Advanced Architecture

Computer Science Honours
Rhodes University – 2012

Philip Machanick

Contents

List of Figures

List of Tables

1 Introduction

COMPUTER ARCHICTURE is a rapidly moving field yet a few things have held good over the last three decades. Before the RISC movement of the late 1970s, much computer architecture was based on gut feel, or poor communication between the hardware and software sides of design teams. For example, the hardware people may decide that setting up the call stack involves tedious repetition and to be nice to the compiler people, they roll this all into one instruction. Only, because they didn't consult the compiler people, the instruction they design isn't useful and is never used. Then when they try to implement a more aggressive version of the design, they find all the complicated instructions they weren't asked to design by the software people make it hard to produce a more aggressive design. Related to this communication issue is a lack of standards for quantifying improvements. In the 1970s, Gene Amdahl, a former IBM engineer who split with IBM to form his own high performance computing company, formulated a speedup limitation [Amdahl 1967] that became known as Amdahl's Law. In essence this says that to calculate the effect of a speed improvement, you need to take into account the entire run time *including parts that are not sped up*. An apparently obvious revelation, it's a point often forgotten when extolling some brilliant enhancement (not only to computer systems).

What created an impetus for improved standards in quantifying performance was a dispute that arose between two schools of computer architecture, the high-level-language oriented approach, and the simplicity-oriented approach. The latter gained credibility as early as the 1960s, when Seymour Cray, at that time working for a relatively small company called Control Data, produced a design that was much faster than the best the market leader IBM could produce. However, many designers still argued that a machine instruction set closer to high level languages was more efficient because even though each instruction may be slower than with a design closer to the hardware, you needed fewer instructions. This argument carried some weight when memories were relatively small, and hardware complexity could be replaced by *microcode*, a very low-level instruction set that interpreted the actual machine instructions. Microcode was stored in a ROM that

was many times faster than DRAM, so frequent accesses of *microstore* as the cost of fewer instruction fetches was a reasonable trade-off. However, as DRAM speeds caught up with ROM speeds and it became viable to implement caches in fast SRAM, the case was less clear. It took a landmark paper in 1980 [Patterson and Ditzel 1980] to fire up a new movement inspired by Seymour Cray's 1960s designs [Thornton 1963] to push the case for simpler instruction sets, and that push led to a more quantitative approach to architecture design and evaluation. In particular, to win the case for simplified instruction sets, the RISC (reduced instruction set computer) movement began a move to more scientific principles in measuring alternative designs, with an emphasis on *repeatable* experiments that were representative of *real workloads*.

In this course we learn about tools and techniques for measuring performance, how architectures are designed and what factors are useful to consider when comparing performance of alternative designs. We related these issues to a range of different areas of architecture design: memory hierarchy, instruction set design, input and output, and parallelism in various forms.

1.1 Measurement

Computer architecture measurement falls into two broad categories: evaluating existing designs and implementations, and evaluating design alternatives. In the first category, we can run standard benchmarks (software that represents a workload of interest) and we can also use simulations so as to produce repeatable run times. In the second category, we mainly rely on simulations because it is too expensive (even using reconfigurable hardware, such as FPGAs) to create multiple real variations in a design to check the effect of changes in design parameters.

Whichever approach we use, we try to adhere to a few essential principles of the scientific method:

- *repeatable* – running the same experiment twice should give the same result and we should report enough detail so others can redo the experiment
- *separation of variables* – where more than one factor can influence performance in a way that cannot be separated out, only vary one such variable at a time
- *representativity* – the experiment should represent something real to those interested in the evaluation, not an artificial exercise that will not rank alternative designs the same way as would real usage

While these principles seem obvious, the quantitative approach was novel enough at the time that two senior academics, David Patterson at the University of

California, Berkeley and John Hennessy at Stanford, felt the need to codify the principles in an academic text in 1990 [Hennessy and Patterson 1990] that has subsequently entered its fifth edition [Hennessy and Patterson 2012], and these principles are now routinely observed in mainstream computer architecture research, which was not the case when I first became interested in the field in 1980.

The fact that we now have well-established scientific principles of architecture measurement doesn't mean that the field is devoid of creativity or innovation. However, that innovation now has to be based on reasonably sound principles. Even so, some large mistakes are still possible, for example, the attempt by Intel to break out of their IA32 architecture with the IA64 (Itanium) design, which failed to achieve its performance goals or wide market acceptance.

A scientifically sound basis for measurement only allows us to be accurate about making comparisons: it does not remove the requirement of thinking up innovations, because someone has to derive the new ideas to compare with old.

1.2 Design Approaches

Given all that, how do we arrive at innovations?

Much of the early computer architecture work up to the 1970s set the scene for widely accepted design alternatives today. A fair fraction of innovation today involves rediscovering old ideas that worked well in a different form factor, and finding that technology today makes those ideas work well once more. Much of Seymour Cray's work in the 1960s was in essence reinvented by RISC designers in the 1970s through to around 2000, as it progressively became possible to fit more of the features of his multichip designs onto a single-chip CPU. Remarkably, very little in modern designs wasn't found in his landmark CDC 6600 of 1962, including hardware to support multiple instructions per clock cycle and out of order execution.

Today, a good starting point for looking our for potential for innovation is to examine various trend lines and work out when new design trade-offs become possible. Possibly the most famous of these trends is *Moore's Law*, an observation that the number of transistors at a given price doubles about every 2 years [Moore 1965]. There are others, like the quartering of the cost of DRAM every 3 years, and the much slower speed improvement of DRAM. Understanding how long these trends can persist and when a change in technology is predictable opens up opportunities for architecture research. For example, in the 1990s, I observed that the speed gap between DRAM and CPUs was heading for similar numbers as measured by lost instruction execution opportunity to the speed gap between CPUs and paging devices when virtual memory was first invented. That led to the *RAMpage* project, of which I cite a fraction of the research outputs

here [Machanick et al. 1998; Machanick 2000; Machanick and Salverda 1998; Machanick 2004].

In another breakthrough, which led to a major change in the whole industry, a Nigerian academic at Stanford University Kunle Olukotun [Olukotun et al. 1996] made a case for replacing very aggressive single-core designs by what are now known as multicore designs. In essence his argument (backed up by design studies and simulations) is that the potential for speedup of a single core design is limited by how much instruction-level parallelism is available, whereas a multicore design can gain speed from several dimensions. A clever compiler can convert instruction-level parallelism into threads, code already designed to run threads or multiple processes can speed up, and multiprogramming workloads (as on a typical operating system where there may be dozens of processes, many not visible to the non-technical user) can also see a speed gain. Multicore designs have in recent years also gained in utility because they create more options for scaling both performance and energy use.

Yet another approach to looking for breakthroughs in architecture is studying roadmaps of predicted future technology[1]. In one example, Trever Mudge at the University of Michigan picked up the likelihood that vertical stacking of dies (a die is a chip without the packaging) was on the horizon, and he explored the implications of this technology for making a package tightly integrating DRAM and a multicore CPU design. The resulting design has a number of advantages. Because through-chip *vias* (conductors) can be as fast as within-chip communications and wide buses are practical to construct in this form, the CPU-DRAM speed gap can be considerably reduced. Since the CPU wastes less time waiting for DRAM, a given level of performance can be achieved with a slower clock, reducing he problem of heat dissipation out of a compact package. The resulting PicoServer design [Kgil et al. 2006] and its successor Centip3De [Fick et al. 2012] may at some stage emerge as a commercial product; even if it does not, it is a good illustration of looking out for technologies that may later become viable.

1.3 Performance Factors

When considering performance, we need to take into account several axes. Depending on the target appllication or market, different axes may be more important. The most significant ones are

[1] International Technology Roadmap for Semiconductors `http://www.itrs.net/` is a good example.

- *cost* – not only of one component such as the CPU, but overall packaging and environment costs
- *speed* – again, not only one component contributes to speed (remember Amdahl?)
- *energy* – in some applications like mobile computing, energy is a first-class concern but even in large-scale computing, energy is a limiting factor
- *scalability* – a design that works at many scales means early expensive versions can be sold into high-margin markets like high-end servers, while older designs can be sold into high-volume markets to maximise amortisation of costs
- *longevity* – one of the most classic errors of hardware designers is to fail to take into account the rate at which technology improves: too small an address space is one of the most common reasons once-successful architectures have had to be abandoned

Cutting across these axes are two approaches to performance that can be in conflict:

- *latency* – time to complete a *specific* operation or service
- *throughput* – average rate of work completion

Low latency is what the user desires; high throughput is what the accountants want. Low latency means you have a responsive system, but that responsiveness can be bought at the expense of lowering throughput, by ensuring that the system is not busy when you want a response.

In this course I examine case studies of performance covering as many of these axes as is practical, depending on the nature of student projects.

1.4 Architecture Areas

Computer architecture is broadly speaking design principles of any area of the computer system including the hardware and any area where hardware and software interface. It's convenient to divide architecture down into different areas, though these necessarily interact. For example, the memory hierarchy includes components that use the IO system, and efficient implementation of IO requires design with the memory hierarchy in mind. So as we divide the architecture world for clarity, remember that the division is not absolute.

1.4.1 Memory Hierarchy

The need for a memory hierarchy arises from the fact that memory components fast enough to keep up with the CPU are many times more expensive than slower memories. Fortunately, the *principle of locality* says that you a program uses a relatively small part of its address space at a time. Locality is generally divided into two types:

- *temporal locality* – a memory location that is referenced is likely to be referenced again soon
- *spatial* – a memory location near a location that is referenced is likely to be referenced soon

The definitions of "soon" and "near" depend on how big the speed gap is between layers. If the speed gap is big, we stretch the definitions out to longer in time and space, because we can less afford the penalty of accessing slower memory.

In an operating systems course, we focus on locality as it applies to virtual memory; here we also consider hardware layers of the memory system, including caches, the TLB (translation lookaside buffer: a small cache of recent page translations) and registers.

1.4.2 Hardware Layers

It is useful to divide computer hardware into logical layers. As seen by the user (or, in today's world, the compiler and related tools like the linker), there is the machine code layer. This layer cannot change much in basic functionality without losing the user base. If you have to recompile or relink your code to run on a new generation of a particular vendor's design, that takes away a reason to stay loyal to that vendor. The *instruction set architecture* or *ISA* is such an important part of a designs identity and its ability to retain a user base that the ISA is often referred to as the architecture (the IA32 architecture, the PowerPC architecture, etc.).

The ISA is not only characterised by a set of instructions but also by the available machine registers, the memory bus size and instruction modes such as supervisor and user mode that implement protection. The idea of an ISA essentially developed with the IBM 360 series of the 1960s [Amdahl et al. 1964], which was the first to feature a whole family of designs launched at once that could run the same programs, subject only to resource limits not differences in the type of code that could execute.

The ISA can be implemented many different ways and remains the same ISA as long as the same programs can run (give or take constraints like memory size and available peripherals).

Some details that can vary include the pipeline, extra copies of the registers to support implementation details like hardware multithreading support and out of order execution, branch prediction and variations in the memory hierarchy. All these variations are below the level of the ISA because they are hidden (other than performance impacts) from anything assuming the ISA as a given.

One area that is not obvious to the user (even a compiler writer) that is hard to change in practice is hardware support for VM. If this changes, unless the old approach is maintained for backward compatibility, every operating system using the new design will need to be modified, since hardware support for VM is tightly integrated into the software side of VM implementation.

1.4.3 Hardware-Software Layers

The operating system provides a layer of abstraction that hides the bare metal from the user, and some parts of the system architecture may involve hardware and software components. The most obvious of these is the virtual memory system that cannot be implemented effectively without hardware support (otherwise, every memory reference would take several times as long as without VM, since it must be looked up and translated, as well as checked for validity).

There are other aspects of the system where hardware and software play a role. In some earlier microprocesor designs including some RISC designs and some implementations of the Intel IA32, significant speed gains could be had from reordering instructions. Such reordering required recompilation in most cases, and was seldom done for the very good reason that the next generation had a different optimal ordering of instructions.

In yet another area, IO involves hardware-software cooperation. IO is very slow compared with the CPU and RAM and that speed gap has to be hidden. An operating system typically schedules IO-bound processes with higher priority than CPU-bound processes for two reasons. If CPU-bound processes run to completion while there are still IO-bound processes in the system, there is no work to be done while waiting for IO. Secondly, if IO-bound processes are able to use the CPU, it's best to give them more time than CPU-bound processes so they can make progress. An operating system has a range of strategies to hide the latency of IO in addition to scheduling policy. Here is a quick summary:

- *scheduling* – IO-bound processes have higher priority than CPU-bound processes
- *buffering* – slightly different effects for input and output:
  - *input* – read more than absolutely needed, relying on spatial locality not to waste the extra IO because it's usually more efficient to transfer

in large blocks

  - *output* – don't wait for writes to complete: dump output to memory and let the device empty the buffer in its own time

- *cacheing* – keep data (or code) in a faster layer of memory as long as possible; buffering can be a form of cacheing if the contents are available for repeated use

- *spooling* – for devices that have to accept a job to completion, spooling is a specialist kind of buffering that stores the data until it's that job's turn (most often used for printing)

- *specialist IO hardware* – in some systems IO is hived off to a separate specialist CPU relieving the main CPU of the detail of IO

These represent some of the strategies used by an operating system; we do not cover much detail here since an OS course has more space to do so.

1.4.4 Hardware-Software Interaction

Given overlaps in hardware and software, how important is it for software to be aware of hardware, and vice-versa? In addition to the issues raised above of communication between parts of the design team, users of a design can benefit from knowing how their software interacts with hardware.

Some areas where this knowledge can apply include:

- *memory-sensitive algorithms design* – understanding of how the cache and VM subsystems work can have a large effect on performance [Lam et al. 1991; Machanick 1996; Xiao et al. 2000; Rahman and Raman 2000]

- *balancing VM use and IO* – in an experiment, I ran quicksort on randomly generated data varying the size until I ran out of RAM and paging occurred. I rewrote the code so it sorted a section at a time, storing most of the data on disk, using mergesort to merge only as much as would fit into RAM at one time. This ran a lot faster than relying on VM. No big surprise. But what was a bit surprising was that it was not significantly slower than quicksort on data that did fit into RAM.

- *efficient use of shared memory* – with multithreaded code or processes with shared memory, a clear understanding of cacheing makes a big difference to performance [Machanick 1996]

- *role of VM hardware support* – understanding how VM is supported in hardware can also make a big difference to performance [Machanick 1996]

In this course we explore some of the issues; since the 1990s when multiprocessor systems were relatively expensive, some hardware-software interaction concerns have found their way to the mass market because of the proliferation of multicore designs.

1.4.5 Instruction Set Design

Instruction set design used to be a core area of computer architecture. It is less so now that it's clear that RISC is fundamentally a good idea, but the Intel IA32 architecture isn't going to go away despite this.

Styles of Instruction Set

Prior to the RISC (reduced instruction set computer) movement, there were two major schools of design:

- *ad hoc* – do something that feels right and hope for the best, making a few trade-offs like make common instructions shorter than less common ones to reduce memory footprint at the expense of making instruction fetch and decoding harder
- *high-level language oriented* or *HLL* – design the instruction set to be easier for compiler writers to generate code

In the first category, we have some of the most enduring designs. The Intel IA32 developed out of a processor with a 16-bit address space, the Intel 8086, which was upgraded to a 32-bit address space with the 80386 in 1985 and now includes 64-bit implementations. The IA32 has endured because it was adopted for IBM's PC design, which developed a massive market, and also because Intel was able not only to throw massive resources at improving its performance against the odds, but had very skilled engineers working around the inherent flaws in the design. The IBM 360 architecture [Amdahl et al. 1964; Gifford and Spector 1987] is another that endured for decades despite clear flaws (in terms of subsequent knowledge on how to design for performance). The 360 endured because IBM was one of the first computer companies to sell on service rather than technology, and because the design had a few key things right: it was designed for 32-bit addressing ahead of much of the competition, and had an adequate number of registers, a critical feature for achieving high performance. IBM, like Intel, had very skilled engineers able to work around inherent flaws in the design.

In the second category, one of the more successful examples is the Burroughs B5000 architecture [Mayer 1982], which used a stack-based instruction set and had hardware support for arrays including bounds checking (a hardware data

structure called a *descriptor* stored details of each dimension of the array). Memory was *tagged* with extra bits representing the type of contents of a machine word, further supporting error checking. Since the hardware could determine the type from the tag bits, there was only one instruction for each basic operation like addition (not a separate instruction for floating point, integer and various precision alternatives). The instruction set was very compact, since stack instructions do not need register names let alone memory addresses except to move data onto or off the stack, and in an exception to common practice, the hardware and software teams worked in close collaboration. Unusually for its time, the system software was written in a high-level language (a version of Algol 60, a language with some following in academia), and the operating system was distributed as source code. The Burroughs machines were not particularly fast if you measured the run of a single program but had a very efficient VM system, and easily outperformed machines that were a lot faster on paper, with real workloads. Unfortunately, Burroughs designed their array support assuming the Algol 60 approach of storing multidimensional arrays in row major order, whereas the scientific community mostly chose to use FORTRAN, which requires arrays to be stored in column major order, causing significant complications in generating efficient FORTRAN code.

The Burroughs example illustrates one of the hazards of HLL-oriented design: high-level languages differ enough that it's hard to do a design that's good for one without serious compromises for implementing other languages.

One of the less successful examples of HLL designs is the Intel 432 [Organick 1983]. The 432 had very fine-grained protection, supposedly to support object-oriented coding, but had very poor performance [Colwell et al. 1988], and didn't ever gain significant market share.

The IA432 illustrates another hazard of HLL-oriented design: it can result in poor performance, especially when insufficient attention is paid to any of practicalities of hardware implementation and usability of features in compilers.

More recently, hardware support for executing Java bytecode has emerged. However, just in time (JIT) compilers reduce the advantage of a Java machine. One implementation of partial hardware support for Java targets small devices with real-time requirements [Schoeberl 2008]. Some niches may justify specialist designs though on the whole it's easier to use a language that's a better fit to the problem than to design hardware to work around limitations of a language (e.g., garbage collection makes for unpredictable execution times, a problem for real time).

By contrast, the RISC movement specifies a very simple regular approach to instruction set design:

- *fixed instruction length* – all instructions are the same length, making it easy to fetch and decode multiple instructions in parallel

- *load-store architecture* – all memory references are loads (copy to register) or stores (copy from register); arithmetic and logic unit (ALU) operations are always on registers
- *standard operands* – ALU operations always operate on 1 or 2 source registers and one destination register
- *bounded execution time* – with the exception of excursions down the memory hierarchy, instructions have clearly defined execution time
- *simple control* – the number and type of branch instructions is limited (usually unconditional jump, and a few conditional branches)
- *large general-purpose register file* – a small number of registers, registers with specific purposes or setting logic conditions in condition codes makes it harder for compiler writers to generate code, and harder for hardware to reorder instructions

Some RISC designs compromise on details, e.g., a number do use condition codes. Nonetheless RISC architectures are generally very similar, unlike other classes of ISA design that differ widely.

We see next why the RISC movement claims advantages over the other approaches. At this point, note that even with the Burroughs design where the hardware and software teams did work in close collaboration, the fact that their pesky customers chose to use a different language for programming meant that much of their good work was wasted.

Design for Performance

The RISC movement is based on a few key observations of how performance is achieved. First, to make the overall system fast, you need to have the highest possible clock speed and rate of instruction flow through the system. The latter works best if you can implement an efficient pipeline. A pipeline is inherently inefficient if the stages are not all the same length (the longer stages will force the shorter ones to idle). To implement a fast clock speed, relatively short pipeline stages help. If there are many variations in type and size of instruction, these things become harder to achieve.

Second, an important principle is *make the common case fast*. This seems contrary to the lesson of Amdahl's Law that the best speed gains arise from making everything faster. However if you calculate speed improvement based on accurate performance measurement, you can quantify this effect. For example, having to run 50% more instructions as the price for doubling clock speed is probably a

win (though you need a comprehensive measurement that takes into account other effects like changes in memory hierarchy use). By contrast, introducing a few special-case instructions that are rarely used but make it hard to scale up the clock speed is seldom a win. Again, quantifying makes the case, not gut feel or hand waving.

To take an example, RISC architectures generally implement a call with several (general-purpose) instructions, rather than using a special instruction to set up a stack frame and store the return address. While this increases the instruction count, the absence of special instructions makes it easier to implement an aggressive pipeline. At the cost of of occasionally using more instructions, the overall system is faster. We can quantify this effect if we know how much faster the clock speed can be made, or how the pipeline can be improved in other ways by simplifying the instruction set design, and calculate the net gain. The need to do this sort of calculation to convince RISC sceptics was the start of the modern approach to quantitative design.

1.4.6 Input and Output

IO is an important part of systems design because most IO devices are so much slower than the rest of the system. A disk for example may take of the order of 10ms to do a seek (move the head to the right track). DRAM access is almost a million times faster, and a 2GHz CPU if executing only one instruction per clock takes 0.5ns per instruction peak is 20-million times faster. If you have an aggressive pipeline executing several instructions per clock the speed gap is even greater (before you consider keeping up with multiple cores).

In this course we do not study IO in detail; many of the performance issues are better dealt with in an operating systems course. At the hardware level we can consider various modes of interfacing, the relationship between latency and throughput (or bandwidth in the IO context) and trends in device technology.

1.4.7 Parallelism

Every now and then when progress in a given approach to technology appears to be heading for a dead end, parallelism appears as the solution. What usually happens is a new approach to sequential programming appears, and all the complexities of parallel programming lose their attraction. Over time many models of parallelism have appeared, and some have proved enduring, while others keep resurfacing as packaging trade-offs change, and the reason they were abandoned is forgotten.

As long as Moore's Law was effectively delivering double the performance every 2 years or so, there was little benefit in writing parallel code for performance unless you could afford a large-scale system. Any system that achieved a speedup

over a serial implementation (measured as $time_{serial}/time_{parallel}$) of less than 4 would be overtaken by faster hardware in a year or two, sometimes sooner than the time it took to achieve an efficient parallelisation.

Instruction-Level Parallelism

Before the multicore era, the most common form of parallelism was instruction-level parallelism (ILP), because it took no effort from the programmer. Provided the hardware can find more than one instruction ready to run at the same time, ILP provides speedup at the expense of hardware complexity, a trade-off increasingly justifiable as the number of transistors per chip at a given price point increases.

ILP however has some inherent limits. There's a limit to how much inherent parallelism that exists at instruction level because of dependencies between instructions [Wall 1991; Lam and Wilson 1992; Postiff et al. 1998], and there are limits to the extent to which practical architectures can find available parallelism (e.g., instructions with no dependency between them may be relatively far apart). A problem that has arisen more recently is that the increasing complexity required for more aggressive ILP has a high cost in energy use [Yeap 2002] and hence also heat.

Another limitation to pursuing performance using more and more aggressive ILP with higher and higher clock speeds is the growing gap between the speed of CPUs and DRAM, resulting in limited gains as a higher fraction of the CPU's time is spent waiting for DRAM, a problem called the *memory wall*, predicted in 1995 [Wulf and McKee 1995].

Multiprocessor and Multicore

In the past multiprocessor architectures differed widely in characteristics. Some emphasised data parallelism (the same instructions on several or many different data items in registers or memory), others instruction parallelism (different instruction streams on each ALU). Memory organization also varied. A distributed-memory architecture had no shared memory and communication primitives like messaging were used. A shared-memory architecture had one global memory and required mechanisms to ensure consistency of caches. A distributed shared memory system [Bennet et al. 1990; Dwarkadas et al. 1993; Bordawekar 2000] is physically distributed but gives programmers as model that looks like shared memory. There are also programming tools and libraries like MPI that hide some of the detail of the memory model, but some understanding of the memory model is essential to achieve performance.

More recently, as limits of ILP and scaling up the clock speed (in part because of the memory wall but also because of limits to ILP and the increasing cost of

hot high-energy consumption designs), multicore designs have become popular, and these generally are shared-memory designs, often with a shared lower-level cache. Writing parallel code is now an option for the mass market, so the specialist skills needed for programming big iron in the 1990s [Cheriton et al. 1991, 1993; Machanick 1996] now apply more widely, but the problems are no easier. A good understanding of how the memory system works in shared-memory multiprocessors is even more important to achieving good performance than with a uniprocessor.

GPUs

The conversion of specialist processors designed for graphics to general-purpose computing is not a new idea. The Intel i860, marketed as a general-purpose CPU with graphics support [Grimes et al. 1989] was clearly designed more with graphics support than general purpose use in mind. Among other things, it suffered very high latency for context switches, and VM support is minimal. On a page fault, it only reports that a fault occurred, not whether it's a read, write or instruction fetch, meaning the page fault handler has to reconstruct the cause, in effect interpreting the instruction that caused the fault to work out what happened [Anderson et al. 1991]. The i860 was reasonably successful as a graphics processor in the days when a high-end graphics system was a multichip design, with features presaging vector extensions to the IA32 line. Overall the i860, despite being deployed on some large-scale supercomputer designs [Berrendorf et al. 1994], was not a great success as a high-performance CPU.

So is the general purpose GPU (CPGPU) concept reasonable, given that specialist processors have not historically been a win? There are arguments for and against. Against, Amdahl's Law tells us that a 100× speedup of a small section of our code will have a small overall effect on run time, and coding for these specialist processors, even with high-level tools like Nvidia's Cuda [Wynters 2011][2], is hard. What's more these toolkits tend to be vendor-specific. On the for side, the massive market for GPUs means that there's a lot more critical mass behind this movement than other attempts at using specialist processors for more general purposes than originally intended. Computers with powerful GPUs are increasingly ubiquitous in the mass maket, making it at least possible that massive-scale computation using such GPUs will continue into the future, whereas past specialist designs did not have that critical mass.

[2] More at `http://developer.nvidia.com/nvidia-gpu-computing-documentation`.

Warehouse-Scale Computing and the Cloud

One of the ongoing debates in computer architecture is whether large-scale computing is best achieved with massive numbers of inexpensive boxes with redundancy designed in, or dedicated highly-scalable designs. One of the earlier ideas of this type is *RAID*, originally *redundant array of inexpensive disks* [Patterson et al. 1988] (now usually "independent" instead if "inexpensive", possibly so manufacturers can claim their disks are "enterprise grade" and hence not inexpensive).

Possibly the best-known example of the extension of this idea to a *redundant array of inexpensive computers* (no one uses RAIC as a name for some reason) is Google's approach or warehouses full of inexpensive computers, with many fail-safes to allow for hardware and software faults [Barroso et al. 2003]. This kind of infrastructure is becoming increasingly important as the Internet expands to ever new services including some that might for a brief period require hundreds or even thousands of servers, then settle back to more modest requirements [Liu and Wee 2009]. How best to put these services together is still a work in progress, and there is no doubt that expertise in this field will be useful for some time ahead.

The "cloud" term is somewhat vague in meaning, and is really marketing speak for distributed services, sometimes storage, sometimes computation, sometimes both. The key feature of a *distributed* system, as opposed to a *networked* system, is naming or location *transparency*, i.e., you don't know (or need to know) whether data or a process is running locally, over a network or even over several computers – that is a performance detail. By contrast, a networked service requires naming the location where the service occurs. What really distinguishes the cloud from earlier distributed services is that the infrastructure is provided on a closed proprietary system, rather than as a file system or operating system that installs on your own computer. While some such services like Google Drive or DropBox occupy file space on a computer on which you use the service, they do not integrate cleanly. Can you put a DrobBox folder into a Google Drive folder? If so, will this still work next week? Can you mix any of these with Amazon's AWS or Apple's iCloud?

Large-scale cloud services are linked to warehouse-scale computing in that they need large highly-scalable geographically dispersed implementations. That's not to say all implementations use the same infrastructure, or that someone won't find a better way. This is a relatively new area and one with a lot of potential for innovation – even though the core concept of a distributed system is quite old, with some of the theory dating back to the 1970s [Lamport 1978].

1.5 The Other Edge

Much of the previous discussion assumes we want the fastest possible system, constrained by cost, power consumption etc. However, Moore's Law can be read the other way. As hinted at by Gordon Bell, new classes of computer become viable as a given level of functionality becomes available at a price point [Bell 2008].

In the 1970s, a personal computer capable of doing interesting work – including the first spreadsheet, VisiCalc [Bricklin and Frankston 1979], became viable because a single-chip microprocessor powerful enough to run an elementary operating system and programming tools reached an affordable price point.

Since then, other breakthroughs have included:

- *scalable PC* – the original IBM PC was not a huge advance on the previous generation but Intel's ability to add enhancements like 32-bit addressing as new thresholds in the number of affordable transistors were crossed meant the CPU remained viable – even if major OS rewrites and recompiles were needed in the transition to 32 bits (and less so to 64 bits)

- *Linux* – when the IA32 became powerful enough to support a UNIX-like operating system it was only a matter of time before a free UNIX emerged; Linux was the first (1991) but there are others like FreeBSD (which appeared only two years later)

- *RISC* – the ability to implement a version of Seymour Cray's 1960s ideas on a single chip, starting from the 1980s, culminating in the collapse of most medium to large-scale computing options that didn't use microprocessors

- *mobile devices* – starting from increasingly sophisticated notebook computers, mobile devices today include smart phones and tablets. Each new form factor derives from another step in the amount of functionality available at a lower price point – and able to run longer between charges or on a smaller and hence cheaper battery

- *pico-PCs* – a likely development out of smart phone parts is ultra-small PCs, of which the Raspberry Pi is an example

In some ways the "other edge" space is even more exciting than the big iron end of the design space because it creates the potential to transform the lives of many people, always remembering that technology is only a tool, and a tool only works if competently applied and to the right problem.

1.6 Structure

In the remainder of this course I examine the above topics in more detail. First, I use memory hierarchy as a starting point for understanding quantitative principles of system design and research, as well as trends and how to analyse their long-term effects. Next I examine parallelism in its various forms, starting with instruction-level parallelism. This is a large topic on its own encompassing pipelines, out of order execution, minimizing delays from branches and the rationale behind the multicore movement. Next I look at alternative models of parallelism including data parallel architectures and more specifically GPUs. I examine thread-level parallelism and how it relates to areas previously covered including memory hierarchy and multicore designs in the multiprocessors chapter, and review features of traditional vector machines as a contrast to multimedia extensions and GPUs. Finally I look at the two ends of the scale: warehouse-scale computing and emerging small-scale systems as representing two very different consequences of technology trends, the smaller devices included in discussion of how to understand and take advantage of trends.

