important topics that we decided we did not have enough pages to do them justice. Examples include Measurement methodology, GPUs, RISC technology, Accelerators, etc.

1.3 Once Over Lightly

Finally, before leaving chapter 1, we introduce a number of topics to give you an idea of what computer architecture is about, in some sense a preview of coming attractions.

1.3.1 The Computer System

We generally think of the hardware that makes up a computer system as consisting of three parts: the processor, the memory system, and the input and output devices.

The processor manages the computer system, processes the instructions, directs the access to information (load/store) from memory and I/O units, computes (operate instructions such as ADD, MUL) with functional units designed for that purpose, and maintains instruction flow (control instructions).

The memory system consists of main memory and zero or more levels of on-chip storage called caches. Access time to memory is very long (hundreds of cycles) so caches, which have much shorter access times usually require far less time to load or store information.

Input and output devices are sometimes referred to as peripherals since they are out of the usual processor/memory flow. Some I/O devices are very simple, like the keyboard (input) and monitor (output) parts of your laptop.

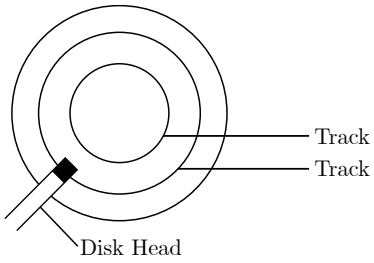


Figure 1.1: Disk Activity

Some are more complex such as a disk, which requires the long latency of moving the disk head until it is hovering above the track that contains the information one wishes to access. The track rotates until the disk head lines up with the start of the information desired. Then the desired information is either loaded from the track or stored on the track, depending on the direction of the transfer.

Some input/output devices are sufficiently rich in what they can accomplish that they are processors in their own right. They are often referred to as I/O processors.

1.3.2 The Transformation Hierarchy

It is the electrons moving from one voltage potential to another that actually solve computer problems, and if humans could speak the language of electrons, we could tell the electrons what we want done, and they could then do the job. Unfortunately, we cannot speak electron, so we are reduced to systematically transforming the statement of our problem from a natural language (like English, French, German, Chinese, Japanese, etc.) formulation to some Algorithmic language, which gets rid of the uglies (like ambiguity) of natural language.

From there the algorithmic representation is transformed into a computer program in a mechanical language (like C++, Java, Python, or perhaps LC-3b assembly language), which is then translated via a compiler, assembler, or interpreter to the 0s and 1s of the ISA. The microarchitecture understands the

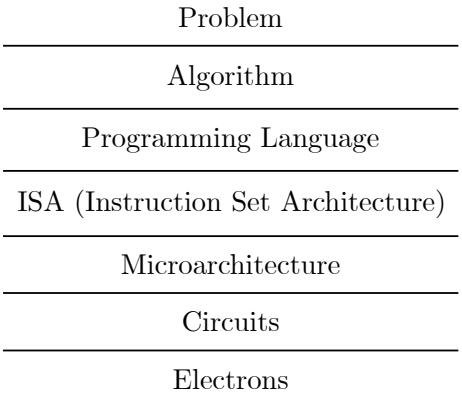


Figure 1.2: Transformation Hierarchy

0s and 1s of the ISA,and transforms the program into the logic gates that process the program, wherein each logic gate consists of electronic circuits that can move an electron from one potential to another thereby solving the problem.

At each stage of the hierarchy there are choices. For example, if the problem involves sorting, the Algorithm has more than a dozen choices, among them quick sort, heap sort, merge sort, insertion sort, bubble sort, etc.

The ISA provides another choice as to what the 0s and 1s mean. Intel's x86, IBM's POWER, Apple's use of ARM, etc. are all choices of ISA. And for each ISA there are multiple microarchitectures that implement that ISA. Thoroughly understanding the choices at each level of the Transformation hierarchy can provide an opportunity to combine them in a way that produces a higher performance engine.

1.3.3 Architecture and Microarchitecture

Architecture means ISA. Microarchitecture is an implementation that can process instructions that obey the specification of the ISA. The key difference between the two is visibility to the software. The ISA is the interface between the hardware and the software so clearly everything in the ISA is visible to the software: the address space, addressability, the set of opcodes, addressing modes, data types, instructions supporting multiprocessors (such as TSET), and instructions that support multiprogramming (such as LDCTX).

