12
Reliability

Till now, we have assumed that all our elaborate designs and protocols work flawlessly. This is sadly not the case and unfortunately modern hardware suffers from a variety of faults, which can impact its correctness and ultimate lifetime. This chapter is devoted to studying the different aspects of hardware reliability, and how we can detect and recover from faults.

Before we proceed further, it is important to differentiate between three terms here: fault, error, and failure. A *fault* refers to a defect: a part of the architecture, implementation or protocol that deviates from the ideal specification. Note that a lot of faults can be benign in the sense that they do not manifest into wrong internal states. However, some of them lead to erroneous internal states of the system – these are known as *errors*. Even if an internal state such as the value of a register is wrong, it does not mean that the final output is wrong. It is possible that the value may get *masked*. For example, we might have an instruction the divides the value in the register by itself. In this case the final output will still be correct even if the value in the register is incorrect. However, in many other cases the final output will be wrong. This is known as a *failure*. A failure is basically defined as any change in the operation of the system (observed externally) that deviates from ideal specifications. Note that a failure is not limited to a wrong output, even a system crashing is a failure.

Definition 98

Fault *A fault is a defect in the system. It is an aspect of the hardware or software that is undesirable. Note that all faults do not necessarily lead to erroneous results. Many faults can be* masked *(their effects are not visible).*

Error *An error is an internal state of the system that is perceived to be incorrect. Similar to the case of faults, errors do not necessarily lead to wrong outputs. They can get masked.*

Failure *A failure is an externally visible event, where the behavior of the system deviates from its specifications and this is visible to the user of the system. It can include an erroneous output, an unresponsive system, or the case of a system going down. Most of the time we assume a* fail-stop *failure mode, where sub-systems are capable of checking themselves and simply cease their operation if their outputs are wrong. However, there are other failure modes as well*

where sub-systems produce erroneous outputs often maliciously, and deliberately confuse other sub-systems. In this case it is also possible for sub-systems to collude and send potentially confusing and erroneous messages. Such malicious failures are known as Byzantine failures. They are relatively rare in hardware systems.

There are a few well known reliability metrics in the context of failures that need to be explained first: FIT, MTTF, MTBF, and MTTR. The FIT metric measures the expected number of failures per billion hours of operation. MTTF is defined as the mean time to failure, which essentially represents the expected time it will take for the system to fail. The assumption here is that the system is not repairable; however, if we make the assumption that the system can be repaired then typically the metric MTBF (mean time between failures) is used. This measures the expected duration between two failures. Once the system has failed, there is a need to repair its state, and restore it. The time it takes to repair the state is known as the mean time to repair (MTTR).

We will keep on referring to these metrics throughout this chapter.

Definition 99

- *Failure rates are typically measured in the units of FITs (failures in time). One FIT is one failure per billion hours.*
- *MTTF is the mean time to failure (assuming the system cannot be repaired).*
- *MTBF is the mean time between failures (assuming a system that can be repaired).*
- *MTTR is the mean time to repair (fix the state of the system).*

Given this understanding, we shall delve into the different sources of erroneous execution in this chapter. Notably, we shall look at two kinds of faults: transient and permanent. Transient faults are ephemeral in nature, come in to existence for a very short period of time and then disappear. However, permanent faults never disappear. We can further divide permanent faults into two types: congenital and ageing related. Congenital faults are there since the time of fabrication, and are typically caused because of imperfections in the fabrication process. However, ageing related faults happen because of a gradual deterioration of transistors' properties. Over time transistors or wires can fully deteriorate and either become open circuits or closed circuits. In this case, these faults can permanently alter the operation of the processor, and unless there is some redundancy, the computer system needs to be decommissioned.

We shall first look at three sources of transient errors/faults: soft errors, faults due to inductive noise, and faults due to non-determinism. Then we shall consider congenital faults: process variation and RTL-level bugs. Finally, we shall consider ageing related faults that lead to permanent breakdowns.

12.1 Soft Errors

12.1.1 Physics of Soft Errors

In the 60s and 70s, engineers noticed a strange phenomenon. A lot of electronic devices in the proximity of nuclear sites, and in space missions were mysteriously crashing. They thus set out to find the reason

and also understand the connection between such failures. They found that high-energy particles such as neutrons and alpha particles that are a part of nuclear or cosmic radiation collide with the silicon die at great speed. These particles displace charge, which causes a temporary current pulse. The current pulse is strong enough to flip the value of bits, particularly in memory cells. In the early 80s, engineers started noticing failures in DRAM devices even at sea level. With increased miniaturization, such failures started affecting SRAM cells and even logic gates. Such errors are known as *soft errors* that are mostly single-event upsets (SEUs), which means that a particle with a very high momentum makes an impact, and the resulting current pulse flips the value of a bit in the circuit.

It should be noted that such single event upsets are not radiation-induced all the time, we can have several intrinsic factors such as power supply noise and inductive noise, which we shall consider in Section 12.2. However, in this chapter we shall limit ourselves to extrinsic factors, which are particle strikes because of cosmic radiation, or because of inherent impurities present in the packaging material.

Sources of Radiation

In the early 70s, alpha particles (two neutrons and two protons) were the most common sources of radiation-induced soft errors. They were emitted by trace uranium and thorium impurities in the packaging material. These are radioactive elements and spontaneously emit alpha particles. Alpha particles have a high mass, and consequently they need to have a high energy to penetrate deep into silicon. The particles generated by these impurities have energies in the range of 4-9 MeV, and thus the maximum penetration range is limited to 100 μm. Hence, we need not worry about alpha particles that are generated outside the package.

Another source of alpha particle emissions is an unstable isotope of lead, ^{210}Pb. It decays into ^{210}Po, which emits alpha particles. These alpha particles can displace a lot of charge. With sophisticated fabrication technologies the proportion of such impurities has gradually decreased and as of today high-energy cosmic rays that comprise mostly of neutrons are the primary source of radiation-induced soft errors.

Next, let us consider a secondary source of radiation, which materializes due to the interaction of neutrons with a relatively rare isotope of boron called ^{10}B. Boron is heavily used in creating p-type materials and is also a part of boron phosposilicate glass that is used as a dielectric material. Whenever a neutron impacts such a boron atom, it breaks it into an alpha particle and a lithium nucleus that travel in opposite directions. Both the alpha particle and the lithium nucleus are capable of inducing soft errors is in the circuit.

Dynamics of a Particle Strike

Whenever a particle strikes the silicon substrate it approaches it with a very high momentum and gradually loses momentum. Most approaches model the loss of energy as a linear process, and the rate of energy loss per unit distance is known as the LET (linear energy transfer) rate. It is often divided by the density of the target material. Normally in CMOS circuits, transistors are most susceptible to such single event upsets (SEUs) in the *off state*. An LET rate of more than 20 MeV-cm^2/mg [Wang and Agrawal, 2008] is sufficient to create a current pulse that can propagate in the circuit and change the value stored in a latch.

The mechanism is as follows. Whenever there is a particle strike, it displaces charge and creates a lot of electron-hole pairs. If the trajectory of the particle strike passes through the depletion regions in a transistor, carriers get rapidly collected over there. This collection phase lasts for tens of picoseconds. The resultant displacement in charge creates a current pulse. The initial phase of charge collection is because of drift, and later on the amount of current reduces and the mechanism of the movement of carriers changes to diffusion. A representative current pulse with the drift and diffusion dominated regions is shown in Figure 12.1. Typically, the entire current pulse lasts for roughly 200-600 ps, which accounts for the majority of the clock period (with a 3GHz clock). Most of the charge collection, henceforth denoted as Q_{total} happens within 2-3 μm of the junction region. Q_{total} depends on the

energy of the particle, nature of the particle, the trajectory, the impact site, and the voltages of the different terminals. The current pulse can be approximated by an equation of the form [Wang and Agrawal, 2008]:

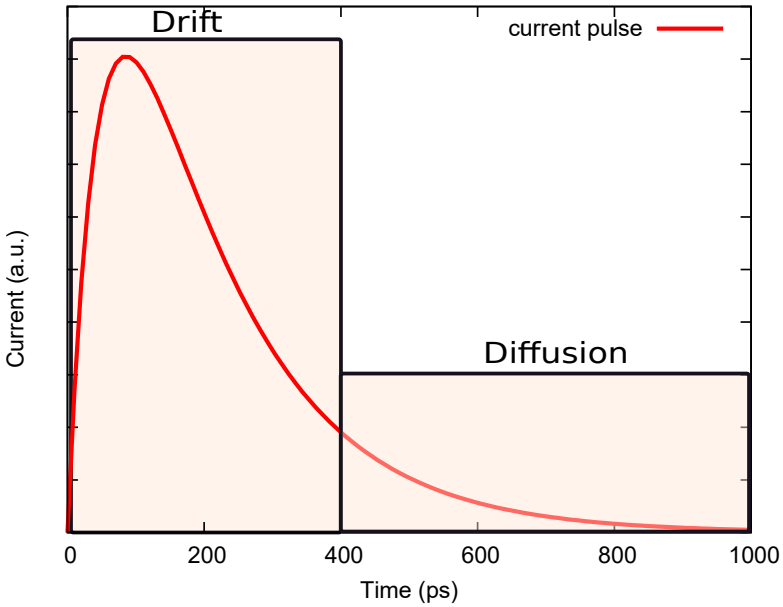

Figure 12.1: Representative current pulse. (a.u. stands for arbitrary units)

$$I(t) = \frac{Q_{total}}{\tau_\alpha - \tau_\beta}(e^{-\frac{t}{\tau_\alpha}} - e^{-\frac{t}{\tau_\beta}}) \tag{12.1}$$

Note that this equation is a difference of two exponential functions. Typically, τ_α is dependent on the properties of the transistor and represents the charge collection time, whereas τ_β is relatively independent of the manufacturing process and depends on the trajectory of the particle strike. As mentioned in [Wang and Agrawal, 2008] typical values are 164 ps for τ_α and 50 ps for τ_β. The value of Q_{total} is somewhere between 50-500 fC (femto coulombs) depending on the nature of the particle.

Alternative one-parameter models also exist [Hazucha and Svensson, 2000] for the current pulse (η is a single empirically determined parameter here).

$$I(t) = \frac{2}{\eta\sqrt{\pi}}\sqrt{\frac{t}{\eta}}e^{-\frac{t}{\eta}} \tag{12.2}$$

Hazucha-Svensson Model

Unfortunately just computing the total collected charge is not enough to estimate the soft error rate. This is because the output terminals of the transistor have an associated capacitance, which can attenuate the current pulse significantly. While computing the soft error rate, it is necessary to take into account the environment in which the particle strikes. For the purposes of calibration and comparison, we typically consider soft errors in the context of simple 6-transistor SRAM cells.