The material covered is loosely based on Hennessy and Patterson [2012] with additions based on my own experience and research.

Exercises

1. In terms of a programmer's view, how do ILP and thread-level parallelism differ?

2. How big is the gap between a RISC architecture and a typical high-level language? Does an instruction set primarily composed of loads and stores, ALU operations and branches fit a wide range of languages?

3. Is it possible to have a program with good temporal locality but poor spatial locality? Is the opposite scenario, good spatial but poor temporal locality possible? Explain.

4. Why do multicore designs still have relatively aggressive pipelines, with over 100 instructions in consideration at any one time, register renaming and out of order execution?

2 Memory and Quantitative Design

MEMORY HIERARCHY is a critical part of computer system design because a memory large enough to contain a whole program and its data, and also fast enough not to stall the CPU, in most cases would be prohibitively expensive and almost certainly physically impossible to design. While we can rely on the principle of locality as outlined in Chapter 1 in general terms, we cannot set the size and organization of the various layers of the memory system with reasonable precision (achieve a required cost-performance trade-off) without measuring variations.

In this Chapter, I present a range of design alternatives and techniques for measurement focused on evaluating the design alternatives for memory. These same techniques can apply with differences in detail to measuring differences in design alternatives in other areas of system design.

2.1 Memory Systems

Memory systems encompass the biggest range in performance difference of any one logical component of a computer system. For this reason, there are different organization details at each layer, though there are common principles. First I present these common principles, then illustrate how they apply at each level.

2.1.1 Organization Principles

Aside from obvious classifications like speed, size and cost, memory systems are generally organised by how they can be accessed and managed. The following in general terms apply to all memory systems, with significant variations in the detail (summarised in Table 2.1):

- *naming* – some kinds of memory have unique names (generally this applies to registers), others use an addressing scheme where a location is identified

term	**definition**
block	unit of storage or management *caches*: also called *line* *VM*: fixed-size *page* (older systems had variable-sized *segments*)
hit	block is found at the requested level
miss	block is *not* found at the requested level
replacement	if there is no vacant block to place a miss another must be *evicted*
victim	block to be replaced
dirty	block modified with respect to one or more lower layers
write through	writes reflected at next layer down
write back	dirty block copied only on replacement
associativity	measure of how many different locations a block can occupy: *direct-mapped*: only 1 location for any block *n-way set associative*: *n* different locations for a block *fully-associative*: a block can be placed anywhere

Table 2.1: Common terminology. *There is some variation across layers but these terms generally apply.*

by a numeric offset from the start

- *accessible unit* – some kinds of memory are accessible in fixed-size units (again, mostly registers, though some have variants like single and double precision) whereas others can be accessed at various granularities such as a byte, two bytes, etc. The latter category may have alignment restrictions (e.g., if memory addresses refer to bytes, a two-byte access must start at an even address) and preferred sizes (a machine word is usually the width of the data bus)

- *transfer unit* – some kinds of memory only transfer to the next layer up or down in fixed size units (e.g., a cache typically has a *block*, sometimes called a *line* of fixed size; a VM system has a usually fixed page size)

- *management unit* – some kinds of memory are managed in fixed size chunks, including issues like *protection*, recording whether the contents is *modified* (sometimes called *dirty*), *valid* meaning that the unit of memory can be used without generating an interrupt, *present* meaning that the unit of memory is available at that level of the hierarchy or *shared*, meaning that more than one way exists to access that memory (usually a property of multiprocessor systems)

- *replacement* – how do we determine which unit to evict if we run out of space? If we do so, what is the policy on writing dirty data to the level below?

As we examine levels of the hierarchy we will see how these properties apply and differ. Computer architects consider faster elements of the hierarchy to be "higher" and if the same kind of memory is split into more layers, the highest level is numbered 1.

2.1.2 Levels of the Hierarchy

In considering levels of the hierarchy, it is logical to start from the top and work down. When a program starts executing, the first thing that happens is the program counter (PC) register is loaded with the start address (actually the last thing from the point of view of the software that loads the program). The ALU then attempts to fetch the instruction from the next level down, the L1 cache – but only after translating the address (on a VM machine) using the TLB, a level above the L1 cache in terms of speed. Levels below these are only accessed if the required data, page translation or instruction is not available at the topmost level. For this reason I describe the hierarchy from the top (fastest) down, though I defer discussion of some of the more complex strategies to the lower layers, since the interface between the very slowest layers and the next level up justifies sophisticated strategies to minimise access to the slowest levels of the hierarchy.

Registers

The top level of the hierarchy is registers. Registers are tightly integrated into the ALU and pipeline, and can usually be accessed in a fraction of a clock cycle. In terms of our universal principles:

- *naming* – register names are encoded into machine instructions, and generally can't be computed at run time
- *accessible unit* – registers are a fixed size though they may sometimes support precision variations (e.g., single, double)
- *transfer unit* – registers only transfer values in fixed sizes up to their widest precision
- *management unit* – registers are sometimes collectively managed in hardware, e.g., if there is hardware support for multithreading, each hardware context has its own copy of the registers. More commonly detailed management of registers is in software: the compiler manages what is within

them within a single process, and the OS manages saving and restoring registers between context switches (some older designs have hardware support for context switches)

- *replacement* – deciding which register to spill is usually totally under software (in practice, the compiler or on a context switch, the operating system) control

TLB

The next level of the hierarchy is the *translation lookaside buffer* or *TLB*, which contains recent page translations. The TLB is usually integrated into the pipeline and can be accessed in a fraction of a clock cycle. A TLB is often organised as an *associative memory*, in essence a hardware hash table that doesn't have collisions. The key being looked up is effectively the address: in this case, the virtual page number. A TLB is in the critical path of logic: if a page translation can be found, it is used immediately to check if the memory location is represented in the L1 cache. In some architectures, virtually addressed caches [Inouye et al. 1992; Wheeler and Bershad 1992] are used, making TLB speed less critical, possibly completely eliminating the need for a TLB [chan Kang et al. 2011].

- *naming* – virtual page numbers identify entries
- *accessible unit* – each item in the table is a pair: a virtual page number (to check compare against when indexing) and a physical page number
- *transfer unit* – the TLB is generally filled and replaced in units of 1 page translation though it is possible to flush it (depending on the system, this may be necessary on a context switch)
- *management unit* – as with transfers TLBs are usually managed per entry. With a virtually-addressed cache, if a TLB is present, it will need to be tagged with process IDs or be flushed on a context switch
- *replacement* – TLB replacement can in theory encompass the range of possibilities used in page replacement polices (see below: least recently used, first in first out, etc.) but in practice since the TLB is in the critical path for performance, a strategy that is fast to implement such as random replacement has some appeal

Any machine that is designed to achieve reasonable performance with VM needs hardware support for page table lookups to speed up handling TLB misses [Jacob and Mudge 1998]. For example, Intel's IA32 architecture has a hardware page table

walker that assumes a 2-level page table, reducing the time to handle a TLB miss to data references and no code in routine cases. Hardware page table walkers limit OS designers' ability to experiment with new strategies for page table design. In the worst case, a page table lookup, even with hardware support, can involve a trip to backing store, since some systems allow parts of the page table to be swapped out.

Minimising TLB misses is an aspect of performance tuning that is often neglected, and the consequences can be high. Assume an average TLB miss adds 50 cycles execution time (miss penalty). That is not an unreasonable assumption given the cost of accessing DRAM *vs*. CPU cycle time. Then if 1% of instructions result in a TLB miss on a machine that would otherwise execute 1 instruction per clock cycle, average execution time becomes

$$t_e = 1 + 0.01 \times 50$$

or 1.5 cycles, a significant drop over 1 cycle per instruction.

How can a high TLB miss rate be avoided?

A TLB represents one page translation. If you have a memory access pattern that spends very little time on one page, you will access many pages without accessing a high fraction of total memory. For example, if a page is 4KiB (the most common size), and you have a loop that looks like this:

```
for (i = 0; i < 1024 * 1024 * 1024; i+= 4 * 1024)
  a[i] = 42;
```

each assignment is on a different page. This is of course a contrived example, but it's possible to write code that scatters data references around memory if not in quite such an extreme way. For example, object-oriented code with many small objects that are not referenced in the order they are placed in memory can exhibit this problem [Machanick 1996].

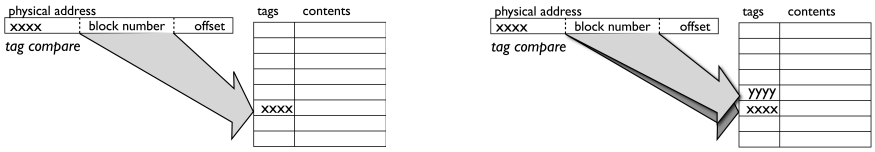
Caches

The next level of the hierarchy is caches. A cache is usually made of static RAM (SRAM), which uses transistors as its building blocks and hence draws on the same technology advances as CPUs. SRAM does not have any significant delay for an access over and above than the time to transfer its contents, so there is no special advantage to doing access in large units. A wide bus will deliver contents faster than a narrow bus because it can do so in fewer transactions, but there is no lengthy setup time to amortize.

The *top-level* (*level 1*, or *L1*) cache is usually in recent designs tightly integrated into the pipeline and can be accessed in one clock cycle. To continue

with our logical progression down the hierarchy, I describe caches before virtual memory, though VM is the natural place to describe some of the more complicated strategies since VM is closer to the operating system and hence has a higher software component.

- *naming* – cache contents is generally *tagged* with a value representing the machine address of the cache block
- *accessible unit* – when accessing a cache, the CPU uses the same units of addressing as apply to main memory
- *transfer unit* – caches contents are moved or copied in blocks (also called lines) that are typically multiple machine words long. Typical values are 32 to 128 bytes. Some caches feature *critical word first*, in which the part of the block that caused the miss is transferred first to reduce the time the CPU is stalled [Zivkov et al. 1994; Moudgill et al. 1999; Aasaraai and Moshovos 2010]
- *management unit* – most caches have tags representing the address of the contents and state (modified, valid, etc.) for each block
- *replacement* – cache replacement policy depends on how the cache is organized:
  - *direct-mapped* – a given address can only go into one location so if that location is already occupied, whatever is there is replaced: very simple to implement
  - *n-way set associative* – a given address can go in one of n locations, so if none of those is available, one has to be selected for replacement; given the relatively high speeds involved cache replacement strategies tend to be simple, though some have investigated software-based approaches [Cheriton et al. 1986] that approach the sophistication of virtual memory page replacement; for small n, hardware is still reasonably simple
  - *fully associative* – some have proposed making the lowest level of cache look more like virtual memory, and hence advocate approaches that approach the sophistication of virtual memory page replacement [Machanick et al. 1998; Hallnor and Reinhardt 2000], including allowing a cache block to be placed anywhere in the cache: to implement full associativity purely in hardware is expensive and impractical for a large cache since every location has to be searched to compare the address tag with the request



(a) Direct-mapped (b) 2-way associative

Figure 2.1: Cache addressing. *The low-order bits of the address are used to find the right byte or bytes within the cache block. The next-lower bits are used as an index into the cache, and the high-order bits in excess of those needed to identify a cache block are stored in the tag to identify which of the possible blocks is actually in the cache. For higher associativity, cache indexing produces more than one result and a hit is detected by checking if any of the stored tags matches the required block.*

In Figure 2.1 I illustrate how a machine address is broken up to check for a hit in a cache. In the event a hit is found, the cache returns the required memory items (or in the case of a write, overwrites the portion of the cache block pointed at by the address). In the case of a miss, the cache controller must identify a victim (in a direct-mapped cache (2.1(a)), that's always the same location; in other organisations (2-way associative, 2.1(b)), a victim may be picked at random since time is short at this level of the hierarchy), and request the block from the next level down. If the victim block is *modified* (or *dirty*), it must be written back to the next level down (*cleaned*). Caches can have one of two write policies:

- *write through* – all writes immediately are reflected at the next level down: seldom used because memory traffic is high
- *write back* – a dirty block is cleaned on replacement

In addition to the address tag, cache blocks have status bits. These can vary but common examples include:

- *modified* – a bit indicating the block is dirty
- *exclusive* – a bit indicating the block is not shared with another CPU or core
- *shared* – a bit set if the block is shared across more than one CPU or core
- *invalid* – a bit set if the block does not have valid contents

A system with this exact set of states (not necessarily that number of bits) is referred to as using the *MESI protocol*[1]. You may wonder why you need a

[1] A few other details, specifically restrictions on the allowed combinations, apply to the definition of MESI.

shared and an exclusive state. If a block is not initially shared, setting it exclusive makes this clear. We go more into shared caches when considering multiprocessor systems (Chapter 4).

In most systems currently available, there is more than one level of cache. The L1 cache is relatively small and tightly integrated into the ALU so it can keep up with the pipeline. The L2 cache is larger and not as fast; some systems have 3 or even more levels of cache, on the principle that as much cache as possible is good but a large one cannot be fast without high costs in energy, a significant factor in design.

In aggressive ILP designs, a cache miss can cause a major slowdown. With a clock speed of 2GHz, one cycle is 0.5μs. If you can execute 4 instructions on one clock, the average time per instruction is 0.125μs so even if your second-level cache is very fast with hits taking only 1μs, a miss costs a delay of 8 instructions. To address this problem, non-blocking caches allow any instructions that are ready to go to continue without waiting for a cache miss [Chen and Baer 1992; Belayneh and Kaeli 1996; Aasaraai and Moshovos 2010]. If your ILP design already includes out of order execution, likely with an aggressive design, support for non-blocking caches is a relatively cheap addition.

For multicore systems, a common approach is to have an L1 cache that is local to each core and a shared L2 cache. Shared caches is an idea explored in research into high-end systems in the past [Cheriton et al. 1988, 1989; Nayfeh and Olukotun 1994] – illustrating the value of a thorough understanding of technology history as technology change makes it possible to package old high-end ideas at new affordable price points.

Main Memory

The main memory in current systems is generally made of DRAM. DRAM uses a capacitor as its storage element. Unlike SRAM, DRAM has to be refreshed periodically because a capacitor's charge drains. Because the underlying technology is different, DRAM has its own price-performance trend, and that is driven more by price per bit than by speed. Hence, DRAM speed improvement lags CPU speed improvement (less so since the move from aggressive ILP and higher clock speeds to multicore, but multicore designs still are growing faster than DRAM speed, if you aggregate the rate at which memory requests occur across the cores). Also unlike SRAM, there is a lengthy delay before the contents can be accessed, so most current DRAMs have streaming modes where, once an access is set up, further sequential accesses moving along from that location are a lot faster. For this reason, moving to or from DRAM in large units is attractive if it does not cause other delays.

- *naming* – a memory address usually refers to a byte, numbered from the start; many machines require aligned access for large units (e.g., to do a 2-byte access, you must start on an even address)

- *accessible unit* – most DRAM systems are accessible at the byte level though in practice to handle cache misses, write-backs and write-throughs, a larger unit is transferred

- *transfer unit* – the transfer unit is the same as the access unit in practice, since most DRAM access are via the cache.

- *management unit* – at the low level, DRAM can be managed down to the byte level but in practice, with a VM system, what is in the DRAM or not is managed in pages

- *replacement* – replacement strategy in VM is complex and must take into account the mix of processes, other IO (since paging uses an IO device) and the extremely high latency of backing store. Some strategies include:

  - *least recently used* or *LRU* – the page used longest ago is evicted
  - *first in first out* or *FIFO* – the oldest page is evicted
  - *working set* or *WS* – each process is limited to pages it used over some fixed time period
  - *clock* – a way of approximating LRU by systematically marking pages as unused, working around the list of pages in the style of a clock hand, and selecting a victim that is not marked as used (indicating the page was not used since the clock hand last passed that page)
  - *page standby list* – a list of pages recently target for eviction [Russinovich 2007]

In some systems there may be a mix of *global* and *local* policies: a global policy balances DRAM use across processes, while a local policy attempts to ensure that a given process has enough DRAM to make progress. A local policy generally attempts to implement the *working set* principle: a process generally only access a subset of its pages for a reasonably long time before shifting to another part of its code or data address space. While the working set concept is quite old [Denning 1968], the principle still applies and will as long as memory has a hierarchy with several orders of magnitude difference in speed. A global policy may sometimes simply shut down processes if there is insufficient RAM (in the worst case, terminate them).

A complete coverage of virtual memory properly belongs in an operating systems course since it's at the interface between hardware and software, and software plays a much larger role than in higher levels of the hierarchy.

Paging Device

Paging devices historically have been mechanical magnetic storage devices of various forms. Early paging devices were dedicated magnetic *drums*, conceptually the same as a disk but with the recording surface on the outside of a cylinder. The earliest commercial VM system, the British Ferranti Atlas [Lavington 1978], had a drum memory with rotational time of 12ms (and thus an average rotational delay of 6ms), and no seek time since the heads were fixed, making it competitive with technology of 50 years later on speed if not capacity. The basic cycle time of the CPU was 2μs, only about 10^3 faster, compared with today's speed gap of a factor of over 10^6. It is the observation that in the late 1990s the delay in handling a cache miss to DRAM was approaching 3 orders of magnitude slower than CPU cycle times that led me to starting the RAMpage project, in which I move the virtual memory system up a layer to handle misses from SRAM to DRAM [Machanick et al. 1998] – so knowing a bit of history is useful.

Today paging is usually on standard disks. There are two major variants: the traditional UNIX approach of a swap partition, and using free space in the boot partition. Mac OS X uses the latter; Linux can use either. On iOS devices, which usually use flash instead of disk, paging is limited to evicting easily recreated content such as code from RAM. There are two reasons for this strategy: flash is small on these devices compared to a disk, and repeated modification of the same bits in flash can wear them out. Programmers of iOS apps are advised by Apple to accept low memory messages and reduce their memory footprint as required [Apple 2012]. Other types of device (including Apple's notebook line) use flash instead of disk and do use flash for paging; reducing the tendency to wear out over-used bits using *wear levelling* [Chang 2007] may be easier with Apple's strategy of sharing the file system with backing store rather than using a separate swap partition.

- *naming* – a page on backing store can be anywhere on the device and is identified by a page table, using the virtual address (or more properly the virtual page number) as an index
- *accessible unit* – a VM system usually deals in whole pages
- *transfer unit* – pages may be transferred singly or the OS may move several contiguous pages to reduce overall latency

- *management unit* – pages are managed as a unit but also by process; if a process completes or dies, all its pages are freed

- *replacement* – since this is the bottom of the hierarchy, there is no replacement until a process exits the system; however, some systems do not keep pages on backing store if they exist in RAM and in that sense pages may not always exist on disk.

In a difference to cacheing terminology, a miss is called a *page fault*. In most real systems, a page fault results in a *context switch*: there is no point stalling the CPU for millions of cycles so despite the fact that a context switch has other significant costs like losing contents of caches, it is faster overall to allow another process to use the CPU while waiting for a page fault to be processed.

Having wended our way all the way down from the world of registers and TLBs that are accessible in a fraction of a clock cycle to paging devices that area accessible in millions of cycles, let's see how we measure the effects of all of this.

2.2 Measurement

There are many levels at which we can measure computer systems performance. We can measure individual components, we can measure times taken by small blocks of code, we can time a whole program, and we can time a workload of interest. Aside from timing overall, we can apportion costs, so as to work out what to improve. Then in addition to timing, we can measure other attributes of interest like energy use, memory requirements if we change some detail (e.g. simplify the instruction set) and frequency of use of specific features.

2.2.1 Architecture-Oriented Measures

Depending on what we are measuring and how much detail we want, there are many variations, including:

- *logic-level simulation* – useful for checking design details like timing and energy use, but too slow to measure non-trivial program runs though work on speeding up such simulations may make larger runs viable [Chatterjee et al. 2009; Mironov et al. 2010]

- *execution-driven simulation* – a program runs on a simulator which can measure at a particular (sometimes parameterizable) level of detail including

  - *cycle-accurate simulation* – simulation run in software designed to give an accurate representation of machine time or energy use [Simunic

et al. 1999]; slow for large runs though recent enhanced techniques make such methods more viable for whole workloads [Lee et al. 2008]

  - *whole-system simulation* – while not necessarily cycle-accurate, these simulators are fast enough to evualate whole workloads

- *trace-driven simulation* – a record of memory accesses (usually classified as read, write or instruction fetch) is read by these simulators, allowing memory system variation to be modelled (instruction variation can only be modelled in a limited way since the actual instructions are not recorded, and changes in execution order cannot easily be modelled)

- *emulation* – emulation differs from simulation in that it only aims to run a non-native instruction set rather than to provide accurate performance data

- *profiling* – measurement of relative times spend on different parts of a program; profiling can be implemented as a feature of a simulator [Cmelik and Keppel 1994] but it is more commonly implemented by instrumenting code [Reddi et al. 2004]

- *back of the envelope* – quick calculations that quantify relatively simple effects; limited in applicability since a whole system includes complex interactions between all influences on performance

From the difference in goals of emulation and simulation arises an interesting question: is it possible for a simulation to be too good? While real systems have variations in execution time that can't be eliminated arising from interactions between processes and interactions with external events, to produce repeatable results for scientific investigations, you need repeatable measurement. For an emulator, you care less about repeatable measurement and more about both accurate implementation of the target system as well as speed and minimal resource requirements. For a simulator, while those factors are important, it may be reasonable to sacrifice a little accuracy or speed for repeatability. In a sense then it *is* possible for a simulation to be *too good*.

2.2.2 Benchmarking

When we are really only concerned with comparing competing systems, rather than pinning down where the time is spent, benchmarking – comparing standard program runs against competing systems – is popular. Benchmarks fall into two broad categories:

- *kernels* – useful for testing how some very specific feature compares across architectures, e.g., floating point multiplication

- *full workloads* – programs that exercise the whole system including the file system, the memory hierarchy and even the network in ways representative of one or more classes of real programs; some examples include
  - *SPEC* – divided into integer and floating point scores [Henning 2006] and widely used specially in the UNIX space to compare systems
  - numerous other benchmark suites to evaluate web server performance (e.g. SPECweb – discontinued in 2012), database scalability (e.g. TPC bechnmarks [Nambiar et al. 2011]), energy [Poess et al. 2010], embedded systems [Guthaus et al. 2001; Schoeberl et al. 2010] and other specific kinds of workload

One of the hot issues in benchmarking is gaming the system. For example, creating a compiler that recognises a specific benchmark and inserts hand-tuned code that no compiler could generate, or including a special instruction that is hard to use in general are tricks used in the past. Kernels have to some extent fallen into disuse because they are so easy to hand-tune or otherwise arrive at fake results that do not predict real system performance. Even with SPEC benchmarks, which are whole programs of the size of a compiler run, I've had the experience of running my own code on two machines one of which had double the SPEC rating of the other, and my own code reversed this to the extent of the "slower" machine on published SPEC results running in half the time of the "faster" machine.

In my experience the best benchmark is the workload of interest to you, run under conditions representative of your usual work (e.g., running a canned installer, or compiling it yourself, then running it with a system loaded the way you usually run).

Since benchmarks are most useful for comparing competing machines rather than elucidating performance details of system components, I do not include them as an example for measuring memory systems.

2.3 Putting it All Together: Measuring Memory Systems Performance

Since memory references occur at least once for each instruction (an instruction must be fetched from memory and may also move data to or from main memory), an accurate simulation of memory systems performance making it possible to compare different options should really simulate most aspects of the pipeline. I examine here the variations that can be useful, starting from those that simulate the least detail.

term	**definition**
miss rate	fraction of references at a level that miss
global miss rate	miss rate over all references
local miss rate	miss rate at a given level
miss penalty	extra time arising from a miss
hit cost	time for a hit

Table 2.2: Performance parameters. *The most important thing is elapsed time; minimising miss rate for example is not an end in itself.*

Table 2.2 lists some terminology of use when evaluating memory performance. At the top of the hierarchy, the cost of a hit is often absorbed into a pipeline stage and hence not counted. At lower levels, we usually count the hit cost as part of the miss penalty for the layer above. When evaluating memory system alternatives, we care most about overall run time. Minimising miss rate for example may seem like a good idea, especially if as big speed gap is involved, but if doing so slows down the faster layer, there may not be an overall win.

2.3.1 Back of the Envelope Calculation

To get a quick feel for the effect of design parameters we can do simple calculations of the likely effect, remembering always that such calculations can be misleading because they do not take into account the full range of interactions of components. For example, with an aggressive pipeline that allows instructions to continue through the pipeline when others are stalled waiting for a cache miss, a simple calculation of the effect of increasing or decreasing the miss rate is at best a crude approximation.

Let's nonetheless look at an example in detail and at the same time introduce some terminology for speed comparison.

In Table 2.3 I list common measures of speed improvement. A speedup greater

measure	**definition**
speedup	$\frac{time_{original}}{time_{new}}$
improvement	$1 - \frac{time_{original} - time_{new}}{time_{original}}$

Table 2.3: Performance improvement measures. *Improvement is often given as a percentage. Dividing by t_{new} is very misleading and greatly exaggerates the % improvement.*

level	**hit or miss**										penalty
1	h	h	h	h	m	h	h	h	h	m	10
2					h					m	100
3					m					h	

Figure 2.2: Example of miss rate calculation. *We need to account for misses to L2 and L3, since there are no misses from L3. Assume hits in level 1 take 1 time unit, and penalties are relative to that.*

than 1 means you are doing better; a speed improvement greater than 0 means you are doing better. Speed improvement is a risky measure to use because "50% faster" doesn't sound nearly as impressive as "150%" faster so many people especially in marketing forget to subtract the 1. You also get a very different answer if you look at improvement relative to the faster rather than the slower system. When quantifying speed improvement, make sure you define your terms.

To calculate the effect of misses, we need an execution time formula, which I generalize to allow more than one level of cache (and the main memory could also simply be counted as another level; going to a paging device is more complicated because the operating system is involved and hence a simple miss penalty does not apply):

$$t_e = t_{h_1} + \sum_{i=1}^{n} p_{m_i} \times r_{m_i} \tag{2.1}$$

where t_e is *relative execution time*, normalized to 1=no misses; actual execution time is $t_e \times IC \times t_{clock}$, where p_{m_i} is the penalty of misses from level i and r_{m_i} is the rate of misses from level i (we use a global miss rate here since we want to quantify the effect on overall run time). It is useful to leave out the instruction count IC because that way we can compare scenarios where we don't vary the instruction count without knowing exactly how many instructions were executed. We also leave out the clock cycle time t_{clock} since that allows us to compare scenarios of similar clock speed without needing to fix the clock cycle time. This general formula can be adapted to include other causes of stalls.

For example, in Figure 2.2, we have miss rates from L1 of 0.2 and from L1 of 0.1. Applying the formula with the penalties given results in:

$$\begin{aligned} t_e &= 1 + 10 \times 0.2 + 100 \times 0.1 \\ &= 13 \end{aligned}$$

In a real system, you would expect much lower miss rates than this, especially to the lower level (and slower) parts of the hierarchy.

For simplicity I assume that the L1 hit time accounts for all execution time, which is true in the case of a pipelined architecture. There is the possibility of

misses for both data and instruction references, and we also need to ensure that we do not double-count hit time at level $i+1$ so we should not we treat an access at that level as part of the miss penalty of level i if we count it as part of the hit time at level $i+1$. However you do this make sure you make it completely clear what you are including in the calculation and why.

Case Study

A simple example illustrates design trade-offs. Assume we have two ways of designing a cache. A direct-mapped cache has very simple logic (a given address can only map to one block in a cache) but has the drawback that it can have a high miss rate, since some combination of addresses used repeated in close proximity that possibly coincidentally map to the same block can evict each other when the cache is nowhere near full. A 4-way associative cache (4 different ways you can place any given address) can avoid this problem at the cost of slower cache reference time. Assume:

- *effect on hits* – the 4-way associative hit time is 10% slower than the direct-mapped hit time
- *effect on miss rate* – the 4-way associative cache has 20% fewer misses
- *miss penalty* – a miss from this level of cache costs 100× a hit in the direct-mapped cache

Calculate the miss rate at which the two caches have the same performance, and hence the point at which it becomes useful to use the 4-way cache.

Solution

We assume that the miss rate r_m is *relative* to this level of cache since we don't know anything about the rest of the hierarchy. We don't know absolute times so make the direct-mapped hit time t_d and base everything on that:

- *hit time at level* $i \equiv t_{h_i}$; for this example:
  - *direct mapped hit time* $\equiv t_d$
  - *4-way associative hit time* $\equiv t_4 = 1.1t_d$
- *miss rate at level* $i \equiv r_{m_i}$; for this example:
  - *direct mapped miss rate* $\equiv r_d$
  - *associative miss rate* $\equiv r_4$

- *miss penalty from level i* $\equiv p_{m_i}$; for this example, only one level with $p_m = 100t_d$

To find the break-even point, we can adapt the execution time formula (2.1) to make it easier to compare our two cache variants without excessive notation. For this example I need only 1 level and drop the i subscript, and derive variants for each case, which I need to set equal to find the break-even point:

$$t_{e_d} = t_d + p_m \times r_d \tag{2.2}$$

$$t_{e_4} = t_4 + p_m \times r_4 \tag{2.3}$$

We know that the 4-way associative cache has 20% fewer misses, and its hit time is 10% slower than the direct-mapped cache, so we can rewrite Equation 2.3 as follows:

$$t_{e_4} = t_d \times 1.1 + p_m \times r_d \times 0.8 \tag{2.4}$$

and the miss rate at which the two equations have the same execution time occurs when Equation 2.4 = Equation 2.2. So we need to solve for r_d in:

$$t_d + p_m \times r_d = t_d \times 1.1 + p_m \times r_d \times 0.8 \tag{2.5}$$

Put all the r_d terms on one side, and put everything in units of direct-mapped hit time t_d, noting that $p_m = 100t_d$:

$$p_m \times r_d - p_m \times r_d \times 0.8 = t_d \times 1.1 - t_d$$

and simplify:

$$100t_d \times r_d(1-0.8) = t_d(1.1-1)$$

$$0.2 \times 100t_d \times r_d = 0.1t_d$$

$$20t_d \times r_d = 0.1t_d$$

So the break-even point is where

$$r_d = 0.005 \tag{2.6}$$

To put the answer in English, in this scenario, we need at least 0.5% of the hits in the direct-mapped cache to be misses before changing the design to a 4-way associative cache is a win.

