Appendix A
Computer Architecture Fundamentals

Abstract No book dealing with FPGAs and embedded systems would be complete without a discussion of computer architecture. To better understand the hardware-oriented security methods presented in this book, this appendix discusses the fundamental concepts of computer architecture that are applicable to FPGAs, ASICs, and CPUs.

A.1 What Do Computer Architects Do All Day?

Computer architects map applications to physical devices, and the requirements of the application dictate the design of the architecture. For some applications, a general-purpose CPU architecture is adequate, and the application can be implemented in software. However, sometimes a generic CPU is not enough, and custom hardware is required to deliver the necessary performance or other requirements. Examples of applications requiring custom hardware include networking and graphics, which have high throughput requirements. In addition, embedded systems often require custom hardware due to their resource-constrained nature. It is the computer architect's job to determine the optimal architecture for a particular application by balancing tradeoffs. Among the many tools in the computer architect's arsenal are measurement and metrics, cost-benefit analysis, simulation, standard programs called *benchmarks*, and logical analysis. Computer architects perform experiments that vary the parameters of the design. Since the list of possible experiments to try is always much larger than the available time, the computer architect must determine the most important parameters of the design and strip away all others.

Although designing custom hardware is hard work, large performance gains over general-purpose systems can be realized. Custom designs can be developed for many disciplines, including machine learning and neuroscience, biometrics, medicine [17, 18], cryptography [29], security, networks, computer vision, and program analysis. For a particular application, the computer architect performs analysis to determine the most common operations and then optimizes them. Understanding the structure of the problem makes it possible to extract parallelism from the application by performing common operations on multiple hardware units in parallel.

T. Huffmire et al., *Handbook of FPGA Design Security*,
DOI 10.1007/978-90-481-9157-4,

It is an exciting time to be a computer architect. In the words of Mark Oskin, "Seven years ago, when I started as a young assistant professor, my computer science colleagues felt computer architecture was a solved problem. Words like 'incremental' and 'narrow' were often used to describe research under way in the field.... From the perspective of the rest of computer science, architecture was a solved problem" [19] (p. 70). However, there is renewed hope for the future of the field. Techniques for improving performance that have worked in the past have run into technical "brick walls," including the "Power Wall," the "Memory Wall," and the "ILP Wall" [1]. In addition, design complexity and reliability for large, out-of-order processors presents challenges for implementation and verification. Industry has embraced chip multi-processors (CMPs) to address these problems, but developing multithreaded software that can achieve performance gains by making use of the multicore hardware has been a hard problem for decades. As researchers from the Berkeley Par Lab explain, "Industry needs help from the research community to succeed in its recent dramatic shift to parallel computing" [2] (p. 56). Rather than trying to parallelize existing scalar software code, one sensible strategy is to think about the problem from the standpoint of *what can I do with all of these cores* that cannot be done with unicore mechanisms? Making it easy for programmers to exploit multicore hardware is an open research challenge [7]. In addition to the programming problem, realistic multicore benchmarks for evaluation are also needed. Oskin concludes, "In my lifetime, this is the most exciting time for computer architecture research; indeed, people far older and wiser than me contend this is the most exciting time for architecture since the invention of the computer" [19] (p. 78).

A.2 Tradeoffs Between CPUs, FPGAs, and ASICs

Figure A.1 shows the relative generality of CPUs, FPGAs, and ASICs. There are several tradeoffs between CPUs, FPGAs, and ASICs, including software vs. hardware, generality vs. performance, cost vs. performance, and generality vs. security. CPUs run software, which is relatively inexpensive to program but comes with high overhead. ASICs are custom hardware, they have relatively high performance, and they are relatively expensive. FPGAs are more difficult to program than CPUs, but they are less expensive than ASICs. In addition, FPGAs can achieve higher throughput than CPUs but not as high as ASICs. The gap between FPGAs and ASICs is narrowing because FPGAs can be built more economically in the latest feature size, while lower-volume ASICs are sometimes fabricated with larger feature sizes, which are cheaper.

Fig. A.1 On a continuum of generality, CPUs are the most programmable, and ASICs are the least

The tradeoff between generality and security is more complex than the other tradeoffs. Attackers like generality because they can run malware on general-purpose systems. However, this doesn't mean that ASICs are free of security problems. If that were the case, *application-specific hardware* would solve the world's security problems. If a device is hard-wired to do only one thing, how can it be hacked? Sadly, this type of thinking is an example of *security through obscurity.* Application-specific devices are useful to the security community, and limiting the functionality of a system can benefit security. However, a system's security should not rest entirely upon the fact that a system has been designed to perform a limited number of functions or that the design is kept secret.