The Hazucha-Svensson model to estimate the soft error rate is as follows.

$$SER = F * CS \tag{12.3}$$

Here, SER is the rate of soft errors. It is typically measured in the unit of FITs (failures in time), where one FIT is one failure per billion hours. F is the neutron or alpha-particle flux (depends on the type of particle we are considering). CS (critical section) is the effective area that is susceptible to particle strikes. We can think of it as the relevant part of the transistor's structure that is vulnerable to particle strikes. It is given as follows.

$$CS \propto A \times e^{-\frac{Q_{crit}}{Q_S}} \tag{12.4}$$

It is unfortunately hard to compute the size of the critical section exactly. Hence, Equation 12.4 uses a proportional sign. CS is clearly proportional to the area of the circuit that is sensitive to particle strikes (around the junctions and so on). Additionally, it is a negative exponential function of the ratio of two variables: Q_{crit} and Q_S. Q_{crit} is the minimum amount of charge that needs to be displaced to generate a current pulse that is large enough to flip the value stored in the SRAM cell. Q_S is known as the charge collection efficiency, which is a measure of the charge generated by a particle strike (units in fC). If $Q_S \ll Q_{crit}$ then the term $e^{-\frac{Q_{crit}}{Q_S}}$ tends to 0. This means that the rate of soft errors tends to 0. Whereas, if $Q_S \gg Q_{crit}$ then the rate of soft errors is proportional to the area.

Note that Equation 12.4 was derived empirically by Hazucha and Svensson based on experiments and observations. This was valid for their setting with a gate length of 600 nm. For different technologies this equation will have a different form, and thus for any upcoming technology it is necessary to perform similar experiments to derive a new soft error model. We can either perform experiments on a device simulator, where we can irradiate the device with different particle streams that have different velocities and trajectories, and measure the resultant soft error rate. Another approach is to conduct a physical experiment where we have a neutron source. It is used to bombard a test circuit with neutrons and then measure the rate of bit flips.

12.1.2 Circuit and Device Level Techniques to Mitigate Soft Errors

Device Level Techniques

Reducing the susceptibility of a device to soft errors is known as *radiation hardening*. Radiation hardening techniques at the device level are the most preferred, particularly if there is no concomitant cost. This is because it reduces the amount of effort that is needed at later stages: the circuit and architectural levels. At the device level, the first approach is to eliminate all those materials that are involved in soft errors: uranium and thorium impurities, and impurities with the ^{10}B or ^{210}Pb isotopes. The use of the BPSG dielectric that contains ^{10}B can also be curtailed, and it can be limited to layers that are not close to the silicon layer.

Another set of approaches focus on radiation hardened transistor technologies. Here, the main aim is to reduce Q_S. A common approach is to use the triple well structure as shown in Figure 12.2(a). Here, there is a deep n-type doped region below the substrate to reduce the total amount of collected charge.

The other popular approach is to use the silicon-on-insulator (SoI) technology. In this case, there is a buried oxide layer below the channel that effectively cuts it off from the rest of the substrate (see Figure 12.2(b)). This reduces the volume in which charge can be collected. As a result, the chances of a soft error reduce drastically.

Let us now discuss a few approaches to increase Q_{crit}. It primarily depends on the transistor size, supply voltage, and output capacitance. With an increase in any of these quantities, the critical charge increases. Sadly, if we increase these quantities the circuit takes up more area and dissipates more power. On the flip side, we observe that with increasing miniaturization, the critical charge will continue to decrease and this will increase the susceptibility to soft errors.

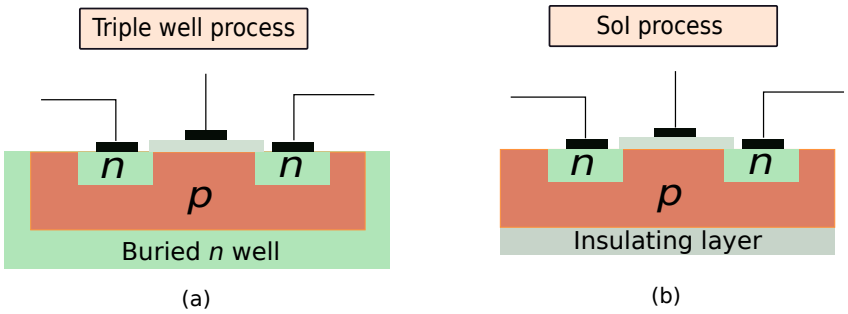

Figure 12.2: (a) A triple well process, (b) An SoI process

Circuit Level Techniques

While studying the effect of soft errors, we typically differentiate between memory and logic circuits. Soft errors in memory such as SRAM and DRAM arrays are easy to model and detect.

Let us thus first look at the effect of soft errors in a combinational logic circuit. Recall from our discussion on in-order pipelines that we typically break a large circuit into smaller chunks and pipeline them. To achieve this, we need to add a pipeline latch after a block of logic to store its value at the end of the clock cycle. Let us look at the effect of soft errors on logic circuits in this setting. We will consider a particle strike as leading to a soft error only if a bit in the pipeline latch gets flipped. Furthermore, this can only happen if a particle strikes a transistor in the logic circuit, and the current pulse reaches the pipeline latch, where it flips a bit. For example, instead of storing a logical 0, we might end up storing a logical 1.

Masking Mechanisms

The good news is that most current pulses are not potent enough to flip a stored bit. There are various reasons why such current pulses can get *masked*. Let us look at some of the most common masking mechanisms. The first masking mechanism is known as *electrical masking*. Assume that a particle strikes a transistor in a combinational logic circuit. As the disturbance propagates towards the pipeline latches, the pulse passes through a multiplicity of logic gates, and it gradually gets attenuated. By the time it reaches the latches, its strength reduces considerably and thus may not be potent enough to change the state. This is known as electrical masking (see Figure 12.3(a)).

Now, consider *logical masking* (see Figure 12.3(b)). Look at the first input of the AND gate in the figure. Assume it makes a transition from a logical 0 to a logical 1. Since the value of the other input is a logical 0, this transition does not have any effect on the output. This occurs because the final output remains a logical 0 due to the input staying at 0. Hence, regardless of the transition made by one of the inputs (because of soft errors), the final value still remains the same. This is where the potential error is said to be logically masked.

The third category of masking is known as *timing window masking* (see Figure 12.3(c)). Note that for a latch to actually change its value, the updated value has to appear only in a critical time window around the negative edge of the clock (between the setup time and the hold time). At any other time the changed input does not lead to a change in the stored state of the latch. This means that if there is a particle strike, the resultant current pulse has to reach the output latch only during this critical period. If this does not happen then the changes will not flip the value of any stored bit, and thus the error will get masked. Due to a combination of these three mechanisms, the actual probability of a bit flip reduces significantly.

Reducing the probability of soft errors in memory arrays is relatively easy. We just need to add ECC (error correction codes) bits to each line. These can be used to detect and correct bit flips. In comparison, bit flips in logic circuits are much harder to detect and correct. Some of the most effective

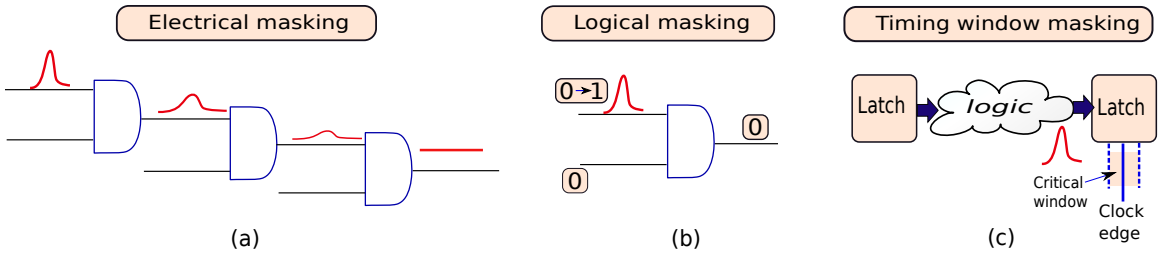

Figure 12.3: Different masking mechanisms. (a) Electrical masking, (b) Logical masking, (c) Timing window masking

approaches actually try to reduce the probability of the current pulse propagating to the latches. This can be done by identifying *sensitive paths* in the circuit (paths that are more susceptible to propagating current pulses). The transistors on the sensitive paths can then be modified to reduce their susceptibility. There are several ways of doing this: increase the size, increase output capacitance, or increase the supply voltage (if there is a choice).

12.1.3 Architectural Techniques to Mitigate Soft Errors

Basic Concepts

There are numerous architectural techniques to mitigate the damage caused by soft errors. First we need to understand the basic mechanisms by which soft errors can impact architectural state. We should first note that not all errors will actually impact the architectural state. For example, if the unit is not being used or if the value of an instruction's result gets corrupted, but the instruction itself is on the wrong path, then the error is not visible. There can be many such instances where the effect of soft errors does not lead to failures: the final output is not wrong, and the program does not crash.

To model the failure rate, SE_{fail}, correctly let us consider the following equation.

$$SE_{fail} = SER \times TVF \times AVF \tag{12.5}$$

SER is the soft error rate. TVF is a factor (between 0 and 1) known as the timing vulnerability factor, which captures the effect of the unit being off (unused at that point of time). AVF is the architectural vulnerability factor, and is defined as the probability that the soft error leads to an erroneous output.

Let us understand the architectural vulnerability factor in detail (refer to Figure 12.4). The aim here is to find if a given bit is vulnerable or not, which means that if it gets flipped, whether it affects the final architectural state or not. If the bit is not read, clearly it is not vulnerable. There is thus no error in this case. However, if it is read then it is possible that it is protected by parity or ECC bits. For example, in many modern processors even small structures such as the register file are protected by ECC bits. If the bit is protected then it is possible that the error is only detected and not corrected. This can happen if we are only using parity bits, or if let's say we can correct only one-bit errors, and there are errors in two bits. Such errors are known as Detected but Unrecoverable Errors (DUE errors). Whereas, if we can correct the bit flips then there is no error.

Now let us consider the case when the bit is not protected. In this case, we need to assess if the bit is relevant to the architectural state or not. If it is a part of the architectural state and helps in determining the output then the resultant bit flip is known as Silent Data Corruption (SDC).