Is this result surprising?

Having done a calculation like this, look back at the numbers to see if the answer makes sense. A miss penalty of 100 is pretty big in relation to the penalty of 10% slower hits for the 4-way associative cache so it shouldn't take a high

number of misses for a reduction of 20% to be a win even given a small increase in hit time. The answer therefore looks plausible. Now go back to Equation 2.5 and check that $r_d = 0.005$ does indeed make the two sides equal and that a larger value of r_d does make the direct-mapped formula (Equation 2.2) for run time bigger than the 4-way associative formula (Equation 2.3).

Another way to check this sort of calculation is to see if you end up with the right units. We want a number expressed as a fraction without units like seconds or number of instructions executed, since a miss rate is just a dimensionless fraction. If you end up with something that has the wrong units, you've probably forgotten to cancel something out or made a mistake in moving terms around.

In practice, most CPUs have two or more levels of cache to reduce the need for this sort of design trade-off. The L1 cache can be as fast as possible, and the L2 cache can be designed with a few compromises on raw speed to reduce miss rate.

2.3.2 Profiling

Profiling is most useful to ascertain where time is spent on an existing architecture for a given workload, and is most often used as a tool to tune performance of a given program or set of programs rather than as an architecture design tool. The reason for this is that profiling does not allow the option of varying design parameters on a real system, and there is little point in doing profiling at the application level on a simulator, since you can instrument the simulator.

That said an understanding of architecture can inform your approach to profiling. If you understand the role of various system components like caches and the TLB, you are in a better position to understand where to look for improvements.

2.3.3 Trace-Driven Simulation

A trace-drive simulation takes as input a *trace file*, containing addresses tagged as one of a *read*, *write* or *instruction fetch*. It is possible to simulate multitasking workloads by interleaving traces, including traces simulating operating system functions, though the OS component necessarily must be an approximation.

Given speed improvements in direct execution simulation, trace-driven simulation is not as popular as it used to be [Borg et al. 1990; Uhlig and Mudge 1997; Engblom and Ermedahl 1999], though there is still a fair amount of research conducted using traces. It is nonetheless a useful tool for testing new ideas independently of CPU details. It is not very hard to create a simple trace-driven simulation, and there are tools to generate traces (e.g., PIN [Reddi et al. 2004]).

To measure memory system variation, the same trace file can be run through different models of the memory hierarchy (e.g., different sizes, organisations and speeds of caches). A simulation may also be sped up by starting the trace at a

point of interest in the code (e.g., skipping initialization). Although there is some inaccuracy, you aim to make than inaccuracy minimal as a fraction of the total run. With the aid of profiling it may be possible to isolate out parts of a program that contribute most to run time and focus on those, not forgetting that effects of the parts of the program not measured can perturb the results.

2.3.4 Whole-System Simulation

Since performance of direct-execution simulations improved so that they run at a reasonably small slowdown over running on real hardware, it has become increasingly common for such simulators to support running a full system including an operating system, making for higher accuracy in measuring inter-process and system influences on performance. A good example of an academic project for full-systems simulation is M5 [Binkert et al. 2003] from University of Michigan and its successor gem5 [Binkert et al. 2011]. Gem5 has full-system support for the Alpha, ARM, SPARC and Intel x86 instructions sets. Alpha historically was a popular architecture for research because it is one of the cleaner RISC designs, though it is no longer in production.

A full-system simulation allows not only detailed variation of the cache architecture but also parameterization of memory system performance down to disk and even network layer, and potentially changing the page table structure, if you have the fortitude to rewrite the operating system interface to the hardware.

A factor that mitigates against the slowdown of full-system simulation is ubiquitous PCs capable of running Linux. Rather than run one simulation faster (as you can do with a less detailed model), you can run many instances of the simulation with different parameters if you are lucky enough to be in a university with large numbers of PCs in student labs that researchers can take over at off-peak times.

2.3.5 More Detailed Approaches

It is seldom that low-level cycle-accurate simulation is necessary for evaluating memory system variations. The biggest performance effects are excursions down the hierarchy, rather than at the level of registers or the pipeline, so small inaccuracies in timing at those levels have an insignificant effect compared with a small change in miss rate. If you are checking a design for correctness, that's a different matter and cycle-accurate simulation as well as mathematical approaches to formal verification play a significant role.

2.3.6 Summary

For most research today, a full system simulation is the approach of choice. For classroom examples, we do paper exercises. For small-scale design studies, trace-driven simulation still has a lot to recommend it. We seldom need more detailed simulations purely to evaluate overall system performance but if producing a new design, we may want to do cycle-accurate simulation to check design assumptions, e.g., for the time a specific implementation should take for given operations (as well as to validate the design, as I describe above). For example, you need to at least work through the timing of the extra logic needed for a 4-way associative cache to know what percentage slower it is than a direct-mapped cache (the 10% number in the case study is not based on a real example).

Learning about a few publically available research tools is useful: do a web search to build on the examples listed here. Also practice at examples of back of the envelope calculation. These are useful to build an appreciation of how performance trade-offs work, even if they are poor indicators of overall system performance.

Exercises

1. Assume in the absence of misses a machine executes on average 1 instruction per clock. You are investigating a new page table organization that reduces page faults by 1%; to implement this you will lose hardware support for TLB misses. Assume a page fault takes 1-million cycles to handle, and a TLB miss without this improvement takes 50 cycles with hardware support. Without hardware support, a TLB miss takes 100 cycles. Apply the general multilevel miss formula (Equation 2.1) here with the TLB as level 1 and the page fault as level 2:

   (a) What is the net speed gain (or loss) of the improvement if 0.1% of TLB references are misses?

   (b) Is this a useful calculation for a real system? Consider what a real system does on a page fault.

2. Assume we have a machine that in the absence of misses executes on average 2 instructions per cycle. Such a machine would have a higher peak throughput but would be limited by other limits on ILP such as branches.

   (a) Redo the calculation of the case study (p 33, section 2.3.1) under this assumption.

   (b) Now allow for a non-blocking cache that can avoid a stall on average for 5 instructions before having to stall.

(c) Is a non-blocking cache a useful improvement given the miss cost of this example? When might your answer change?

3. We are considering splitting the cache of the case study of page 33 into two levels as alternative to the 4-way associative cache. The miss penalty from L1 to L2 in this new alternative is 10 times L1 direct-mapped hit time (t_d in the case study), and the miss penalty to DRAM is unchanged. In this new scenario:

   (a) Work out the miss rate from L2 required to break even with the single-level 4-way associative cache cache.

   (b) Comment in general on the value or otherwise of an L2 cache as opposed to fine-tuning the parameters of L1.

4. *Blocking* is a programming technique (not to be confused with cache *blocks*, or non-*blocking* caches) where a fraction of the total data structure to be processed in an algorithm is used as much as possible before moving on to another part of the data structure [Lam et al. 1991]. Discuss how blocking can aid performance.

3 Pipelines and ILP

PIPELINES ARE AT THE CORE of instruction-level parallelism so I discuss the two together. A pipeline, sometimes pipe for short, is based on the same principle as assembly-line mass production. If you break a task down into smaller tasks, each requiring the same time to complete, you can dramatically speed up overall operation, even if completing one task is not sped up, because you overlap multiple tasks each at a different stage of the pipeline (or production line).

The key to pipeline performance is *balanced stages*. If one stage takes a lot longer than the others, that stage determines performance. Another consideration is overheads in moving from one stage to the next, which limits how deep a pipeline is practical. Another limitation on how deep a pipeline is practical is the cost of flushing the pipeline when instructions at various stages turn out not to be needed, usually on a branch instruction.

Instruction-level parallelism builds on pipelining by adding options of out-of-order execution and more than one instruction per clock. These additions, as noted in Chapter 1, go back to the early work of Seymour Cray in the 1960s. Because RISC architectures lend themselves naturally to aggressive pipelines, some commentators erroneously label such features as "RISC-like", including in versions of the Intel IA32 (and of course IA32-64) architecture, which clearly does not have the attributes of typical RISC ISA. A RISC architecture makes aggressive ILP easier to design, but there is no reason in principle that any other ISA should not also feature an aggressive ILP implementation.

In this chapter, I review basics of pipelining and go on to show how ILP can be added onto a basic design. Much of the discussion is based on pipelines that complete at most a single instruction per cycle, and that have the same total execution time. Pipelines that allow multiple instructions per cycle (*superscalar* pipelines), and floating-point pipelines with instructions that have multiple execute cycles considerably increase complexity.

3.1 Simple Pipelines

Pipelines can be organized with many variations on the number and type of stages. To keep things simple, I start out with a 5-stage pipeline that is relatively easy to implement for integer instructions using a RISC ISA. The stages are (in some cases, allowing for variations in instruction types):

1. *instruction fetch* (*IF*) – use the program counter register (PC) to load the next instruction and increment the PC
2. *instruction decode* (*ID*) – decode the instruction and also read register values from source operands; compute the branch target address; sign-extend immediate operand values
3. *execution* (*EX*) – complete ALU operations using previously prepared operands including:
   (a) *memory reference* – add the offset to the base address
   (b) *branch* – determine branch outcome
   (c) *register-register ALU operation*
   (d) *register-immediate ALU operation*
4. *memory access* (*MEM*) – for a load instruction, fetch the data from memory; for a store, send the data to memory from the register whose value is to be stored
5. *write-back* (*WB*) – for ALU operations and memory loads, copy the result to the destination register

In a RISC ISA, much of this is radically simplified. For example, in IF, we can do all the possible options simultaneously and drop any not needed, because register operands are always in the same place in an instruction. We need sign extension on immediate operands because a negative value has all 1s in the most significant bits if we extend the precision. An immediate operand is built into the instruction and is therefore smaller than a machine word.

We see the value of the load-store architecture of a RISC ISA here. Because no instruction does both a memory reference and an ALU operation, a single pipe stage can do any part of either kind of operation.

This design does not complete all instructions in uniform time. A branch can complete in the second stage, a store in the fourth and all other instructions need all five stages. Nonetheless it is a simple design and easy to pipeline simply by starting an IF on every clock.

	clock number								
instruction no.	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i+1$		IF	ID	EX	MEM	WB			
$i+2$			IF	ID	EX	MEM	WB		
$i+3$				IF	ID	EX	MEM	WB	
$i+4$					IF	ID	EX	MEM	WB

Figure 3.1: Progress through a 5-stage pipeline.

In Figure 3.1, I illustrate progress through a pipeline, assuming each instruction can start without delay. This is a common notation for illustrating progress through a pipeline and counting up total elapsed clock cycles. In this example, each instruction can start immediately and continues for all five stages without a break. In real examples, instructions my *stall* for various reasons, adding a *bubble* to the pipeline. A more realistic example should take into account *dependences* between instructions, e.g., if one instruction creates a value, a following instruction cannot enter the pipe stage where it needs that value until it's ready.

Another notation used to illustrate a pipeline uses a picture of a datapath, repeated starting once for each stage, showing the components active at each stage. The advantage of this notation is that it's easy to visualise dependences between stages. Figure 3.2 based on the style used by Hennessy and Patterson [2012] illustrates how the datapath can be visualised in this time-shifted way. The grey boxes between stages represent *pipeline registers*, which pass values between

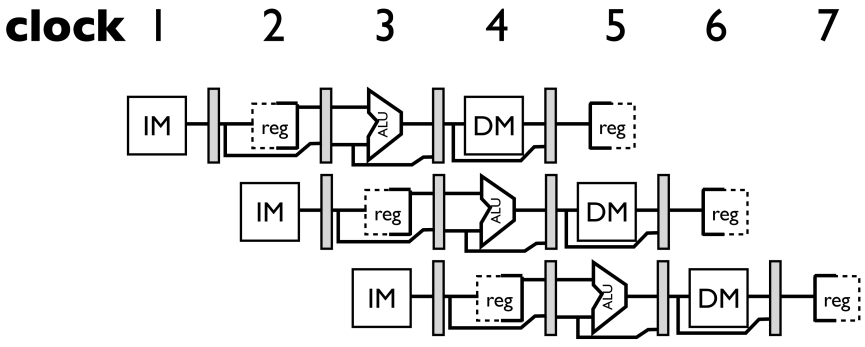


Figure 3.2: Pipeline progress with datapaths. *The register file is repeated: the first time it appears where it's read, the second time where it's written. The grey boxes represent inter-stage registers. Instruction (IM) and data (DM) memory are represented separately though they are logically in the same address space, capturing the property of most L1 caches that are divided between instruction (I) and data (D) caches.*

stages. Because the register file is accessed at two different stages, it appears twice, with a broken line on the left if it's being read and on the right if it's being written. You can clearly see with this notation if a dependency may exist because the pipe stages where registers are accessed. The notation for the register file is useful because it contains a hint that a modification to a register and a read may be possible on the same cycle if the modification happens in the first half of the cycle and the read in the second half.

3.1.1 Pipeline Limitations

Our 5-stage pipeline isn't the only organisation possible. Some designs have fewer stages and the later versions of the Pentium 4 architecture had as many as 31 [Zukowski et al. 2006]. Very deeply pipelined machines are sometimes referred to as *superpipelined*. The theoretical gain from a deeper pipeline – more instructions in parallel hence theoretically greater speedup – is offset by various costs. These include:

- *clock skew* – longest delay between the clock arriving at any pair of registers
- *propagation delay* – the pipeline registers are fast but each new stage adds delay
- *cost of pipeline flushes* – the deeper the pipeline the more instructions are lost when the wrong instructions are in the pipeline; this adds not only to the cost of branches but also of context switches

In general super-deep pipelines have been explored and not had big enough performance wins to remain in the mainstream.

Another complication in pipeline design is floating-point instructions. For practical purposes, it is not possible to complete some of the more complex operations like floating point divide in one cycle, meaning that the clean simplicity of a RISC pipeline with uniform instruction handling is broken.

3.1.2 Pipeline Performance

Once we've worked out the number of stages and any delays between stages, we can work out a theoretical peak execution rate, which is just the clock rate. The clock rate is limited by the time of the longest pipe stage plus overhead. A 5-stage pipeline can at most result in a speed up of 5 over a non-pipelined machine. A real machine though will have *bubbles* in the pipeline induced by *stalls*, and therefore not achieve its theoretical peak throughput.

Case Study

Let's look at an example. The timing for each stage has to be worked out by doing a proper logic design, and working out the longest logic path at that stage. Here, I use invented numbers to illustrate the principle. Assume inter-stage logic has an overhead of 0.1ns, and the following times for each stage:

1. IF – 0.5ns
2. ID – 0.4ns
3. EX – 0.3ns
4. MEM – 0.5ns
5. WB – 0.2ns

The longest stage takes 0.5ns, and overhead is 0.1ns, so this sets cycle time at 0.6ns (1.67GHz; to convert between Ghz and ns: GHz = $\frac{1}{ns}$). How much speedup is this over a non-pipelined implementation? Superficially, we can add the cycle times of the nonpipelined machine, but we should also take into account the fact that some instructions don't use all stages and in a nonpipelined implementation could therefore finish faster. In our 5-stage pipeline, only memory operations need all 5 stages (other instructions are idle in the MEM stage). To work out what average instruction execution time a non-pipelined machine takes, we need an *instruction mix*. Assume instructions break down as follows (as a fraction of all instructions executed in a particular workload):

- load – 20%
- store – 10%
- branch – 20%
- ALU operation – 50%

We can now work out an average for a non-pipelined instruction, in which 30% (loads plus stores) use all 5 stages, and the rest skip the MEM stage:

$$\begin{aligned} t_{nopipe} &= 0.7 \times (0.5+0.4+0.3+0.2) + 0.3 \times (0.5+0.4+0.3+0.5+0.2) \\ &= 0.7 \times 1.4 + 0.3 \times 1.9 \\ &= 0.98 + 0.57 \\ &= 1.55ns \end{aligned}$$

So our actual speedup is $\frac{1.55}{0.6} = 2.58$, significantly less than a speedup of 5 that you would predict from a superficial understanding of pipelining.

It is tempting given the numbers in our example to split the pipeline stages. Assuming we can split each longer stage into two stages, each half the size of the original (of course with overhead as before, but now for more stages), can we do better? Let's work the numbers, aiming for a new maximum stage of 0.25ns:

1. IF1 – 0.25ns
2. IF2 – 0.25ns
3. ID1 – 0.2ns
4. ID2 – 0.2ns
5. EX1 – 0.15ns
6. EX2 – 0.15ns
7. MEM1 – 0.25ns
8. MEM2 – 0.25ns
9. WB – 0.2ns

We how have 9 stages, and the longest is 0.25ns, so our cycle time is 0.35ns with overheads, a speedup of 4.4 over the non-pipelined design, and 1.7 over the 5-stage pipeline. That looks worthwhile but as we will see later, this is not the whole pipeline story, and we need to take into account pipeline stalls before declaring a clear win.

What if we take this to the limit, and make each stage 0.1ns, the same as the overhead? In this case, we have 19 stages and the cycle time is 0.2ns, a speedup of 3 over the 5-stage pipeline, and 7.8 over the non-pipelined design. However, we have thrown a lot more hardware at the problem and we incur other significant costs, e.g., as we see when we deal with branches, we have significant costs of having the wrong instructions in the pipeline. With these numbers, it should be clear that further reducing the stage size has little benefit.

Hazards

Now we hit the hard part of pipelining, quantifying the costs when we have bubbles in the pipeline. A pipeline has an empty time slot when it can't proceed because of a dependency or resource constraint, generally called a *hazard*. Hazards fall into three categories:

- *data hazards* – data dependences prevent progress, divided into:

  - *read after write* or (*RAW*) – any use of a data value after its changed including registers and memory locations, though mostly registers in our examples: the main challenge is ensuring the updated value is read
  - *write after write* or (*WAW*) – any attempt to change a data value after another change: making sure the last change sticks is the main challenge
  - *write after read* or (*WAR*) – this hazard in less aggressive designs can be avoided by writing to registers and memory in a late stage; see Figure 3.2 for example where the "DM" box representing the MEM pipeline stage where movement of data between memory and registers happens, and the second "Reg" box representing the WB pipeline stage are the two latest pipeline stages

- *control hazards* – a change (or possible change) in order of execution prevents progress

- *structural hazards* – a limit on hardware resources prevents progress (e.g., a functional unit is not available to two instructions that need it on the same cycle, something not a problem with our simple pipeline)

To quantify simple examples, we need a machine code instruction set. We base ours on a generic RISC architecture, with ALU operations that either take two register source operands and one destination, or the source operands can include an *immediate* operand, a value encoded into the instruction. Memory data references are all either loads (copy from memory to register) or stores (copy from register to memory). We assume 32 registers (named `R0`...`R31`, with `R0` always the value 0), and a 32-bit instruction word.

A few things to note:

- *operand widths* are specified in the instruction as "`l`" for a long word of 8 bytes, "`w`" for a 4-byte word, "`s`" for a 2-byte short and "`b`" for a single-byte operand at the end of the operation name

- *unsigned operands* are specified in the instruction as "`u`" after the operand name

- *immediate* operands are encoded into the instruction and limited to 16 bits so, to extend the range of possible values, when they are used as address offsets for aligned access, the low bits are not present (which is why the "<< 2" calculation is used before adding them to a word address); immediate operand instructions are written with a "`.i`" suffix

instruction	**effect**
`loadw` R_d`,[`R_a`+`R_b`]`	$R_d \leftarrow mem[R_a+R_b]$
`loadw.i` R_d`,[`R_a`+`*offset*`]`	$R_d \leftarrow mem[R_b+offset << 2]$
`storew [`R_{d_1}`+`R_{d_2}`],`R_s	$mem[R_{d_1}+R_{d_2}] \leftarrow R_s$
`storew.i [`R_d`+`*offset*`<< 2],`R_s	$mem[R_d+offset << 2] \leftarrow R_s$
`addw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a + R_b$
`addw.i` R_d`,`R_a`,`*value*	$R_d \leftarrow R_a +$ *value*
`subw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a - R_b$
`subw.i` R_d`,`R_a`,`*value*	$R_d \leftarrow R_a -$ *value*
`multw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a \times R_b$
`multw.i` R_d`,`R_a`,`*value*	$R_d \leftarrow R_a \times$ *value*
`divw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a \div R_b$
`divw.i` R_d`,`R_a`,`*value*	$R_d \leftarrow R_a \div$ *value*
`andw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a \wedge R_b$
`orw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a \vee R_b$
`xorw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a \oplus R_b$
`lshiftw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a << R_b$
`rshiftw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a >> R_b$
`cmpeqw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a = R_b$
`cmpnew` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a \neq R_b$
`cmpltw` R_d`,`R_a`,`R_b	$R_d \leftarrow R_a < R_b$
`breqw.i` R_a`,`R_b`,`*offset*	$R_a = R_b$? $PC \leftarrow PC+$ *offset* $<< 2$
`brnew.i` R_a`,`R_b`,`*offset*	$R_a \neq R_b$? $PC \leftarrow PC+$ *offset* $<< 2$
`j` R_a`,`R_b	$PC \leftarrow R_a + R_b$
`j.i` *address*	$PC \leftarrow$ `address` $<< 2$
`save` R_d`,`R_a	$R_d \leftarrow PC + R_a$
`save.i` R_d`,`*offset*	$R_d \leftarrow PC+$ *offset* $<< 2$

Table 3.1: Simple instruction set for examples. *Both `offset` and `value` are signed 16-bit values. Instructions ending in "`w`" operate on a 32-bit integer word, and a "`.i`" suffix implies an immediate operand. We don't need both $<$ and $>$ because we can reverse the operands. You can obtain a logical negation by using* `xorw` R_d`,`R_a`,`R_0. *You can check for negative values by* `cmpltw` R_d`,`R_a`,`R_0. *To keep the notation consistent with an assignment, the destination operand is always written first.*

- *register* operands are 32 bits wide and can potentially generate unaligned accesses, which are trapped by hardware since these are errors for this architecture
- *branch* instructions generally are relative to the current program counter (PC); in assembly language for convenience we use symbolic labels to indicate the branch target but, in machine code, the target is a signed offset
- *jump* instructions are unconditional and usually allow longer addresses than the short offsets allowed in branches; the `j.i` instruction uses all the bits not required for an opcode as a word-aligned address
- you can get the effect of a jump instruction to an offset by using a condition

on a branch that's always true

- a *return address* is usually stored using a variant on a branch or jump; in our simple instruction set, we instead have a `save` or `save.i` instruction that allows us to store a location some offset from the PC

I only include word-length instructions with signed operations in the Table 3.1; an example of another variation, an unsigned add of 1 half-word ("s" for "short") is:

```
addsu R6,R5,R4
```

This is a very simple instruction set; simpler in some ways even than the MIPS instruction set, one of the more regular RISC examples[1].

Let's look at a simple code snippet, translated to assembly language in our notation, and see how it proceeds through the pipeline:

```
for (int i = 0; i < N; i++) {
    a[i] += b[i] - 42;
}
```

To translate to our machine instruction is reasonably straightforward. We need to note a few things:

- word size is 4 bytes so we need to go up in steps of 4 to iterate through an array
- the variable `i` is local to the loop and only used in array references, so we can replace it by an offset incrementing in steps of 4
- we need to test the stop condition before the first iteration to be consistent with the definition of a C-style `for` loop

In assembly language it looks something like this, with the original code interleaved as comments:

```
# assume the value of N is in R1, the base address of a is in R2,
# the base address of b is in R3
      multw.i R7,R1,4 # loop end point (N scaled by 4)
#     for (int i = 0; i < N; i++) {
      addw.i R4,R0,0 # initial array offset (i scaled by 4)
test: cmplt R8,R4,R7 # < end of loop test?
      breqw.i R8,R0,end # < test false? get out
```

[1] See `http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html` for some details of the MIPS instruction set.

	clock number									
instruction	1	2	3	4	5	6	7	8	9	10
`loadw R6,R2,R4`	F	D	X	M	W					
`loadw R5,R3,R4`		F	D	X	M	W				
`addw R6,R6,R5`			F	D	X	M	W			
`subw.i R6,R6,42`				F	D	X	M	W		
`storew R2,R4,R6`					F	D	X	M	W	
`addw.i R4,R4,4`						F	D	X	M	W

Figure 3.3: Our code without pipeline bubbles. *I mark registers modified in previous steps in* `red`*. We now have to work out where stalls should occur. For brevity I shorten the stage names to 1 letter.*

```
#           a[i] += b[i] - 42;
      loadw R6,R2,R4
      loadw R5,R3,R4
      addw R6,R6,R5
      subw.i R6,R6,42
      storew R2,R4,R6
      addw.i R4,R4,4 # advance by 4 because word = 4 bytes
      breqw.i R0,R0,test
#     }
end:  # next instruction after loop
```

This code could be more efficient (e.g., branching to the test is less efficient than doing the test at the end, and our last branch is in effect unconditional and a jump would be more efficient – since it's a short distance away, `j.i test` would work) but it serves to illustrate progress of code through a pipeline, and gives us a simple example to explore control hazards. To start with, we will only look at the body of the loop without conditional code, to see how data hazards arise. The body of the loop on its own is as follows:

```
      loadw R6,R2,R4
      loadw R5,R3,R4
      addw R6,R6,R5
      subw.i R6,R6,42
      storew R2,R4,R6
      addw.i R4,R4,4 # advance by 4 because word = 4 bytes
```

To see what dependences there are, let's write out a timing diagram then refer back to our definition of timing in the pipeline. In Figure 3.3, I list the instructions without bubbles in the pipeline but instructions that depend on previous instructions highlighted. The first `addw` instruction depends on the previous two loads, but only

	clock number																		
instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`loadw R6,R2,R4`	F	D	X	M	W														
`loadw R5,R3,R4`		F	D	X	M	W													
`addw R6,R6,R5`			F	–	–	–	D	X	M	W									
`subw.i R6,R6,42`							F	–	–	–	D	X	M	W					
`storew R2,R4,R6`											F	–	–	–	D	X	M	W	
`addw.i R4,R4,4`															F	D	X	M	W

Figure 3.4: Our code with stalls (marked as "–") causing pipeline bubbles.

the delay caused by the second load matters, since any delay added there will be at least 1 cycle longer than needed for the first delay. In our simple pipeline, a load and ALU result is available in the target register at the end of the WB cycle and an ALU operation needs a register value in the ID stage. That means we must stall the pipeline for three cycles in each case where an ALU or store operation follows another instruction that changes a register it needs.

The result as illustrated in Figure 3.4 is an increase from 10 to 19 cycles to complete the sequence of code.

That's a rather large slowdown[2]: $\frac{10}{19} = 0.53$. Can we do better? Waiting for the end of a cycle when a result is written to in a register is not really necessary if we can write a register in the first half of a cycle and read it in the second half. Also, we can go a step further add hardware resources to determine that a value is needed, we can *bypass* the register file, an approach also called *forwarding*. By making the first improvement, we can reduce each stall by 1 cycle. If we introduce forwarding hardware, we can use each result as soon as it's ready rather than routing it via the register file. In the case of an ALU operation, it is ready the cycle after EX. In the case of a load, it is ready after the MEM stage. Also, we can route the result

[2]Technically, this is a "speedup" though the word looks wrong applied to a case where we've lost speed.

	clock number															
instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`loadw R6,R2,R4`	F	D	X	M	W											
`loadw R5,R3,R4`		F	D	X	M	W										
`addw R6,R6,R5`			F	–	–	D	X	M	W							
`subw.i R6,R6,42`						F	–	–	D	X	M	W				
`storew R2,R4,R6`									F	–	–	D	X	M	W	
`addw.i R4,R4,4`												F	D	X	M	W
`loadw R6,R2,R4`	F	D	X	M	W											
`loadw R5,R3,R4`		F	D	X	M	W										
`addw R6,R6,R5`			F	–	D	X	M	W								
`subw.i R6,R6,42`					F	D	X	M	W							
`storew R2,R4,R6`						F	D	X	M	W						
`addw.i R4,R4,4`							F	D	X	M	W					

Figure 3.5: Approaches to reducing stalls. *The example above the horizontal line illustrates the effect of being allowed to read a register in the second half of the cycle when it's written. The version below the line illustrates the benefit of forwarding.*

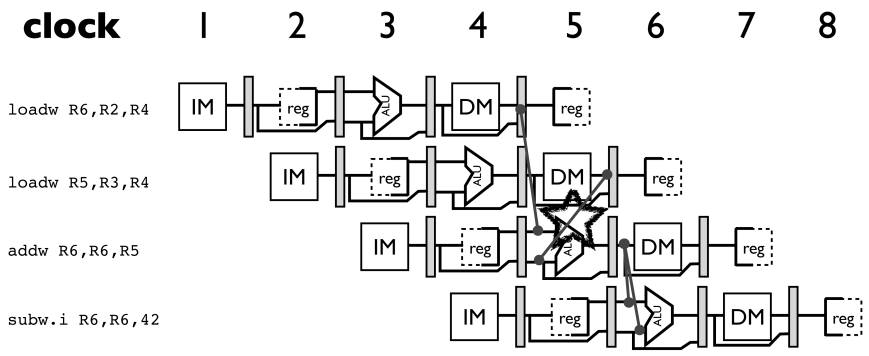


Figure 3.6: Limits of forwarding. *The grey lines show how values can be forwarded; the ragged star shows where forwarding would require sending a value back in time, since the load result is not ready in time for the following ALU operation.*

at the cycle it's needed rather than the cycle before, e.g., for an ALU operation, if the result is ready before EX, forwarding can make it available at the start of EX even if it's not available at the start of ID. A store instruction only needs its value at the start of MEM. I illustrate a minimal version of stall reduction in the top half of Figure 3.5, and a more aggressive version using forwarding in the lower half.

In this example, we are able to eliminate all but one stall by aggressive use of forwarding. The cost of forwarding is a more complex decode stage, which must determine if any needed registers are pending results and if so set up bypass logic, which can include receiving values from the ALU or from a memory read. It is this kind of detail that illustrates the benefit of the extremely regular design of a RISC architecture. Register operands are always encoded the same way, so relatively little effort is required to determine which registers need values in the decode stage. In Figure 3.6, I illustrate why all stalls can be eliminated except for the add immediately following a load.