On the other hand, the microarchitecture is not visible to the software. Structures in the microarchitecture are not tied to the ISA. They have been put in the microarchitecture to provide some benefit, such as a pipeline that speeds up processing, or a branch predictor that predicts the direction of a branch instruction before all the information needed to know the direction is available. ...or the on-chip storage (caches) which eliminates much of the need to access the much slower memory.

It is worth noting that for some elements, their visibility has changed over

the years. Cache memory is a perfect example. Historically, caches were part of the microarchitecture. They would speed up processing each time a cache access eliminated the need to access memory. But they were not part of the ISA. They would be part of the microarchitecture or not, depending on what the architect wanted. At some point some companies (not all) realized that if the actual locations contained in the caches were known to the ISA, instructions could be added to the ISA whose job would be to position those locations for optimal performance. For example, the Digital Equipment Corporation VAX ISA had a FETCH instruction that could place locations in an optimal place (on chip, or NOT on chip) depending on when the data in those locations would next be needed.

One final thought for the moment about ISAs and microarchitectures: Suppose you have the job of designing a microarchitecture to implement an ISA and you come up with a brilliant idea to improve performance by adding a structure to the ISA or to the microarchitecture. Which would you do? Answer: Add it to the microarchitecture. Why? Answer: The idea may turn out to be not as good as you originally thought. If you add it to the microarchitecture and it turns out to be a bad idea, you simply do not implement it on future microarchitectures. If you add it to the ISA you are stuck with it for many future implementations, probably for the lifetime of the ISA.

1.3.4 Requirements for Optimal Performance

To get optimal performance, we need a large number of micro-ops accessing the core every clock cycle, a large number of data elements available in the core every clock cycle, and enough functional units to do the actual concurrent processing of each micro-op when all its data is available.

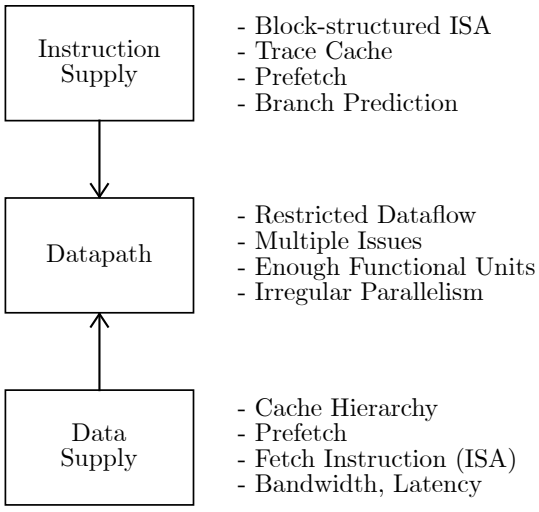


Figure 1.3: Requirements in Microarchitecture

Instruction Supply

Supply of micro-ops means not having to access memory to obtain micro-ops because the micro-ops needed are present in the on-chip storage (i.e., the instruction cache). Access to on-chip storage takes a few clock cycles. Access to memory takes hundreds of clock cycles. Thus, the requirement is getting the instructions which get decoded into micro-ops from the instruction cache, and not having to access memory.

Three mechanisms enhance the ability to get the instructions from the Instruction Cache. (1) A mechanism to not miss in the instruction cache. The best solution so far is prefetching, which gets the instructions into the Instruction Cache before they are needed to be fetched. (2) A mechanism to take advantage of the full fetch/decode width of the microarchitecture. That is, if the microarchitecture has hardware to fetch n instructions at a time, we want the engine to fetch n instructions each clock cycle. The problem is the dynamic instruction stream that we are fetching is not the static instruction stream that is stored in the cache, which means control instructions may allow subsequent instructions to be fetched, but the time needed to perform that fetch may not allow them to be fetched in the same clock cycle. The fetch ends with the control instruction, and the target of the control instruction gets fetched in the next clock cycle. We refer to this as a packet break. Packet breaks prevent n instructions from being fetched in a single clock cycle. A solution to the packet break problem is a trace cache, which stores the n dynamic sequence of instructions in sequential order, rather than in the static order created by the program itself. (3) A mechanism to not having to throw away instructions that never should have been fetched in the first place, i.e., the result of a branch misprediction. That would require a perfect branch predictor, something we will probably never obtain, although we keep getting closer and closer to 100%.