The good news is that a lot of errors happen in parts of the circuit that do not determine the final output. For example, we have numerous bits in the ALU logic, decode logic, and pipeline latches that are not used. If any of these bits get flipped, then there are no errors. Additionally, there are a lot of structures in the pipeline that impact performance but do not affect correctness. Consider the branch

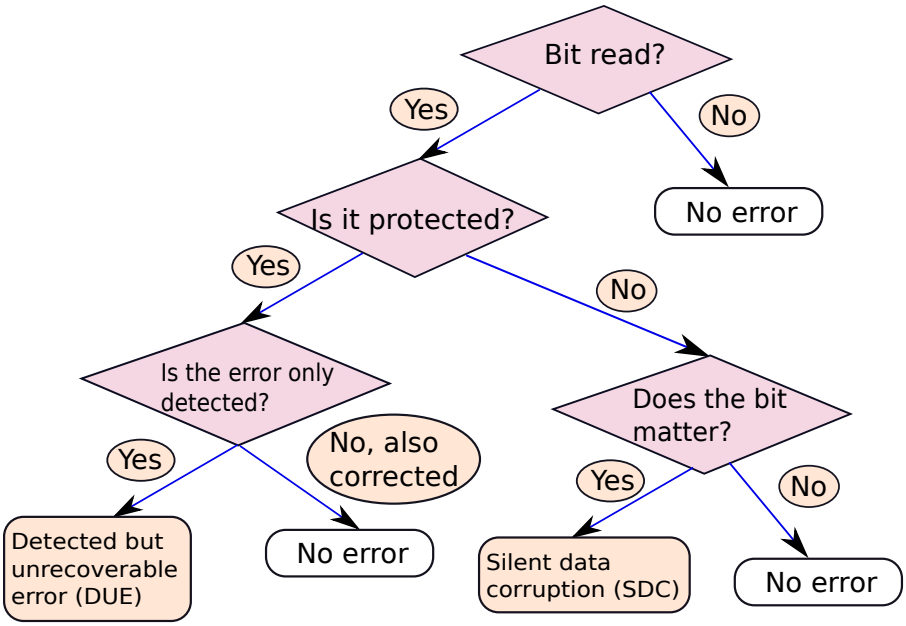

Figure 12.4: Components of the AVF

predictor, the prefetch buffers, and entries in the instruction window for issued instructions. Even if there is a bit flip in the contents of these structures, correctness is not affected. Moreover, there are many instructions in the pipeline that are either dynamically dead (do not affect the output), are on the wrong path, fetch the value of performance counters, or are a part of some misspeculated state – the contents of all of these instructions are immune to soft errors. For example, the AVF of a branch predictor entry is 0. The AVF of an instruction window entry is between 0 and 1 (depending on the behavior of the program).

Design Space of Checkers

Soft errors do not pose much of a problem in regular desktops and laptops. However, they are very serious issues in large servers and supercomputers. The single bit flip can have catastrophic consequences. Imagine a bank account where a bit flip converts a billionaire to a pauper – all because of a single neutron! Additionally, radiation induced soft errors are serious issues in aircraft and definitely in components that go to space. Even with extensive shielding soft errors do happen, and thus we need architectural techniques to mitigate their pernicious effects.

Given that there is no way to predict soft errors, we need checkers that can be dedicated software or hardware units whose job is to check the execution of a program (particularly SDC errors); if they detect any transient fault, then they can initiate a process of recovery. We shall use the same terminology as originally used in the survey paper by Kalayappan et al. [Kalayappan and Sarangi, 2013] to classify the design space of checkers based on their degree of coverage.

Complete In this case the execution of the entire thread is checked. Such approaches are typically very expensive. However, in most cases where we cannot take a risk, this approach is required.

Subset Here we check the results of a subset of instructions.

Invariant In this case we check if certain properties (invariants) hold for the outputs. For example, if it is always the case that regardless of the values, the sum of the outputs is equal to a known

value, then all that we need to do is add the outputs and check if it is equal to the expected value.

Symptom Here we check if something went wrong during the execution of a thread. For example, if we do not expect a segmentation fault (illegal memory access), and if there is one it might point towards a bit flip.

Let us now look at another axis where we focus on the structure of the checker. Here we can have three different configurations: *MultiMaster*, *SingleSlave*, and *MultiSlave*. In this case, a *Master* is an independent computing unit that executes the program as it would have executed on a regular system. The *Slave* on the other hand is a system that is impaired either in terms of hardware resources or performance. It is often used to check for invariants, or check for error symptoms. Note that we did not list the SingleMaster configuration because it simply represents the default implementation, which does not have any in-built checking mechanism.

MultiMaster Systems

The *MultiMaster* configuration uses redundant threads. In this case, we have parallel threads either running on the same core or on different cores that perform exactly the same computations. We can then periodically compare the results. If there is a discrepancy, we can infer a soft error. One of the earliest designs in this space was the IBM G5 processor [Slegel et al., 1999] that used two parallel pipelines that executed instructions from the same thread. They compared the values for all the stores and register writes every cycle. If a discrepancy was detected, the pipeline was flushed. It is important to note here that when we have only one redundant checking unit (a parallel pipeline in this case), it is not possible to determine which value is correct. Hence, it is necessary to flush both. Here, the error remains confined to the pipeline, it is not allowed to propagate to the memory system. We are also making the implicit assumption that the memory system is protected with ECC bits, and DUE errors are not an issue.

Let us now analyze the problems in the scheme. In this case, we have a straightforward 100% overhead in terms of hardware. Furthermore, non-trivial changes need to be made to an OOO pipeline to support this feature. We need to log every single value that changes the architectural state; these values need to be compared at commit time. To minimize the communication overhead, we can run both the threads on the same core (simultaneous multithreaded execution) by partitioning its resources between the threads; however, this will halve the computational throughput of the core and result in a significant slowdown. Hence, the other approach is to run the second thread on a different core. In this case, we will solve the issue of the slowdown, but add two more problems. The first is that values need to be communicated at the end of every cycle. This will place a substantial load on the NoC. The second is that it is in general very difficult to run two cores in lockstep in modern multicores with complex NoCs. Hence, we need to occasionally spend additional time and effort to ensure that the cores remain in synchrony.

Because of these issues, later approaches such as the HP NonStop [Bernick et al., 2005] systems allow errors to propagate to the memory system. Instead of comparing values every cycle, they only compare the values of I/O operations that are visible to the external world. This minimizes the communication overhead, and allows us to utilize the full computational throughput of the cores. Furthermore, the different checking threads can run in separate address spaces (as separate programs) and there is no need for lockstepped execution. This is by far a more practical and efficient setup.

Now, in this configuration we have some unique challenges and opportunities as well. The first is that if we detect a discrepancy, then we still cannot find out which core is at fault. This is because we have only two cores, which means we have dual modular redundancy (DMR). If we detect a transient fault, we need to roll back both the threads on the two separate cores to a safe checkpoint. The *checkpoint* in this case is defined as an earlier point in the execution, which is deemed to be correct; moreover, it is possible to restore the state to that point in the execution. Recall that we have discussed various checkpointing schemes in Section 9.7. Any of them can be used here.

In this case rollback and recovery is a fairly expensive operation. The good news is that soft errors are very rare, hence in practice this does not represent a large overhead. However, in specialized cases such as in environments with a high particle flux such as in aircraft or in space, where either we cannot afford the long recovery time or the check pointing overhead is prohibitive, we need to opt for triple modular redundancy (TMR). In this case, we run three parallel threads on three separate cores and if there is a discrepancy, then the results are decided on the basis of voting. The implicit assumption here is that since soft errors are rare, the chances of an error happening simultaneously in two threads is also very rare. Hence, if we have three threads, the probability that two of them will simultaneously be afflicted by a soft error and that too in the same checkpointing interval is extremely improbable. Hence, it is expected that voting will be successful almost all the time.

There are variations of this scheme, where we divide time into fixed size intervals called *epochs*, and for every epoch we compute a hash of all the values that a thread computes (known as its signature). At the end of an epoch, different threads compare their signatures. Here again, a DMR or TMR based mechanism can be used.

To summarize, with a variety of interventions it is possible to reduce the time overhead of such checking schemes by a significant amount, however, the hardware overhead is still significant. The next two families of approaches that we shall describe will help us in reducing that.

SingleSlave Systems

In *SingleSlave* systems, there is a smaller processor known as the slave processor that takes some inputs from the master and tries to verify the master's computed results. The aim is to finish the verification process quickly such that the critical path is not lengthened. The key insight here is that verification can be a faster process than the computation itself. Consider a simple example. Finding the roots of a set of non-linear equations with multiple variables is a very difficult problem. However, given a solution we can always verify it very quickly. Such *SingleSlave* systems are built on similar lines. In some cases we check for invariants, which means that the master sends its computed results to the slave, which simultaneously checks if the solution is correct or not.

Let us now describe a general system that is based on the original research idea called *DIVA* presented by Austin [Austin, 1999]. Here, the assumption is that the checker is made of larger transistors that are by far more immune to soft errors.

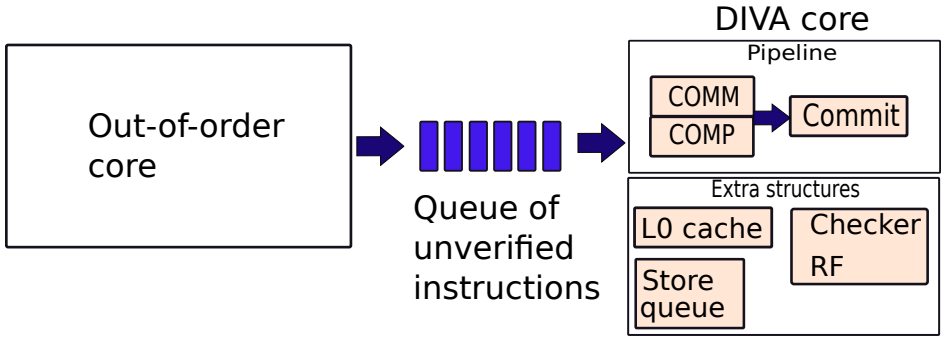

Figure 12.5: The DIVA checker core

Figure 12.5 shows the schematic of a master-slave system. The master core sends an instruction packet to the slave core, which contains the program counter, opcode of the instruction, the values of the operands, and the computed result. The checker processor is a small 2-stage in-order processor that maintains a copy of the architectural register file. The first stage of this pipeline is broken into two sub-stages: communication and computation. These sub-stages run in parallel. The communication

sub-stage checks if the operand values and the memory values have been read correctly. There is a need to read the register values again from the architectural register file in the checker, and for memory values the addresses need to be read again from the cache. Accessing the L1 cache again will increase the pressure on the cache, and this can cause severe performance issues. Hence, later versions of DIVA have proposed to have a small L0 cache within the checker itself that can cache many of the frequently read values.

Next, let us consider the computation sub-stage. Here, a dedicated ALU in the checker re-computes the result and verifies if it is the same as the result passed by the master. If the values are the same, then this means that the computation is correct, otherwise we can infer an error. After the computation and communicate are verified, the instruction proceeds through the DIVA core's pipeline and gets *committed* – writes the results to the architectural state.

If there is any mismatch in the operands' values or results, then there is a need to flush the pipelines. A few more subtle points need to be considered here. Instructions in the master remain within the ROB and the store queue till the checker verifies them. This means that we need to have larger ROBs and store queues to support DIVA. This is indeed an overhead. Furthermore, the store queue is used as a temporary cache to store values written to memory by uncommitted instructions. This is required by both the master and the checker, and it is not a good idea to make the checker access the store queue of the master. Hence, we need to replicate the store queue in the checker as well. For the values written by uncommitted stores, the checker can access entries in its private store queue.