Another aspect of the tradeoff between generality and security involves the trusted foundry problem. An ASIC's intellectual property is vulnerable to theft and malicious inclusions because the design is typically sent to a third-party foundry. CPUs and FPGAs, on the other hand, are programmed after fabrication. However, one problem is replaced by another because of the risk that the hardware design may be stolen from a fielded FPGA (by circumventing the bitstream decryption mechanisms) or that the software may be stolen from a fielded general-purpose system. Stealing the design from a fielded ASIC is a much more expensive task, requiring reverse-engineering and physical sand-and-scan attacks.

A.3 Computer Architecture and Computer Science

Computer architecture spans all of computer science, and it is just as essential as theory, algorithms, languages, operating systems, networking, and security. Many of the performance gains in processors are the result of progress in algorithm development, not just the ability to fit more and more transistors on a single die. Transistors are only able to compute when they are arranged in a large-scale, correct, efficient, and fault-tolerant computer architecture. With the number of transistors on a single chip approaching and even exceeding one billion, the orchestration of such a large number of individual elements requires complex algorithms for optimization, scheduling, placement, routing, verification, branch prediction, cache replacement, instruction prefetching, control, pipelining, parallelism, concurrency, multithreading, hyperthreading, multiprocessing, speculation, simulation, profiling, coherence, and out-of-order execution, to name a few. Understanding how processors are designed and what is inside a modern processor are fundamental questions of computer architecture.

Processors are the most complicated things that humans build. Complexity is the reason that a majority of engineers at chip manufacturing companies are verification engineers, with the design engineers making up a minority. With the increasing non-recurring engineering (NRE) cost of chip manufacturing for smaller and smaller feature sizes, a mistake in the design of a chip can be fatal for even a large company. Since the number of possible states for a billion transistor design is astronomical, verification requires advanced algorithms, and its difficulty is proportional to design complexity.

A.4 Program Analysis

Computer architects study the question of what makes computers go fast. To write fast programs, it is necessary to profile what is going on inside the processor. In a modern processor, a huge number of events occur every second, on the order of one billion, and analyzing these events in real time is no small task.

A.4.1 The Science of Processor Simulation

Simulation is an essential program analysis technique, and processor simulation encompasses an entire field of study. Computer architects consume a lot of time and processor cycles running simulations. Not only is simulation useful for analyzing the performance of a processor as it runs a program, but simulating the processor is also necessary to verify its correctness. Just as an architect must create blueprints before breaking ground, a processor must be simulated prior to fabricating a chip. In between simulation and fabrication, the additional step of prototyping and testing the design on an FPGA is often taken. Simulation of the design should always be performed prior to FPGA implementation (due to the large effort required to prototype a design on an FPGA) and chip fabrication (due to the high NRE cost of fabrication).

The choice of simulation technique depends on the problem that needs to be solved. Simulation at a fine level of granularity is computationally intensive. For example, cycle-level simulation, in which events at the granularity of a single clock tick are duplicated by a software program running in a different hardware environment, is very detailed and therefore computationally intensive. Instruction-level simulation (a.k.a. *emulation*), in which events at the granularity of an instruction are represented in another processing environmet, is less expensive than cycle-level simulation but is less accurate. Emulation is useful for narrowing in on a good idea because it is fast and simple but less useful for deeper analysis because it is less accurate. For example, emulation is useful for determining the number of memory accesses and cache misses for a particular benchmark program. All simulators, both cycle-level and instruction-level, run at a much lower speed than the processors they simulate.

A key simulation parameter is how long and what to simulate. Since it is usually infeasible to simulate every event of a long-running program, it is necessary to choose a sample that represents the most pertinent behavior of the program. Taking this sample from the very beginning of the program may be a bad idea because computer programs usually begin by zeroing out arrays. Often, the critical action occurs towards the middle of the program. A sample size on the order of one hundred million instructions is common with instruction-level simulation. For cycle-level simulation, a smaller sample size is needed. Another key simulation parameter is the baseline or benchmark used for the experiment. Benchmarks are standard programs for comparing processors. The program is run by the processor being simulated.

It is extremely important to document the simulation parameters when describing the methodology of the experiment in order to understand the meaning of the results.

To study the effects of changing various aspects of the processor being simulated, computer architects modify the source code of the simulator so that the new behavior can be observed. One way of accomplishing this observation is to generate trace files and then analyze them later offline, an approach called *trace-driven* simulation. In trace-driven simulation, the simulator is modified to write specific events to a *trace file*. Trace files can get very large, on the order of gigabytes for one second of program execution, and writing to trace files slows down the simulation further. After the trace file is generated, the computer architect uses a different program to analyze the trace file off-line. For example, this program could determine the optimal cache replacement policy for a simulation that exercised different memory management strategies. Alternatively, it could determine the optimal branch prediction technique for a simulation of branch events. However, trace-driven simulation is limited by the fact that it is static, which is not useful for studying behaviors that involve significant speculative execution [30].