Data Supply

Having to access memory to obtain source operands takes hundreds of clock cycles. Accessing the source operands from on-chip storage (in this case the Data Cache) takes a few cycles. Data supply does not have the branch misprediction problem or the packet break problem. But it does have in a very real way the quest to not miss in the Data Cache. Several approaches are currently being advanced: Better specification of the cache hierarchy, better cache replacement policies, prefetching, specifying instructions in the ISA that more effectively positions locations of the cache, gathering multiple data accesses so they can access memory concurrently so we pay the cost of a long latency operation once, where the operation consists of multiple concurrent accesses.

Functional Units

Finally, we need enough functional units to perform the work required of the micro-ops. Fortunately, this third requirement is less problematic given that today's microprocessors have billions of transistors on-chip (which means we

can easily have as many functional units as we need) and that micro-ops are able to execute when they and the source operands are available, avoiding the former requirement that micro-ops must execute in program (i.e., fixed) order).

1.3.5 A Changing World Introduced by "Clock Cycle Time."

It is useful to pay attention to changes in technology which often have the effect of making us pay attention to something that we used to be able to take for granted. An obvious case in point is the effect of frequency on our designs. A chip running at 5 GHz sees things differently than one running at 66 MHz. A few examples: Wire delay we used to take for granted. We used to x be able to move the data and still have time to perform the operation in the same clock cycle. With today's frequencies, we often have to waste one or more clock cycles moving data from where it is produced to where it is needed.

Power and energy depend on the operating frequency (the higher the frequency, the more energy is consumed), so this has to be considered when determining the budget available for the chip.

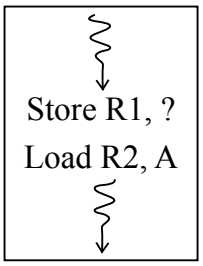
Soft errors, those that are due to the operating frequency of a chip causing a bit to be flipped, means that every design needs to pay attention to this erroneous behavior which occurs even if the logic design of he chip is bug free.

1.3.6 Speculation

Speculation is doing something before you have all the information you need to know you really have to do it. My favorite non-computer example of this is an automobile trip you are taking. You come to a fork in the road. Which way should you go? You don't remember exactly what you did the last time you took this trip, although you have a pretty good idea which road you took. Do you stop, get out a map (we are assuming you do not have GPS at your disposal) and find on the map the road you should take. Or, do you speculate? That is, do you guess, expecting to guess correctly, thereby saving the time it would have taken you to stop and check the map. Of course if you guess, and guess wrong, you need to retrace and that causes you to waste time. Computer systems contain many examples of instances where we have confidence that we will guess correctly and so we will perform speculative execution. That is, we will guess and hopefully save time and get the problem solved more quickly. Of course, if we guess wrong, we waste time delaying the execution of the program.

The most common example of speculative execution practiced today is predicting the direction of a conditional branch instruction before we know the condition codes on which the direction is based. If your prediction is incorrect, you will also have wasted a lot of energy executing instructions that you should not have executed. Years ago, the cost of those incorrect instructions would not have been a major penalty to accrue. Today, with the strong emphasis on the amount of energy expended, it is not a simple matter to know whether speculative execution is the right approach. One must examine the two choices (go, or wait until you know) before you decide which tradeoff makes sense.

Professor Gurindar Sohi and his PhD students years ago provided a brilliant example that highlights the speculation tradeoff. Figure 1.4 shows a piece of code where instructions are allowed to execute out of the order specified by the program. How that is possible we shall deal with later in this book.