Let us now consider another core issue. The checker needs to match the master in terms of IPC. If the checker is slower, the master will also get slowed down. The good news is that even in cores with a large issue width, for many integer codes the average IPC is never more than 1.5 in practice. Hence, a 2-issue in-order pipeline in the checker is sufficient. If there are temporary peaks in the IPC of the master, a queue between the master and slave can be used to buffer some instructions. However, if we are running floating point code in the master and the IPC is high (let's say more than 3), then a simple checker will not be able to keep up. It will thus slow the system down. Hence, we need to look at another set of designs known as *MultiSlave* systems.

MultiSlave Systems

In a *MultiSlave* system we have multiple checkers. A representative execution is shown in Figure 12.6.

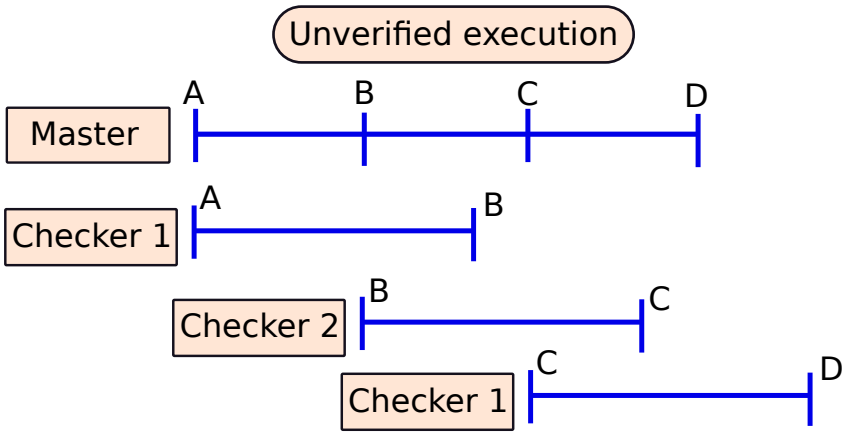

Figure 12.6: Single master, multiple checkers. Here each individual checker is slower than the master.

In such a design we divide the execution into multiple fixed size intervals called *epochs*. Consecutive epochs are checked by different checkers. This solves the problem of matching the rates of the master

and the checker. In this case, we have several checkers that simultaneously verify the results of different epochs. This allows the master to continue at its regular pace, and the checkers can carry on with their work in parallel. If the IPC ratio is four times, then we simply need to have four checkers for each master. Note that in this case rolling back may not be as simple as flushing the pipeline, a slightly more complicated checkpoint recovery mechanism is necessary.

12.2 Inductive Noise

12.2.1 Basic Physics

Recall that we discussed in Section 11.3 that processors have a complex power delivery mechanism. Every processor has a power grid that is connected to the external power supply using hundreds of pins. The reason we need to have a power grid is because if there is a sudden current requirement, the additional current can come from many pins at the same time. Now we have assumed that this is an *ideal voltage source*, which means that it can provide any amount of current while maintaining a constant voltage. Furthermore, we have assumed that the internal impedance of this voltage source is zero. This is sadly not true. Let us now describe the constraints of realistic power grids. For following the rest of the discussion a basic understanding of concepts such as reactance and impedance is required.

The power grid has a lot of wires connected in a grid like fashion, which also means that it has a large inductance. Any inductor opposes the flow of current through it by creating a back EMF (Lenz's law). This means that if there is a sudden current requirement, the power grid will take some time to actually provide it. To offset this effect, designers typically add capacitors known as *decoupling capacitors* that can provide the additional current and also maintain the voltage level. A power grid can thus be replaced by an equivalent RLC circuit. In any such circuit, the behavior is dependent upon the frequency of the variation of the voltage or current. For the purposes of analysis, we typically assume it to be a sinusoidal variation, and quantify it in terms of its angular frequency ω. The reactance of the capacitors and inductors in the circuit is a function of ω. The final impedance takes into account the complex reactances of the inductors and capacitors, and the real-valued resistances.

Any such RLC circuit has an interesting property known as *resonance*. Here, the impedance typically becomes real (imaginary component is zero), and the impedance as a whole reaches an extremum (maximum or minimum). In other words at this point the circuit offers either the maximum resistance to the flow of current or the minimum. In either case, we will have large voltage spikes and these spikes can indeed induce errors in the circuit. Hence, the job of a processor designer is to ensure that the power delivery mechanism never operates close to its *resonance point*. Maintaining the stability of the operating voltage at this point is difficult. To ensure that we are always off resonance the frequency of variation in the current requirement of the CPU should be far away from the resonant frequency. Secondly, the amplitude of variation in the current requirement should be commensurately small to eliminate large voltage fluctuations.

12.2.2 Pipeline Damping

The resonant frequency is typically between 10-100 MHz [Powell and Vijaykumar, 2003a]. If we are unlucky we can have code snippets that run in a loop, and the frequency of the resultant current variation is close to the resonant frequency. In such cases, it is necessary to perform *pipeline damping* – deliberately homogenize the activity by reducing the rate of computation at some point in time (damping). It has been observed that the front end of the pipeline (fetch and decode logic) has a fairly homogeneous power usage (across time), and thus its activity does not have to be damped.

However, there is a need to monitor the activity of the back end: issue, execute, and commit logic. Here the primary aim is to reduce the amplitude of the fluctuation in the current requirement across fixed size intervals of time (known as epochs). Even if the frequency of variation is close to the resonant frequency, because of the homogenization of activity (stability in the current demand), the voltage

fluctuations are limited. To implement this, the idea is as follows. The manufacturer needs to create a model that links the activity of different structures within the CPU (ALU operations, selection operations, instruction issue, etc.) with the current requirement. Every epoch a dedicated circuit keeps a count of all the activity and computes a dynamic estimate of the current drawn since the beginning of the epoch. The aim is to limit the difference in the current drawn across epochs to a value Δ. This can be done by throttling the select rate or issue rate, or by introducing extraneous (fake) instructions to just draw more current. Both of these approaches allow us to introduce a degree of controlled jitter in the execution, which limits the occurrence of large voltage spikes.

12.3 Faults due to Inherent Nondeterminism

Till now, we have assumed deterministic latencies for all the components inside the chip inclusive of the cores, the caches, and the network on-chip. This assumption has helped us design systems that are simple, easy to model, and easy to debug. However, as we shall see in this section, this abstraction breaks down in most practical systems that are in use today because of various sources of nondeterminism. Sadly, getting an accurate, deterministic snapshot of the system is a vital requirement for debugging the chip. Most modern chips have scan chains, which are serial buses that can be used to read out the entire internal state of the chip after a given event. To ensure that such debugging data is deterministic (same data produced for the same input) we need to make a lot of changes to the processor. Otherwise, if we observe different internal states in each run and that too for the same input, debugging a processor will be very difficult. To accurately detect faults, it is necessary to get an exact idea of the internal state of the processor and assess if it is on expected lines or not. In this section, we shall analyze the sources of nondeterminism in a system (with a CPU, memory, and buses), and discuss solutions to increase the amount of determinism.

12.3.1 Sources of Nondeterminism

Nondeterminism in CPUs and Caches

Some sources identified by your author in one of his previous papers [Sarangi et al., 2006a] are as follows: random replacement policies in caches, random choices made by bus arbiters, and uninitialized state elements in the processor. Furthermore, power gating, clock gating, and voltage-frequency scaling can introduce more nondeterminism.

This means that when these mechanisms are turned on, we cannot guarantee that the internal state of the processor will be the same even if we start from a known state and apply the same input. In the debug mode these features need to be turned off. Even if a few of these mechanisms are turned on such as clock gating, we need to somehow ensure that the internal state remains the same.

Nondeterminism in DRAM Memory

The two most common reasons for nondeterministic behavior are DRAM refresh operations, and ECC scrubbing. As we have noted in Chapter 10, refresh operations are scheduled opportunistically based on the memory traffic. Furthermore, in many modern systems refresh operations are scheduled based on runtime conditions, and thus the refresh times tend to vary across runs. The other reason is called *ECC scrubbing*, which means fixing the errors in a DRAM row based on errors indicated by the ECC bits. Such errors can happen because of soft errors. Since such errors cannot be predicted, the time we shall spend on ECC scrubbing is fundamentally nondeterministic.

Nondeterminism in I/O and Interrupts

Systems with mechanical components such as hard disks are extremely unpredictable when it comes to exact timing. Their rotational delay and seek time is dependent on many factors, and cannot be reliably

predicted at the nanosecond level.

Nondeterminism in Buses

This is a major source of nondeterminism at the board level. For example, the buses connecting different chips on the motherboard, or the buses connecting a chip on the motherboard with main memory, can have nondeterministic delays for a variety of reasons. Because of ambient conditions we might see some degree of jitter in the transmission. Additionally, it is possible that the clock of the sender and the clock of the receiver might gradually drift (relatively). Because of such jitter and drift issues, it is possible that let's say in one run we get a message in cycle 10, and in the next run we get a message in cycle 11.

Nondeterminism due to Faults

Lastly we can have nondeterminism due to faults in the circuit such as faults due to inductive noise or soft errors that lead to rollbacks and restarts.

12.3.2 Methods to Enforce Determinism

Deterministic CPUs

There should be a method to set a CPU's internal state to a known state such that a given sequence of inputs will always lead to the same output state. Note that here we are differentiating between the outputs of a program, and the CPU's internal state. Even if the program's outputs are the same, the internal state might still be different for different inputs. Even though this will make no difference externally, it is significant from the point of view of verification and validation engineers. Many processors already have debugging instructions to reset the state of all flip-flops, flush the caches and clean the TLBs. In addition, if we wish to deterministically replay a set of inputs, then we need to also log the exact nature and timing of nondeterministic events such as interrupts, voltage-frequency changes, and clock gating.

Events in the memory and the I/O subsystem can be handled similarly. We simply need to log the exact time of different events, and replay them in a CPU debug run if there is a need.

Deterministic Buses

Ensuring determinism in buses is a hard problem. This is because the sender and receiver might have different clocks that are loosely synchronized with each other and the transmission process itself might have some jitter. First consider the case where their frequencies are the same. In this case, the standard approach is that the sender sends its local cycle count to the receiver. Let us say that the sender's time is t_s as per its local clock, and due to nondeterminism the time at which the receiver gets the message (receiver's local time) can be anywhere between $t_s + \theta_1$ and $t_s + \theta_2$. Then the idea is to delay the message till $t_s + \theta_2$ (the absolute worst case delay) at the receiver. If the receiver reads the message at its own time $t_s + \theta_2$, it is guaranteed to do so in all runs with the same inputs. Note that this approach does increase the latency of a message at the cost of ensuring determinism. It is indeed much easier for engineers to debug and perform post-silicon validation (validate the design after it has been fabricated) if determinism is ensured.