SimpleScalar is a suite of uni-processor simulators and tools for compilation [4]. The suite consists of *SIM–FAST*, a functional simulator that provides no timing, *SIM–OUTORDER*, a functional and timing simulator with a detailed timing model, *SIM–CACHESIM*, for simulating memory behavior, and *SIM–BPRED*, for simulating branch prediction. SimpleScalar can run programs with different ISAs, where the *MACHINE.DEF* file specifies the ISA, which can be Pisa, Alpha, Arm, or x86. Pisa is a made-up architecture based on MIPS, and a specific version of gcc will produce Pisa code. A real compiler and a set of binaries is available for Alpha. Arm is primarily used in embedded systems, and x86 is ubiquitous. *SIM–MAIN* is the main simulator loop.

A.4.2 On-Chip Profiling Engines

Because simulation typically slows down execution by at least one order of magnitude, capturing and analyzing events in real-time using on-chip profiling hardware is very useful for computer architecture research. An on-chip profiling module can capture and analyze events in real time and at high bandwidth without the need for dumping every event to a trace file on the hard disk. Although some processors have profiling features such as special profiling registers, more full-featured profiling engines desired by computer architecture researchers are rare because of economic priorities. Chip manufacturers have their hands full verifying the processor design itself without the extra task of designing and verifying a profiling module, and most end users will never use the profiling module, not being computer architecture researchers.

To overcome this problem, one option is to employ a co-processor for off-loading computationally-intensive analysis tasks. The co-processor can be a chip on the

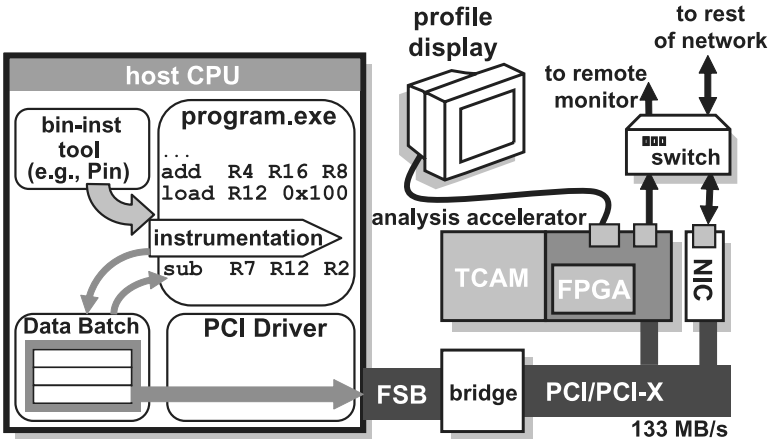

Fig. A.2 Online program analysis architecture. Computationally-intensive program analysis tasks can be off-loaded to an FPGA co-processor

same board as the main processor, or it can be an FPGA board that is connected to the Peripheral Component Interconnect (PCI) interface of the motherboard, as shown in Fig. A.2. As the processor executes the program, software instrumentation gathers profile data, which is batched in a buffer. Rather than performing computationally-intensive analysis of the profile data in software on the host CPU, the data is then written to the PCI driver. Next, the FPGA analysis module processes the profile data, and the output is used by an optimizer, a human operator, a display, or a remote monitoring unit. Clearly, this online program analysis architecture has lower bandwidth than an on-chip profiling module. An approach that can achieve higher bandwidth applies 3-D integration, an established technology in which a commodity integrated circuit is enhanced with a separate chip after fabrication [15, 16]. This additional integrated circuit contains a profiling module for analyzing events on the commodity chip. The application of 3-D Integration to security has also been proposed [14]. In this approach, specialized security functions reside in one IC, called the *3-D control plane*, which monitors and enforces a security policy on the commodity IC, called the *computation plane*.

A.4.3 Binary Instrumentation

Binary instrumentation is another useful program analysis technique [12] and is used by itself or, for example, as a component of on-chip profiling engines, as shown in Fig. A.2. An unmodified binary can be instrumented with calls to custom functions in response to specific events. These software functions can perform analysis of the event or simply write specific details of the event to a trace file. Like sim-