What of the branches? We have only so far considered data hazards. There are two places where control hazards occur: the test at the top of the loop and the branch at the end. We consider only the second example. The first is useful as an exercise for later. In this case (Figure 3.7) it is not strictly necessary to stall since the ID phase doesn't do anything that can't be undone. However with aggressive forwarding there is a fair amount of logic that would be wastefully exercised, a consideration for low-energy design. In this case, the branch is mostly not *taken*, i.e., the branch condition is false. So eliminating the stall would be a win. Alternatively if a branch is mostly taken, starting to load the target instruction immediately that the branch target is known (in our architecture, at the end of ID) rather than wait for the outcome to be known, would be a win.

	clock number									
instruction	1	2	3	4	5	6	7	8	9	10
`multw.i R7,R1,4`	F	D	X	M	W					
`addw.i R4,R0,0`		F	D	X	M	W				
`cmplt R8,R4,R7`			F	D	X	M	W			
`breqw.i R8,R0,end`				F	D	X	M	W		
`loadw R6,R2,R4`					F	–	D	X	M	W

Figure 3.7: Branch-induced stalls. *After fetching the last instruction, we know the previous instruction is a branch, and stall until the outcome is known.*

Clearly, the loop control branch instruction will most often go the same way. We know here that the `breqw.i` instruction controls a loop, but that's because we have the source code. How do we know in general when a branch is less or more likely to be taken? Many recent designs have hardware *branch predictors*. We can see from this example that a branch predictor will not be a huge win. If we predict the branch as not taken, that eliminates 1 stall (the only stall in the example), provided the prediction is correct. If the prediction is incorrect, we lose the opportunity to load the target instruction as early as possible, and lose 1 cycle.

The simplest approach to branch prediction is *static prediction*, based on the observation that loops repeat by branching backwards. If you predict all forward branches as not taken and all backward branches as taken, you capture a large fraction of easily-predicted branch behaviour [Piguet 2006]. Our example may not be typical of machine code: would this branch predictor work?

A simple approach to *dynamic* branch prediction is to use 1 bit to record whether a branch is taken or not. In a case like a loop, 1 bit of prediction is potentially useful; in a case where prediction depends on the outcome of other branches a more complex strategy may be better. A simple way of storing state is in a *branch history table*, indexed by low-order bits of the instruction address. The more address bits used, the less chance two branches' predictions are confused with each other. A table of 4Ki entries suffices for smaller programs; current architectures may use bigger tables and more sophisticated schemes. There was a lot of research into branch prediction in the 1990s, when aggressive ILP was a major design goal [Yeh and Patt 1992, 1993; Kaeli and Emma 1997; Young and Smith 1999; Skadron et al. 1999]. If a branch is taken, the bit is set to 1, otherwise 0, and whatever was previously set is used to predict the branch outcome. A 1-bit scheme has the drawback that, since it changes every time the direction of the branch changes, if a branch mostly goes the same way, it mispredicts not only on the rare occasion when it goes the other way, but the next time when the direction reverts to the usual way (taken or not). A simple solution is to use 2 bits, in a scheme that requires the branch go twice in a different direction before the

bits	prediction	event	new bits
00	not taken	not taken	00
00		taken	01
01		not taken	00
01		taken	11
10	taken	not taken	00
10		taken	11
11		not taken	10
11		taken	11

Figure 3.8: Two-bit branch predictor state transitions. *Each predictor is stored in a table indexed by low address bits of branch instructions.*

prediction changes.

Figure 3.8 illustrates state transitions of a 2-bit predictor. Each state is identified by two bits. If both bits are zeroes, that represents a prediction that the branch is not taken, which requires two successive instances of the branch being taken to flip the prediction. Both bits being ones means it takes two successive instances of the branch to not be taken to flip the prediction. The other two states each require only 1 disagreeing branch direction to change the next prediction, and can be pushed back to the "00" or "11" state in a single step. The scheme used here is a *2-bit saturating up and down counter*. It is "saturating" because it stops when it hits an end point, and "up and down" because it has two end points, one for counting up, the other for counting down. A branch predictor, like many simple hardware constructs, can be described as a *finite state machine* (*FSM*), which I represent here as a table. You can also represent an FSM as a diagram with one node per state, and arrows labeled with events indicating state transitions.

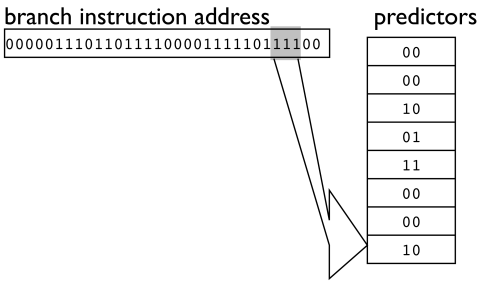


Figure 3.9: Finding a branch prediction. *Low bits of a branch instruction address are used to index a single global pattern table. In this toy example (table size only 8), 3 bits are needed. Instructions are word-aligned, so the low 2 bits of the instruction address are always 0 and not used in the index.*

I illustrate how a branch instruction's address is used to look up a prediction in Figure 3.9. In this case the branch table only has 8 entries, not big enough to be useful, but we can see the whole example in one picture.

Branch prediction becomes a much more significant issue with superscalar pipelines, where deciding early to flush the pipeline and go on a new path makes a big difference, if you mostly choose the right option.

When we consider more exotic pipelines the benefits of branch prediction and other approaches to reduce branch latency become clearer.

3.2 More Exotic Pipelines

Three variations on pipelines add complication (assuming we are starting with a simple, regular instruction set: the IA32 is pretty complicated even in a simple implementation, for example):

- *deeper pipelines* – hazards have a higher cost the deeper the pipeline because there are more instructions in flight
- *multi-cycle instructions* – even in a simple RISC architecture, floating-point instructions cannot all be implemented in one execute cycle (particularly divide)
- *multiple instructions per clock* – a superscalar pipeline multiplies the opportunities for hazards

Aside from the standard kinds of hazards, handling interrupts becomes more complex the more complicated the pipeline. Ideally, you want your architecture to maintain *precise exceptions*: any instruction that entered the pipeline before that causing the exception should finish, and any instruction that enters the pipeline later should not finish, and should not have any effect on the machine state.

There is considerable complexity in handling floating point because some instructions take multiple cycles and hence make it hard to maintain precise exceptions (e.g. a divide overflow exception may take a few cycles to become apparent, implying that other logically later instructions that subsequently completed should be rolled back). Long instruction completions also make it possible even with our simple pipeline model to have WAR hazards.

Timing of deeper pipelines depends exactly how the stages are split.

Here I only consider multiple instructions per clock. Since this technique is in competition with multiple cores, it is useful to understand the basic concepts and how far they can go. I also examine tactics that can reduce dependences or pull them further apart. These techniques include instruction reordering, register

renaming and loop unrolling. You can reduce stalls either by *static* or *dynamic scheduling*:

- *static scheduling* – the compiler (or a fanatical human who goes down to the machine code layer) can optimize ordering of instructions for a given pipeline
- *dynamic scheduling* (also *dynamic dispatch*) – the hardware determines the order of instructions at run time

A pipeline that can start more than one instruction per clock (more correctly, have more than one instruction start the EX stage per clock, sometimes called *issue*) is called *superscalar*. In the simplest scheme, the next k instructions are fetched and if there are no dependences between them limiting parallel execution, all are dispatched or issued simultaneously. A limitation of this scheme is that it's not necessarily a given that adjacent instructions have no dependences but other instructions further apart may be free to go. Another limitation of a simple scheme is that branches limit simple ILP. I a typical MIPS integer workload, between 15 and 25% of instructions (counted dynamically, i.e., as fraction of instructions executed) are branches, meaning you can typically expect 3–6 instructions between branches [Hennessy and Patterson 2012, p 149]. While floating point code often has longer sequences of instructions between branches, working around branches is a key aspect of achieving significant ILP.

In some schemes, dispatch and issue are treated as separate steps[3]:

- *dispatch* – queue the instruction for execution
- *issue* – allocate a functional unit to the instruction and start its execute step

it is useful to split dispatch and issue in out of order machines; in machines that start instructions strictly in order, there is no need to separate out these steps.

A superscalar pipeline requires duplicated resources for any pair of operations that could occur in parallel. Typically, the ALU is divided into *functional units*, a major grouping of related instructions, such as integer or floating point, and the number of each type of functional unit limits the number of that type of instruction that can simultaneously be dispatched.

Before we go on, we need a little more terminology. We already know about data, control and structural hazards. Another type is a *name* hazard, a situation where instructions share the same data *resource*, usually a register, but do not actually interchange data. Name dependences usually arise because a machine

[3]Mark Smotherman has a nice summary of the terminology here: `http://www.cs.clemson.edu/~mark/464/dynsched.txt`

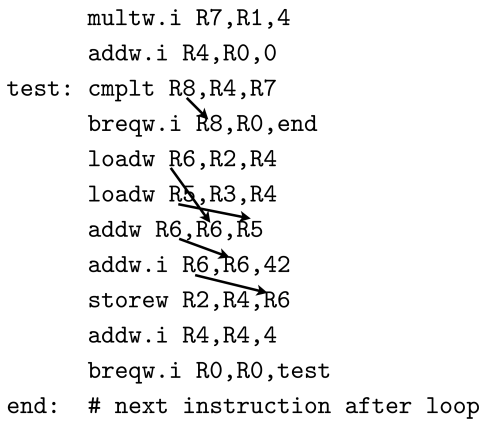


Figure 3.10: Dependences in one iteration of the loop. *To reduce clutter I do not depict dependences between the initialization code and loop body. Clearly all instances of* `R4` *in the loop body depend on the initialization step in the second instruction, and the loop test depends on the value of* `R7`*.*

does not have a limitless register set, so registers have to be recycled. Another example is the call stack, which is recycled between calls, and limits any hardware attempt to convert function or method calls into threads [Postiff et al. 1998].

Static scheduling

Let us now return to our simple example, and see what happens if we attempt to execute two instructions per clock. To start with, I look at reordering instructions and other changes that could be done at compile time.

In Figure 3.10, I illustrate data dependences using an arrow from the place the data is updated to the place it's used. To avoid cluttering the picture, I leave out dependences between the loop initialization and the body; of more interest is what happens when we repeat the loop. A question we need to ask is if these are true dependences, or name dependences. In one iteration of the loop, they are true dependences, limiting ILP. In a two-instruction per clock pipeline, we cannot dispatch two successive instructions if the second depends on the first. In the body of the loop, the only cases where pairs of instructions do not have a dependence are the two loads and the last three instructions. Provided the store can access `R4` before the add modifies that register, those two instructions can proceed in parallel.

Using the same subset of the program as in Figure 3.5, let us see how much parallelism we can extract in a simple scheme that fetches two instructions at a time and if there is no dependence, dispatches both at once. If there is a dependence, the second waits until the dependence is cleared. Figure 3.11 illustrates the outcome.

If we compare the result against eliminating stalls using forwarding but in a

	clock number									
instruction	1	2	3	4	5	6	7	8	9	10
`loadw R6,R2,R4`	F	D	X	M	W					
`loadw R5,R3,R4`	F	D	X	M	W					
`addw R6,R6,R5`		F	–	D	X	M	W			
`subw.i R6,R6,42`		F	–	D	–	X	M	W		
`storew R2,R4,R6`						F	D	X	M	W
`addw.i R4,R4,4`						F	D	X	M	W

Figure 3.11: Simple two-instruction dispatch schedule. *If two instructions can execute on the same cycle they do, otherwise the second stalls. We fetch two instructions every cycle where there isn't a pending stall. Assume forwarding makes it possible to use a result at the end of the stage when it is created.*

scalar pipeline in Figure 3.5, we've reduced total cycles from 11 to 10, not a huge win for significantly greater hardware resources.

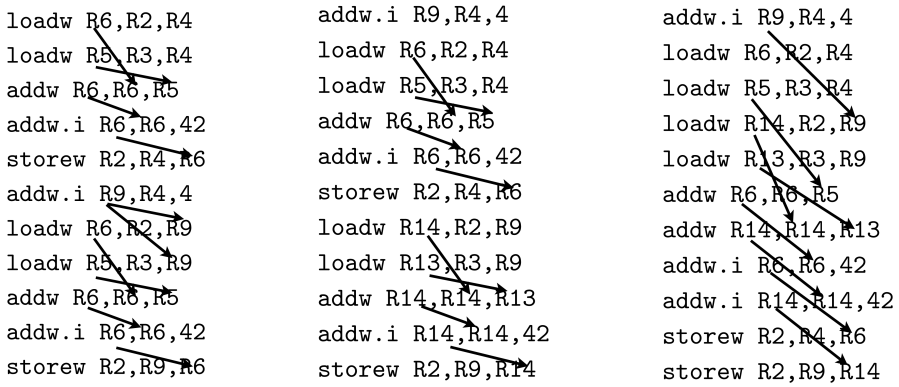
Can we do better? So far, we have fudged the issue of multiple iterations of the loop. If we return to the original C-style code:

```
for (int i = 0; i < N; i++) {
    a[i] += b[i] - 42;
}
```

a simple observation is that the calculation for each value of `i` is independent so this code has more natural parallelism than is at first apparent. There's no reason if our hardware isn't clever enough that we shouldn't be able to do as many loop bodies as we have hardware resources for in parallel. The reuse of the variable `i` is an example of a name dependence, which we can break by systematically renaming `i` each iteration of the loop. The only real dependence is that we need to conpute each new value of `i` based on the last one, but that's only one dependence rather than a long chain that imposes a strict ordering on our code.

So back to the machine code version: if we write out two iterations of the loop, leaving out the condition and branch, we have dependences between `R4`, the index variable, across iterations, but are these true data dependences? Not really, because we can replace `R4` by a different register. In Figure 3.12, I illustrate how two instances of the loop have minimal dependences between them – though the new register, `R9`, has many dependences to successor instructions (as does `R4` in the original code, had I shown them). I then go on to show that I can increase the gap between dependences by increasingly aggressive renaming.

What's the win here? The dependence between `R9` and successor instructions need not be as close to them as in Figure 3.12(a). We can move the initialization of `R9` to the top, and we can get further gains by interleaving the code for the two instances of the loop, with further renaming of registers. We now have the potential



(a) Rename R4. (b) More renaming. (c) Even more renaming.

Figure 3.12: Dependences in two instances of the loop. *To reduce clutter I do not show dependences more than 3 instructions apart. I rename* R4 *as* R9*, then show with more aggressive renaming dependences can be moved further apart.*

if we generalise to more than one instance of the loop to achieve a respectable level of ILP.

So far, I've assumed that we can have two instances of a loop. That is not in general true: if the loop executes an even number of times, we can do this, and adjust the stopping condition. What I've presented here is an example of *loop unrolling*. A compiler can generate code using the principles I illustrate here, but only for a loop where the stopping condition is a limit on a counter as in a typcial `for` loop. In that case, the compiler can generate two instances of the loop: one that runs for $N\%k$ times, the other for $N \div k$ times. In this case, where $k = 2$, the compiler would generate code equivalent to

```
int i = 0;
int k = 2;
// a N%2 == 1 or 0; could use an if statement
// but the following generalises to k > 2
for (int j = 0; j < N%k; j++) {
    a[i] += b[i] - 42;
    i++;
}
for (int j = 0; j < N/k; j++) {
    a[i] += b[i] - 42;
    a[i+1] += b[i+1] - 42;
    i+=2;
}
```

We can potentially improve our unrolled code even further by using two registers for the different instances of the loop index from the start, and incrementing each separately. However we have enough detail at this point to see how unrolling works in general, and how if can be extended to multiple instances of the loop body. What we do not have is a way to do loop unrolling when the stopping condition is more complicated, i.e., we don't know even at run time (by the first iteration of the loop).

Dynamic scheduling and better branch prediction

There are two big downsides to static scheduling: an ideal schedule for one pipeline may not be ideal for another, and some limits on parallelism may only be possible to resolve at run time. The Control Data CDC 6600 was the first machine to tackle the concepts of out of order execution. It had 10 functional units [Thornton 1963], and had a hardware structure called a *scoreboard* that kept track of dependences and identified which instructions could issue [Thornton 1980]. Recompiling code may be an option for software created in-house or on frequent release cycles, but maintaining versions of code for multiple pipelines is impractical for most software in common use.

Once it became possible to add the equivalent in logic to another functional unit to implement a scoreboard-like feature, commodity processors switched to out of order execution.

Another part of the picture, for a change not originally designed by Seymour Cray, is hardware loop unrolling. Robert Tomasulo, an IBM engineer, developed a hardware algorithm [Tomasulo 1967] that bears his name that was the first example of register renaming. The keys to the algorithm is *reservation stations* that hold an instruction until all its operands are available, and internal register renaming. In an example like our unrolled loop, it would not be necessary to find new registers for the second (or subsequent) instances of the loop; the hardware would allocate virtual registers to the successive instances of the loop.

The important thing about both a scoreboard and Tomasulo's algorithm is that they make it possible to issue out of order, even though aggressively superscalar architectures were not feasible in the 1960s. A scoreboard makes it possible to issue instructions when data dependences are met; Tomasulo goes one step further and makes it possible to eliminate name dependences (at least between registers: name dependences as relate to memory addresses are another whole problem).

The major thing that these innovations add is that the sort of scheduling exercise illustrated in Figure 3.12 can work as well as the hardware available: provided there is a sufficiently large hardware instruction window, dependences can be limited to real dependences, and as many functional units as are available can be kept busy, up to the limit imposed by true dependences. That leaves us with

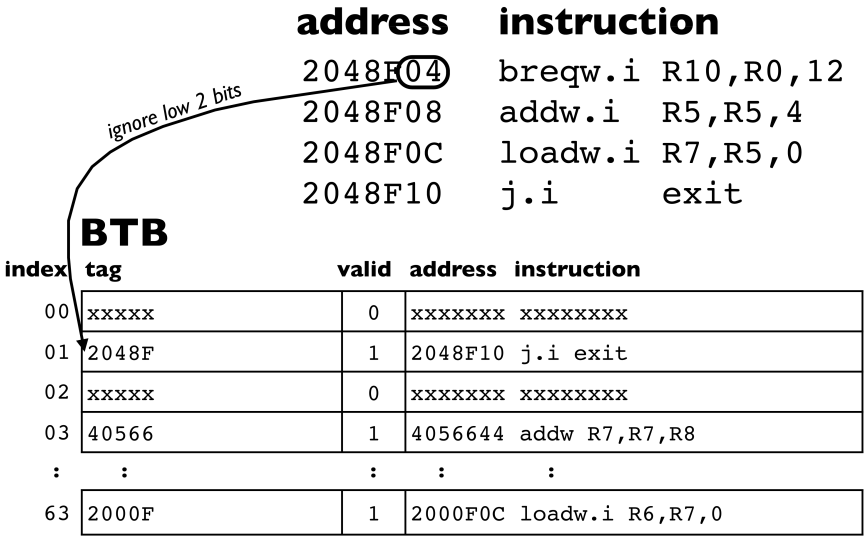


Figure 3.13: Possible branch table buffer organization. *The instruction would of course be represented in binary rather than a human-readable form. Some BTBs only represent the target address, and others also include the branch prediction. Note that in a word-aligned machine with 32-bit instructions, it isn't necessary to store the low 2 bits of the instruction or use them in the index. In this example, the index uses 6 bits, leading to a 64-entry table.*

one major cause of stalls we need to reduce: branches.

So far the best we have is a 2-bit branch predictor, that can capture up of the order of 93% of branch behaviour. The remaining 7% is significant if the penalties are high. If for example we have a very aggressive design capable of issuing up to 8 instructions per clock and we mispredict a branch, we not only have to flush up to 7 instructions from the pipeline, but we could have started 7 instructions in the correct path of the branch.

Before going to a more sophisticated branch prediction strategy, I introduce one more improvement in branching: a *branch target buffer* (or *BTB*). The win from a BTB, which stores the target instruction of the branch, is that as soon as the branch is resolved as taken, the target instruction can be inserted into the pipeline. In some schemes, a BTB may include branch prediction [Perleberg and Smith 1993], but I prefer to keep the terms separate. A BTB may store varying degrees of information from a prediction of the branch target address through to the target address and the actual instruction. Unlike a branch predictor, a BTB needs an accurate representation of the target instruction since it would be useless to start executing a completely wrong instruction, so a BTB typically includes a tag that allows the address of the *source branch* to be reconstructed. In other

words, similarly to a cache, a BTB is indexed by part of the address of the branch being predicted, and the rest of the branch address is used in a tag to check that the right target has been found. This scheme can obviously only work if branch instructions cannot vary the target address, i.e., it's always based on an immediate operand, not a register. Figure 3.13 illustrates a possible BTB organization. If the top branch instruction in the illustrated snippet of code is about to be executed, the BTB logic looks up the target instruction. If the branch is predicted as taken, it can be fetched immediately. In an aggressive scheme, the BTB can be looked up before the instruction is decoded, since the tag ensures that a lookup will miss if the current instruction turns out not to be a branch.

A BTB, much like a cache, can experience misses. In the event that the branch is predicted as taken, a miss requires waiting for the target instruction but the BTB is updated for next time. If the branch is predicted as not taken, the BTB can be ignored.

On now to more sophisticated branch schemes. There are many of these [Yeh and Patt 1993; Skadron et al. 1999; Tyson 1994; Kaeli and Emma 1997] and there was considerable research on these in the 1990s at the height of ILP research, and I only consider one in depth, and adaptive two-level predictor [Yeh and Patt 1991]. With the shift to multicore designs, branch prediction is unlikely to need more sophisticated schemes in the near future than the schemes of the 1990s. In this scheme, there is a predictor for each branch. Each predictor maintains a k bits of history of a branch. These k bits are used as an index into a pattern table that predicts what the next branch should do. The entry for each pattern is a saturating counter, much as for our single-level 2-bit predictor. The main difference is that a previous pattern of branches like *taken, taken, taken, not taken, taken*, of it's the same as the current pattern (for $k = 5$ in this example) selects the prediction, rather than the address of the branch. We still use the branch address as an index into the branch history, but use a global pattern table. Figure 3.14 illustrates the basic idea of the scheme. This scheme with a 512-entry 4-way associative history table was shown to have 97% accuracy in predicting branches in the original Yeh and Patt [1991] study.

A final wrinkle on branch prediction is *speculative execution*. If a branch outcome cannot be determined in time to keep the pipeline busy, in a speculative machine, instructions that may have to be discarded are executed, with results in shadow registers, that are copied to the real registers when the instructions are committed. If the branch prediction is incorrect, the speculated instructions are discarded [Lee et al. 1995; Hiraki et al. 1998; Krishnan and Torrellas 1999]. Speculative execution can include speculative loads [Rogers and Li 1992] and even to threads [Martínez and Torrellas 2002; Ceze et al. 2006]. Any memory access that's speculative should not cause a page fault, as that's a huge overhead

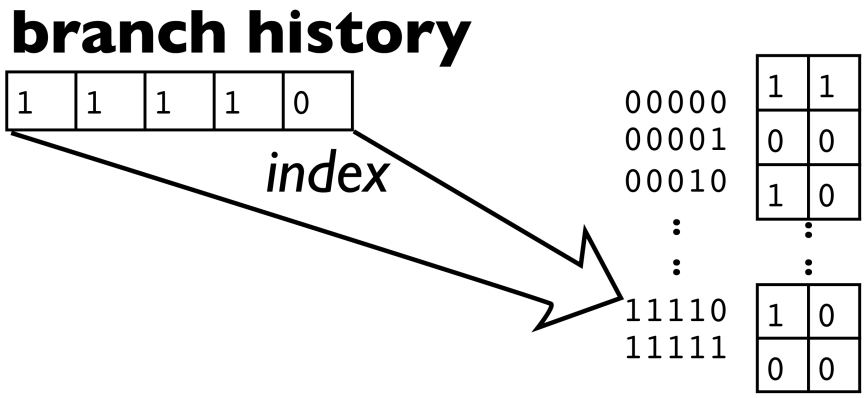


Figure 3.14: Two-level predictive branch. *After each branch, the pattern history for that branch is shifted left and the latest actual outcome becomes the low-order bit. The individual branch histories may be stored in various ways including a structure indexed and tagged like a cache.*

compared with a pipeline stall, so implementation of speculation is very complex, yet it made it to commodity designs like the Pentium 4. The Pentium 4 could issue 3 instructions per clock but could have up to 60 pending issue at any one time [Sohi 2001].

With all of this out of the way it now becomes possible to explore a reasonable level of ILP in a superscalar architecture. Paper exercises similar to that of 3.12 are instructive, though real design studies showing the effects of cache misses, TLB misses and unavoidable stalls for dependences show that in practice, it is hard to achieve much more than two instructions per clock on average.

Compiler-Exposed ILP

One more idea is to have the compiler expose ILP. A pioneering approach to this is packing multiple instructions in one long machine word. This idea, called *very long insrtuction word* (*VLIW*) was used in the Multiflow machines of the late 1980s. The intial design had 256-bit instructions containing 7 operations, followed by a more ambitious design with double the instruction width and 14 operations per instruction word. The idea was that a compiler technique called trace scheduling would expose enough ILP to fill a high fraction of the operations with useful work (otherwise a null operation or `NOOP` had to be inserted) [Colwell et al. 1990].

The Intel IA64 was designed based on similar principles. Despite considerable money being thrown at the project not only by Intel but partners like HP, performance was disappointing. One of the lead architects on the Multiflow project, Bob Colwell, on joining Intel, led the design of the Pentium Pro and

successors [Colwell and Steck 1995], possibly an indication that he'd learnt his lesson at Multiflow. That lesson is that if compiler technology cannot deliver for an exotic architecture that very smart people are working on, the chances are it will not deliver for anyone else.

The supposed gain of VLIW is to remove the hardware complexity of dynamic scheduling by having a smart compiler that can expose ILP. So why did VLIW fail? You could argue that Multiflow failed because it was a startup, and that what startups do: so why couldn't Intel and HP get it right? Mainly because ILP is not only dependent on statically-determined dependences. Memory delays are also a factor, and in the area where the IA64 was competing, cache misses are a significant factor. If any instruction in your long word has to stall for any reason, all the rest must stall, unless you go back to where you started, hardware to support dynamic scheduling. Multiflow avoided that particular problem by not using caches, not a practical approach for a general-purpose architecture. The IA64 included a few other innovations like bits that the compiler could set as hints to the hardware on available parallelism, and *predicated instructions* [Tyson 1994]. A predicated instruction is tagged with a condition that must be true otherwise the instruction is not executed. Predicated instructions are intended to avoid the overheads of branches in short sequences of conditional code.

3.3 Summary

Increasing ILP was the key focus of computer architecture in the 1990s. Much of what I describe here was implemented in increasingly aggressive forms. By 2000, it was starting to become apparent that aggressive ILP was hitting limits, and any new attempts at increasing ILP would have limited gains and significant costs in energy and heat, and clocks become increasingly hard to scale as the total wiring on the chip increases [Agarwal et al. 2000]. IBM's Power5 CPU pretty much did everything: it had 8 execution units, with a peak issue rate of 8 instructions per cycle (one per unit), hardware multithreading and speculative out of order execution and two cores [Kalla et al. 2004]. Unlike its predecessor the Power4 [Tendler et al. 2002], Power5 did not lead to mass-market designs, as IBM and Motorola lost the Apple account, the one major market for PC-scale CPUs outside the Intel camp. At that point, it seemed that RISC had lost to CISC, though a better explanation is that aggressive ILP had peaked.

Exercises

1. Rework the pipeline timing diagram of our simple loop example under the following assumptions:

   (a) We can issue any pair of instructions, subject only to dependences; assume the most aggressive achievable model of forwarding.

   (b) Add now the ability to do register renaming; assume the hardware is smart enough to rename any register after a write to it and no limit to the number of virtual registers (do you run out of paper?).

2. Now go back to the simpler dual-issue pipeline without register renaming and evaluate the effect of a simple 2-bit branch predictor.

   (a) Will a more sophisticated scheme like a 2-level adaptive scheme make a difference here? Explain.

   (b) Now assume we have an `if` statement in the loop that only does the assignment on odd values of `i`. Write out the assembly language for this case (you may fudge some details as long as the branches are plausible). Will a more sophisticated branch predictor help in this case? Explain.

3. Assume we have a floating-point pipeline in which a multiply takes 2 cycles and a divide 4 cycles (both in the execution stage; other stages are the same as for integer instructions). Explain how these instructions introduce new types of hazard not present in the integer pipeline and why they present problems for interrupts.

4. Look up how precise exceptions or precise interrupts are handled in multiple-issue implementations.

5. VLIW was based on the premise that a compiler technique, *trace scheduling*, could expose significant ILP. Look up trace scheduling and analyse its strengths and weaknesses.

4 Multiprocessors

MULTIPROCESSOR SYSTEMS ARE NOT A NEW CONCEPT – what is comparatively new is multicore designs. Multicore systems are not fundamentally different from older multiprocessor systems. They have two major advantages: lower cost, and lower latency interprocessor communication. Otherwise they present many of the same performance and software challenges.

In the days of big iron multiprocessor systems, many models of parallelism were explored, and the winner was shared-memory multiprocessors. I review here are few of the other variations, then focus on shared-memory systems and relate the general field to current multicore designs. I save other models of parallelism currently in use, vector instruction sets and GPUs, for the next chapter, since they are significantly different in implementation and efficiency issues, and only briefly review them here..