There are however two problems with this scheme. The first is that the two clocks need not have the same frequency. In this case, there are various proposals [Sarangi et al., 2006a, Chen et al., 2012] that slightly modify the basic idea to factor in the effects of different frequencies as long as their ratio remains more or less the same. The second shortcoming is that we need to send a large number (current cycle count) with every message. Note that given that the relative clock skew is a bounded quantity (between θ_1 and θ_2) we can significantly reduce the number of bits that are sent (refer to [Sarangi et al., 2006a]).

12.4 Design Faults

How many of us have actually heard of a bug or a mistake in hardware? The answer is probably very few or maybe none. On the other hand, we all can recount horror stories of software bugs. We all remember that time when our screen went blue, the system was frozen, and maybe the file system crashed – all before an assignment deadline. We have grown up thinking that hardware is always correct. This is sadly not true. Hardware is after all a very long piece of code written in a high-level synthesis language such as Verilog or VHDL. If we can make mistakes in C++ or Java code, we can make mistakes in Verilog or VHDL code as well. With software the vendor can issue patches, but with hardware once there is a bug, almost nothing can be done. Such faults are known as design faults.

Given that faults in hardware are very expensive primarily because they cannot be fixed, or have very expensive workarounds in software, designers spend a disproportionate amount of time in trying to verify and validate their designs. Incidentally, the terms *verification* and *validation* are two widely used terms in this space, and thus they deserve to be more accurately defined. Verification is the process of checking whether a certain part of the design conforms to specifications. It is typically done at design time. In comparison, validation is an activity that is performed after the chip has been fabricated. In this case, we typically provide several arrays of test vectors to a chip as inputs and test to see if the outputs are according to published specifications or not. Validation engineers often test a chip in extreme scenarios and rate its fitness in the field.

After the verification and validation stages, processors are released in the market. Unfortunately, some bugs nevertheless manage to slip through into the final product. Even though we do not get to hear about design faults a lot; however, when they happen, they cause major losses to the company. One of the most widely documented events was a bug in the division unit of Intel's Pentium® processor [Pratt, 1995]. In some rare cases, floating point division produced an error in the fourth (or beyond) decimal digit, which was ultimately traced to incorrect entries in an internal lookup table. This was considered a highly critical bug, especially because division is widely used by Intel's customers, and such errors can accumulate over time. As a result, Intel had to recall all the defective processors and provide replacements. Way back in 1994, Intel had to spend 475 million US dollars to replace all the defective processors. This was a substantial penalty. Another widely reported incident was a bug in AMD's Barcelona and Phenom™ processors, which caused the processors to lock up. The bug was traced to the TLB.

For most bugs of this type, vendors often issue "patches", which are updates to the BIOS, OS, or compiler. They are really invasive in nature, and thoroughly constrain the behavior of the system such that it is not affected by the bug. The resulting performance penalty can be very severe. Hence, it often makes commercial sense to just recall the processors and deliver a new one without the bug – totally free of cost. Given the risks, it is very important for companies to thoroughly vet their designs before releasing a processor. In this section, we shall first look at some common verification and validation steps, then we shall understand the nature of design bugs and how they can be detected in the field, and finally discuss solutions.

12.4.1 Verification and Validation

Verification

Unlike traditional software design, verification and validation are very important steps in hardware design. It is not uncommon to find that 50-60% of the time and a majority of team members are devoted to just checking the correctness of the hardware.

Once the architecture has been decided, and the high level design has been planned, each of the individual components are assigned to different teams. Some companies even outsource this job to third parties. The specifications of each module, and its interaction with other modules needs to be specified very accurately. It is the job of each team to thoroughly verify its module before releasing it to the rest of the company. Many small circuits such as floating point units or adders are formally verified, which

is the gold standard for verification. In this case, the operation of the unit is first described in a formal language that clearly specifies the relationship between the outputs and the inputs. Subsequently, either the circuit or some abstracted version of it is checked to see if the specifications (relationship between the inputs and outputs) hold or not.

There are two approaches in this space: automated theorem proving and model checking. In automated theorem proving we start with a set of axioms, define a theory, and then we check if it is possible to prove that a given circuit obeys a certain property. For example, we can use this approach to prove that the output of an adder is correct. Sadly, such approaches require a lot of skills, and are often very slow. Hence, methods based on model checking are preferred. In this case we create a model of the system using a theoretical language. We start from a known state, and the aim is to show that a given set of states that satisfy a certain property are reachable. The main advantage of this technique is that it produces counterexamples – starting states that lead to finishing states that don't satisfy a certain property. This is a very useful debugging tool because the designers can then see what is wrong with the circuit, and for which inputs a wrong output is produced.

After model checking was proposed in the early 1980s, different methods have been proposed to substantially speed up the process. However, formal verification techniques are still plagued by their slow speed. Hence, it is very common to also test small subsystems via generating test vectors.

For each test vector (initial state), we simulate the circuit and verify if the output is equal to the ideal output or not. The simplest method of verification uses random test vectors. However, now we have many automatic test pattern generators that try to intelligently generate test vectors to maximize the overall coverage. The coverage is defined as the fraction of the input space that has been verified to generate correct outputs. There are two ways to simulate the behavior of circuits with test vectors: RTL simulation, and FPGA prototyping. The latter is a much faster approach as compared to the former.

Validation

After the chip has been manufactured we still need to test it for manufacturing defects. This process is known as *validation*. The first step of testing happens at the wafer level. After the wafer is ready, special probes are attached to dedicated points in the fabricated dies and test vectors are applied. If all the outputs are correct for a given die in the wafer, it is marked to be correct. Here, the main aim is to verify the properties of the fabricated transistors and do some high-level tests on the generated outputs. The next phase of testing is known as burn-in testing. Here the packaged die is tested at elevated temperatures. If it works correctly, it means that it can operate in extreme conditions and is ready to proceed to the next level.

For the next phase a special purpose motherboard is used that allows us to arbitrarily vary the clock frequency, pause the execution, read the internal state of the processor, and reset the processor to a known state. Subsequently, a wide range of test vectors are applied to the chip that are often generated by automatic test pattern generation software. In this phase we use scan chains. A *scan chain* is simply a serial bus that is connected to most of the flip-flops in the chip. It is possible to set their values by sending bits along the scan chain. Furthermore, after setting the initial state of the flip-flops, we can run the chip with representative inputs, and again use the scan chain mechanism to read the state of all the connected flip-flops. We can thus verify if the internal state is correct or not. If there is a discrepancy, this mechanism will accurately tell us which flip-flop's state is erroneous. Another test that is performed in this stage is known as Iddq testing. Here, we verify that in the quiescent state (no switching activity) the supply current (Idd) that is drawn is roughly equal to the total estimated leakage current. If this is not the case, then it means that there is a short circuit from the supply to the ground.

The next phase of validation is called characterization. In this phase, the chip is run at different voltages and frequencies (known as *shmooing*), and a plot is generated for the voltage-frequency pairs at which the chip executes correctly. This is used to characterize the chip and set its operating frequency and voltage along with its DVFS settings. For example, it is possible that different chips produced in the same fabrication facility run at different frequencies because of minute variations in the fabrication

environment. This process of setting the voltage and frequency is also known as *binning*.

Till now all the tests have been *structural*; they basically tested if the circuit was fabricated correctly or not. However, we can also have *functional testing*, where we test if the behavior is as per existing specifications. Regardless of whether the internal state is correct or not, the outputs that are visible to external observers should be correct. Here, the idea is to first set the state of the chip to a deterministic state and then apply test vectors that are possibly associated with long executions. The final output should be correct, only then is the chip deemed to be executing correctly, and is ready to be shipped. After this discussion, the reader should be able to appreciate why it is so important to remove nondeterministic effects.

Sadly, in spite of so much of verification and validation, design faults still slip into products that are in the market. Hence, almost all processor vendors release extensive errata sheets, where they document the nature of the defects, the factors that trigger them, and possible workarounds.

12.4.2 Nature of Design Faults

Types of Faults

To understand the nature of design faults, let us look at a few and try to appreciate the broad patterns. Table 12.1 lists a few design faults taken from publicly released errata sheets.

	Processor	Defect
Fault 1	IBM G3	If the L1 cache suffers a miss, at the same time the processor is flushing the L2 cache, and power management is turned on, then some L2 lines may get corrupted.
Fault 2	Pentium 4	If the following conditions are simultaneously true – there is a cache hit on a line in the M state, a snoop access is going on, there are pending requests to reinitialize the bus – then this can lead to a deadlock.
Fault 3	AMD Athlon 64	If within a window of 4 instructions we have two adjust-after-multiply (AAM) instructions or within a window of 7 instructions we have a DIV instruction and a following AAM instruction, the ALU might produce incorrect results.

Table 12.1: Three design faults [Sarangi et al., 2006b]

Sarangi et al. divide such design faults into three classes: non-critical, critical-complex, and critical-concurrent.

Non-critical Such faults do not affect the correctness of the program. Faults in the performance counters, and debug registers fall in this category.

Critical-concurrent Look at Fault 1 and Fault 2 in Table 12.1. They are fundamentally different from Fault 3 in the sense that all the events need to happen concurrently. For example, for Fault 1, at the very same time we need the L1 cache to be processing a miss, the L2 cache should be in the process of flushing itself, and power management should be turned on. Hence, these are concurrent faults because multiple conditions are active at the same time. In other words, these are simple Boolean combinations of signals, where a *signal* is defined as a micro-architectural event such as an L1 cache miss or the fact that power management is enabled. We typically think of a signal as a Boolean event, hence, we shall make the same assumption here.

Critical-complex Fault 3 is an example of such a fault. Here, we have a dependence in terms of time. The relationship is not purely combinational, there is a temporal dependence. For example, we say that event A needs to happen within n cycles of event B. Such defects are difficult to detect and debug.

We can alternatively classify the defects based on their root cause as proposed by Constantinides et al. [Constantinides et al., 2008].

Logic design faults These faults are caused because of errors in the RTL code, which lead to erroneous logic.

Algorithmic faults Here, the faults are deeper. It means that there are major algorithm (design level) deviations from the specifications. The fixes are not limited to fixing the output of a logic gate, or changing a value from a logical 0 to 1. The workarounds typically involve major modifications to the circuit.

Timing faults In this case the timing paths are not analyzed properly. It is possible that signals either do not reach the latches at the right time, or are sometimes latched too early. We need to either reduce the frequency, make modifications to the circuit to add extra flip-flops or fix the signal latching logic.

Note that the two types of classifications are orthogonal. It is possible that performance counters have timing faults, and critical-complex faults have logic design faults. Whenever, we detect the conditions that might lead to an error we also need to consider if the error has already manifested or not. In many cases, we can stop the error in its tracks before the value propagates to the memory or the register file. In some cases, the error might be restricted to the pipeline, and in this case we simply need to flush the pipeline. However, if the error has propagated to the memory system or the I/O ports, then there is a need to invoke higher level checkpointing mechanisms, or simply let the OS know that there is a need to restart the system.