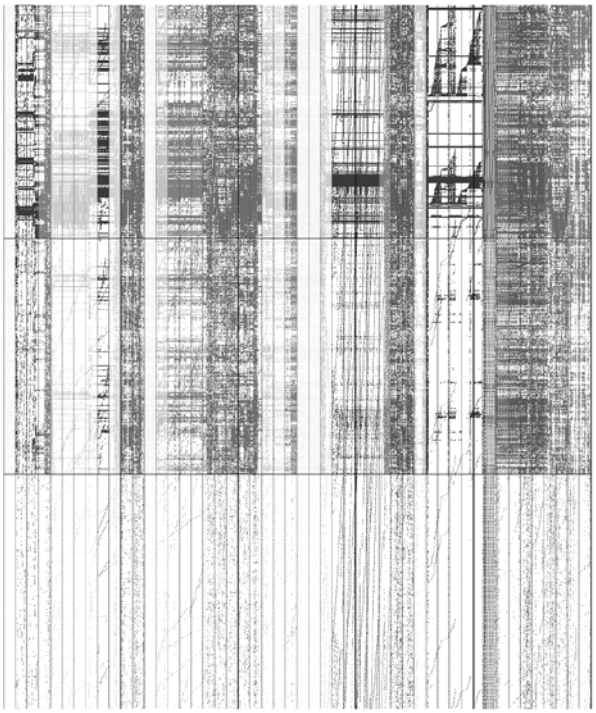

Fig. A.3 A plot of the cache behavior of Firefox over fifty million instructions as it loads a web page. The x-axis is time, and the y-axis is a function of the address of the memory access. Each *vertical slice*, or interval, represents one million instructions. The *top band* shows L1 cache hits, the *middle band* shows L1 cache misses, and the *bottom band* shows L2 misses. Intervals are colored according to the wavelet-based phase classification algorithm [13]

ulation, there is at least one order of magnitude slowdown, and writing to trace files slows things down even further. However, binary instrumentation provides the opportunity to study the complex, multi-threaded behavior of applications like web browsers, word processors, and graphics editing programs. Specific features of these applications can be invoked by interacting with the user interface (e.g., invoking menu commands, dialog boxes, etc.)

A.4.4 Phase Classification

Phase analysis is another useful tool in the computer architect's arsenal [21, 22]. Computer programs exhibit repeating behaviors over the course of their execution. Identifying these phases, which are time intervals that share similar behavior, provides several opportunities for guiding run-time optimizations and reducing simulation time [8, 9, 20, 23]. Phase classification works by counting the frequency each

basic block[1] is executed during each time interval. A Basic Block Vector (BBV) is simply an array with one entry for each basic block that stores the frequency that each basic block is executed, weighted by the number of instructions in the basic block and normalized by the total number of basic blocks executed during the interval. A similarity metric called *Manhattan Distance* is used to compare BBVs, and it is the sum of the absolute value of the difference between each element of two BBVs. Random Projection is used to reduce the dimensionality of the data, and k-means clustering is used to group the BBVs into clusters. All BBVs in a cluster belong to the same phase. Other structures for phase classification include those that capture memory access stride, structures that employ the notion of working sets, and even structures that use wavelet coefficients [13]. Figure A.3 shows the memory behavior of Firefox over fifty million instructions as it loads a web page, where intervals are colored according to the wavelet-based phase classification algorithm.

A.5 Novel Computer Architectures

Making effective use of billions of transistors and a multitude of cores on a single chip is the goal of next-generation processor design proposals. The key to success is managing complexity by using robust design abstractions that do not hide the technical nuances. Computation will become less expensive, but communication will become more expensive. Interconnection networks that manage the communication among large numbers of cores will likely consume a large portion of the on-chip resources. Just as processors use a hierarchy of memories, it is likely that future processors will use a hierarchy of interconnect.

A.5.1 The DIVA Architecture

The DIVA architecture [3] is an attempt to manage complexity. Verification of processors that use speculative execution is very computationally intensive. The key idea is that verifying the correctness of the result of a computation is less computationally demanding than computing that result. Therefore, the runtime correctness of a complex processor that uses out-of-order execution can be verified by a smaller *checker* hardware module. This checker unit's small size makes it much easier to verify. The checker unit resides on the chip along with the more complex processor that uses speculation. There are several parallels between the DIVA concept and the reference monitor concept. Both are small, making them easier to verify, both help to manage complexity, and both are run-time mechanisms that reside on the chip.

[1] A basic block is a straight-line sequence of code with one entry point, one exit point, and no jump instructions.

A.5.2 The Raw Microprocessor

To manage complexity, future processors will likely employ large numbers of computational cores. These cores will need to communicate in an efficient manner, requiring alternatives to traditional bus interconnect. A shift from computation-centric to communication-centric design is underway. The idea of networks-on-a-chip (NoC) has been around for a while [6], but few NoCs have been realized due to their complexity. Implementation of a network-on-chip requires building scaled-down networking routers that are located throughout the chip. The chip is divided into tiles, where each tile has a computational core, a network interface (NI), and a switch. Various topologies have been studied, such as the hierarchical ring-based interconnect [5], although realistic evaluation benchmarks, network traffic models, and simulation environments are currently lacking. While the primary application of NoCs is embedded systems, Intel's technology roadmap for future interconnect calls for a scaled-down network protocol stack for on-chip interconnection networks [11]. The Raw Microprocessor Architecture, developed at MIT, uses both static and dynamic routing [25, 28]. Tilera is a company that has developed a 64-core processor aimed at the embedded market. Tilera's processor is called TILE64 and is based on the MIT Raw Architecture.