Computer Program

Figure 1.4: Instructions Executed Out of Order

In this case we have a STORE instruction that has not yet completed the computation of the address where the contents of R1 is to be stored. A LOAD instruction later in the program has computed the address of the data to be loaded into R2. Since instructions in our microarchitecture can be executed out of the sequential order of the program, the question is: should the LOAD instruction be allowed to load into R2 the data from Memory location A? If the address computed for the STORE instruction is totally independent of A (the address of the LOAD instruction), the LOAD should be allowed because it enables subsequent instructions to be executed sooner, decreasing the time to complete the execution of the program. However, if the address computed for the STORE instruction (when finally known) is A, then we should wait. In fact, in this case, the LOAD would not have to access memory at all. It could simply load the contents of R1 into R2.

If we speculate (i.e., guess) that the address that will be computed for the STORE instruction will not be A, and therefore we perform the LOAD, we will load old data. We will do useless work, waste energy, and slow down the execution of the program..

Professor Sohi and his former students have a patent governing exactly this situation, when to speculate and when the better trade off is to wait.

1.3.7 The Preoccupation with Numbers

Computer architecture folks seem to be obsessed with numbers. Unfortunately, that can lead to erroneous results. I include here several examples where obsession with numbers can leave one twisting in the wind.

For example, I have noticed lately more research results that do not have a baseline. They report numbers, but don't compare them to the state of the art. If I told you I can run a marathon 1000 times faster than your grandmother, would you be impressed? Why is that any different from reporting a

pseudo-brilliant computer result that is actually much worse that what is readily available. "Good numbers" come from building on the state-of-the-art, and doing better.

Most of our research involves simulation. If we are to believe the numbers, the simulator needs to be bug free.

Some numbers come without any understanding as to how they occurred. It is not enough to say, "Look, it works!" Why does it work? My pet example makes this clear. Take the fraction, 16/64. How do we reduce it? Can we reduce it by canceling the sixes? ..or, 19/95 by canceling the nines or 26/65 by canceling the sixes. "See it works" is really not good enough. Why does it work?

The numbers you see should not be the numbers the researcher wants you to see if there are not necessary safeguards. The researcher chooses the experiments to perform, the instructions that are executed, the data one reports. There are too many opportunities for chicanery. Finally, an experiment may produce a data point that is anomalous. Never gloss over such data points. They are usually the most important data points in the entire experiment.

1.3.8 Moore's Law

Gordon Moore hypothesized in 1965 that the number of transistors on each silicon die would double every year. In 1975 he modified his prediction to doubling every two years. As time has passed the time required for doubling has been not quite as aggressive, but still aggressive enough to obtain exponential growth on the number of transistors on a chip. The observation has been known as Moore's Law. It has been improperly given credit for all of the benefit we obtain from more transistors on a chip. The reality is probably closer to half the benefit from the device technology (Moore's Law) and half from cleverness exhibited in the microarchitecture. The demise of the law has been predicted every few years since the 1980s. Today, it appears that Moore's will finally end within the next few process generations. Two obvious reasons: (1) The cost of producing chips with the geometries required to sustain Moore's Law is much too expensive, and (2) the geometries are getting too small. Researchers are currently working with devices expected to be in the 2 nanometer range. As you may recall from your study of chemistry, two nanometers is 20 angstroms. Nonetheless, before we bury Moore's Law, we should acknowledge what it has provided: (1) Parallelism. In 1971, when the first microprocessor, the Intel 4004 arrived, there were 2300 transistors on a chip. Today that number is several billion. (2) Switching speed. Smaller transistors mean faster switching time. The Intel 4004 was a 106 KHz device. Today frequencies are in the GHz range. We are computing orders of magnitude faster at orders of magnitude greater in concurrency. It will be tough to say goodbye!

The current effect of this will be much more demanded of those computer architects still around. Charles Leiserson and several of his colleagues, referring to my levels of transformation, say "There is plenty of room at the top." That is, compilers and algorithms will need to pick up the slack if we are to maintain

our historic rate of performance improvement. I am quick to agree, but also note that there is still plenty to do at the bottom, noting that we have plenty of opportunity to better harness the device and microarchitectural levels of the hierarchy.