4.1 Multiprocessor Models

Models of multiprocessor classically have been defined by whether they have more than one instruction stream, more than one data stream, or both:

- *SISD* – single instruction single data stream: a uniprocessor
- *SIMD* – single instruction multiple data stream: vector architectures for example, but there are other types
- *MIMD* – multiple instruction multiple data stream: more general types of multiprocessor, which run multiple threads or processes each relatively independent of each other

It's not clear that MISD – multiple instruction single data stream – makes sense. Another classification that cuts across these to some extent is memory organization:

- *shared memory* – all processes can access a single global memory

- *distributed memory* – processes have local memories that cannot be directly accessed; there are two models for distributed memory programming:
  - *message passing* – all communication is by messages similar to those you'd send over a network
  - *distributed shared memory* – the effect of a single global memory is faked using software, often using a combination of the virtual memory system and networking

The shared-memory model MIMD proved to be most popular because it most easily adapts to a variety of workloads, including multitasking a large number of single-threaded processes. It's possible to program in a message-passing style on a shared-memory machine, while distributed shared memory needs operating systems support for efficient implementation. In that sense a shared-memory machine is more general than a distributed-memory machine. MPI, now in common use as OpenMPI [Gabriel et al. 2004][1], is a message-passing API that can work efficiently on a variety of architectures, including networked systems and shared-memory systems.

Examples of vector additions to standard instruction sets include

- MMX [Peleg et al. 1997], SSE (Streaming SIMD Extension) extensions to the IA32 instruction set, and successors (SSE1, 2, etc.) and AVX [Firasta et al. 2008]
- AltiVec extensions to the PowerPC [Diefendorff et al. 2000]

In one of the more extreme examples that has made it to a commodity product, the Cell processor designed by IBM, Toshiba and Sony has 8 vector units, each with a local memory. The Cell seems to be an attempt at recreating all the hardware design errors of the past. Vector instruction sets only work well on specialised workloads, local memories put a lot of load on the programmer to get the right data in the right place at the rights time and combining vector units with another model of parallelism (multiple cores) is an untried programming model. The Cell was designed with two purposes in mind: developing HDTV codecs, and the Playstation 3. For the former, it was likely a success because computation is highly regular. Despite exaggerated expectations [Macedonia 2004], a handful of games developed specifically for the Playstation 3 was available at launch, and it was notoriously difficult to program.

SIMD systems include vector architectures, that take two forms: applying the same operations to multiple memory locations, and applying the same operation

[1] See also `http://www.open-mpi.org`.

to multiple registers grouped together as a vector register. Traditional large-scale supercomputers such as those made by Cray were vector machines, and had refinements like applying the same operation to sequential addresses, or locations with a fixed distance apart. Vector registers are common in GPU and similar instruction sets, such as the vector extensions of the Intel and PowerPC instruction sets. Vector instruction sets save a lot of time in avoiding the need to process instructions through a pipeline, and take advantage of high bandwidth of sequential or other regular memory access patterns – or the speed of registers. However, they rely on problems that are well suited to highly regular computation on sequences of data.

In the past, there was another class of SIMD machines that were described as "massively parallel", exemplified by the Thinking Machines CM-1 and CM-2 ("CM" for "connection machine") that had up to 64Ki relatively simple 1-bit processors, that could work simultaneously on the same instruction on different data; the effect was of 2048 parallel floating point processors. These machines seldom came close to their peak throughput, and were notoriously hard to program. The nodes were arranged in a *hypercube* [Womble et al. 1999], a structured designed to minimise distances between nodes while also minimising the total number of interconnections.

Since GPUs have taken on a new life as an alternative to conventional performance-oriented architectures, I consider them separately. SIMD and vector architectures feed into the design of GPUs, so I add a little more detail as applies to GPUs in the next chapter.

4.2 Shared Memory Principles

Shared-memory systems have significant performance advantages over distributed memory systems up to the point where they run into scalability issues (though you can argue that distributed memory systems only appear more scalable because they are unsuited to problems with a large amount of interprocess communication, IPC). Nonetheless shared memory can cause significant performance penalties if not well understood. Those issues start with performance problems generic to memory hierarchies, and extend to those specific to shared memory.

In what follows, I talk about a "CPU" as synonymous with a core, since there is no logical difference.

First, let's review some memory hierarchy basics. At the top level registers are specific to a CPU and not an issue for sharing. The TLB too tends to be specific to a CPU, and isn't specific to multiprocessing[2], though failure to understand the

[2]This is not strictly accurate since shared memory involves sharing a page table, but the performance

TLB can cause major performance problems. Once we get to caches, we start to run into significant performance problems. Even though the L1 cache may be local to a CPU, we need to take into account shared memory and ensure that the caches remain consistent. Despite all these innovations, the IA64 was a market failure, and the time when Intel was focused on that approach allowed AMD to dictate the design of 64-bit extensions to the IA32 architecture [Keltcher et al. 2003].

Maintaining *cache coherence* is one of the bigger problems of shared-memory multiprocessors. In addition to the usual cache tag scheme where we need a sufficient portion of the address to determine what memory locations a block represents, and status bits to indicate validity and whether the block is dirty, we also need to know if a block is shared. The simplest way to do this would be to add a shared bit. However, keeping track of whether the block is shared is a useful addition, because a non-shared block can immediately change to modified (or dirty) without waiting for any other caches to report back. One of the most common cache protocols is called MESI for having 4 states, modified, exclusive, shared and invalid. MESI is specifically well suited to a *write back* cache, i.e., one where blocks can be dirty. If a block is written *through*, i.e., all modifications immediately go to the next level down, a different protocol is needed. However, write-through caches are not in wide use, and have seldom been used in real systems [Archibald and Baer 1986]. Early designs with relatively slow CPUs used write-through caches (e.g. some early Sequent systems – a company with a brief period of success mainly in the database server market) but they do not scale to faster designs, as the number of writes saturates the bus.

- *multilevel inclusion* – bigger low-level components of the memory hierarchy include everything in the smaller higher levels (especially caches): this makes coherence a lot easier to manage as absence in a lower level automatically means absence in a higher level; caches without inclusion have the *subset* property

- *snooping* – each cache controller watches the shared bus for transactions that relate to its content; snooping doesn't scale to very large systems, and various *directory* schemes have been developed for very large shared-memory systems.

There are several variations on how cache coherence is implemented in practice. Using *snooping*, each CPU's cache controller watches for activity on a shared bus, and either intervenes in other caches, or modifies the state of its own if necessary. The MESI protocol is designed to reduce the need for snooping, because once a block is marked exclusive in a cache, the owner need not broadcast any actions on

issues of a TLB tend not to be significantly exacerbated but this effect.

that block. It must however react if any other cache broadcasts an action. Let's examine in detail how the MESI protocol works in a variety of scenarios. In each case, assume that a miss results in initiating a read from main memory, and this is aborted if another cache has a copy. If 1 other cache has a copy, it puts it on the bus for the requester; if it's shared, the owners race to put a copy on the bus.

- *read*
  - *hit* – no action
  - *miss* – state currently I
    - *no copy* – state $\rightarrow E$
    - *another cache S* – state $\rightarrow S$
    - *another cache E* – state $\rightarrow S$; snoop makes owner set its state $\rightarrow S$
    - *another cache EM* – state $\rightarrow S$; snoop makes owner set its state $\rightarrow S$; write back to main memory
- *write*
  - *hit*
    - *state EM* – no action
    - *state E* – state $\rightarrow EM$
    - *state S* – *invalidate* signal sent on bus; state $\rightarrow EM$
  - *miss*
    - *no copy* – state $\rightarrow EM$
    - *another cache E or S* – state $\rightarrow EM$; *invalidate* signal sent on bus
    - *another cache EM* – state $\rightarrow EM$; snoop makes owner set its state $\rightarrow I$; write back to main memory
- *replacement* – what we do to a block we evict from the cache (on a read or write)
  - *state EM* – write to main memory, continue as for miss
  - *state E or S* – no action, continue as for miss; if state is S and only 1 other cache holds the block, it will still hold it in state S

The protocol doesn't have a way of turning a block that's shared back to exclusive if all other processors lose theirs. To do so, we would have to broadcast on the bus every time a shared block was evicted, and keep a count of sharers. Note also that "main memory" really means the next level down, and in current multicore designs is usually a shared L2 or L3 cache. This version minimises copying from

main memory; more conservative designs copy to main memory whenever a copy is requested from another cache.

The Intel Nehalem architecture (launched with the Core i7, late 2008), with the major functions illustrated in Figure 4.1 and the die layout in Figure 4.2, is an example of a recent design with shared caches, in this case, L3. The version illustrated is from the first series with 8MiB of L3 cache; in more recent designs with 12MiB of L3 cache, caches would take up more than half the real eastate of the die. In the Nehalem design, the MESI protocol routes most requests via the L3

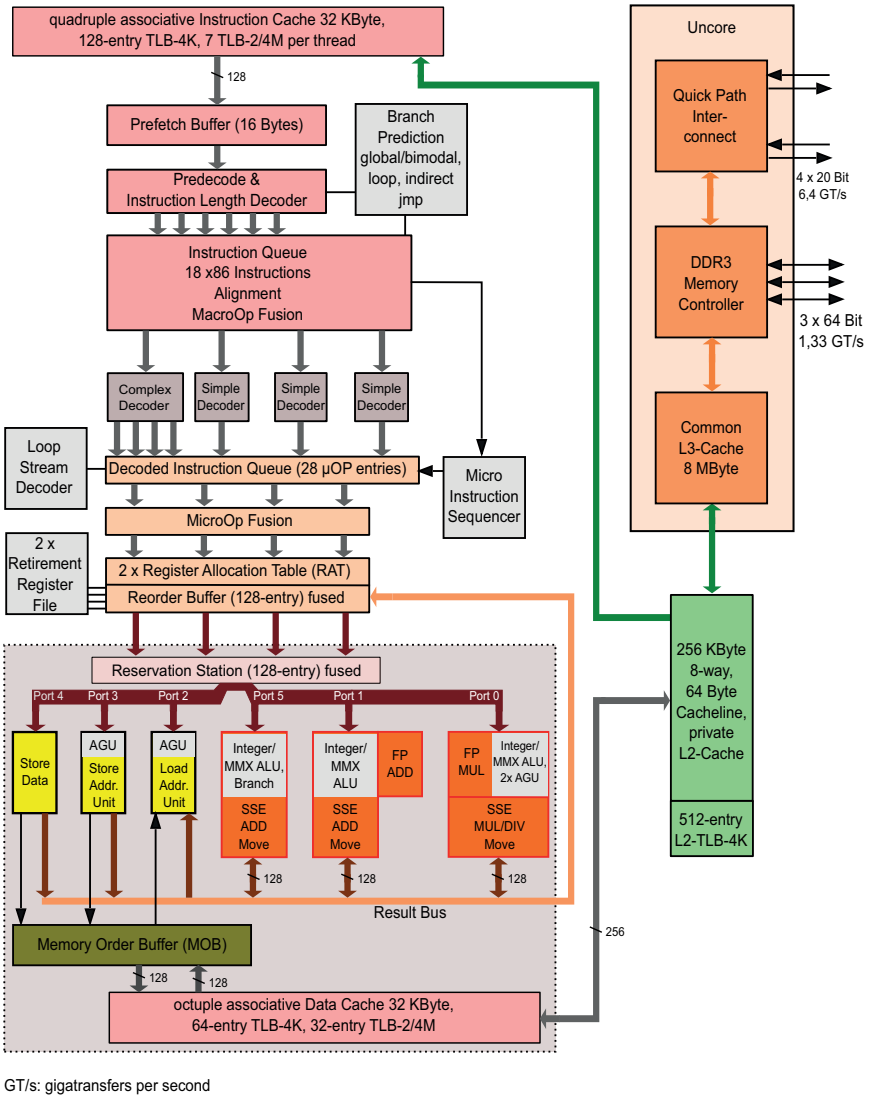


Figure 4.1: The Intel Nehalem archictecture. *Source: `http://en.wikipedia.org/wiki/Nehalem_(microarchitecture)`.*

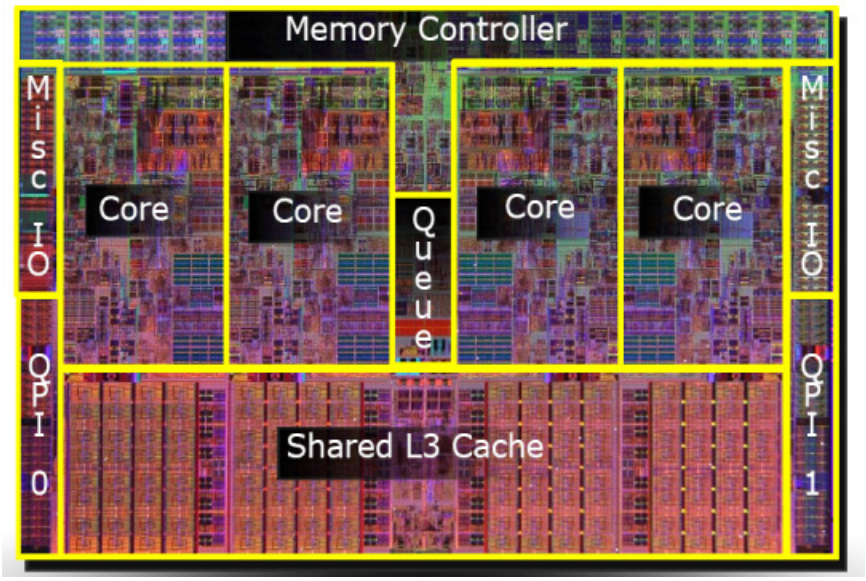


Figure 4.2: The Intel Nehalem die showing major components. *Source:* *http://arstechnica.com/uncategorized/2008/11/intels-3-2ghz-monster-nehalem-roars-onto-the-scene/.*

cache rather than having direct transactions between L2 caches, but is extended to something closer to the above, with an extra feature confusing called *forwarding*, making it a MESIF protocol, implementing the scheme I describe where caches forward a value to another that requests it rather than going via main memory (or in this case the L3 cache). The forwarding feature is limited to processors outside a single multicore unit. In local core-core cache transactions, the L3 cache acts as a central repository for transactions with tag bits indicating the state of blocks in the individual cores, reducing the need for snooping [Molka et al. 2009].

4.3 Shared Memory Performance

There are many performance factors in a shared-memory system. The less sharing there is, the fewer problems there are with scaling. Some problems are avoidable with careful programming, but any workload with a high rate of communication between components will not achieve a good speedup on any multiprocessor system.

Here are a few key factors in performance of shared-memory multiprocessors, that apply (almost) equally to multicore systems:

- *high rate of sharing* – as you should be able to see from the MESI protocol, modifying a variable in one CPU (or core) then reading it in another creates

significant bus traffic. Even if you don't need to wait for the write to the lower level to complete, you need to wait for the other cache to broadcast on the bus. That, in some systems, may not be a huge penalty compared with waiting for the lower level of memory. Still, if it happens a lot the bus can saturate.

- *false sharing* – if two or more variables that are actually not shared are in the same cache block, the coherence protocol doesn't know that: it only sees whole cache blocks; in this scenario a lot of unnecessary delay and bus traffic can result

- *contention for locks* – if a lock is implemented as a simple spinlock relying on the cache coherence scheme to ensure that updates are propagated, the amount of us traffic when a lock is released and set by one of several contending processes can be very high

These factors are in addition to the usual problems of scaling up a multiprocessor workload: load balance (ensuring the work is evenly split) and ensuring program correctness.

False Sharing

Let's use numbers from a real system, an Intel Nehalem design. I list some key numbers in Table 4.1. A few things need explanation: *snoop latency* is the extra time L3 must take before responding to a miss if a block is exclusive in a higher-level cache, since it must also check if the block has been modified. If a block is shared, L3 can immediately provide the block to the missing cache with the *shared access* latency. In this scheme, misses in L1 or L2 are handled out of L3, rather than the more aggressive scheme suggested in my definition of MESI. The reason for this is to relieve the high-level caches of the need to service requests from other caches. Since L3 is relatively fast compared with DRAM, this is a reasonable trade-off to avoid either the complication of another port on the L2 caches or forcing them to stall if they have a competing request from another core as well as servicing their own L1 misses. The Nehalem architecture includes other features we do not explore here including a fast interconnect for building multi-chip multiprocessor systems.

Given the Nehalem numbers, let's consider costs of cache misses and performance bugs such as false sharing. Suppose we have two sequences of code on two cores that each modify a separate variable that's in the same cache block. Let's take a short sequence of code in a loop as our example (using our simple RISC instruction set but with the Nehalem latencies):

level	latency	
	cycles	*ns*
L1	4	1.4
L2	10	3.4
L3	38	13
DRAM	191	65
multiprocessor overheads		
shared access	38	13
snoop latency	27	9.2

Table 4.1: Intel Nehalem latencies. *These are for a specific model, an Intel Xeon X5570 with core frerquency 2.933 GHz (cycle time hence 0.34ns), as determined by Molka et al. [2009]: at a different clock speed latencies will vary.*

#	core 0	core 1
1	`loadw.i R6,R2,0`	`loadw.i R6,R2,4`
2	`addw R6,R6,R5`	`addw R6,R6,R5`
3	`storew.i R6,0,R2`	`storew.i R6,4,R2`
4	`addw.i R4,R4,4`	`addw.i R4,R4,4`
5	`cmplt R8,R4,R7`	`cmplt R8,R4,R7`
6	`brnew.i R8,R0,-20`	`brnew.i R8,R0,-20`

These two examples pretty much do the same thing, except one has a variable at offset 0 from the address pointed to by R2, and the other a variable at offset 4 from the address in R2. The two cores' registers are independent, though I assume here that R2 has the same value in both cores. Unless we are extremely lucky and R2 is pointing at the last 4 bytes of a cache block, we will get false sharing here (assuming blocks are more than 4 bytes).

At the start of the loop, to keep things simple, neither core has a copy of the cache block and it is in the shared L3 cache, and that they are running in lock-step until one has a cache miss. At instruction number 1, both processors have a miss, and whichever acquires the bus first issues a miss to L3. We must add the latencies for L3, L2 and L1; once the block is copied from L3 to L2, the shared bus is released and the other core can access the block from L3. Since the outcome doesn't differ, let's assume core 0 wins the race. The sequence of events starts out as in Figure 4.3, and goes downhill from there. After the illustrated steps, the bus protocol should allow core 1 to acquire the block (forcing core 0 to write it back so the state can be *S* again) to complete the load and it can then do the add (#2), since that only involves registers, even if core 0 invalidates core 1's copy of the block again. The chances are core 1 will have another miss when it tries to write

the block at which point it retaliates by forcing core 0 to give up the block (writing it back as well, as part of the write miss from core 1). It's a useful exercise to calculate in this scenario how long it takes for both loops to complete as few as 10 iterations.

So how can we avoid this scenario?

One approach is to use processes rather than threads, with explicitly allocated shared memory, and take care that you manage how data structures are used between cores or CPUs. While processes are heavier-weight entities than threads, it doesn't take a lot of inadvertent false sharing to cancel out the gains. Another approach is to pad all variables to a size that's a multiple of the cache block size, making sure that all variables start on a cache block boundary. To do this, you may need your own memory allocator.

Locks

For multithreaded and multi-process programming, we need *mutual exclusion* to ensure predictable behaviour of updating shared variables. For example, if we have a global counter and two threads update it, we don't want a scenario where one thread loads it into a register, another thread also loads it into a register, then each increment the variable and store it back in memory, resulting in the counter only increasing by 1. A result that depends purely on the order competing processes or threads complete an action is called a *race condition*, and is usually a programming error. Since synchronization is critical to parallel program correctness, we need efficient implementations so synchronization doesn't become a bottleneck.

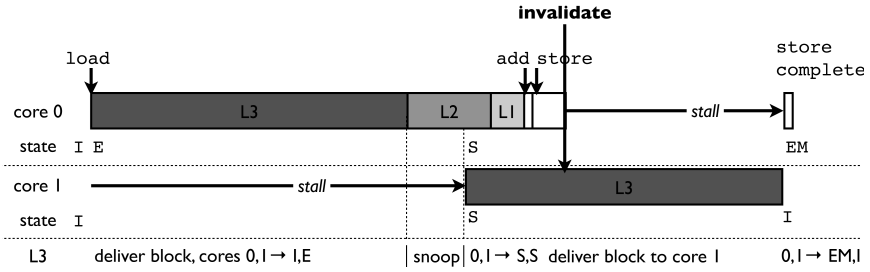


Figure 4.3: False sharing example. *Core 0 wins the race to acquire the block, and reaches instruction #3 before core 1 completes handling its miss to L2. L3 maintains global state of shared blocks, and issues a snoop on the bus to inform core 0 that its copy of the block is changed to state S before if can supply the block to core 1. Core 0 is stalled on cycle 4 of its store (#3) because it can only invalidate a block once the bus is free, before it can modify it. Latencies are drawn to scale; shading representing the memory hierarchy (darker ≡ slower).*

A simple lock structure, a spinlock, is based on an atomic memory operation. There are several that can work. Earlier architectures used *test and set*, that tested a value for a specific condition and set it based on the outcome, and the instruction was guaranteed to complete without interruption (or to have that effect). A more recent variant is *atomic swap*, which allows swapping contents of a memory location with a register. Again, the operation is guaranteed to complete without being interrupted. A spinlock using an atomic swap operation could look like this (assuming there is a location in memory we can refer pointed at by register `R5` that's initially 0, and `R6` is initialised with 1):

```
lock: swapw.i  R6,R5,0
      bnew.i R6,R0,lock
```

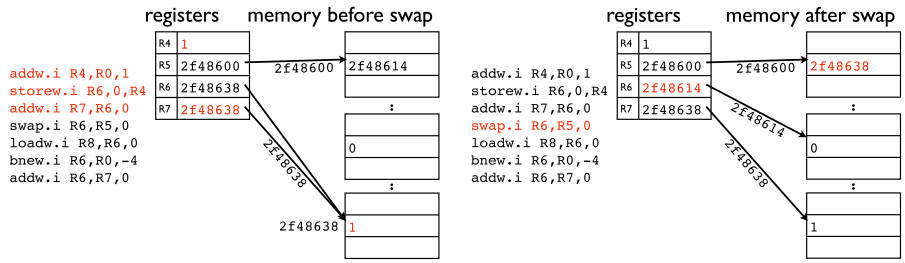
If our swap operation gets back a 0, that means the lock wasn't previously held, and we've now set it. If it's not a 0, we have to try again. For either outcome, we set the value of the lock to 1. If someone else held it already, we don't effectively change it, since it would be 1 already. This tight little loop continues until whoever else got in first releases the lock, which is done like this:

```
      storew.i  R5,0,R0
```

We rely on cache coherence to ensure that locking an unlocking is serialised, and updated to each processor or core when the lock variable's value changes.

This looks nice and simple; the bad stuff happens when a lot of processes or threads are trying to acquire the lock. Each time one of them tries to acquire the lock, since the swap instruction is a memory write, it must be invalidated out of any cache that holds it. The first core or CPU that tries to issue a swap instruction will cause an invalidate, forcing a write back, then get a copy in its cache, where it will immediately write to it (even if the effect of the write is to overwrite the lock variable with the exact same value as it had before, it's still a write). Every other process trying to acquire the lock will experience a cache miss, and queue up on the bus. If the process that holds the lock meanwhile has a cache miss for any other purpose, it will also have to wait its turn for the bus, further slowing things down. When finally the holder of the lock releases it, it will also have a miss, invalidate the lock variable from any other cache that has a copy, and modify it, causing a flurry of invalidates as each contender races to be the next one to acquire the lock.

If you think this sounds pretty bad, you'd be right. That's why there has been considerable research into more scalable locking strategies. One example of this is a *ticket lock*. There are various ways of implementing ticket locks (and some that are inconsistent with the basic definition, suggesting that the name sounds appealing and has been appropriated by anyone wanting to claim they have a scalable lock design). The key idea is that each contender for the lock picks up



(a) Just before swap. (b) After swap.

Figure 4.4: Simple ticket lock. *Contention for updates is reduced to grabbing the address of the next available "ticket".*

one ticket that corresponds to their place in the queue, and they can either spin on that ticket, which is a variable only shared between that thread or process and the one immediately ahead of it in the queue, or suspend pending the release of that specific ticket. Compared with a spinlock, a ticket lock has these benefits:

- *minimal multiprocessor contention* – there's a race to acquire the next ticket but from there on no spinning on a single shared variable
- *fairness* – the first contender to win a specific ticket will be serviced in the order of winning the ticket so starvation cannot occur

A simple implementation of a ticket lock is as follows. Each contending thread or process has to have a unique variable in shared memory, the address of which we store in register `R6`. We start out with the address of the global ticket in register `R5`. Initially, the global ticket pointer (the address contained in `R5`) must point to a location containing the value 0, so the first contender to win will find the lock not set. To lock:

```
      addw.i R4,R0,1
      storew.i  R6,0,R4
      addw.i R7,R6,0    # save for next time
      swapw.i  R6,R5,0  # find out who beat us
test: loadw.i R8,R6,0
      bnew.i R8,R0,test # wait for the winner if any
      addw.i R6,R7,0 # restore the address of our lock
```

We release the lock as follows:

```
      storew.i  R6,0,R0 # release our lock
```

What we have done here is to set up our own lock variable with the value 1, and swap its address with whatever address is already stored in the global ticket pointer. We now spin on the value at the address *previously* pointed at by the ticket pointer. The global ticket pointer has to be initialised to point to a memory location with the lock value set at 0 so the first contender can get in. The effect of this approach is that each contender is spinning on a *different* lock variable that will be released only when the process or thread that was immediately ahead of it in attempting to acquire the lock releases it. While there will be contention for swapping the address of our local lock with that stored in the global ticket pointer location, that only happens once for each attempt at acquiring the lock. Subsequent spinning is on a memory location only known by the current stalled thread and the owner of that location. Of course we need to ensure there is no false sharing though even this is not that important because we are spinning on a memory read rather than a write; if more than one core experiences an invalidation there will not be consequential series of follow-on invalidations. I illustrate the two crucial steps of the lock: initializing and the swap instruction in Figure 4.4. In Figure 4.4(a), I illustrate the way state is set up in a scenario where the initial ticket is still zero, representing the case where the lock is not held. In Figure 4.4(b), I show how the state changes when the current contender has acquired the lock.

Ticket locks are but one of many approaches to scalable synchronization primitives [Mellor-Crummey and Scott 1991]. Parallel programs use a range of primitives including *barriers*: a barrier with parameter N causes $N-1$ thread or processes to stall and when the Nth arrives at the barrier, all proceed. Implementing a barrier using spinlocks is very inefficient, even on a small number of processors, and threads packages such as Pthreads implement barriers by queueing waiting threads and putting them to sleep. Even so the underlying implementation will be unscalable if it's based on spinlocks as a basic building block. One approach to implementing scalable barriers uses a tree structure [Markatos et al. 1991]; another is to synchronize with nearest neighbours, limiting global communication [Machanick 1996].

4.4 Summary

Although there are many multiprocessor architectures, I focus here on shared memory MIMD architectures since they represent the mainstream of conventional computing. SIMD architectures in various forms have come and gone, and remain persistent mainly because GPUs are such a large share of the market.

Shared-memory architectures will likely be with us for some time given that they are a natural organization for multi-core devices, because they accommodate so many different types of workloads, including parallel applications and multi-

tasking. The performance issues covered here, before the multicore era, were primarily the concern of relatively high-end systems. Given that multiple cores are commonplace, understanding the performance problems and avoiding them is increasingly important both in user-level vode and system code.

Exercises

1. Use the latencies in Table 4.1, and the timing illustrated in Figure 4.3. You can assume every instruction can complete in one clock if fully pipelined, a store modifies memory in the MEM stage (4th stage of the pipeline), and an invalidate takes 2 cycles for the invalidating core if the bus is not occupied.

   (a) Calculate the total time it takes before core 0 manages to complete the illustrated store instruction.

   (b) Now assume that core 1 acquires the block as soon as possible after core 0's store completes. Calculate the total time from the start of Figure 4.3 until core 1 completes its first store, assuming that core 0 continues with the loop with minimal stalls.

   (c) Calculate the total time it takes for 10 iterations of the loop for both cores, stating assumptions about timing of competing events.

   (d) Why is it a reasonable assumption that an invalidate requires a relatively short stall (here, 2 cycles), not a longer delay, e.g., the 27-cycle delay required for a snoop?

2. Will the ticket lock as described here work as you expect if N threads enter it and leave it, then try to re-enter at a later stage? Hint: what will the global pointer have as its value, and what will be stored at that location?

3. Spinlocks are often used as a core primitive to implement more complex, scalable synchronization techniques like semaphores that put a process or thread to sleep and wake it when it reaches the head of a queue. Are spinlocks a reasonable choice in that scenario, or would you still look for a more scalable option like a ticket lock?

4. Is a test and set instruction superior or inferior to an atomic swap? Explain, considering the design philosophy of a RISC ISA.

5 GPUs

GPUs HAVE BEEN AROUND FOR A LONG TIME and represent an untidy mix of architectural ideas – so why are they worth considering separately? First, because they are a mix of architectural styles, they represent a case study in comparing the benefits and weaknesses of various models of parallelism. Secondly, because they are so widely available, there is more chance that, despite any difficulties in programming them, they may become established as an alternative platform for high-speed computation. It is that market, rather than the obvious one (given that the name means "graphics processing unit"), that hold some interest, because there are limits to how much further graphics performance needs to be taken. Once you can do realistic three-dimensional imaging in real time, where else can you go?

The idea of adapting a part designed for graphics processing to general purpose computation is not new; as I describe on page 14, the Intel i860 featured as a component in large-scale supercomputers in the 1990s. I also note there that it was not a great success. Will GPUs be more successful as high-performance computation engines? If only because they are deployed on a vast scale whether used in that mode or not, there is a lot of ongoing investment in pursuing this question. Since the primary market for GPUs remains graphics, design compromises will tend to favour that application. Early attempts at using GPUs as compute engines ran into the problem that design compromises favouring speed in graphics rendering meant that CPUs could not in general be expected to implement the IEEE floating point standard as strictly as a general-purpose floating-point unit [Chinchilla et al. 2004; Meredith et al. 2009] (the odd wrong pixel is less noticeable than failing to render the next frame in time). Given the growth in the market for general-purpose use of GPUs (GPGPU), manufacturers have started to pay attention to quality of their floating-point implementation [Krakiwsky et al. 2004; Whitehead and Fit-Florea 2011].

In this chapter, I briefly survey some of the architectures that contribute to the design of GPUs, adding to the discussion of Chapter 4.