A.5.3 The WaveScalar Architecture

WaveScalar [24] is a dataflow architecture that is an alternative to the von Neumann architecture. The problem with the von Neumann architecture is that all of the data, including data residing in low levels of the memory hierarchy (e.g., off-chip DRAM), has to be brought into a central location (the CPU) to be processed, before being sent back out to memory. The key idea of WaveScalar is to execute the program in place in the memory system. Whenever a variable's value changes, this automatically triggers the recalculation of the values of other variables that are dependent on it. WaveScalar exploits the *principle of locality*, meaning that if the program accesses a memory value at location i, it is likely that the program will access one of i's neighbors in the near future. The basic building block of the WaveScalar processor is the WaveCache, consisting of both memory and processing elements.

A.5.4 Architectures for Medicine

There are many medical applications of computer architecture. For example, subthreshold voltage processors in medical devices can be used to prevent blindness [17, 18]. Subthreshold voltage processors trade processor performance for energy savings. The device uses a subthreshold voltage processor to measure intraocular pressure to delay the onset of blindness in glaucoma and diabetes patients.

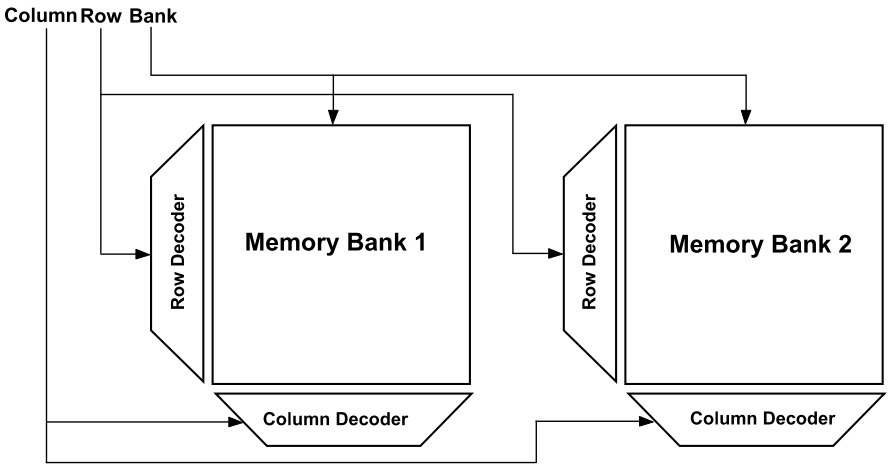

Fig. A.4 Memory is arranged into banks, with row decoders and column decoders

"The system is designed to grip the inner surface (vitreous) of the eyeball. The system will be installed via out-patient surgery, and it will provide patients with real-time feedback on the interior eye pressure. Recent medical studies have shown that careful monitoring and subsequent control of intra-ocular pressure can delay the onset of blindness in glaucoma and diabetes patients. The intra-ocular pressure measurement system includes a subthreshold sensor processor, 384 bytes of memory, 1024 bytes of ROM, a MEMOS-based pressure sensor, a Peltier-based energy scavenging mechanism which utilizes temperature gradients within the eyeball to produce electricity, and a communication system based on inductive coupling" [18].

A group at Stanford is working on a retinal prosthesis implant that is placed behind the retina and has an array of tiny solar cells, and other groups send power to their devices via RF signals [10].

A.6 Memory

A good analogy to the memory hierarchy of a computer is a kitchen. When the cook needs an ingredient, the first place to look is the refrigerator. The refrigerator is like a cache: finding the needed ingredient is analogous to a cache hit, and not finding it is analogous to a cache miss. If the ingredient is not in the refrigerator, it is necessary to check the pantry. The pantry is analogous to an L2 cache. If the required ingredient is not in the pantry, this is analogous to an L2 cache miss, and a trip to the supermarket is required. The supermarket is analogous to off-chip memory because of the relatively large number of clock cycles required to perform an access to off-chip memory. Fetching an item from the pantry takes progressively longer than fetching an item from the refrigerator in the kitchen, and driving to the supermarket takes progressively longer than fetching the ingredient from the pantry.

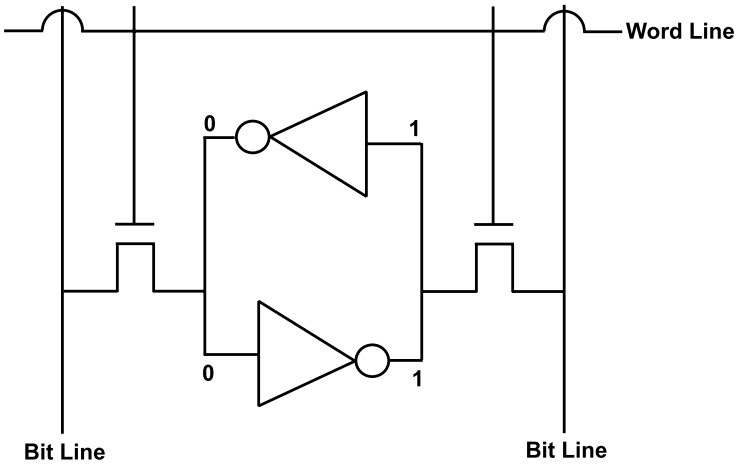

Fig. A.5 An SRAM cell stores a single bit. Feedback between two NOT gates, which enter a stable equilibrium, is the storage mechanism