Type of Workarounds

Once a processor is released with such design faults, they remain with it forever. One option for the vendor is to recall all the processors and then provide the customers with fault-free versions of the processors. This is a very expensive exercise as we have seen in the case of the Intel Pentium bug. As a result, it is a much better idea to anticipate that we will have design faults, and to create circuits within the processor that can be activated to provide fixes and workarounds to these faults as and when they are detected. This implies that even after the processor is released, the vendor continues to verify and validate the design. If any fault is detected there is a need to characterize it, and find the cheapest possible workaround. Often, the faults can be fixed in software. For example, it is possible to instruct the compiler and operating system writers to ensure that certain conditions are avoided.

Let's say that a certain bug is triggered if prefetching is enabled while the power management module is computing the next DVFS configuration, then the operating system can be instructed to not let this happen. It can simply turn off prefetching while computing the next DVFS configuration.

Similarly, in the case of Fault 3, the compiler can just ensure that such instruction sequences are not there in any program. However, note that this is only possible for programs that are compiled after this particular defect has been characterized. To ensure that released codes work correctly on processors with this defect, it is necessary for the vendor to issue periodic patches that change the parts of the binary that have such error-prone instruction sequences. In general, handling design faults in software (OS or compiler) is easy, particularly if the resultant performance loss is limited.

However, in many cases such as Fault 1 and 2, it is not possible to provide fixes in software. There is a need to make changes to the hardware itself. After the processors have been released the hardware cannot be changed. However, if it is possible to at least detect that the conditions for an error are about

to get enabled, we can then try to turn one of the conditions off. Note that this is far more efficient than permanently turning off one of the conditions such as permanently turning off power management or prefetching; in this case the conditions are only turned off for a very short duration of time. Hence, the net impact on the performance loss is minimal.

Even though such approaches look promising, they are bedeviled by the fact that we at least need to add the circuitry to collect all the signals (events of interest) from the chip at design time. This means that the designer needs to anticipate, which part of the chip will suffer from more faults, and then she needs to tap the signals. Examples of such signals are an L1 cache miss, ALU add operation, receiving a snoop message from the bus, etc. Once the signals are available, then it is possible to configure programmable logic arrays that can compute Boolean functions over the signals and give an indication if certain errors are going to happen or not in the near future. If there is a chance of an error, then it is possible to turn off one of the enabling conditions, and then turn it back on after some time. This is essentially a mechanism to provide a *hardware patch*, where even after the processor is released, we can ensure that the triggering event-combinations of certain hardware faults do not occur.

Even though this idea [Sarangi et al., 2006b, Constantinides et al., 2008] is promising; however, it is dependent on the signals that are tapped and furthermore this approach is the most useful for critical-concurrent bugs. To detect conditions that have a temporal relationship between them is hard. Another way of looking at this is that this mechanism can be thought of as a sophisticated scan chain where instead of reading a large number of flip-flops via a serial bus, we focus our attention to fewer Boolean events known as signals. If there is a high likelihood of a fault happening in the areas where the signals are tapped, then we can detect the combination of signals before an error actually manifests. For example, many studies have shown that the power management and cache coherence subsystems typically have a lot of bugs given their complexity. Hence, we can tap more signals from these units.

However, as of 2020 this is primarily a research idea mainly because we have thousands of potential signals on the chip, effectively tapping them and computing Boolean combinations on them is prohibitively expensive. However, interventions are now made at the level of the microcode array and the firmware. Recall that in a CISC instruction set, complex instructions are translated into a sequence of microcodes, and these translations are stored in a microcode array. By changing these translations it is possible to avoid the conditions that activate a design fault.

12.4.3 Using Signals for Debugging and Post-Silicon Validation

The notion of tapping signals can also be used for post-silicon validation and subsequent debugging. The basic insights are the same, which is that we need to create an intelligent version of scan chains where we can tap more signals in regions of the chip that are difficult to verify, and it is expected that they would be involved in a disproportionate number of design faults. For many such situations using traditional scan chains is difficult because we wish to do at-speed debug – run the CPU at native speeds without stalling in the middle. This means that we cannot stall the processor while reading out the scan chain. To run the processor at its native speed, designers insert design-for-debug (DfD) hardware units throughout the chip. These are a network of buffers called *trace buffers*, which are circular buffers that store the values of specific signals and any associated data. If an error is detected, the contents of the trace buffers can be inspected to figure out the set of conditions that led to the error. This is a reverse problem as compared to detecting errors early. In this case we wish to find the conditions that led to a certain error as a part of post-silicon validation.

Here, the most important issue is that the trace buffers have a limited size, and they very soon start to overflow. Hence, there are proposals to only log a subset of signals [Xu and Liu, 2010, Ma et al., 2015] – those that are deemed to be important. We can additionally have runtime techniques where we dynamically filter [Neishaburi and Zilic, 2011] the information that is collected and discard information that is not relevant. Another approach is to compress [Anis and Nicolici, 2007] this information using traditional compression techniques.

Out of this family of approaches, the most popular set of techniques try to filter the information and

only store the values of relevant signals in the trace buffers. This process is known as *Summarization*. We can either have *spatial Summarization*, where a single trace is filtered, or we can have *temporal summarization* where we jointly filter all the traces that have roughly been collected at the same point in time. Note that for both these types of summarization, it is necessary for the validation engineer to indicate the nature of filtering that is desired. The debug hardware has dedicated circuits to filter out this information. For example, for spatial summarization the debug engineer can specify the set of signals that she is interested in. She can also specify the regions of activity that may be of interest. Such regions typically begin with a known signal pattern, and have a similar signature when they end. The DfD hardware can track such regions by monitoring the signals.

In the case of temporal summarization, the approaches for spatial summarization still work. In addition, it is possible to also filter out a lot of events if the processor reaches a given state. For example, let us say that the processor goes to sleep, and this is the expected behavior. Then we can flush all the events in the trace buffers that led to this action. On similar lines, we can define other properties and the trace events that help ensure them. If the property is verified we can delete the corresponding trace events from all the trace buffers.

12.5 Faults due to Parameter Variation

12.5.1 Introduction to Different types of Parameter Variation

Process Variation

Before reading this section it is advisable that the reader gets a very broad idea of the VLSI fabrication process, particularly photolithography. The reader can consult [Glendinning and Helbert, 2012].

In the VLSI fabrication process we create structures on silicon such as wires and transistors. To start with, we coat the silicon wafer (thin circular slice of silicon 200-300 mm in diameter) with a photoresist, which is a light-sensitive material, and then illuminate the photoresist with a *pattern* by placing a photomask in front of a light source. When the photoresist is exposed to light, the parts of it that are exposed change their chemical structure. Subsequently, a solvent is applied to remove those areas that were not exposed to light (the reverse is also possible). This process is known as *etching*. We can then spray dopants (p-type or n-type) that permeate into the exposed regions, and thus we can create doped regions in the silicon substrate. This process can be used to fabricate billions of transistors. Moreover, we need to repeat this process several times for creating a multilayered VLSI chip.

For creating the metal layers, a similar technique can be applied where we deposit metallic vapors into the areas that were not exposed. We can also deposit insulating materials such as silicon dioxide between the layers and etch vias (vertical metallic connections) through it for creating connections across layers.

We need to understand that we are trying to create very small (nanometer level) features on silicon. The dimensions of a transistor as of 2020 are in the range of 10 to 20 nm, whereas the light that is used to etch patterns has a wavelength of 193 nm. This is like trying to make a fine engraving with a very blunt knife. It is obvious that we will not be completely successful and the patterns that will actually be created on silicon will be non-ideal. Additionally, because of the particle nature of light the number of photons that strike different regions of the die will not be uniform, and there will be fluctuations in the dopant density. All of these effects will cumulatively introduce a degree of variation into the transistors' geometrical characteristics. We can have inter-die variation (die-to-die or D2D variation) and within-die (WID) variation. D2D variation happens because of a slight variation in the process parameters across dies. However, we are typically more concerned about WID variation, which can further be classified into two types: systematic variation and random variation. The former is more predictable, and has a degree of spatial correlation, whereas the latter type of variation is of a random nature even at the level of neighboring transistors.

Regardless of the source of process variation, the end result is that the physical and electrical properties of transistors and wires vary across the chip. Most notably, the threshold voltages of transistors change. If the threshold voltage reduces, the transistors become faster and more leaky, and vice versa. We thus have problems of increased leakage power dissipation in some regions of the die and an increased susceptibility to timing faults in other regions because of slow transistors.

Systematic Variation

There are two reasons for systematic variation: CMP (chemical mechanical polishing), and photolithographic effects.

The surface of the wafer has structures with different mechanical properties such as silicon dioxide layers that are hard and the silicon substrate that is soft. As a result, the topography – thickness of the copper and silicon dioxide structures – of a given region of the wafer is dependent on the relative ratio of the amount of silicon dioxide and copper. This amounts to a degree of variation in the geometry and thickness of the devices across a fabricated chip. It is possible to predict the nature of this variation to a large extent, however the mathematical models and the resultant computation is very complex. To reduce this type of variation, designers typically fill up unused regions of the die with metallic fills, such that the regions become more uniform. Subsequently, it is necessary to polish the surface of the wafer using a rotating pad and a mix of chemicals to even out the surface. This is known as chemical mechanical polishing, abbreviated as CMP. This process does not eliminate the degree of variation in transistors' dimensions completely. The residual variation is nonetheless predictable to some extent.

The second reason for systematic variation is because of the relatively high wavelength of light as compared to the size of the fabricated feature. The size of the smallest feature that we can fabricate with adequate quality guarantees is known as the *resolution*. By the Rayleigh criterion, the resolution is proportional to λ/NA, where λ is the wavelength of light, and NA is the numerical aperture. Sadly, despite intensive efforts we have not been able to reduce the wavelength of light beyond 193 nm. Because of this, the features that are fabricated are inaccurate (see Figure 12.7(a)). For example, we observe the rounded corners of shapes that should have been perfectly rectangular. Efforts to reduce the wavelength of light such as extreme ultraviolet (EUV) lithography using light with a wavelength of 13.5 nm have not proven to be successful yet. Because of this limitation, we suffer from effects that are a consequence of the wave nature of light such as diffraction. The photomask essentially contains millions of tiny slits, and light propagating through these slits has a high degree of diffraction, which makes it hard to print sharp features on silicon. This problem is by far more serious than CMP based variation.

Random Variation

We have had an issue with the wave nature of light; we unfortunately also have an issue with the particle nature of light! When we are trying to fabricate a straight line on silicon we want the photon flux to be uniform on the silicon surface. At the nanometer level, there is a certain uncertainty with regard to whether a photon will strike a given region or not in a given window of time. Because of this it is possible that there will be a certain degree of non-uniformity in the photon flux, and thus the corresponding areas of the photoresist where we want to fabricate a straight line will also have a varying degree of exposure. Additionally, during the process of etching it is possible that because of non-uniform acid diffusion, the photoresist is not removed uniformly. Both of these effects lead to *rough lines* as shown in Figure 12.7(b).