5.1 Vector Processing

Vector processors fall into two broad categories: vector register architectures, and memory-based vector machines. The latter generally require vector registers to perform at a reasonable level, so I start with vector registers. Vector machines of the class of the designs created by Seymour Cray generally have very aggressive memory architectures. I briefly describe how these work; in the heyday of this class of machine, there was considerable research into designing memories for them [Cheung and Smith 1986; Weiss 1989; Valero et al. 1992; Seznec and Lenfant 1992], possibly a pointer to difficulties with vector register machines with no special memory architecture. Vector instructions sets as found in multimedia extensions to standard ISA generally lack some of the more sophisticated models for copying between registers and memory that are found on big vector machines.

A vector machine has registers that replicate a specific data type, and in some cases can be reconfigured as more lower-precision (narrower) or fewer higher-precision (wider) registers. For simplicity, I assume a vector register is comprised of a fixed number of scalar registers of a fixed size. To make the discussion concrete, I assume a vector register has length 64 (i.e., it can contain 64 distinct values). Vector architectures can generally either operate on a pair of vectors producing a vector result, or a vector and a scalar producing a vector result. In the latter case, the same scalar is an operand for each operation on the source vector. For example, you may want to multiple each vector element by a constant.

In a case where the available vector length (here 64) is sufficient, you can use a single instruction to do the main work (e.g. multiplying all 64 elements by a constant, or adding all 4 elements to the equivalent entries of another vector). This single instruction has a latency dependent on how long the hardware can perform 64 operations. By contrast, if you use a scalar architecture, you will do the same 64 operations but need to wrap a loop around them, and add ancillary code for array indexing. The saving in number of instructions executed in this case should be about a factor of 100. That is not as big a saving as it sounds, since the vector unit still has to do 64 operations, and those cannot happen instantaneously. However contrast the requirements of speeding up the vector operations by adding more parallel hardware with doing the same for the scalar code. The scalar code has a loop, so you will need to unroll the loop, either by a coding technique (compiler optimisation or hand-unrolling it), or hardware support such as Tomasulo's algorithm. If you go the first route, you need to know in advance how many times it's worth unrolling the loop; in the latter case, you add significant hardware complexity including register renaming. On the other hand, to speed up the register code in hardware is relatively simple. You can add more functional units and provided there are no dependences between computing vector

elements, as many calculations as there are available functional units can start at once.

You can apply the same trick as with a scalar pipeline to reduce inter-calculation delays, forwarding a result to the input of a functional unit rather than going via the register file. In a vector architecture, forwarding is called *chaining*.

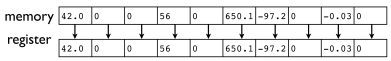
In a simple implementation of a vector machine, each ALU operation takes as long as in a single-issue scalar machine. The major saving is in fewer instruction issues and removing branches for loops. In the case of a simple loop with two ALU operations, in a typical RISC instruction set the loop body and condition would add up to about 10 instructions. For a 64-long register vector machine (without for now going into how the operands find their way to registers), the equivalent code would be about half the number of instructions and also would not require repetition. The scalar code would therefore require about 128 times the number of instructions, though the number of ALU operations would be the same, meaning that the practical speed gain would be relatively modest, especially if the ALU operations are multi-cycle floating point operations. The big gain from vector instructions comes from the extremely regular nature of the parallelism, which makes it possible to split the work across multiple functional units without complications such as data and control hazards.

In big-iron vector machines such as those designed by Cray, vector ALU operations are accompanied by vector memory operations, which is where things start to get more interesting. In a simple example where the data size exactly matches the register length, a vector load instruction that fetches the next N (64 in our example) elements of an array, sequentially from a given base address, is a good fit to the problem. There is a range of scenarios that cover cases where the data size is not an exact fit:

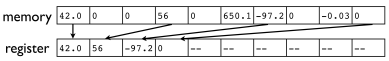
- *data shorter than the vector size* – one approach is to have a *vector length register* (*VLR*) that can be any value up to the hardware vector length; vector instructions' length is controlled by the VLR's value. Some machines also have a *maximum vector length* set in the hardware. The MVL can change in new hardware, avoiding the need to change the instruction set when the vector length changes

- *data longer than the vector size* – use the MVL value to create an outer loop that splits the problem of size N into the portion that is an exact multiple of MVL, repeated $\lfloor \frac{N}{MVL} \rfloor$ times, and the remainder (done once). This technique is called *strip mining*

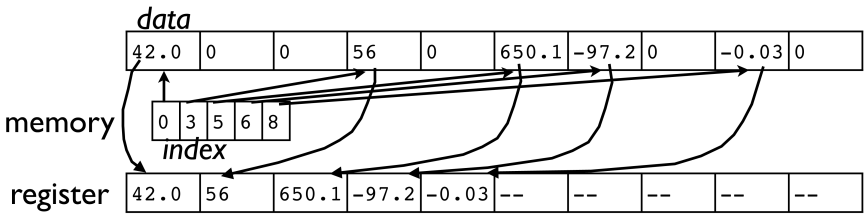
- *elements not adjacent in memory* – we need a way to specify a *stride*, a distance apart of memory locations. Big-iron vector machines have this



(a) Simple vector load



(b) Vector load stride 3



(c) Vector load gather.

Figure 5.1: Variations on vector loads. *Stores have similar variations; the store version of a gather is a scatter. In stride mode, the load fetches data stride S apart. In gather mode, an index vector is used to find the offset from the start of the main array in memory.*

capability (e.g., a stride register could set the distance apart of successive elements for vector load or store)

- *sparse vectors* – in a case where a data structure has a lot of zeros (or other elements not of interest), it may be stored using indirect indices, e.g., element `i` is found at `A[index[i]]`. To handle this scenario, vector architectures may use *gather-scatter*:
  - *gather* – use an index vector to add to a base address to do loads
  - *scatter* – use an index vector to add to a base address to do stores

Gather-scatter can be used to save memory, if the index array is a smaller data size than the actual data, and also as a way of accessing memory in different orders without having to sort the original data each time a different ordering is needed. To implement all of these operations with an aggressive vector CPU that can do multiple ALU operations per cycle requires high memory bandwidth. To get some idea how much, if we consider a clock cycle time of the order of 2ns (500MHz: fast at the time of the later Cray vector machines), with main memory SRAMs with a cycle time of 15ns (available at the time), one vector load would saturate the memory system. If we start going more aggressive and allow more than one load or store per cycle, or add processors, the system is going to be memory-bound despite the fast SRAM main memory. The solution is to *multi-bank* the main memory. A bank is a division of memory that can be separately addressed. While one bank can't supply results as fast as the clock speed, pipelining access to multiple *banks* can. Supercomputers of that era could have of the order of 1024 banks. A similar

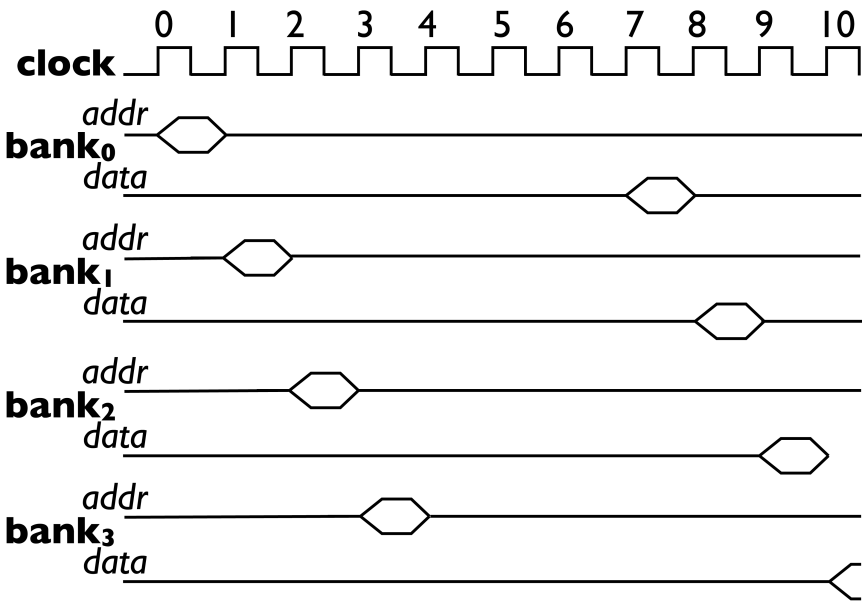


Figure 5.2: The principle of multiple memory banks. *An access is started on each bank in each successive cycle. Although there may be a sizeable delay before each bank responds, each bank will deliver its request in successive cycles after that delay.*

effect can be achieve with multi-banked DRAM, though the startup delay is much higher. Modern architectures tend to reduce the need for memory banks by using very fast caches that can keep up with the CPU. DRAM chips may have internal banks, and DRAM may be arranged in banks (typically one per DIMM), but all of this is to improved latency for the sort of contiguous access required for caches, rather than to support concurrent access to very different regions of the address space.

In Figure 5.2 I illustrate the general principle of multiple banks. In this case, there are four, and the total latency of a memory operation is 8 cycles. By starting bank requests on successive cycles, the total latency for 4 accesses is 11 cycles, rather than 32 cycles, as would be the case if each access had to be strictly sequential. In the illustrated scenario (where each bus transaction takes only 1 cycle), there is no contention for the interconnect (a hexagon in a timing diagram as seen here represents a state where values could be 1 or 0, and we don't care which, only that a transition occurred), so data and addresses could use a common bus, even though I illustrate the separately. In high-end systems of the Cray era, a more complex interconnect was required.

Another inhibition on vector mode is conditional code. If you have a program that for example should only do an ALU operation if a vector operand is non-

zero, you want your vector code to apply the operation to every element except those that are zero. A common approach is to use a *vector mask*, a special register with 1 bit per element of a vector register. If the bit is 1, the operation is wanted. If not, it isn't. To implement vector masks, you need an instruction that resets the mask to all 1s (the default, meaning all operations are wanted), and vector compare instructions that set the corresponding bit of a vector mask based on whether the compare is true (`1`) or false (`0`). The mask then applies to whether the outcome of the ALU operation is stored on not; the time for the ALU operation is unchanged. The mask in effect only says whether it's written to the destination vector register element or not. While time and resources are wasted for results that aren't needed, the overall effect in most cases is still faster than executing the ALU operations in scalar mode with a loop.

Why aren't vector machines mainstream? Cray's machines peaked in the 1990s. Seymour Cray split from his company Cray Research in 1990 when the Cray 3 project was put on the back burner, and his new company, Cray Computers, failed to sell more than one of its first model, the Cray 3, and folded before it could deliver its successor, the Cray 4. Cray Research continued for a while but with the end of the cold war, generous funding for high-speed computers of limited applicability faded and at around that time, RISC architectures started to deliver a significant fraction of the performance of specialist architectures at a fraction of the price. In the mass market, large SRAM memories with a thousand or more banks are not practical (not until someone finds a way to package them cheaply anyway).

The big question that all of this leads to is whether vector architectures embedded in GPUs and multimedia media extensions to instruction sets make any sense without the massive memory bandwidth available to a machine of the old Cray type. The Cray X1, for example, a successor to the T90, has 16 memory controllers per node supplying 204 Gibytes/s to each 4-core vector unit [Dunigan et al. 2005] – and this is an architecture with caches, unlike earlier Cray designs.

5.2 SIMD Extensions to Instruction Sets

Many media applications need relatively short data types, e.g. 8 bits per colour, and it's relatively easy to partition an arithmetic unit so that instead of 32 or 64-bit arithmetic, it can do multiple instances of a narrower unit like 8 bits. In the Intel IA32, multimedia extensions were added by a relatively modest extension of the existing ALU based on this principle. Where Cray vector registers were in the range of 64-128 long supporting full-size floating point, the data types arising from such modest extensions are limited to what can fit into a double-precision (64-bit) register. Intel's original MMX additions were based on that simple model. Later extensions, Streaming SIMD Extensions (SSE) double the register width to 128,

and the latest iteration, Advanced Vector Extensions, increased register width to 258, allowing up to 32 8-bit operations per register.

Because these are ad-hoc extensions with big jumps from each design, and without the advantage of the older vector architectures of hardware support for varying the vector length, the number of new instructions is large, several hundred counted across all Intel's variations [Firasta et al. 2008]. There are about 90 AVX instructions, if you do not count all variants of the same basic instruction separately, and the AVX reference runs to 750 pages [Intel 2009].

While compiler support for these instructions is improving, it is not nearly as easy for a compiler to spot opportunities to use them automatically as with a traditional vector architecture. They tend to be used more commonly in hand-coded drivers or plug-ins for programs with intensive graphics requirements. Despite these problems, this form of limited SIMD does have some advantages. Unlike vector machines, a page fault across a load or store boundary can't happen – or at least it couldn't until Intel allowed loads to be explicitly unaligned with SSE [Lomont 2011], and relaxed the requirement for loads to be aligned in most cases with the AVX design [Intel 2009]. Also, the limited vector size is a better match to commodity memory systems that would battle to keep up with the demands of a full vector instruction set, with 64 or more double-precision floating point numbers per vector register.

Overall, while SIMD extension instruction sets initially set out to be simple, avoid the complications that attend traditional vector designs, hundreds of instructions requiring a good fraction of 1,000 pages to document suggests something not quite right.

5.3 GPUs

Graphics processing units are increasingly migrating to the general purpose space (hence GPGPU: general-purpose use of GPUs). As with SIMD extensions to instruction sets, they suffer ad hoc design and repeated changes. That style of change has a venerable history. Silicon Graphics, the pioneer of high-speed 3D graphics, went through architecture iterations that reprised a good fraction of the major models of parallelism:

- *pipeline* – early versions of the SGI Geometry Engine were deeply pipelined, with some SIMD aspects [Harrell and Fouladi 1993]

- *heteorgenous architecture* – the Reality Engine used a small number of relatively non-exotic Intel i860 processors with hundreds of specialised cores [Akeley 1993]

- *SIMD* – the InfiniteReality system of the late 1990s uses a SIMD architecture [Montrym et al. 1997]

SGI early on realised the need for a high-level programming interface that hid the hardware, and developed GL, the basis for OpenGL, as an abstraction layer. That approach made it possible to change the underlying implementation radically as design trade-offs changed.

However, SGI did not ever envisage their graphics hardware being used for high-speed computation: they had a different department covering that, and they had very competitive machines in the 1990s, that were part of the reason that traditional supercomputer makers like Cray ran into trouble.

In more recent times, the underlying reason for rapid change in graphics hardware has not changed. As hardware becomes cheaper, approaches to graphics processing that previously were impractical become viable. Unlike with the history of SGI though, those changes are accompanied by an increasing demand to make it possible to run non-graphics applications on GPUs.

NVIDIA has addressed the problem of rapid hardware change providing C and C++ extensions called *CUDA* (Compute Unified Device Architecture) that allow programming that divides code between the host CPU and the graphics system. OpenCL (Open Computing Language) is a more generic alternative (extending C) that aims to be portable across a wider range of hardware, not only GPUs [Stone et al. 2010]. Aside from the usual portability concern (ideally, a recompiled should be sufficient to run on a different CPU), *performance portability* is a hard problem [Du et al. 2012] even within one manufacturer's line: assumptions underlying your coding strategy may not apply on a different model.

A few basics apply to current designs. First, streaming access to memory hides latency. As with multiple banks in older designs, in current DRAM designs, every access after the first has no additional delay (up to the limit of a column access).

I examine briefly some of the features of a typical GPU from NVIDIA, and the performance portability problems that can arise.

First, the memory hierarchy of a GPU is complicated. In recent designs that support multiple SIMD threads to hide memory latency, there is a small cache for local variables that don't fit the streaming model. There may also be a local memory that is used for synchronization between threads (e.g. NVIDIA has a hardware barrier instruction [NVIDIA 2011]). Then there is a global memory that is separate from the main CPU's main memory. Second, NVIDIA hides frequent changes in the hardware by using an abstraction layer in the form of the PTX (Parallel Thread Execution) instruction set, that has to be translated at load time to the actual underlying machine instruction set.

PTX has about 40 basic instructions that Hennessy and Patterson [2012] use in examples. There are many other specialist instructions and when you add in all the available variations, the number blows out to hundreds, though the reference manual only runs to about 200 pages [NVIDIA 2011], potentially an improvement on Intel's AVX design at least in that respect. PTX hides some of the complexity of identifying threads and branching, allowing these to vary from implementation to implementation.

Here is a contrast between traditional vector and PTX code, implementing the following function (DAXPY stands for double precision *a* times *x* *p*lus *y*, and is part of the popular Linpack benchmark suite):

```
void daxpy (int n, double a, double *x, double *y) {
      for (int i = 0; i < n; i++)
            y[i] = a * x[i] + y[i];
}
```

First, let's look at how a typical old-school vector instruction set would implement this. In pseudocode, it would be something like this for the body of the loop:

```
Vload Rx, x[i]    # get VL items starting at i
Vload Ry, y[i]    # get VL items starting at i
VSmuld Rx, a, Rx  # do the vector*scalar multiply
VVaddd Ry, Ry, Rx  # do the vector add
Vstore y[i],Ry    # vector store result
```

In a strip-mining solution, we need to take care of details like how often to repeat the loop and a fragment where the full vector length isn't needed.

A PTX version is significantly more complicated though superficially it looks similar. Part of the reason for that is that memory access always uses gather-scatter. Also, in creating SIMD code, you create a large number of threads, as part of the strategy of hiding memory latency by using threads. Rather than use vector registers, you allocate a block of threads, then do a calculation simultaneously with each thread doing a different part of the calculation. This would appear to throw away the performance advantage of sequential memory access, but if a program is written so adjacent threads access adjacent memory, the memory subsystem coalesces memory references. The basic steps in pseudocode are:

```
use thread id to create offset in array
load x[i+offset]
load y[i+offset]
do x*a multiplication
add to y
store y[i+offset]
```

This code is replicated across threads, each with a different id and hence offset in the array.

An important difference between GPU threads in the NVIDIA world and threads in a general CPU is that all threads are either executing the same instruction or are idle. A combination of mechanisms makes this possible, including masks similar to those in vector machines and predicates, similar to those in VLIW machines. Branches allow threads to *diverge*, with hardware support to handle managing this. The unwary programmer can create code where most threads are idle.

Although CUDA and OpenCL provide an even higher-level abstraction than PTX, some basic understanding of the underlying hardware is necessary to program efficiently.

5.4 Review

Let's compare GPUs and multimedia extensions with what we generally know about instruction set design. Here are some core principles derived from the RISC movement and experience with supercomputers:

- *Amdahl's Law* – speedup depends on the whole workload, not only the subset that can be improved
- *make the common case fast* – a large instruction set with rarely-used instructions makes it harder to achieve overall speed improvement
- *minimise instruction format variation* – keep fetch and decode simple to make aggressive pipelines easier to implement
- *optimize for average throughput not peak throughput* – as Cray demonstrated in the days of big iron vector machines, a high peak throughput is meaningless if the average case isn't close to the peak
- *simple memory model for programming* – even if there's a complex memory hierarchy that varies from generation to generation of the hardware, a simple uniform model for programmers ensures code longevity and performance portability over time

Why then do multimedia extensions (Intel is not the only guilty party: the AltiVec extensions to PowerPC are also large and complicated, with a reference manual running to over 260 pages[1], if more regular in design than Intel's efforts [Freescale 1999]) and GPU instruction sets violate these principles?

[1] You can find a summary on Apple's developer web site – `https://developer.apple.com/hardwaredrivers/ve/instruction_crossref.html` – a legacy of when Apple used PowerPC.

A key consideration is the *real time argument*. In *hard real time*, if a deadline is not met, the system is broken; in *soft real time*, failing to meet a deadline is a performance bug but tolerable (e.g., if the picture pixelates but no so often as to be annoying, you keep watching your digital TV). While graphics is not strictly a hard real time application, the faster the graphics system, the better the quality picture. In graphics-intensive applications like a photo editor, implementing a filter fast enough to be usable adds value, even if the careful hand-coding necessary doesn't speed up the overall application, a very different consideration than applying Amdahl's Law. If the system is fast enough, expectations expand, but there is a limit to human perception. At some point, perception saturates and there is no point making graphics any faster. Once we approach that point, selling faster GPUs requires another market, hence the interest of GPU makers in selling to a broader base.

However, once we exceed the limit of human perception, a model of GPU that has lower peak throughput but a much simpler instruction set that can be used effectively by compilers has a lot to recommend it. If such an instruction set had 80% of the peak throughput of a much more complex design, either it would be sufficient when the more complex design was sufficiently ahead of human perception, or it could be implemented as two independent cores with at least the same performance as the more complex design, with the option to use one of the other cores for non-graphics tasks. If the ISA were general enough to apply to ordinary workloads, instead of a separate GPU, a multicore design could have some cores used exclusively for graphics and others for computation, with the option to choose dynamically which to use. Another design challenge is how to organize memory so that both graphics and ordinary usage would be satisfied; high-end graphics systems generally avoid this problem with dedicated memory. A cost of dedicated memory in graphics systems is a memory hierarchy that's difficult for programmers.

A detailed design is necessary to evaluate these ideas, as was the case with the original argument for multiple cores (then called a chip multiprocessor [Olukotun et al. 1996]). A useful starting point would be a minimal RISC instruction set, with design studies to determine extra styles of instruction that add the maximum value for parallel execution modes. We can safely avoid ideas that failed in the past like VLIW, be cautious about adopting ideas that are hard to program like SIMD, give careful consideration to ideas that work well in limited cases like vector instructions, and shun ideas that make life for programmers hard, like local scratch memories under programmer control.

Exercises

1. You can find some specifications of the Cray T-90 here (Table 1): `ftp://ftp.cs.ucsd.edu/pub/faculty/carter/cug.html`. Based on numbers you find:

   (a) What is the maximum number of loads and stores possible in one clock cycle?

   (b) In an 8-processor configuration, with the maximum possible numbers of loads and stores, how many banks of 15ns RAM are required to keep up with demand? Assume each load or store can be divided into as many banks as are needed.

   (c) The top model of this range, the T932, had up to 32 processors and a slightly faster clock speed than that in the above reference (2.167ns). It had 1024 banks of RAM, and the RAM was upgraded from a 15ns cycle time to twice as fast. Was this upgrade necessary?

2. Look up details of the AltiVec instruction set. How does it compare with the other architecture styles we've examined? Is it a reasonable fit to the RISC philosophy?

3. Find a detailed example of NVIDIA PTX code. Explain how parallelism is achieved in the example.

4. If you were designing the Intel AVX instructions from scratch, rather than as an extension of previous designs, how different would your approach be?

6 Warehouse-Scale Computing

MASSIVE-SCALE COMPUTING in the 1990s was the province of high-performance computing (HPC), mainly a concern for computational sciences and large-scale engineering projects (e.g. simulating wind tunnels). Much of that market has disappeared into various models of scaling up commodity parts, e.g. clusters. In some cases, these designs use extra-fast interconnects, but many use commodity networks. A big change since the 1990s is the emergence of massive-scale computing mainly targeting ordinary consumers, not large commercial or research enterprises.

A key difference in the new kind of large-scale computing is economy of scale. Large service providers like Google and Amazon deal in customer bases in many millions, and achieve economy of scale on three fronts:

- *mass-market commodity hardware* – whereas supercomputer makers like Cray and Thinking Machines designed their own parts for a very limited market, this new category of computing draws on the low cost inherent in massive markets

- *purchasing at scale* – even given that these operations use commodity hardware, they score by being able to buy in massive quantities, and hence achieving a much lower price point per unit of work than even a home PC; this large scale also makes it possible for them to design custom configurations of commodity hardware and still arrive at a reasonable cost [Barroso et al. 2003]

- *massive user base* – unlike past HPC-oriented large-scale computing, the new services spread their costs over an enormous user base

All this being the case, some of the complexities of scaling up to extremely large systems remain. Google, Amazon, *et al.* to some extent have the luxury of choosing the services they offer, since many are offered at no charge, and as a way of building advertising revenue or directing users to for-money services (like buying books or apps).

As a generic term for such large-scale services, we use the term *warehouse-scale computer* (*WSC*). Google famously uses relatively entry-level computers, and lots of them. In an operation on this scale with over 50,000 computers, managing individual computers is not possible. A WSC operation has to have considerable support for automated managing of configurations, detecting errors and moving calculations when a computer fails. The range of applications run on these systems is highly variable, and that variation to some extent makes them viable. For example, a large part of Google's operation is web crawling to build search indexing. That kind of workload is both highly parallel and not interactive, and can soak up any available computational resources and network capacity. Multiplexing that kind of workload with requirements for more rapid response time is a good mix, as temporary demand for interactive response time can easily be accommodated by reducing resources for the other type of workload. Contrast this with an electricity grid where instant responsiveness requires not only a lot of spare capacity, but generators capable of rapid cycling up. In one such system for example (the Australian state of Queensland where I used to live), the last 1% of demand costs 100 times base load per kilowatt hour. Power utilities could learn a thing or too about load and demand balancing from WSC operators.

All of this is not however without significant challenges. Users of interactive services, especially those where they care about losing data and want access when they need it, expect a highly dependable service. Downtime of 1 day a year requires 99.7% availability and downtime of at most an hour requires 99.99% availability.

6.1 Fault tolerance and dependability

A key aspect of large-scale systems built out of reasonably reliable components is that the probability of failure increases as you scale up, because there are more parts to fail. First I start with some terminology in Table 6.1 and the following definition:

$$Availability = \frac{MTBF}{MTBF + MTTR} \tag{6.1}$$

or alternatively,

$$Availability = \frac{t_{total} - t_{down}}{t_{total}} \tag{6.2}$$

A key thing to understand is the difference between *dependability* and *reliability*. Something that's reliable has a low chance of failing. Something that's dependable has a low chance of not being usable despite failures. A way of ensuring that dependability is higher than reliability is by *fault tolerance*. Fault tolerance is often achieved using *redundancy* along with error checking and correction. For example,

term	definition
$MTBF$	mean time between failures: expected time before a module fails
$MTTR$	mean time to repair: expected time to fix a faulty module
reliability	measure of probability of no failures
dependability	measure of likelihood of being useful
fault tolerance	ability to work despite failures
availability	fraction of time a system is able to do work
durability	total time a system is useful
nines	availability of 99.9% is 3 nines for example

Table 6.1: Dependability terminology.

probability / year	failure type
0.02	disk failure
0.01	uncorrectable DRAM failure
0.3	bad software configuration

Table 6.2: Dependability example. *There are many other sources of failure like software crashes and uncorrected power glitches (assuming use of backup power).*

a RAID disk system may be configured so that if one drive fails, its content may be recreated. Although the disk subsystem has had a failure, it still works and is therefore dependable, even if it's not reliable.

Similar considerations apply to WSC with tens of thousands of computers. Not only the computers themselves, but networking, building power supplies and software can all fail. To make this concrete, let's take a centre with 2,500 computers and apply the failure rates in Table 6.3. Assume a hardware fault takes 1 hour to repair, and reboot takes 60s. With the figures in Table 6.3, in an average year with 2,500 computers we get the expected number of failures in Table 6.3. Optimistically assume we can fix a bad software configuration with a reboot, and the others require a hardware repair taking an hour. Then the total time systems are

expected failures / year	failure type
50	disk failure
25	uncorrectable DRAM failure
750	bad software configuration

Table 6.3: Expected number of failures for 2,500 computers.

out of action is

$$\begin{aligned} hours_{outage} &= (50+25)\times 1 + 750 \times \frac{1}{60} \\ &= 87.5 \end{aligned}$$

Applying Equation 6.2, and noting there are 8766 hours in an average 365.25 day year:

$$\begin{aligned} availability &= \frac{8766 - 87.5}{8766} \\ &= 0.99 \end{aligned}$$

So any service requiring continuous use of all 2,500 computers would experience two nines of availability. A real system would have more modes of failure than those listed here, so availability without error correcting and fault tolerance would be considerably lower in practice.

A system like Google's relies on a combination of features to ensure dependability. First, there is considerable checking for potential faults. Second, when a highly distributed computation has a few outstanding results, rather than wait for them, they are farmed out again to the network. Third, there is a high degree of replication of data, to ensure that a hard failure can be recovered. This replication is also required for performance, so fault tolerance falls out of the basic design, rather than being an expensive add-on [Barroso et al. 2003]. In general ensuring high availability in such a large-scale system is a complex task, and the ability of large operations like Google and Amazon to maintain services with high dependability, especially as Google has history of rapid evolution of their user-level software offerings, is a considerable achievement.

Fault tolerance is a large and complex topic; whole courses are given on the subject. I leave it here with a few of the key concepts, rather than an in-depth coverage.

6.2 Programming model

WSC provides parallelism on an unprecedented scale. Given that ordinary-scale parallelism can be hard to use, as we've seen in preceding chapters, does WSC provide a model for large-scale parallelism, or is it only good for thousands of uniprocessor workloads (in itself a useful feature)?

A lot hinges on the programming model and the nature of parallel workloads. Google uses an approach derived from two LISP programming constructs, `map` and `reduce`. In LISP (a predecessor of modern functional languages), `map` is a family of functions that apply another function to each element of a data structure,

producing another data structure usually of similar size. The LISP `reduce` function applies a function pairwise to elements of a data structure to produce a single value. An example of application of a LISP-style `map` operation would be to take a list of words and return a list of the length of each. An example of a LISP-style `reduce` operation would be to take a list of numbers and return their sum (here, the applied function would be "+").

Google's MapReduce[1] and the free equivalent, Hadoop MapReduce (Hadoop is an Apache project, including a distributed file system with related tools and services[2]), are based on the LISP `map` and `reduce` concepts. In Common LISP, a map operation looks like this:

```
(map 'list 'length '("fred" "jim" "james"))
=> (4 3 5)
```

and a reduce operation looks like this:

```
(reduce '+ '(4 3 5))
=> 12
```

with a lot of variations possible[3]. The single-quote symbol in LISP forces the next item to be passed to the calling function without evaluation.

In a MapReduce implementation, a map operation takes as input a function and a list of values. The function produces an intermediate value in the form of a list of keys and values, and a reduce operation applies another function to the result of map. In a typical application, the items in the list of values would be large enough to schedule as work units on separate machines, so the map and reduce stages provide a model of parallelism. Part of the fault tolerance in the design is periodic checks on whether the sub-tasks have completed. If they don't after a timeout, the master process restarts them on another node. Coordination and synchronization occurs in effect by a combination of the reduce tasks waiting for map outputs, and the master process waiting for the reduce tasks to complete. Scalability depends on reasonably large chunks of work in each of the map and reduce list elements, and on a reasonable load balance. The overall approach appears to be very successful, given the scale of Google's operation. Within 5 years of the development of the initial implementation in 2003, Google had more than 10,000 internal MapReduce programs, and each day 100,000 MapReduce programs processed about twenty petabytes (PB = $10^{15}B$) of data [Dean and Ghemawat 2008].