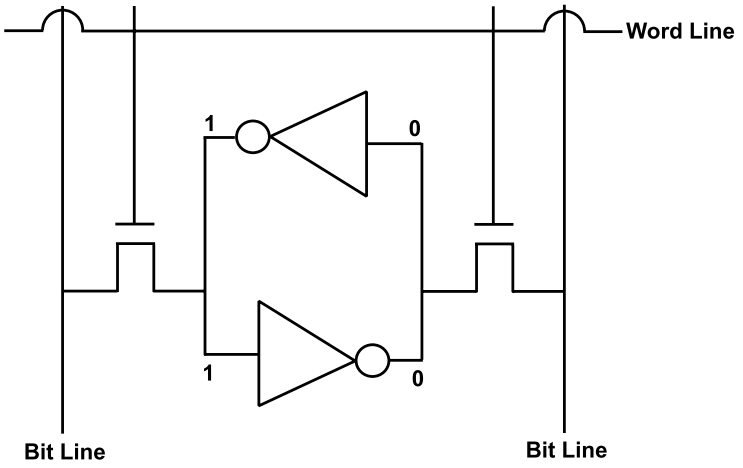

Fig. A.6 Another stable equilibrium of an SRAM cell

Memory is arranged into banks, as shown in Fig. A.4. A memory address must specify which bank along with the row and column of the desired location within that bank. A row decoder and a column decoder select the row and column specified in the address. The bit is stored at the junction of the row and column. Figure A.5 shows how a bit is stored in an SRAM cell. Feedback between two NOT gates, which enter a stable equilibrium, is the storage mechanism. Figure A.6 shows another stable equilibrium. Six transistors are needed for each SRAM cell: two for each NOT gate, and two additional transistors. Figure A.7 shows a multi-port SRAM

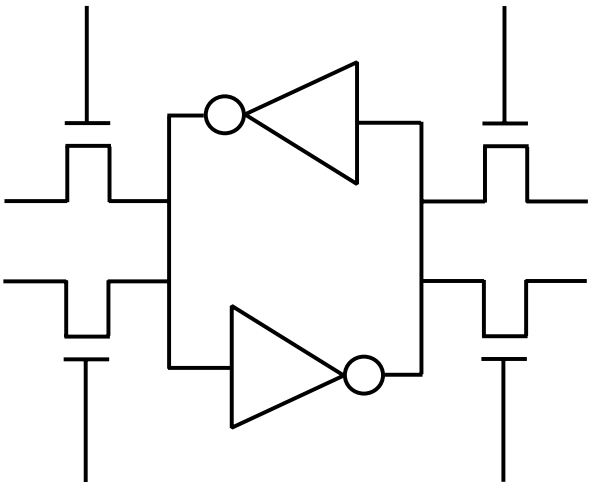

Fig. A.7 A Multi-port SRAM cell allowing multiple simultaneous access

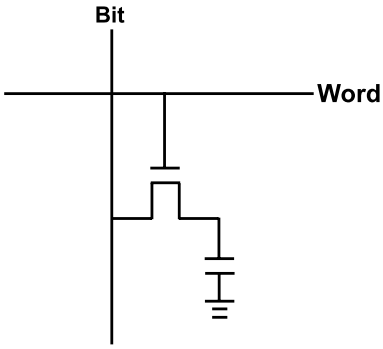

Fig. A.8 A DRAM cell

cell that allows multiple simultaneous access, which is useful for video cards that read and write at the same time or perform multiple accesses to the same location.

Despite its large power and area requirements, SRAM is used for on-chip L1 caches and registers, where high performance is needed. Off-chip DRAM, on the other hand, uses fewer transistors per bit, reducing the power and area for each bit. This makes it possible to build memories very densely but also reduces the performance (1 ns for SRAM and 100 ns for DRAM). A DRAM cell uses a capacitor, which is a *bucket* of electrons. A write to DRAM involves either charging or discharging, but a read is more tricky. Since the capacitor leaks slowly, buckets that have the value one need to be refilled. The currents involved are tiny, with just one hundred electrons representing a single bit. Figure A.8 shows a DRAM cell, where the capacitor is drawn in the lower right portion of the figure.