There are other problems as well. Current feature sizes are close to 10 nm. They are expected to go down to 5 nm or maybe even 3 nm over the next few years. Since the size of a silicon atom is roughly 0.22 nm, the number of atoms that will be used to build a transistor will be in the tens and not in the thousands or millions. Placing atoms so accurately is very hard, and a few errors are bound to happen. When transistors were large (of the order of microns), these errors had no significance. However, for nanometer scale transistors, they can affect the dimensions of the transistors by as much as 10 to 20%, which can lead to large deviations in the electrical parameters. Unfortunately, we do not have a very accurate way of placing atoms. For example, we have fluctuations in the dopant density and thus a varying number of dopant atoms diffuse through the silicon. This causes random variations even at the

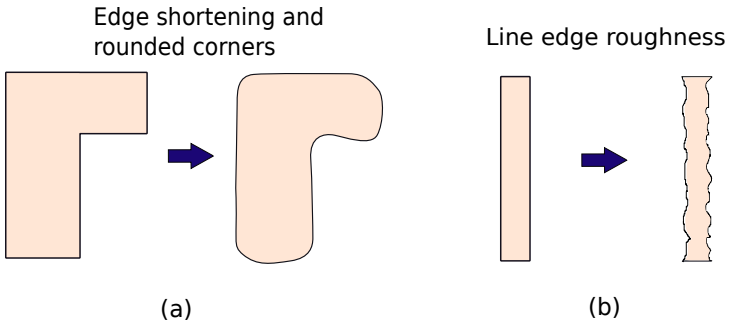

Figure 12.7: (a) Edge shortening and corner rounding and (b) Line edge roughness

level of neighboring transistors. We see a similar effect also in the case of the thickness of the gate oxide, which is of the order of several nanometers. Both of these effects are modeled as random processes.

Voltage and Temperature Variation

Process variation is not the only reason for the electrical characteristics of transistors to change. They can change because of other reasons as well. Two of the notable reasons are variations in the operating temperature and the supply voltage. As we have studied in Chapter 11, the temperature of the chip can vary depending on several factors such as the dynamic power and the ambient temperature. Because of temperature variability, the speed and leakage power can also change. This will lead to the same kind of problems as process variation such as high leakage power and timing faults.

In addition to temperature variations, rapid changes in current requirements can lead to voltage fluctuations on the chip, as discussed in Section 12.2. Changes in the supply voltage can affect the speed of transistors. For example, an increase in the supply voltage makes transistors faster and vice versa. All these sources of variation inclusive of process variation comprise *parameter variation*.

12.5.2 A Mathematical Model of Parameter Variation

We shall describe the Varius [Sarangi et al., 2008] model in this section.

Model for the Gate Delay

The switching time of a gate, T_g, is given by the following equation known as the Alpha power law.

$$T_g \propto \frac{L_{eff} V}{\mu (V - V_{th})^{\alpha}} \tag{12.6}$$

Here, L_{eff} is the effective channel length, V is the supply voltage, V_{th} is the threshold voltage, α is roughly 1.3 (between 1.1 and 1.5 in general), and μ is the mobility of carriers. In this equation process variation affects V_{th} and L_{eff}. It is also obvious to see the effect of the variation of the supply voltage; however, the effect of the variation in temperature applies indirectly. The mobility is typically a function of temperature. In silicon $\mu(T) \propto T^{-1.5}$, which means that with an increase in temperature the mobility decreases and the gate delay increases. Additionally, the threshold voltage, V_{th}, also reduces with increasing temperature (typically at the rate of 2.5mV/°C). We can thus see that with increasing temperature, V_{th} reduces, $V - V_{th}$ increases, and the gate delay reduces. In general, the first factor (mobility) dominates, and thus with increasing temperature the gate delay increases.

Components of Process Variation

We can divide the components of process variation into three parts: die to die, intra-die systematic and intra-die random variation. They are modeled as additive components. Let ΔP denote the deviation in a given parameter such as the threshold voltage due to process variation. Then we have,

$$\Delta P = \Delta P_{D2D} + \Delta P_{WID} = \Delta P_{D2D} + \Delta P_{rand} + \Delta P_{sys} \tag{12.7}$$

The first component is a constant additive component for a given die; hence, most of the modeling effort is focused on the intra-die systematic and random variation. They are modeled as normal distributions.

Systematic Variation

To model systematic variation, we divide the entire chip into a two-dimensional grid. Each point has an x-coordinate and a y-coordinate. Furthermore, we assume that the following parameters have a systematic variation component: the threshold voltage (V_{th}) and the effective channel length (L_{eff}). For the case of systematic variation we assume that V_{th} and L_{eff} are linearly related. The Varius model further assumes that the relative variation in L_{eff} is half of the variation in V_{th}. However, this is a technology dependent effect and might change in future technologies. Given that the systematic variation in these quantities is linearly related we can first compute a systematic variation map that stores the deviation with respect to the mean for each grid cell. Let the mean threshold voltage be V_{th0} and the computed deviation for a grid cell be d. The threshold voltage at the grid cell is given by

$$V_{th} = V_{th0} + d \times V_{th0} \tag{12.8}$$

We typically model the dimensionless quantity d using a two-dimensional multi-variate normal distribution. The formula for such a normal distribution is as follows. Let us consider a k-element vector $\mathbf{X}$ that contains the values of the deviation for each grid cell.

Its probability density function is given by the following equation:

$$pdf(\mathbf{X}) = \frac{1}{\sqrt{(2\pi)^k \mid \mathbf{\Sigma} \mid}} exp\left(-\frac{1}{2}(\mathbf{X}-\boldsymbol{\mu})^T \mathbf{\Sigma}^{-1}(\mathbf{X}-\boldsymbol{\mu})\right) \tag{12.9}$$

Here, $\boldsymbol{\mu}$ is a vector that contains the mean value for each grid cell. We can assume that all its elements have a value equal to 0. The covariance matrix $\mathbf{\Sigma}$ is of special interest to us because it represents the covariance of the deviations across two grid cells. The systematic variation is dependent on its structure. Lastly, the symbol $|\mathbf{\Sigma}|$ refers to the determinant of the $\mathbf{\Sigma}$ matrix.

We make two simplifying assumptions. The first is that the covariance is isotropic (does not depend upon the direction), and second assumption is that it only depends on the Euclidean distance between the grid cells. With these assumptions, we can simplify the modeling as follows. We first note that the covariance is related to the correlation as follows for two random variables U and V.

$$\boldsymbol{\rho}(U,V) = \frac{\mathbf{\Sigma}(U,V)}{\sigma_U \sigma_V} \tag{12.10}$$

Here $\boldsymbol{\rho}$ is the correlation matrix, $\mathbf{\Sigma}$ is the covariance matrix, σ_U is the standard deviation of random variable U, and σ_V is the standard deviation of random variable V. Let U and V represent the deviations at grid cells u and v respectively. If we assume that the standard deviations are the same (equal to σ), which is normally the case, then we have the following formula.

$$\mathbf{\Sigma}(U,V) = \sigma^2 \boldsymbol{\rho}(U,V) \tag{12.11}$$

σ^2 is the variance, which is assumed to be a constant. It is a measure of the systematic variation and will be denoted henceforth as σ_{sys}. The other important takeaway from this equation is that the

correlation matrix and the covariance matrix are related by a factor σ_{sys}^2. It is often easier to compute the correlation matrix, and then use it to compute the covariance matrix.

Many experimental studies have indicated that the correlation matrix has a spherical structure, which means that if the Euclidean distance between two grid cells is r, then their correlation is given by Equation 12.12. Here we consider a function ρ, which yields the correlation between the deviations at two grid cells separated by a distance r. We normalize the distance to the chip's width, which is assumed to be 1.

$$\rho(r) = \begin{cases} r < \phi & 1 - \frac{3r}{2\phi} + \frac{r^3}{2\phi^3} \\ r \geq \phi & 0 \end{cases} \tag{12.12}$$

ϕ is a constant and can be anywhere from 0.1 to 0.5 (assumptions in the Varius model). This means that the correlation is 1 at the grid cell, decreases linearly when r is small, then it decreases sublinearly, and finally becomes 0 (uncorrelated) when $r \geq \phi$. We can consider many such correlation functions to model different kinds of processes. Statistical packages can be used to generate samples from such a multivariate distribution. Each such sample is a variation map.

Random Variation

Random variation in comparison to systematic variation occurs at a far smaller scale – at the level of individual transistors. Furthermore, the variations in V_{th} and L_{eff} are not linearly related anymore. They are modeled as independent quantities.

We typically model both these quantities as univariate normal distributions. The formula for a regular univariate normal distribution is comparatively much simpler.

$$pdf(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(\frac{-(x-\mu)^2}{2\sigma_{rand}^2}\right) \tag{12.13}$$

Here, σ_{rand} is the standard deviation for the randomly varying component.

Putting it all Together

We can use the standard formula for adding the variance of two normal distributions.

$$\sigma_{total} = \sqrt{\sigma_{rand}^2 + \sigma_{sys}^2} \tag{12.14}$$

This will give the net variance for any quantity of interest (V_{th} or L_{eff}). For different values of σ_{total}/μ (standard deviation/mean) we can generate different kinds of variation maps. Most researchers as of 2020 perform simulations assuming $\sigma_{total}/\mu = 0.09$, and $\sigma_{sys} = \sigma_{rand}$.

Given a logic circuit we can create a timing model for it by considering the effects of both random and systematic variation. If the circuit is small we can consider the systematic component to be a constant, and then we can add the delays of each wire and gate on the critical path of the circuit to compute its delay, which in this case will not be a single value because of random variation – it will be a probability distribution instead.

To start with, we model the delay of each gate as a normal distribution. This makes computing the delay of the critical path of the circuit easy because the sum of normally distributed variables also follows a normal distribution. We thus have a distribution for the time taken by a circuit. However, if there are multiple critical paths, then the analysis is more complicated because the time that the circuit will take to compute its result is the time taken by the slowest critical path. Given that the delay in this case is actually a distribution we a priori do not know which critical path will turn out to be the slowest. We thus need to compute the maximum of several normally distributed variables. Even though the delay of each path can be modeled as a normal distribution, the maximum of several normally distributed variables is not normally distributed. It follows the Gumbel distribution. Furthermore, if the circuit is

large and different parts of the circuit have different systematic components, then this effect also needs to be factored. After considering all of these effects we typically arrive at the distribution of the delay of the circuit.

Comparatively, it is far easier to compute the statistical distribution of the delay of an SRAM array. It has a regular structure. Similar to the Cacti model (introduced in Chapter 7), we can break down the components of the total delay, model each one separately, and then add them up to compute the final distribution of the delay of the memory array.

Let us now describe how this model can be used to mitigate the effects of parameter variation.