[1] You can find a MapReduce tutorial here: `http://code.google.com/edu/parallel/mapreduce-tutorial.html`.

[2] `http://hadoop.apache.org/`

[3] More on `map` here: `http://www.lispworks.com/documentation/HyperSpec/Body/f_map.htm` and on `reduce` here: `http://www.lispworks.com/documentation/HyperSpec/Body/f_reduce.htm`.

MapReduce is of course not the only programming model possible for large-scale distributed computing; a comprehensive study of the options is worthy of a whole course. Whatever approach is used has to observe a few key principles to achieve scale:

- *minimum communication* – each scheduled unit of work should be reasonably large and able to complete without sharing data with other work units

- *coordination* – there should be a strategy to ensure that outstanding work is completed and there is a reasonable balance between waiting for uncomplete work and scheduling new work; coordination decouples communication and cooperation from computation [Gelernter and Carriero 1992; Tanenbaum and van Steen 2002, p 700]

- *load balance* – work should be spread reasonably evenly over available resources; while rebalancing load by migrating workloads is theoretically possible, the costs in time lost to communication seldom make the move worthwhile

- *fault tolerance* – there should be a fallback strategy to cope with parts of the workload failing to complete

The programming model is interesting to the computer architect because there has to be one for an architecture to be usable (hence the drive to find usable models for GPUs that can use a reasonable fraction of their theoretical throughput, rather than hand-tuning assembly language). MapReduce is successful and therefore validates the broad concept of WSC; that doesn't mean a better model can't be found, but it is not a problem for the computer architect.

MapReduce has another aspect of interest to a computer architect: it is similar in some ways to a dataflow architecture, in which operations are fired by availability of operands, rather than being driven by order of the code. Dataflow was a style of architecture that attracted some research interest in the 1990s [Ghosal and Bhuyan 1990; Arvind and Nikhil 1990; Lieverse et al. 1999]. Despite some attempts to revive the concept [Swanson et al. 2006; Petersen et al. 2006], dataflow has not been widely adopted because it's too hard to build hardware that fully exploits theoretically available parallelism in the model without changes to programming languages. Though dataflow languages were also an area of active research for two decades [Traub 1986; Johnston et al. 2004], in practice it is hard to sell a new architecture without the option of (mostly) running existing code. Some versions of Intel's IA32 pipelines use dataflow [Papworth 1996], though the parallelism in that case is relatively local (instruction or micro-op

operation	latency
network switch	$10\mu s$
local RAM access	$100ns = 0.1\mu s$
disk latency	12ms

Table 6.4: Performance parameters for scalability. *Disk latency is based on half a rotation for a 7,200rpm disk (4ms) plus a conservative 8ms average seek time, assuming a cheaper design than a fast enterprise drive. Local RAM access assumes a miss to DRAM.*

reordering). In the case of MapReduce, dataflow is more a coordination (large-scale parallelism) concept than a highly local form of parallelism, and seems to work well at that scale; use of dataflow languages for coordination is an idea developed independently of MapReduce [Lombide Carreton and D'Hondt 2010]. Dataflow architectures today survive in specialist domains [Vo 2011] and in FPGA-based designs, where the programming model is nonstandard anyway [Silva and Lopes 2010; Voigt et al. 2010; Ferlin et al. 2011].

6.3 Hardware Design

One of the most important considerations of a system on this scale is cost per unit of work. In the early 1990s, when the RISC revolution was at its height, I made the observation that a high-end box was seldom worth the extra cost because the maximum performance was had from a machine a step or two down from that with as much RAM and disk as you could afford. Any machine that you could not afford to populate to the maximum with RAM would no longer be worth the cost of upgrades in a year or two. Google has made a similar discovery: they generally use components typical of a mid-range rather than top of the line PC. An important consideration in their design is overall cost, including power consumption and heat. Another important consideration in design for scale is network bandwidth and latency. If the network within a building has high bandwidth and low latency, workloads that require some communication can be accommodated within a building or if the requirements for communication are higher, within a single rack with a single fast ethernet switch.

To get some idea of how things scale, let's take some numbers. Actual latency of ethernet depends on how loaded the network is as well as how many switches there are between the nodes sharing information and an accurate model of performance should be based on real workloads [Jin and Caesar 2010]; network latency in Table 6.4 is a little on the optimistic side. On the other hand, disk

latencies and memory are on the high side: I assume that as with the Google philosophy, we are aiming for a midrange PC configuration, rather than enterprise-grade drives, and that all memory access are misses to DRAM. This combination of assumptions reduces the penalties for remote access, and puts an upper bound on scalability estimates.

Taking all this into account, let's estimate the fraction of memory accesses that can be remote without doubling average memory access time. That is a break-even point of a fashion: with that amount of overhead, it would be better if we could make the work go to the remote node rather than access its data. Let's go back to our relative execution time formula (Equation 2.1), remembering that we are not really calculating execution time. In this case, we are not even calculating relative execution time as before, just comparing local and remote memory accesses. Assume that the basic latency numbers are a close enough approximation to overall transaction time, which is true of relatively small accesses, and we have a workload where we only have local and remote memory accesses, and no disk accesses. Then our average memory access time is:

$$t_{MEM} = t_{local} + t_{remote} \tag{6.3}$$

I define local access time t_{local} using the fraction of memory references that are local mem_{local} and time to access RAM t_{RAM}

$$t_{local} = mem_{local} \times t_{RAM} \tag{6.4}$$

and remote access time t_{local} using the fraction of memory references that are remote mem_{local} and time to access the network t_{NW} (using the above assumptions, as defined in Table 6.4):

$$t_{remote} = mem_{remote} \times t_{NW} \tag{6.5}$$

We want to find the point where $t_{MEM} = 2 \times t_{local}$, which leads to

$$\begin{aligned} 2 \times t_{local} &= t_{local} + t_{remote} \\ t_{local} &= t_{remote} \\ mem_{local} \times t_{RAM} &= mem_{remote} \times t_{NW} \\ 0.1 mem_{local} &= 10 mem_{remote} \\ \frac{mem_{local}}{mem_{remote}} &= 100 \end{aligned} \tag{6.6}$$

In other words, if more than 1% of memory references are remote, we are going to see a slowdown of at least 2 versus purely local computation, and we need to rethink our programming strategy.

The calculation I present here is very optimistic: in a real machine in which most memory references are cached, going over the network is a much larger performance hit, even if we don't add all the components of network latency I've missed here. What I have not gone into is how memory is accessed over a network. In most cases, there will be more to it than putting a packet on a network: a process at the other node will have to interpret the packet and send a response.

Let's now consider a simple memory hierarchy in which latency for a cache hit is hidden by the pipeline and so is effectively the same as the clock speed. Let's set the clock speed to 2GHz, or $0.5ns$. Let's conservatively allow 10% of memory references to miss to DRAM (a high miss rate in most practical systems, e.g. recent Intel designs with 8-12MiB of L3 cache; here I only consider 1 level of cache for simplicity). Then applying Equation 2.1, the average local memory access time is

$$\begin{aligned} t_e &= t_{h_1} + \sum_{i=1}^{n} p_{m_i} \times r_{m_i} \\ &= 0.5ns + 100ns \times 0.1 \\ &= 10.5ns \end{aligned}$$

In the units used to derive Equation 6.3, $10.5ns = 0.015\mu s$. So rewriting the local memory term:

$$\begin{aligned} 0.015 mem_{local} &= 10 mem_{remote} \\ \frac{mem_{local}}{mem_{remote}} &= 666.7 \end{aligned} \tag{6.7}$$

These numbers should give some indication, without working through in full detail, that a model like MapReduce has to distribute relatively large chunks of work that can be computed independently, only communicating results after reasonably long computation.

To make things worse, the minimal network latency only applies if you stay within one network switch. Typically a network switch will cover one rack; there may be several switches covering a full warehouse, and once you go out to the wider Internet, latency quickly mount up. 4000 km, about the distance across continental United States, is about 0.01 light seconds, so the shortest time (unless you can find a way to work around relativity) that you can access information over that distance is about $20ms$, 2,000 times our extremely optimistic network access time, though to be fair, this time I'm counting the round trip, so let's call it at least a factor of 1,000.

Note in all this I didn't mention disks. Clearly, a delay of the order of 1000 times the minimum delay on a network is also a big factor in performance, but that's a factor without highly distributed systems. Disk latency can to some extent be

hidden by accessing large units, and by cacheing disk contents in RAM. Accessing a disk over a network doesn't significantly increase the latency, but disk bandwidth tends to be higher than network bandwidth, and less subject to contention. In that sense there is a mismatch between the two technologies. A disk works best streaming large quantities of data, but a network works best with smallish packets, not bigger than a few thousand bytes.

6.4 Warehouse Design

Although WSC uses commodity parts, these will generally be packaged into rack-mount systems for ease of maintenance. A rack can be design to use a single network switch, and packaging can be optimized to fit requirements like minimising network cabling, even distribution of power, quick identification of faulty systems and selectively replacing obsolete models.

A critical aspect of the overall design is heat dissipation. Even if the Google approach of using mid-range systems is followed, a few thousand PCs in a warehouse add up to significant heat to extract. A midrange CPU is likely to generate about 100W of heat. To allow for all components, let's take a ballpark figure of 300W (Google reports CPU use as about a third of the total energy budget [Barroso and Hölzle 2009, p 10]). If we have a warehouse of 2,500 computers, that adds up to 750kW of heat (to a good approximation; some of the electricity actually does get used for useful work). Large computer installations may use water as a heat exchange medium, potentially a significant factor in their environmental footprint. As WSC becomes an increasing component of computer services, environmental footprint will become an increasingly important issue, including energy consumption and lifecycle costs [Chang et al. 2012]. By contrast, if you have a single PC in an office or in your home, the impact of its heat dissipation on heating and air conditioning is negligible.

A typical warehouse-scale system will include a large diesel backup power supply, as well as more instantaneous backup UPS power [Barroso and Hölzle 2009].

The overall design of cooling, power supply and component positioning is very complex, and can make a big difference to life cycle costs.

6.5 Historical Perspective

It is interesting to contrast the new world of large-scale computing with previous generations. At a very early stage of the computer industry, it was a widely-held perception that computing would evolve to one giant computer meeting the

entire world's needs. As a play on the name of one of the first commercial computers, Univac I, one science fiction author (Isaac Asimov[4]) called the single world computer Multivac. In 1943, IBM CEO Thomas J Watson is alleged to have predicted a world market for about 5 computers. More recently, large-scale computing used a small-number of high-spec machines, with high power consumption, tight packaging to minimize latencies and advanced cooling needed with a machine like an early Cray drawing 130kW [Kolodzey 1981]. Fast forward to today's world, and the number of discrete computers is in the hundreds of millions, billions if you count mobile and embedded devices, but a small number of players is trying to pull large-scale computation back to central large-scale installations.

The new big in computing may be composed out of consumer parts, but it involves engineering challenges every bit as complex as those faced by architects like Seymour Cray. Energy and heat are huge problems on the scale at which they operate, and reliability is a major issue the more complex the combination of parts and the number of parts. While it doesn't appear that we are headed for something exactly like Asimov's vision of Multivac, a small number of very large players with tens of thousands of computers functioning as a single system is not as far off from that concept as where the industry was previously headed.

Why does it make sense to build systems on this scale? After all if the individual servers are essentially mid-range PCs with large disks, plus a lot of expensive infrastructure to provide them with stable power, remove heat from the building and to scale the network up to tens of thousands of nodes, why is the service superior to just running something on your own PC, which you don't need to share with anyone else, and which doesn't tax the power and thermal requirements of a home or office?

There are two major motivations for a return to highly-central services (even if the implementation is geographically distributed):

- *management* – many of the more complex management issues of running individual computers are taken over by the central service, including backups, installing new releases and managing user credentials
- *instantaneous scalability* – using services of this type makes it possible to adjust the scale of installation based on demand; you cannot realistically buy 10,000 computers for a need that happens once a year

Another major factor in the early viability of these services is that they ride on the back of requirements of their creators. Amazon has a massive requirement for highly-responsive services for their retail operations. A cost of high responsiveness

[4] `http://en.wikipedia.org/wiki/The_Last_Question`

is a lot of idle time, and selling that idle time is a way of recouping the cost (though naturally highly responsive reactions to a customer buying something from the Amazon store is still top priority). Google has the opposite scenario: a massive requirement for batch processing for their web crawlers that build their search indexes. This batch processing can fill the gaps between the demand for more interactive services that Google can make available as another way of using their massive infrastructure.

The combined effect of current trends is likely to be massive growth in WSC, with an increasing range of cloud services, because a wide range of services with different requirements allow load to be balanced. What is an interesting question is what happens when external clients' requirements exceed internal requirements in operations like Amazon and Google. Whichever WSC-based provider adjust first to the reality of putting their computation customers first is likely to dominate, just as IBM dominated the early decades of the computer industry by being the first to place a high priority on customer relations.

Exercises

Note that the standard abbreviation for byte is "B" and for bit is "b". Recall that binary prefixes have an "i" added to differentiate them from decimal multiples (e.g., Ki means 2^{10}, whereas G means 10^9).

1. With an average year of 8766 hours, how many hours of downtime does four nines of availability represent?

2. You would like to offer four nines of availability on a 2,500 server configuration. Which of the following gets you closest to this goal (starting from the base of the figures in Table 6.3, which gave us 2 nines of availability):

   (a) replacing the hard drives by solid-state drives (SSDs), reducing the expected number of failures to 10 a year

   (b) replacing the RAM with DRAMs with error checking and correction (ECC), reducing the number of uncorrectable failures to 20 per year

   (c) running a more robust operating system that reduces failure to 250 per year

   Given the above, comment on Google's actual approach, which is to tolerate failures.

3. Google is a significant investor in clean energy technologies, and Apple has reportedly commissioned one of the largest solar energy facilities no owned by a power utility. Discuss why this may be the case.

4. Use the Intel Nehalem latencies from Table 4.1, with network latency in this chapter (Table 6.4):

   (a) Assume a uniprocessor task running on a local CPU, and redo the calculation for the fraction of remote accesses that double the average memory access time, assuming global miss rates from L1 or 10%, from L2 of 1% and from L3 of 0.1%.

   (b) Now redo the calculation assuming 10% of L2 misses incur snoop latency (implying a multiprocessor task). How does this change your answer?

   (c) Adjust your answers by doubling network latency to allow for the round trip, and adding 10% to allow for network congestion. How much of a difference does this adjustment make?

   (d) Assume network latency adds for each switch. How much difference will it make if you have to go through 3 switches to obtain a data item?

   (e) In general terms, discuss the performance hit going to a remote machine rather than local accesses, even with multiprocessor overheads.

5. Gbit ethernet switches are commodity technology. Let's consider whether 10Gbit switches are worth considering for WSC. Assume switching latency is the same, and the only change is the transfer rate.

   (a) Ignoring switching latency and packet overheads, how long does it take to move a packet of 4KiB at 1Gb/s?

   (b) Ignoring switching latency and packet overheads, how long does it take to move a packet of 4KiB at 10Gb/s?

   (c) How big a difference is there in these two numbers if we add $10\mu s$ switching latency?

   (d) MapReduce operates on relatively large chunks of data. Relate switching latency to transfer time in this example, and explain why MapReduce is generally used that way.

   (e) If you were designing a new WSC facility would you consider 10Gb ethernet switches? Explain.

6. Look up the services that Apple, Amazon, Google and DropBox offer.

   (a) What are similarities and differences in implementation technologies, offerings to customers and their revenue models?

   (b) Have any of these taken the critical step of placing customer needs ahead of their own internal strategy?

7. Assume it takes 12ms to start to access data off your local disk, and a remote site with a network latency of 10ms has the data you request in memory. Your disk can transfer data at 1Gb/s, and the remote location 5Mb/s. Take into account in network accesses that you need to request the data (with a packet small enough to neglect transmission time) as well as receive the data, and assume that no packets are lost. Compare in each question the time for local and remote access.

   (a) If you want one data item 1kB in size, which is faster to access?

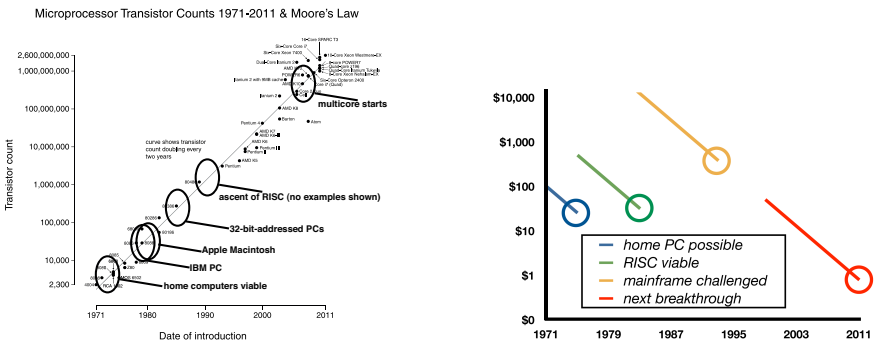
   (b) If you want to access 10 items, each 1kB in size, each of which is on a different location on your local disk, but in RAM on the other node, which is faster, assuming you can request all 10 items in one network packet?

   (c) Now repeat the previous two calculations for data items 1MB in size. Assume you only need to take into account network latency once, even if in practice the multiple packets will be sent.

7 Predicting Breakthroughs

PREDICTING TECHNOLOGY BREAKTHROUGHS is an inexact science. If you project a trend like Moore's Law, you can make fairly obvious predictions. However, predicting a major change in packaging is more difficult. Major changes since the first computers include the development of mid-sized computers, around the size of a filing cabinet, called minicomputers, the appearance of the home PC, the acceptance of the PC in business, the RISC revolution, the switch from desktop to notebook computers and the explosion of mobile devices. Buried in amongst all this is the less visible development of computers in embedded systems.

In Figure 7.1, the first part (7.1(a)) illustrates a conventional view of Moore's Law: if you hold dollars constant, how much more can you buy in the future? The second part of the figure (7.1(b)) illustrates keeping functionality constant, and predicting when it will cross a price point that makes a new form factor or market niche viable.



(a) Constant dollars.

(b) Constant Transistors.

Figure 7.1: Two views of Moore's Law.

7.1 Predicting the Past is Easy

Figure 7.1(b) is of course easy to construct with hindsight. Applying a theory to test whether something in the past is predicted is called *hindcasting*; forecasting is a rather more difficult proposition. For example, at the time when home PCs first appeared, there were many hobbyist kits around but not many people would have predicted that someone would package that technology as something appealing for the average household. The development of the first spreadsheet, VisiCalc, propelled the Apple II from a slicker than average hobbyists' toy to a useful home computer. If we look at today's world, something similar appears to be happening again. There are many inexpensive kit computers based on cell phone parts, often for $100 or less. One of these, the Raspberry Pi [RaspberryPi 2012], illustrated in Figure 7.2, captured the public imagination on its launch, and has sold in large numbers. Compared with the computer that was most successful in the original home PC market, the Raspberry Pi is a very capable system, capable of booting Linux and running a full range of programming languages. What no one can reasonably predict at this stage is whether these cell phone technology-based ultra-cheap computers represent a new general platform, or a sophisticated toy. As with the Apple II, the key is software. If someone develops a killer app for this class of computer, it will take off as a mass-market niche. At launch, there is nothing obvious. Many standard applications like Open Office are too large for the available memory for this class of computer, and slightly less complex applications like web browsers barely run on it. Bet that as it may, it's early days and something may yet emerge. A machine of this class with 256MiB-1GiB of RAM, 1GB or more of flash, ethernet and a 700MHz-1GHz ARM processor is certainly fast enough to run software that was commonplace 20 years ago.

On the other hand, sometimes it's easier to predict where a breakthrough is going. In the early 1990s, I told a computer centre manager that mainframes were in trouble because RISC processors were improving so much faster. He didn't see how this could be possible, and even after IBM made a record loss [Burgess 1993], it took him a few years to switch to RISC-based servers. At that time, prediction was easy because RISC processors ran versions of UNIX, and hence could perform functions traditionally associated with HPC machines. There was no question of a "killer app" to change the game space.

7.2 Limits to Trends

There's a limit to how far it's reasonable to extrapolate trends across major technology changes. For example, in the era of the CDC6600 design [Thornton 1963], memory was relatively fast compared with CPUs, so large caches weren't

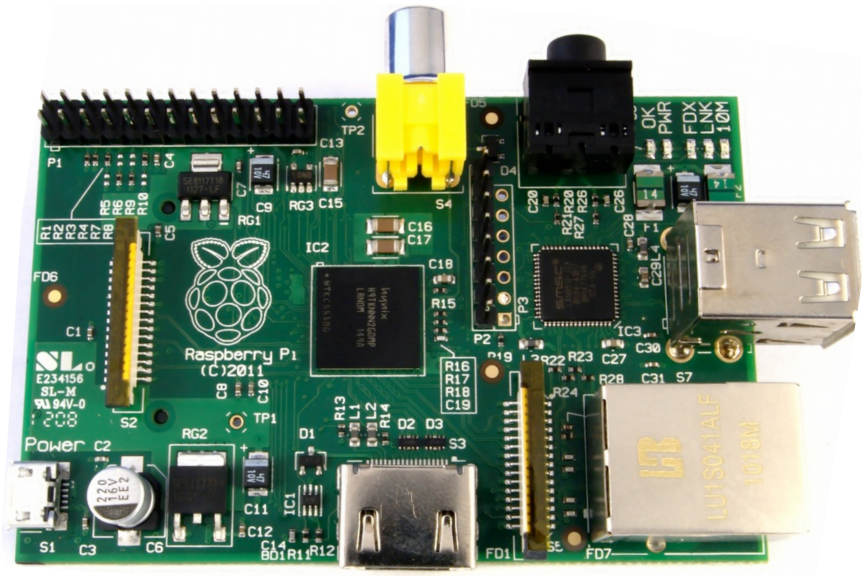


Figure 7.2: Raspberry Pi. *Based on cell phone parts, it has a 700MHz ARM processor, 256MiB of RAM, boots off SD flash and has ethernet, USB and several lower-level interfaces.*

necessary. Since the 1990s, the CPU-DRAM speed gap has increasingly become a factor [Wulf and McKee 1995], meaning the number of transistors in aggressive designs increasingly goes into caches. More recently, the difficulties in scaling up superscalar pipeline performance has led to multicore designs [Olukotun et al. 1996; Geer 2005] becoming mainstream.

How much longer will Moore's Law hold good? One 1997 paper has the Law as ending as early as 2003, with estimates of the end point varying up to 2010 [Schaller 1997]. A 2002 physics paper indicates that features of less than 40nm should run in to problems no later than 2012 [Kish 2002]. A more recent paper [Thompson and Parthasarathy 2006] projects another 30 years of advances. Obviously the earlier predictions are not correct; there is active research on technologies to design for lower feature sizes so [Lin 2012] we still have a few generations to go.

More critical than limits on how much we can shrink components is how we can use more components. In the 1990s the mainstream approach was to increase instruction-level parallelism, but there are limits to ILP [Wall 1991; Lam and Wilson 1992; Postiff et al. 1998] as well as to the complexity that is reasonable to implement as a single logical device. As the circuit complexity grows, wiring effects like capacitance and wiring delays dominate [Agarwal et al. 2000]. Some of these limitations inspired the move to multicore (originally chip multiprocessor

[Olukotun et al. 1996]).

Another limitation on trends is the fact that components do not all improve at the same rate. Historically, DRAM has improved in density by a factor of 4 every 3 years; selling more DRAM for the same money has been the focus, not speed improvement. As a result, the CPU-DRAM speed gap has been growing, also limiting the value of not only more ILP but faster clock speeds. To some extent this effect is countered by going to multicore designs with slower individual cores, though the aggregate effect of memory demand for a dual-core 2GHz design is not necessarily less than a single-core 4GHz design, depending on the nature of the workload. Disk latency too has not scaled as fast as CPU speed.

Looking at a single trend in isolation therefore is risky: you need to look at all factors playing into overall system performance (remember Amdahl?).

7.3 Really New Stuff

How do you find out about potential technologies that are not simple extrapolations from existing designs? One good place to look is the The International Technology Roadmap for Semiconductors (ITRS[1]). ITRS is a very comprehensive survey of technologies including those still in the lab and far from ready for production.

That's not to say that predictions will happen: for a long time, gallium arsenide was predicted to be the successor to silicon for semiconductors, but, other than some exotic designs at the high end of the market, that never happened [Spinardi 2012].

Nonetheless looking at trends and where they break is useful. In the late 1990s, when predicting the memory wall was big, I looked to the past and that led to my RAMpage project (see p 3).

It's also useful to examine future technologies from sources like ITRS: that's where the University of Michigan Picoserver (see p 4) originated [Kgil et al. 2006].

One thing always to keep in mind is that the latest fast, hot technology often gets all the PR, while something small and apparently insignificant lurking behind the scenes is more likely to be the future trendsetter. The PC started out with very modest technology, that was not initially even intended for making a whole computer. RISC designs started out as a better way of making a UNIX workstation, and all-but wiped out the venerable mainframe market.

Understand history, and you have some chance of being among the leaders in designing or adopting a new technology.

[1] `http://www.itrs.net`

Exercises

1. Pick a technology that's commonplace now but expensive, and a price point at which you think it could define a new market. If Moore's Law continues unabated, when will your technology hit the target price point? If you pick a technology that doesn't rely on Moore's Law, which relates to transistors, can you find another trend to use?

2. Assume the more optimistic estimates are correct, and Moore's Law continues for 30 years. That implies an improvement in the number of transistors you can buy of a factor of $2^{15} = 32768$. What class of device really needs that level of improvement? Consider both buying more speed at today's price, and dropping the price of a fixed number of transistors.

3. Assume Moore's Law no longer holds, and we cannot make single chips any faster (or cheaper by making a new version with a smaller die and fixed functionality).

   (a) How would you go about building a faster computer than a current model?

   (b) What would you have to do to build a computer with the same functions as an existing model but at lower cost?

4. What would have to change before a Raspberry Pi-type machine became a mainstream home or work computer (aside from cosmetic improvements like a nice case)?

5. Look up ITRS and see if you can pick any exciting new technology that could be a game changer.

6. Look up why gallium arsenide didn't make it to the mainstream.

References

Aasaraai, K. and Moshovos, A. (2010). An efficient non-blocking data cache for soft processors. In *2010 International Conf. on Reconfigurable Computing and FPGAs (ReConFig)*, pages 19 –24.

Agarwal, V., Hrishikesh, M. S., Keckler, S. W., and Burger, D. (2000). Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. 27th annual int. symp. on Computer architecture (ISCA'00)*, ISCA '00, pages 248–259, New York, NY, USA. ACM.

Akeley, K. (1993). Reality Engine graphics. In *Proc. 20th annual conf. on Computer graphics and interactive techniques (SIGGRAPH'93)*, SIGGRAPH '93, pages 109–116, New York, NY, USA. ACM.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. Spring joint computer conference*, AFIPS '67 (Spring), pages 483–485.

Amdahl, G. M., Blaauw, G. A., and Brooks, F. P. (1964). Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87 –101.

Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. (1991). The interaction of architecture and operating system design. *SIGARCH Comput. Archit. News*, 19(2):108–120.

Apple (2012). About the virtual memory system. Online: `https://developer.apple.com/library/mac/#documentation/performance/conceptual/managingmemory/articles/aboutmemory.html` last accessed 6 July 2012.

Archibald, J. and Baer, J.-L. (1986). Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298.

Arvind, K. and Nikhil, R. S. (1990). Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318.

Barroso, L., Dean, J., and Holzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28.

Barroso, L. A. and Hölzle, U. (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool. online `http://www.morganclaypool.com/doi/pdf/10.2200/S00193ED1V01Y200905CAC006`.

Belayneh, S. and Kaeli, D. (1996). A discussion of non-blocking/lockup-free caches. *Computer Architecture News*, 24(3):18–25.

Bell, G. (2008). Bell's law for the birth and death of computer classes. *Commun. ACM*, 51(1):86–94.

Bennet, J., Carter, J., and Zwaenepoel, W. (1990). Adaptive software cache management for distributed shared memory architectures. In *Proc. 17th Int. Symp. on Computer Architecture (ISCA '90)*, pages 125–134, Seattle, WA.

Berrendorf, R., Burg, H. C., Detert, U., Esser, R., Gerndt, M., and Knecht, R. (1994). Intel Paragon XP/S – architecture, software environment, and performance. Technical Report KFA-ZAM-IB-9409, Jülich Supercomputing Centre, Jülich, Germany.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

Binkert, N. L., Hallnor, E. G., and Reinhardt, S. K. (2003). Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 36–43.

Bordawekar, R. R. (2000). Quantitative characterization and analysis of the I/O behavior of a commercial distributed-shared-memory machine. *IEEE Trans. on parallel and distributed systems*, 11(5):509–526.

Borg, A., Kessler, R., and Wall, D. (1990). Generation and analysis of very long address traces. In *Proc. 17th Int. Symp. on Computer Architecture (ISCA '90)*, pages 270–279.

Bricklin, D. and Frankston, B. (1979). *VisiCalc Computer Software Program for the Apple II and II Plus*. Personal Software, Inc, Sunnyvale, CA.

Burgess, J. (1993). IBM's $5 billion loss highest in American corporate history. *The Tech*, 112(66):3.

Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C. (2006). Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proc. 33rd Int. Symp. on Computer Architecture*, pages 227–238, Boston.

chan Kang, S., Nicopoulos, C., Lee, H., and Kim, J. (2011). A high-performance and energy-efficient virtually tagged stack cache architecture for multi-core environments. In *Proc. IEEE 13th Int. Conf. on High Performance Computing and Communications (HPCC)*, pages 58 –67.