Due to the growing gap between processor and memory performance, system designers employ a hierarchy of memories, a concept devised by von Neumann in 1946. To capture spatial locality, caches are broken up into blocks. In a *fully associative* cache, a block can go anywhere in the cache. This requires a system

of tags in order to know which pieces of memory reside in the cache, and parallel searches of all slots are required. At the other extreme is a *direct mapped* cache, which works like a hash table because a block can only go to one location. Direct mapped caches are very fast but may have collisions of addresses to the same slot. In between the two extremes is a *set associative* cache, which allows a block to go to a set of possible locations. Cache replacement policies include least recently used (LRU), least frequently used (LFU), random, and *oracle*, which makes a prediction about which pieces of memory will be used in the future.

A common program optimization technique is to change the way that a program accesses memory in order to reduce the number of cache misses. The program is redesigned so that needed data fits into the cache better and can be prefetched more effectively.

A.7 Superscalar Processors

Analysis of the dependencies of code determine when more resources (e.g., adders) are needed in order to execute multiple instructions in parallel. Multiple issue processors include *Very Long Instruction Word (VLIW)* machines and superscalar machines. In a Very Long Instruction Word (VLIW) machine, two instructions can be glued together to form one big instruction, and the compiler does the scheduling. In a superscalar machine, the processor hardware schedules the instructions. More things are being done by the hardware these days rather than the compiler. For example, *out-of-order execution*, in which the hardware dynamically rearranges instructions, is becoming common.

Tomasulo's Algorithm is a distributed, scalable algorithm for finding parallelism [26]. For data dependencies, out-of-order execution adapts dynamically to the data flow graph. In a sense, the data flow graph is sucked through a straw, and since there are finite resources on the chip, sometimes there is a miss. Tomasulo's Algorithm uses dynamic loop unrolling, mix & match, and reservation stations. Instructions wait at the reservation stations until they have what they need. After executing, the fact that they now have what they need is broadcast over the broadcast bus. Tomasulo's algorithm was implemented on the IBM 360/91, which was a commercial flop. It had four floating point registers and little compiler support. In the case of an add, the instruction waits for the two operands it needs, checking the register file. The result is broadcast everywhere over a common data bus. The reservation station scheme requires entries for the operation (e.g., addition, subtraction), the operands, the names of the reservation station of origin, and whether the reservation station is occupied.

Problems with Tomasulo's Algorithm include broadcasting to everybody over the common data bus. Since order matters, arbitration is needed to prioritize the communication. Since overloading the bus is a problem, multiple busses may be necessary or a *cross bar* network of point-to-point interconnection. Complexity is another problem with Tomasoulo's Algorithm. Tomasulo's Algorithm teaches us that good solutions are a *compromise*. An alternative scheduling scheme to Tomasulo's Algorithm is *just in time scheduling* used by the Pentium 4.

For name dependencies, the hardware implementing Tomasulo's Algorithm uses register renaming. While the user and the compiler can only see a small number of architectural registers (e.g., 32), the set of physical registers can be larger (e.g., 256), and register renaming maps the smaller set onto the larger set. This register renaming contributes to the complexity of the design and verification of Tomasulo's Algorithm.

A.8 Multithreading

Figure A.9 shows two threads executing on a superscalar processor and illustrates the problem of *vertical waste*, in which redundant hardware resources often go unused [27]. Tullsen et al. define empty issue slots as vertical waste when the processor issues no instructions in a cycle; horizontal waste occurs when all of the available issue slots cannot be filled in a cycle [27]. At one point in Fig. A.9, Thread 2 is not able to use any of the hardware resources. To mitigate this problem, one might try to interleave both threads, a scheme called *fine-grained multithreading*, but this does

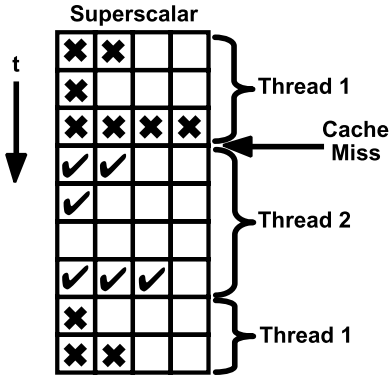

Fig. A.9 Superscalar machines exhibit *vertical waste*

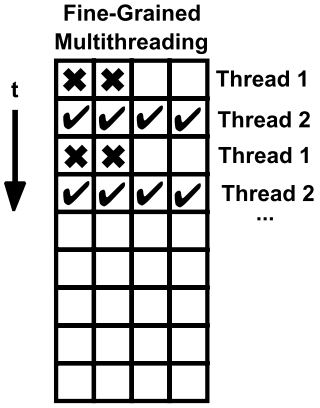

Fig. A.10 Fine-grained multithreading machines exhibit *horizontal waste*

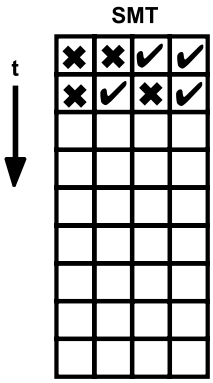

Fig. A.11 A Simultaneous Multithreading (SMT) Machine interleaves instructions every cycle, performing them simultaneously. This approach does a good job of packing the instructions in and more fully utilizing redundant hardware resources

not eliminate the *horizontal waste* of resources, as shown in Fig. A.10. Although interleaving has solved the vertical waste problem, there are still times when all of the hardware resources are not being utilized fully. Simultaneous Multithreading (SMT) machines interleave instructions every cycle, performing them at the same time, as shown in Fig. A.11. SMT machines do a good job of packing the instructions in. Technical issues of SMT machines include:

- page tables and TLBs (a TLB is a cache for the page table)
- register assignments (separate registers are required)
- multiple stacks
- the memory hierarchy (one thread might overwrite the cache)
- separate branch predictors
- parallel instruction issue logic
- sharing one structure between threads

References

1. K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: a view from Berkeley. Technical Report No. UCB/EECS-2006-183, University of California, Berkeley, 18 December 2006
2. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
3. T.M. Austin, DIVA: a reliable substrate for deep submicron microarchitecture design, in *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO-32)*, Haifa, Israel, November 1999
4. T. Austin, E. Larson, D. Ernst, SimpleScalar: an infrastructure for computer system modeling. IEEE Comput. **35**(2), 59–67 (2002)
5. S. Bourduas, Modeling, evaluation, and implementation of ring-based interconnects for network-on-chip. Ph.D. dissertation, McGill University, Dept. of Electrical and Computer Engineering, Montreal, Canada, May 2008
6. W.J. Dally, B. Towles, Route packets, not wires: on-chip interconnection networks, in *Proceedings of the 37th Design Automation Conference (DAC)*, Las Vegas, NV, June 2001

7. J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. L. Eeckhout, R.H. Bell Jr., B. Stougie, K. De Bosschere, L.K. John, Control flow modeling in statistical simulation for accurate and efficient processor design studies, in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, Munich, Germany, June 2004
9. L. Eeckhout, J. Sampson, B. Calder, Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation, in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'05)*, Austin, TX, October 2005
10. W.D. Jones, A form-fitting photovoltaic artificial retina. *IEEE Spectrum*, **46**(12), December 2009. http://spectrum.ieee.org/biomedical/bionics/a-formfitting-photovoltaic-artificial-retina
11. D. Kanter, The Common System Interface: Intel's future interconnect. *White Paper, Real World Technologies*, August 2007
12. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in *Proceedings of the 2005 ACM/SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005
13. T. Huffmire, T. Sherwood, Wavelet-based phase classification, in *Proceedings of the Fifteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Seattle, WA, September 2006
14. T. Huffmire, J. Valamehr, T. Sherwood, R. Kastner, T. Levin, T.D. Nguyen, T. Sherwood, Trustworthy system security through 3-D integrated hardware, in *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, Anaheim, CA, June 2008
15. S. Mysore, B. Agrawal, S.-C. Lin, N. Srivastava, K. Banerjee, T. Sherwood, Introspective 3D chips, in *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2006
16. S. Mysore, B. Agrawal, S.-C. Lin, N. Srivastava, K. Banerjee, T. Sherwood, 3-D integration for introspection, in *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, January–February 2007
17. L. Nazhandali, B. Zhai, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, D. Blaauw, Energy optimization of subthreshold-voltage sensor network processors, in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, Madison, WI, June 2005
18. L. Nazhandali, M. Minuth, B. Zhai, J. Olson, T. Austin, D. Blaauw, A second-generation sensor network processor with application-driven memory optimizations and out-of-order execution, in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, San Francisco, CA, September 2005
19. M. Oskin, The revolution inside the box. Commun. ACM **51**(7), 70–78 (2008)
20. E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, B. Calder, Using SimPoint for accurate and efficient simulation, in *International Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, June 2003
21. T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2002
22. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases, in *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, November–December 2003
23. R. Srinivasan, J. Cook, S. Cooper, Fast, accurate microarchitecture simulation using statistical phase detection, in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, Austin, TX, March 2005
24. S. Swanson, K. Michelson, A. Schwerin, M. Oskin, WaveScalar, in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, San Diego, CA, December 2003

25. M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, The RAW microprocessor: a computational fabric for software circuits and general-purpose programs. IEEE Micro **22**(2), 25–35 (2002)
26. R.M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Develop. **11**(1), 25 (1967)
27. D.M. Tullsen, S.J. Eggers, H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995
28. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal, Baring it all to software: RAW machines. IEEE Comput. **30**(9), 86–93 (1997)
29. L. Wu, C. Weaver, T. Austin, CryptoManiac: a fast flexible architecture for secure communication, in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, Gothenburg, Sweden, July 2001
30. C. Zilles, G. Sohi, Master/slave speculative parallelization, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, November 2002