1.3.9 The von Neumann Machine

The von Neumann machine, the classical model of computing continues to be declared dead and ready to be replaced by several devices referred to as non-Vons. The reality is that just about all those that are referred to as non-Vons are more accurately accelerators rather than computers. In my view, future chips are going to need von Neumann machines. Why? Answer: Future chips will more and more be dominated by accelerators that will have to talk to each other. A very serious communication problem, ready to drown in its own chaos, unless something is available to maintain order. That, in my view, is the role the von Neumann will fill for the foreseeable future.

1.3.10 Is Hardware Sequential or Parallel?

Clearly, hardware is both sequential and parallel. It is sequential in the sense that hardware processes instructions clock cycle by clock cycle. Those clock cycles occur one after the other, i.e., sequentially. But within a clock cycle, just about everything happening is happening in parallel. All the electrons are operating all the time. An old electron does not tell a young electron, "After you," to which the young electron responds, "No way. You are the older electron, you go first." All the devices in the core are functioning all the time; ergo, hardware is the ultimate in parallelism.

Software is also both sequential and parallel. Instructions execute sequentially. But a lot of current activity challenges programmers to think in parallel, and from there, create programs that can execute concurrently.

The bottom line, both hardware and software execute clock cycle after clock cycle, sequentially. But within each individual clock cycle the work can go on in parallel. With respect to hardware, this is inherent in the way hardware works. In the case of software, it more nearly comes as a result of the software producer "thinking in parallel."

1.3.11 Do it in hardware or do it in software (or perhaps use an FPGA)?

Many design decisions resolve as to whether one does the task (or part of the task) in hardware or in software. A simple rule of thumb: doing it in hardware takes time to build the capability, and making changes are usually very time consuming. The good news: generally higher performance. Do it in software and you get the implementation much more quickly. And you can make changes much more quickly. The bad news: generally lower performance.

...which brings up a third choice: do it with FPGAs. Not as fast as hardware but much more flexible, and more easily changeable. Much faster than software, but not as flexible or changeable as software. Another example of tradeoffs.

1.3.12 Design Principles

There are many principles of effective design. We will mention three of them here. Critical path design, Bread and Butter design, and Balanced design.

Critical Path Design

Critical path design takes the position that there are a few things going on in your computer which take so much time that they will define the cycle time. Since cycle time is generally the most critical element driving performance, it makes sense to pay close attention to it. Critical path design dictates measuring the time it takes in each clock cycle to do each task. The one that takes the longest will determine the cycle time UNLESS you can do something about it.

Critical path design follows the following steps:

Step 1. Identify the longest path.

Step 2. Continually shorten that longest path until it is no longer the longest path.

Step 3. Move to the *new* longest path and continually shorten it until it is no longer the longest path.

Iterate the process until you are satisfied with that cycle time, and you are done. Usually, the three longest paths are (1) the time it takes to access cache memory, and store the result in a register, (2) the time it takes to source the registers for source operands, perform an ALU operation on those operands and store the result back to a register, and (3) determine the control signals needed in the next clock cycle, and latch them for the start of the next clock cycle. Often you will have one path much longer than the others, and be unable to shorten it. In that case, make that path a multi-cycle activity, unless you get more benefit from doing it in one clock cycle than the loss you incur from everything else taking a longer clock cycle.

Bread and Butter Design

The "bread and butter" is an American idiom to mean those things that are most important to some task or behavior. Students' bread and butter may be their grade point average (GPA) or some outside activity that to them is more important than grades. Bread and butter design means paying extra attention to the things that matter most. In the case of computer design that means spending an inordinate fraction of the chip resources on those things that will have the greatest effect on the final product. That does not mean ignore the rest of the design completely. Everything in the design has to work. But for things that are not part of the bread and butter, make sure to devote enough

resources to not create a disaster, while focusing most of the effort on the bread and butter.

Balanced Design

The parts of the microprocessor have to work together. I remember a design review many years ago where the chief architect boasted that his front end, the fetch and decode stages of his pipeline, enabled 6 instructions to be fetched and decoded every clock cycle. Unfortunately, the core of his processor could only process three instructions per cycle, resulting in a lot of fetched instructions waiting to be processed. A bad design – Not balanced! All that extra fetch/decode bandwidth used a lot of resources but the core could not handle it so it produced no benefit.