Chang, J., Meza, J., Ranganathan, P., Shah, A., Shih, R., and Bash, C. (2012). Totally green: evaluating and designing servers for lifecycle environmental impact. In *Proc. 17th int. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, ASPLOS '12, pages 25–36, New York, NY, USA. ACM.

Chang, L.-P. (2007). On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1126–1130, New York, NY, USA. ACM.

Chatterjee, D., DeOrio, A., and Bertacco, V. (2009). GCS: High-performance gate-level simulation with GPGPUs. In *Design, Automation Test in Europe Conference Exhibition DATE '09.*, pages 1332–1337.

Chen, T. and Baer, J. (1992). Reducing memory latency via non-blocking and prefetching caches. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pages 51–61.

Cheriton, D., Goosen, H., and Boyle, P. (1989). Multi-level shared caching techniques for scalability VMP-MC. In *Proc. 16th Int. Symp. on Computer Architecture (ISCA '89)*, pages 16–24, Jerusalem.

Cheriton, D., Goosen, H., Holbrook, H., and Machanick, P. (1993). Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: The value of distributed synchronization. In *Proc. 7th Workshop on Parallel and Distributed Simulation*, pages 159–162, San Diego.

Cheriton, D., Goosen, H., and Machanick, P. (1991). Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: A first experience. In *Proc. Int. Symp. on Shared Memory Multiprocessing*, pages 109–118, Tokyo.

Cheriton, D., Gupta, A., Boyle, P., and Goosen, H. (1988). The VMP multiprocessor: Initial experience, refinements and performance evaluation. In

Proc. 15th Int. Symp. on Computer Architecture (ISCA '88), pages 410–421, Honolulu.

Cheriton, D., Slavenburg, G., and Boyle, P. (1986). Software-controlled caches in the VMP multiprocessor. In *Proc. 13th Int. Symp. on Computer Architecture (ISCA '86)*, pages 366–374, Tokyo.

Cheung, T. and Smith, J. (1986). A simulation study of the CRAY X-MP memory system. *IEEE Transactions on computers*, C-35(7):613–622.

Chinchilla, F., Gamblin, T., Sommervoll, M., and Prins, J. F. (2004). Parallel N-body simulation using GPUs. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill. `http://wwwx.cs.unc.edu/~tgamblin/gpgpu/GPGPfinalReport.pdf`.

Cmelik, B. and Keppel, D. (1994). Shade: a fast instruction-set simulator for execution profiling. *SIGMETRICS Perform. Eval. Rev.*, 22(1):128–137.

Colwell, R. and Steck, R. (1995). A 0.6μ BiCMOS processor with dynamic execution. In *Proc. 42nd IEEE Int. Conf. on Solid-State Circuits (ISSCC)*, pages 176–177, 361.

Colwell, R. P., Gehringer, E. F., and Jensen, E. D. (1988). Performance effects of architectural complexity in the Intel 432. *ACM Trans. Comput. Syst.*, 6(3):296–339.

Colwell, R. P., Hall, W. E., Joshi, C. S., Papworth, D. B., Rodman, P. K., and Tornes, J. E. (1990). Architecture and implementation of a VLIW supercomputer. In *Proc. 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 910–919, Los Alamitos, CA, USA. IEEE Computer Society Press.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Denning, P. J. (1968). The working set model for program behavior. *Commun. ACM*, 11(5):323–333.

Diefendorff, K., Dubey, P., Hochsprung, R., and Scale, H. (2000). AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85 – 95.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., and Dongarra, J. (2012). From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407.

Dunigan, T.H., J., Vetter, J., White, J.B., I., and Worley, P. (2005). Performance evaluation of the Cray X1 distributed shared-memory architecture. *IEEE Micro*, 25(1):30 – 40.

Dwarkadas, S., Keleher, P., Cox, A., and Zwaenepoel, W. (1993). Release consistent software distributed shared memory on emerging network technology. In *Proc. 20th Int. Symp. on Computer Architecture (ISCA '93)*, pages 144–155, San Diego, CA.

Engblom, J. and Ermedahl, A. (1999). Pipeline timing analysis using a trace-driven simulator. In *Proc. Sixth Int. Conf, on Real-Time Computing Systems and Applications (RTCSA)*, pages 88–95.

Ferlin, E. P., Lopes, H. S., Lima, C. R. E., and Perretto, M. (2011). Prada; a high-performance reconfigurable parallel architecture based on the dataflow model. *Int. J. High Perform. Syst. Archit.*, 3(1):41–55.

Fick, D., Dreslinski, R., Giridhar, B., Kim, G., Seo, S., Fojtik, M., Satpathy, S., Lee, Y., Kim, D., Liu, N., Wieckowski, M., Chen, G., Mudge, T., Sylvester, D., and Blaauw, D. (2012). Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores. In *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, pages 190–192.

Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S. (2008). Intel® AVX: New frontiers in performance improvements and energy efficiency. Technical report, Intel. `http://toolbox-dzada.googlecode.com/svn/trunk/docs/simd/Intel+AVX_New+Frontiers+in+Performance+Improvements+and+Energy+Efficiency_WP.pdf`.

Freescale (1999). *AltiVec Technology Programming Environments Manual*. Freescale Semiconductor. online `http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf` accessed 7 August 2012.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

Geer, D. (2005). Chip makers turn to multicore processors. *Computer*, 38(5):11–13.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Commun. ACM*, 35(2):97–107.

Ghosal, D. and Bhuyan, L. N. (1990). Performance evaluation of a dataflow architecture. *IEEE Trans. Comput.*, 39(5):615–627.

Gifford, D. and Spector, A. (1987). Case study: IBM's system/360-370 architecture. *Comm. ACM*, 30(4):291–307.

Grimes, J., Kohn, L., and Bharadhwaj, R. (1989). The Intel i860 64-bit processor: A general-purpose CPU with 3D graphics capabilities. *IEEE Comput. Graph. Appl.*, 9(4):85–94.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., and Brown, R. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE Int. Workshop on Workload Characterization, 2001 (WWC-4)*, pages 3–14.

Hallnor, E. G. and Reinhardt, S. K. (2000). A fully associative software-managed cache design. In *Proc. 27th Ann. Int. Symp. on Computer Architecture*, pages 107–116, Vancouver, BC.

Harrell, C. B. and Fouladi, F. (1993). Graphics rendering architecture for a high performance desktop workstation. In *Proc. 20th Ann. Conf on Computer graphics and Interactive Techniques*, pages 93–100.

Hennessy, J. and Patterson, D. (1990). *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann, San Francisco, CA, 1st edition.

Hennessy, J. and Patterson, D. (2012). *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann, San Francisco, CA, 5th edition.

Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

Hiraki, K., Tamatsukuri, J., and Matsumoto, T. (1998). Speculative execution model with duplication. In *Proc. 1998 Int. Conf. on Supercomputing*, pages 321–328, Melbourne, Australia.

Inouye, J., Konuru, R., Walpole, J., and Sears, B. (1992). The effects of virtually addressed caches on virtual memory design and performance. Technical Report CS/E 92-010, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Engineering.

Intel (2009). Intel® advanced vector extensions programming reference. Technical Report 319433-006, Inte.

Jacob, B. L. and Mudge, T. N. (1998). A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 295–306, San Jose, CA.

Jin, D. and Caesar, D. N. M. (2010). Efficient gigabit ethernet switch models for large-scale simulation. In *2010 IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 1–10.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34.

Kaeli, D. and Emma, P. (1997). Improving the accuracy of history-based branch prediction. *IEEE Trans. on Computers*, 46(4):469–472.

Kalla, R., Sinharoy, B., and Tendler, J. (2004). IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47.

Keltcher, C., McGrath, K., Ahmed, A., and Conway, P. (2003). The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66 – 76.

Kgil, T., D'Souza, S., Saidi, A., Binkert, N., Dreslinski, R., Reinhardt, S., Flautner, K., and Mudge, T. (2006). PicoServer: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, San Jose, CA.

Kish, L. B. (2002). End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3–4):144–149".

Kolodzey, J. (1981). Cray-1 computer technology. *IEEE Trans. on Components, Hybrids, and Manufacturing Technology*, 4(2):181 – 186.

Krakiwsky, S., Turner, L., and Okoniewski, M. (2004). Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). In *Proc. IEEE MTT-S Int. Microwave Symp.*, volume 2, pages 1033–1036 Vol.2.

Krishnan, V. and Torrellas, J. (1999). A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. on Computers*, 48(9):866–880.

Lam, M., Rothberg, E., and Wolf, M. (1991). The cache performance and optimizations of blocked algorithms. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-4)*, pages 63–74, Santa Clara, CA.

Lam, M. S. and Wilson, R. P. (1992). Limits of control flow on parallelism. In *Proc. 19th Ann. Int. Symp. on Computer Architecture*, pages 46–57, Queensland, Australia.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Lavington, S. H. (1978). The Manchester Mark I and Atlas: a historical perspective. *Commun. ACM*, 21(1):4–12.

Lee, D., Baer, J.-L., Calder, B., and Grunwald, D. (1995). Instruction cache fetch policies for speculative execution. In *Proc. 22nd Ann. Int. Symp. on Computer Architecture*, pages 357–367, S. Margherita Ligure, Italy.

Lee, J., Kim, J., Jang, C., Kim, S., Egger, B., Kim, K., and Han, S. (2008). FaCSim: a fast and cycle-accurate architecture simulator for embedded systems. In *Proc. 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 89–100, New York, NY, USA. ACM.

Lieverse, P., Deprettere, E. F., Kienhuis, A. C. J., and De Kock, E. A. (1999). A clustering approach to explore grain-sizes in the definitionof processing elements in dataflow architectures. *J. VLSI Signal Process. Syst.*, 22(1):9–20.

Lin, B. J. (2012). Lithography till the end of Moore's law. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, ISPD '12, pages 1–2, New York, NY, USA. ACM.

Liu, H. and Wee, S. (2009). Web server farm in the cloud: Performance evaluation and dynamic architecture. In Jaatun, M., Zhao, G., and Rong, C., editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 369–380. Springer Berlin / Heidelberg.

Lombide Carreton, A. and D'Hondt, T. (2010). A hybrid visual dataflow language for coordination in mobile ad hoc networks. In *Proc. 12th int. conf. on Coordination Models and Languages (COORDINATION'10)*, COORDINATION'10, pages 76–91, Berlin, Heidelberg. Springer-Verlag.

Lomont, C. (2011). Introduction to Intel® Advanced Vector Extensions. Technical report, Intel.

Macedonia, M. (2004). The digital world's midlife crisis. *Computer*, 37(8):100 – 101.

Machanick, P. (1996). *An Object-Oriented Library for Shared-Memory Parallel Simulations*. PhD Thesis, Department of Computer Science, University of Cape Town.

Machanick, P. (2000). Scalability of the RAMpage memory hierarchy. *South African Computer J.*, (25):68–73.

Machanick, P. (2004). Initial Experiences with Dreamy Memory and the RAMpage Memory Hierarchy. In *Proc. Ninth Asia-Pacific Computer Systems Architecture Conf.*, pages 146–159, Beijing.

Machanick, P. and Salverda, P. (1998). Preliminary investigation of the RAMpage memory hierarchy. *South African Computer J.*, (21):16–25.

Machanick, P., Salverda, P., and Pompe, L. (1998). Hardware-software trade-offs in a Direct Rambus implementation of the RAMpage memory hierarchy. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 105–114, San Jose, CA.

Markatos, E., Crovella, M., Das, P., Dubnicki, C., and LeBlanc, T. (1991). The effects of multiprogramming on barrier synchronization. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, pages 662–669.

Martínez, J. F. and Torrellas, J. (2002). Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proc. 10th Int. Conf. on Architectural support for programming languages and operating systems*, pages 18–29, San Jose, CA.

Mayer, A. J. W. (1982). The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10.

Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems (TOCS)*, 9(1):21–65.

Meredith, J. S., Alvarez, G., Maier, T. A., Schulthess, T. C., and Vetter, J. S. (2009). Accuracy and performance of graphics processors: A quantum Monte Carlo application case study. *Parallel Computing*, 35(3):151–163.

Mironov, D., Ubar, R., Devadze, S., Raik, J., and Jutman, A. (2010). Structurally synthesized multiple input BDDs for speeding up logic-level simulation of digital circuits. In *13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 658 –663.

Molka, D., Hackenberg, D., Schone, R., and Muller, M. (2009). Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proc. 18th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 261–270.

Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. (1997). InfiniteReality: a real-time graphics system. In *Proc. 24th annual conf. on Computer graphics and interactive techniques (SIGGRAPH '97)*, SIGGRAPH '97, pages 293–302, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117.

Moudgill, M., Wellman, J.-D., and Moreno, J. (1999). Environment for PowerPC microarchitecture exploration. *IEEE Micro*, 19(3):15 –25.

Nambiar, R., Wakou, N., Carman, F., and Majdalany, M. (2011). Transaction processing performance council (TPC): State of the council 2010. In Nambiar, R. and Poess, M., editors, *Performance Evaluation, Measurement and Characterization of Complex Systems*, volume 6417 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg.

Nayfeh, B. A. and Olukotun, K. (1994). Exploring the design space for a shared-cache multiprocessor. In *ISCA '94: Proc. 21st Ann. Int. Symp. on Computer Architecture*, pages 166–175, Chicago, Illinois, United States.

NVIDIA (2011). PTX: Parallel thread execution ISA version 2.3.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. (1996). The case for a single-chip multiprocessor. In *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 2–11, Cambridge, MA.

Organick, E. I. (1983). *A programmer's view of the Intel 432 system*. McGraw-Hill, Inc., New York, NY, USA.

Papworth, D. (1996). Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15.

Patterson, D. A. and Ditzel, D. R. (1980). The case for the reduced instruction set computer. *Computer Architecture News*, 8(6):25–33.

Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, New York, NY, USA. ACM.

Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel MMX for multimedia pcs. *Commun. ACM*, 40(1):24–38.

Perleberg, C. and Smith, A. (1993). Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412.

Petersen, A., Putnam, A., Mercaldi, M., Schwerin, A., Eggers, S., Swanson, S., and Oskin, M. (2006). Reducing control overhead in dataflow architectures. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 182–191, New York, NY, USA. ACM.

Piguet, C. (2006). Ultra-low-power processor design. In Oklobdzija, V. G. and Krishnamurthy, R. K., editors, *High-Performance Energy-Efficient Microprocessor Design*, Integrated Circuits and Systems, pages 1–30. Springer US. DOI:10.1007/978-0-387-34047-0_1.

Poess, M., Nambiar, R. O., Vaid, K., Stephens, Jr., J. M., Huppler, K., and Haines, E. (2010). Energy benchmarks: a detailed analysis. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 131–140, New York, NY, USA. ACM.

Postiff, M. A., Green, D. A., Tyson, G. S., and Mudge, T. N. (1998). Limits of instruction level parallelism in SPEC95 applications. In *INTERACT-3 Workshop on Interaction between Compilers and Computer Architectures, part of ASPLOS VIII*, pages 31–34, San Jose, CA.

Rahman, N. and Raman, R. (2000). Analysing cache effects in distribution sorting. *J. of Experimental Algorithmics (JEA)*, 5:14.

RaspberryPi (2012). Raspberry Pi FAQs. `http://www.raspberrypi.org/faqs`.

Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). PIN: a binary instrumentation tool for computer architecture research and education. In *Proc. 2004 workshop on Computer architecture education: held in conjunction with the 31st Int. Symp. on Computer Architecture*, WCAE '04, New York, NY, USA. ACM.

Rogers, A. and Li, K. (1992). Software support for speculative loads. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pages 38–50.

Russinovich, M. (2007). Inside the Windows Vista kernel: Part 2. *TechNet Magazine*.

Schaller, R. (1997). Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52 –59.

Schoeberl, M. (2008). A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1–2):265 – 286.

Schoeberl, M., Preußer, T. B., and Uhrig, S. (2010). The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 120–127, New York, NY, USA. ACM.

Seznec, A. and Lenfant, J. (1992). Interleaved parallel schemes: improving memory throughput on supercomputers. In *Proc. 19th annual int. symp. on Computer architecture (ISCA '92)*, ISCA '92, pages 246–255, New York, NY, USA. ACM.

Silva, J. L. E. and Lopes, J. J. (2010). A dynamic dataflow architecture using partial reconfigurable hardware as an option for multiple cores. *W. Trans. on Comp.*, 9(5):429–444.

Simunic, T., Benini, L., and De Micheli, G. (1999). Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. 36th Design Automation Conference*, pages 867–872.

Skadron, K., Ahuja, P. S., Martonosi, M., and Clark, D. W. (1999). Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Trans. on Computers*, 48(11):1260–1281.

Sohi, G. (2001). Microprocessors – 10 years back, 10 years ahead. In Wilhelm, R., editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 209–218. Springer Berlin / Heidelberg.

Spinardi, G. (2012). Road-mapping, disruptive technology, and semiconductor innovation: the case of gallium arsenide development in the uk. *Technology Analysis & Strategic Management*, 24(3):239–251.

Stone, J., Gohara, D., and Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73.

Swanson, S., Putnam, A., Mercaldi, M., Michelson, K., Petersen, A., Schwerin, A., Oskin, M., and Eggers, S. J. (2006). Area-performance trade-offs in tiled dataflow architectures. In *ISCA'06: Proc. 33rd Int. Symp. on Computer Architecture*, pages 314–326, Boston.

Tanenbaum, A. S. and van Steen, M. (2002). *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ.

Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B. (2002). POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25.

Thompson, S. E. and Parthasarathy, S. (2006). Moore's law: the future of Si microelectronics. *Materials Today*, 9(6):20–25.

Thornton, J. E. (1963). Considerations in computer design – leading to the Control Data 6600. Technical report, Control Data Corp. `http://archive.computerhistory.org/resources/text/CDC/CDC.6600.1963.102641207.pdf`.

Thornton, J. E. (1980). The CDC 6600 project. *Annals of the History of Computing*, 2(4):338 –348.

Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Research and Development*, 11(1):25–33.

Traub, K. R. (1986). A compiler for the mit tagged-token dataflow architecture. Technical report, MIT, Cambridge, MA, USA. `http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Amitai%3AMIT-LCS%2F%2FMIT%2FLCS%2FTR-370`.

Tyson, G. S. (1994). The effects of predicated execution on branch prediction. In *Proc. 27th Ann. Int. Symp. on Microarchitecture*, pages 196–206, San Jose, CA.

Uhlig, R. A. and Mudge, T. N. (1997). Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170.

Valero, M., Lang, T., and Ayguadé, E. (1992). Conflict-free access of vectors with power-of-two strides. In *Proc. 6th int. conf. on Supercomputing (ICS '92)*, ICS '92, pages 149–156, New York, NY, USA. ACM.

Vo, H. T. (2011). *Designing a parallel dataflow architecture for streaming large-scale visualization on heterogeneous platforms*. PhD thesis, University of Utah, Salt Lake City, UT, USA. AAI3454865.

Voigt, S., Baesler, M., and Teufel, T. (2010). Dynamically reconfigurable dataflow architecture for high-performance digital signal processing. *J. Syst. Archit.*, 56(11):561–576.

Wall, D. (1991). Limits of instruction level parallelism. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-4)*, pages 176–188, Santa Clara, CA.

Weiss, S. (1989). An aperiodic storage scheme to reduce memory conflicts in vector processors. In *Proc. 16th annual int. symp. on Computer architecture (ISCA '89)*, ISCA '89, pages 380–386, New York, NY, USA. ACM.

Wheeler, B. and Bershad, B. (1992). Consistency management for virtually indexed caches. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pages 124–136.

Whitehead, N. and Fit-Florea, A. (2011). Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA white paper, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Floating_Point_on_NVIDIA_GPU_White_Paper.pdf`.

Womble, D. E., Dosanjh, S. S., Hendrickson, B., Heroux, M. A., Plimpton, S. J., Tomkins, J. L., and Greenberg, D. S. (1999). Massively parallel computing: A sandia perspective. *Parallel Computing*, 25(13–14):1853–1876.

Wulf, W. and McKee, S. (1995). Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24.

Wynters, E. (2011). Parallel processing on NVIDIA graphics processing units using CUDA. *J. Comput. Sci. Coll.*, 26(3):58–66.

Xiao, L., Zhang, X., and Kubricht, S. A. (2000). Improving memory performance of sorting algorithms. *J. of Experimental Algorithmics (JEA)*, 5:3.

Yeap, G. C.-F. (2002). Leakage current in low standby power and high performance devices: trends and challenges. In *Proc. 2002 Int. Symp. on Physical design*, pages 22–27, San Diego, CA.

Yeh, T.-Y. and Patt, Y. N. (1991). Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA. ACM.

Yeh, T.-Y. and Patt, Y. N. (1992). Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Ann. Int. Symp. on Computer Architecture*, pages 124–134.

Yeh, T.-Y. and Patt, Y. N. (1993). A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. 20th Ann. Int. Symp. on Computer Architecture*, pages 257–266, San Diego, CA.

Young, C. and Smith, M. D. (1999). Static correlated branch prediction. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 21(5):1028–1075.

Zivkov, B., Ferguson, B., and Gupta, M. (1994). R4200: a high-performance MIPS microprocessor for portables. In *Compcon Spring '94, Digest of Papers.*, pages 18 –25.

Zukowski, M., Héman, S., Nes, N., and Boncz, P. (2006). Super-scalar RAM-CPU cache compression. In *Proc. 22nd Int. Conf. on Data Engineering (ICDE '06)*, page 59.

A Minimal Instruction Set

FORE PURPOSE OF EXAMPLES, we use a minimal RISC instruction set. This instruction set has sufficient operations to do examples of interest, and could be used in a general-purpose computer, though sometimes with more instructions than even a relatively clean RISC design like MIPS or Alpha. In this appendix, I illustrate how one can go about deriving an encoding for an instructions set. The design compromises we need to consider include placing register operands in a consistent place to simplify the ID stage of our simple 5-stage pipeline, maximising the bits available for immediate operands especially an jump instruction and in general making it easy to decide early in the pipeline how to handle all the variations in what functional units an instruction may require.

With approximately 20 basic instructions it is possible to implement a good fraction of common functionality. I add to these a few system functions: a *syscall* goes to one of a limited range of fixed locations in the operating system, and switches out of user space. For this instruction, the next instruction's location is stored in a special register, the exception program counter (EPC) that is also used to record the faulting instruction on any kind of interrupt. It is up to the operating system to save the exception PC and any other registers it may clobber (usually all of them). To return to user space, the OS must reset internal pointers to the page table, and execute a `sysret` instruction to return from system mode to user mode. The `sysret` instruction should go to the saved value of the EPC, after restoring the saved registers. I also add an atomic `swap` operation, one of several primitives that can be used to implement synchronization.

An approach to designing a real architecture is to start with such a basic instruction set and run simulations or other design studies to determine which combinations are common enough to consider introducing specialist instructions. In Figure A.1, we do not for example have inverses of comparison because these can be achieved by inverting a logical result. The 20 instructions illustrated here do not include other variations like variations on operand size. In a reasonable design for today's requirements, you would want arithmetic and logic on at least byte, 16-bit, 32-bit and 64-bit operands. For some computational requirements, you might

want 128-bit operands, and floating-point variants on arithmetic operations.

What encoding scheme could apply to this instruction set?

An important thing we need to fix is register count. Most early RISC designs had around 32 registers, though a few had significantly more. The main concern with bits allocated to registers is reducing the bits available for immediate operands, since it's unlikely that any number of bits we allocate for registers and opcode encoding will use up all 32. For example, if we increase the register count to 128, we need 7 bits per register, totalling 21 out of 32 bits, leaving 11 bits for opcodes. Given that we are starting out with 20 instructions before adding operand variations, it seems unlikely that we will need as many as 11 bits to encode

class	**type**	**instruction**	**description**
ALU	arithmetic	add	$R_d \leftarrow R_a + R_b$
		subtr	$R_d \leftarrow R_a - R_d$
		mult	$R_d \leftarrow R_a \times R_d$
		div	$R_d \leftarrow R_a \div R_d$
	bitwise logical	lshift	$R_d \leftarrow R_a << R_d$
		rshift	$R_d \leftarrow R_a >> R_d$
		lshifta	$R_d \leftarrow R_a << R_d$ with carry
		rshifta	$R_d \leftarrow R_a >> R_d$ with carry
		and	$R_d \leftarrow R_a \wedge R_d$
		or	$R_d \leftarrow R_a \vee R_d$
		xor	$R_d \leftarrow R_a \oplus R_d$
	comparison	cmpeq	$R_d \leftarrow R_a = R_d$
		cmplt	$R_d \leftarrow R_a < R_d$
control	branch	breq	$R_a = R_b$? $PC \leftarrow PC +$ *offset* $<< 2$
	unconditional	j	$PC \leftarrow R_a + R_b$
		syscall	trap to *value*
		sysret	switch out of user mode and jump
	save PC	save	$R_d \leftarrow PC + R_a$
	save exception PC	saveep	$R_d \leftarrow PC_old + R_a$
memory		load	$R_d \leftarrow mem[R_a+R_b]$
		store	$mem[R_{d_1}+R_{d_2}] \leftarrow R_s$
		swap	swap values of mem[R_a+R_b], R_d
general		nop	*do nothing*

Figure A.1: Minimal RISC instruction set. *Details such as operand size and variations like immediate operands are omitted for clarity. I also trim the options to the minimum, e.g.,* `ne` *variations are not included for comparisons and branches. A* `syscall` *instruction only has 1 form, a trap to an immediate value. The* `saveep` *instruction is used to save the PC at the point where an exception occurred so the instruction can be restarted. A* sysret *instruction allows a return to user space, and usually is followed by a jump to the saved PC (this works because the PC for the next instruction is set up before the change of state back to user space). A* `swap` *is an atomic swap of a memory location's contents and a register value, useful for implementing synchronization primitives.*

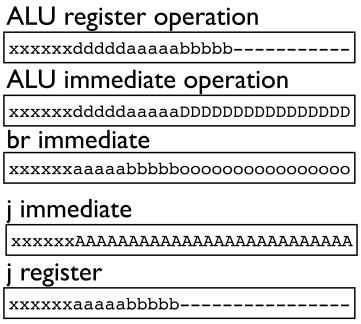


Figure A.2: Possible instruction encoding. *Opcodes always take up 6 bits, labeled with "*`x`*". Registers take up 5 bits each, labeled as "*`d`*" for a destination register, and "*`a`*" or "*`b`*" for a source register. An immediate operand is labeled according to how it's used: a signed data value with "*`D`*", an absolute address with "*`A`*" and a signed offset with "*`o`*". Unused bits are marked with "*`-`*".*

instructions (that allows $2^{11} = 2048$ variations).

Going the other way, if we generously allow ourselves 256 (=2^8) opcode encodings, we leave ourselves 24 bits for registers, meaning we could have 256 registers.

Since running out of registers is one of the key challenges of compiler writing, why not just allow for a large number of general-purpose registers like 128 or 256? There is another cost to a high register count: whenever there's a context switch, the operating system must save registers and restore those of the a restarting process. There are other considerations including the extra cost of support for hardware threads if the register file is bigger, energy use of a larger register file and the size of logic for accessing registers. A designer must balance all these issues in deciding how many registers to design into an architecture.

For purpose of simple exercises, we fix register count at 32, requiring 15 bits to encode three registers, and opcode size at 6 bits, allowing 64 different opcodes. That leaves 26 bits for an immediate address for a `j` instruction, and 16 bits for an immediate operand in an ALU operation with one source and one destination register. Figure A.2 illustrates some examples of this scheme.

Will this scheme work in practice?

Since encoding is tight, we can use a few tricks like encoding the `nop` (in less terse assembly languages, sometimes called "`noop`" for no operation) instruction using a bit pattern that is another instruction that has no effect (e.g., something in which all operands are register 0 – a trick used in the MIPS instruction set). Also note that in Figure A.1 we have 13 ALU operations, even trimming a few (such as only comparing for equal). If we allow each to have an immediate variant, that doubles the number. Adding in each of a byte, short, word and long size

multiplies the total by 4 again, making a total of $13 \times 8 = 104$ variants. However, these variations only apply to ALU operations and loads and stores, none of which need the entire instruction word. We can therefore use a basic opcode size of 6 giving us 64 instructions and encode the operand type separately. Doing this allows us to reserve as many bits as possible for instructions where a large immediate operand is desirable, particularly jumps. Another useful gain from this is that the decode stage can tell easily what the operand size is, since we are encoding that in a standard way. The final principle we need to observe is placing register operands in a consistent place so we can simplify the decode stage, where register operands are set up in our simple 5-stage pipeline. We have the following variations:

- *three registers* – destination (R_d), and two source registers (R_a and R_b); a store instruction is a bit different in that we have two registers to calculate an effective address (R_{d_1} and R_{d_2}) and a register holding the source value to store to memory (R_s); in this case, we can treat the two address operands as R_a and R_b, since they are needed in the EX stage, and as a special case use the register usually used as a destination as R_s, since there's time to reconfigure the logic to move the value of R_s to the memory system rather than set that register up to receive an ALU result

- *single source, single destination register* – as in ALU immediate operations: R_a and R_d respectively; a `save` instruction also has this format but doesn't need the bits for an immediate operand

- *two source registers* – as in a branch, where we have R_a and R_b used to compute the effective address

- *no registers* – a `j.i` and a `syscall` instruction uses all the available bits for an immediate address

If we want register operands to appear in consistent places, we need to place them so we can include them with immediate operands (where one or more registers sometimes are not needed) in a consistent way, while maximising the bits available for immediates.

All this leads to the scheme illustrated in Figure A.3. I still have a basic 6-bit opcode, but use a bit to encode whether an instruction is immediate or not, and 2 bits to encode the word size: bits that are not needed for a `j.i` instruction, so it can use all 26 bits left over from the opcode. Have I left enough bits to add floating-point instructions? If so, how would you add them? If not, what would you have to change?

Figure A.3: Refined instruction encoding. *Opcodes still take up 6 bits with the following changes in labelling compare with Figure A.2. ALU operations use two bits labeled with "*`tt`*" to represent the operand type (byte, short, word or long). A* `store.i` *instruction uses one register for the destination address labeled as "*`d`*" though it's actually the register usually labeled as "*`a`*", and a* `store` *instruction uses two registers for the destination address labeled with "*`1`*" and "*`2`*". The source register for a store is encoded in the position usually used for a destination (labeled "*`s`*")*