Simply put, the front end (fetch/decode) and back end (execution, retirement) should make sense together.

1.3.13 Design Points and Design Methodology

Every product should have a design point. What is the product you are designing expected to accomplish. In my case the design point has always been performance. I want the product to perform faster than other products on the market. But there are other design points. You may want a product that can be sold for very little; that is, cost is the design point. Not every product has to deliver the highest performance. Or the product can be designed to use the least amount of energy, or be built in a way that the computer is working correctly almost 100% of the time. In this case we may not care about energy consumption or performance. We may be willing to sacrifice energy consumption and performance if we can guarantee the computer will almost always be functioning correctly. We call this design point high availability.

Design methodology is the process one goes through in designing a product. The first step is to specify your design point. Then identify your bread and butter. Then optimize the bread and butter. Finally, deal with the rest of the design.

1.3.14 The Role of the Architect

Good computer architects look backward and forward, up and down, as they develop their designs. What do I mean by that?

Look backward means one should examine old programs. What parts of the ISA were heavily utilized and need to be part of the new design and perhaps part of the bread and butter. What parts were never used and should not be part of the new ISA. Digital Equipment Corporation produced the PDP 11 which had 8 addressing modes. One, “auto-decrement deferred” was never used. The next product, the VAX removed that addressing mode from the ISA.

Look forward means listen to the dreamers who would like new features that they can use effectively. Intel produced the Pentium chip, and introduced the

MMX instruction set because the dreamers saw an effective use of those instructions as applications moved more and more toward multimedia applications.

Look up refers to the transformation hierarchy we have discussed. At the top is the problem expressed in natural language. Looking up means seeing what your product will be used for, which should direct your design.

Look down also refers to the transformation hierarchy, where the bottom is the device technology. Looking down means predicting what will be available for your product when it is time to manufacture it. If you predict too aggressively, the technology may not deliver what you predict and you will not have a product. If you predict too conservatively, you will have a product but no one will buy it if a more aggressive technology is available that does the job. Predicting where the technology will be when the product is manufactured is an important task for the architect. Digital Equipment Corporation discovered that in their Alpha 21164 chip when the technology did not deliver what was predicted. The result was a Level 2 cache that was 96 KB, 3-way set associative. Why? Because there was not enough area on the chip to provide a 128 KB, 4-way set associative cache.

1.3.15 Thinking Outside the Box

People are always telling us: "Think outside the box." My response is, "Be careful." It is too easy to get carried away and end up producing something that is guaranteed to never work. I remember a young architect designing a computer that he thought could resolve 25 conditional branches in a single clock cycle. An obvious disaster.

My approach is more cautious: Expand the box. Several examples immediately come to mind. The HPS microarchitecture did not invent out-of-order execution. Robert Tomasulo did that. But HPS built on top of the basic Tomasulo algorithm to produce a much higher performance engine. The Simultaneous Multithreading (SMT) work did not invent multithreading. Burton Smith did that more than 15 years earlier with the HEP. But the SMT people built on that paradigm, again producing a higher performing engine. The Perceptron Branch Predictor did not invent the Perceptron, Frank Rosenblatt did that more than 30 years earlier. But Daniel Jimenez realized the Perceptron was a natural fit for Branch Prediction and he could build on the Perceptron to create a better branch predictor.

1.3.16 Finally, a Few Questions to Complete Chapter 1

Finally, we will complete Chapter 1 with a few questions:

Question 1: What is computer architecture? Answer: It is a contrast between the software(what it demands) and the hardware(what it agrees to deliver.

Question 2: What is microarchtitecture? Answer: It is a science of tradeoffs, balancing what functionality we will deliver, at what performance and at what cost.

Question 3: How do we compute faster than the speed of light? Answer: We do things concurrently.

Question 4: Should we add a feature if it slows the clock by 10%? Answer: Normally no, unless the benefit of the feature taking a single cycle outweighs the fact that everything else takes 10% more time.

Question 5: Is computer architecture dead? Answer: Computer Architecture will always be alive and healthy as long as people can dream. The dreamers are usually not the architects, they are those who want to use machines in new and interesting ways. Computer Architecture is about the interface between what technology can provide and what the market demands.