12.5.3 Methods to Mitigate Parameter Variation at the Architectural Level

Random Variation

We use the inputs of the variation model to compute the distribution of the delay of a circuit (either logic or memory). This is known as statistical static timing analysis (SSTA). Once this is done, we need to place an upper bound on the maximum expected delay of the circuit in a real setting. This means that we need to be conservative and ensure that we give sufficient time to the circuit to compute its result under all kinds of runtime conditions. Hence, while designing the chip we need to ensure that the delays of each unit (in terms of clock cycles) are calculated correctly and account for the worst case.

Systematic and Die-to-Die Variation

Techniques at the Lithographic Level

There are a host of techniques to reduce the amount of process variation because of photolithographic effects. Please refer to Figure 12.8.

Optical Proximity Correction(OPC) The key idea here is that we pre-distort the shape on the photomask such that the shape that is actually printed on silicon resembles the desired output. In Figure 12.8 we show an example where we actually try to print a distorted structure; however, because of the wave nature of light we get the desired rectangle on silicon. Figuring out these OPC features is a very computationally intensive process.

Off-Axis Illumination Here light from the source is incident on the photomask at an oblique angle (it is not perpendicular). Higher diffraction orders strike the surface of the silicon, and thus in layman's terms the pattern on silicon receives more light.

Sub-resolution Assist Features (SRAFs) These are very small structures that we place beside isolated lines on the photomask. Since they are smaller than the resolution limit, they themselves do not get printed on silicon, however the light passing through them forms an interference pattern with the light passing through the main structure and can thus enhance the accuracy of the printing process.

Phase Shift Masking (PSM) It is possible that the light passing through two adjacent lines can constructively interfere and also print a pattern in the intervening space. To avoid this we shift the phase (often by 90°) of the light passing through adjacent structures such that they destructively interfere in the intervening space and do not form any patterns there. To shift the phase we increase the optical path length by changing the transmission properties of adjacent regions in the mask.

Architectural Solutions

Both systematic and die-to-die variation affect the chip at the macroscopic level. This also means that different chips fabricated using the same technology will have different speeds. This is taken care of

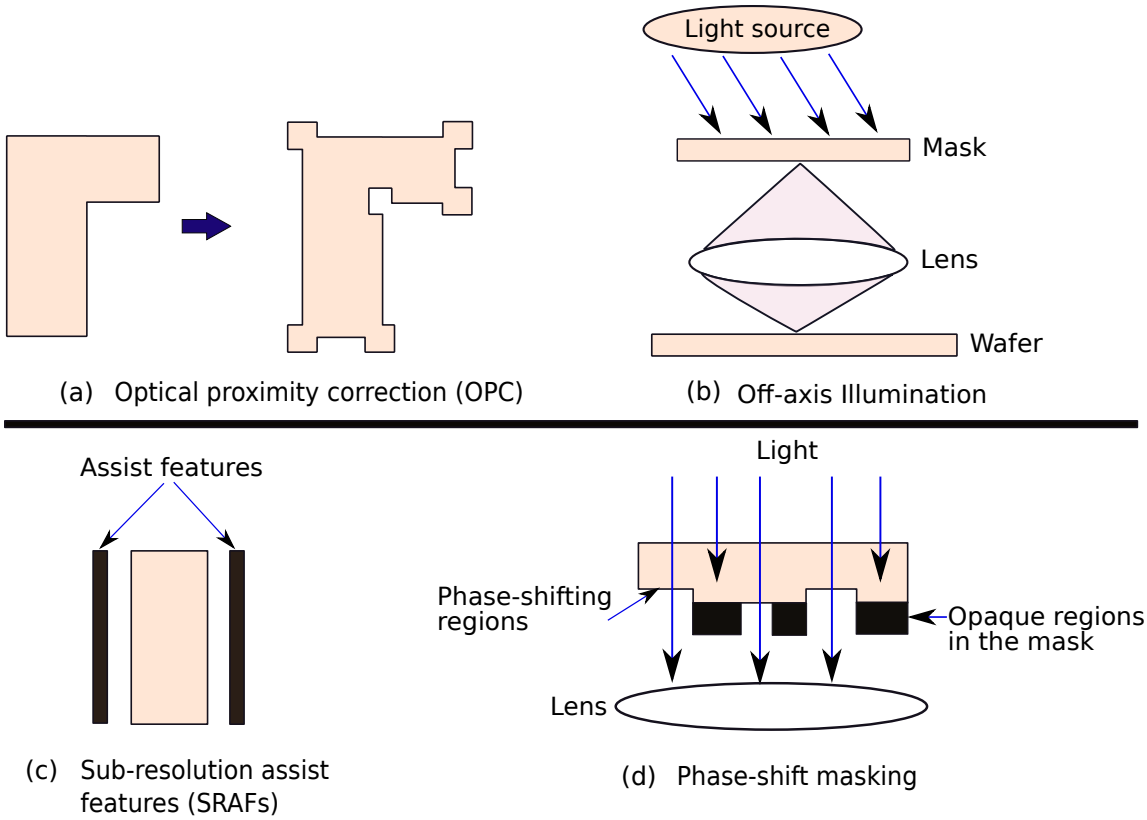

Figure 12.8: Different types of mechanisms to correct variations due to photolithographic effects

during post-fabrication testing where the operating frequency of the chip is determined. As discussed in Section 12.4.1 this process is known as frequency binning. We need to ensure that the chip safely operates at a given frequency regardless of the ambient parameters such as the temperature or the supply voltage (as long as they stay within certain predefined limits). It is further possible to leverage systematic variation by assigning different frequencies to different cores. For example, it is possible that a given region of the chip is faster than other regions. Cores located in the faster regions can have a higher frequency as compared to cores located in the slower regions. However, this is easier said than done because cores in the high frequency regions may also need a higher supply voltage, and furthermore it is hard to synchronize data between cores from the different frequency regions. Finally, with different such voltage-frequency islands within the chip, verification and validation become far more challenging.

12.6 Hard Errors and Aging

Similar to human beings, processors also *age* and finally end up developing permanent faults. Such faults lead to *hard errors*, which means that the circuit suffers from some permanent damage. The damage is not limited to the transistors alone, it is possible that some wires get snapped or some of the I/O connectors develop a very large resistance. This happens gradually over time (*aging*) and ultimately renders the processor non-functional. In this section we shall understand the basic physics underlying the development of hard faults, and the aging processes. Then we shall provide a brief overview of the techniques used to slow down the aging processes and still keep the processor running in the presence of hard errors.

12.6.1 Aging

Negative Bias Temperature Instability (NBTI)

Hydrogen is extensively used in the semiconductor fabrication process. It is not known to react significantly with silicon or change its characteristics, hence it is preferred. However, the effects of hydrogen can be very important at the boundary of the channel and the gate. Recall that the channel is made of silicon with some dopants (depending upon the type of the transistor), and the gate is made of silicon dioxide (SiO_2). Given that silicon has four valence electrons, in the crystalline state it is bonded with four other silicon atoms. However, atoms at the surface have a few dangling bonds, and this is where $Si-H$ bonds form. This is as such a stable state and does not pose any threat to the operation of the transistor.

However, this can induce an aging effect in PMOS transistors [Mahapatra and Parihar, 2018a, Rashkeev et al., 2002]. The mechanism is as follows. When the transistor is enabled, the gate-to-source voltage is negative. This causes some hydrogen atoms to get dislodged and H^+ ions (basically protons) start moving towards the gate. It is however hard to cross the Si-SiO_2 barrier. Hence, they start getting accumulated at the interface. Subsequently, the protons interact with the Si-H bonds, remove the hydrogen nucleus and form a stable hydrogen molecule H_2. This leaves a positive charge at the silicon atoms. Because of a combination of such effects, we have an accumulation of positive charge at the Si-SiO_2 interface. Now, it is possible to have Si-H bonds within the dielectric (SiO_2 layer) that can be filled with holes coming from the channel. This process creates positive charges within the dielectric.

Because of a combination of these effects, the effective threshold voltage rises and thus the transistor degrades. This is known as the *stress phase*. However, the good news is that once the transistor is turned off or disconnected from the power lines, a large part of this effect can be reversed. This is known as the *recovery phase*. However, the reversal is not complete and there is a long-term degradation primarily because H^+ ions get *trapped* at the interface. Many of these traps go away in the recovery phase, however the degradation in terms of an increase in the threshold voltage continues to get worse over time. Traditionally the change in the threshold voltage has been assumed to be proportional to $t_{stress}^{0.25}$, where t_{stress} is the time that the PMOS transistor is under stress (negative gate-to-source voltage) [Tiwari and Torrellas, 2008]. The recovery (in terms of the threshold voltage) is proportional to $(1 - \sqrt{\eta \times t_{rec}/(t_{rec} + t_{stress})})$ where t_{rec} is the time under recovery and η is a constant (assumed to be 0.35 in [Tiwari and Torrellas, 2008]).

Please bear in mind that many of these models are approximate and empirical; they tend to change with changes in technology. In general, we can assume that most of the degradation happens early on, and the degradation rate subsequently slows down (modeled as a power-law relationship). Such effects are particularly important for analog circuits where we need to match the electrical characteristics of neighboring transistors. If the characteristics change then the quality of the output may also change.

Hot Carrier Injection(HCI)

Recall that hot carrier injection is a form of leakage current (Section 11.1.2) where carriers gain sufficient kinetic energy such that they can overcome the potential difference at the channel-gate boundary. This happens with electrons when they gain sufficient kinetic energy because of a positive potential at the gate. They tunnel through the channel-gate boundary and get permanently lodged or trapped in the SiO_2 layer within the gate. This increases the threshold voltage in NMOS transistors. Additionally, charged electrons can also collide with Si-H bonds and change the chemical structure of the Si-SiO_2 interface. This also leads to a change in the transistor's parameters. The rate of degradation (measured as the change in the threshold voltage) is for obvious reasons proportional to the activity factor and the frequency. However, the most important point to note in existing models is that $\Delta V_{th} \propto t^{0.5}$ (t is the time). Unlike NBTI, the recovery is very little, and changes mostly accumulate over time.

12.6.2 Hard Errors

We shall heavily refer to the RAMP reliability model [Srinivasan et al., 2004, Srinivasan et al., 2005] in this section.

Electromigration

In copper and aluminum based interconnects that carry DC current within a chip electrons flow from one direction to the other. Gradually over time because of this mass transport, some momentum gets transferred to the metal atoms and a few of them also start moving along with electrons. This is as such a very slow process but after years of use a wire will appear to thin at the source of electrons and fatten up at the destination. This increases the resistance of the wire at one end gradually over time; the wire can ultimately get snapped and develop a permanent fault. Additionally, atoms can also drift towards nearby conductors causing short circuits (known as *extrusions*). Such failures lead to erroneous internal states that are aptly referred to as hard errors.

This mechanism is known as electromigration. The mean time to failure is given by the following equation, which is known as the Black's equation [Black, 1969].

$$MTTF_{em} \propto (J - J_{crit})^{-n} e^{\frac{E_{aEM}}{kT}} \tag{12.15}$$

J is the current density, J_{crit} is a constant for a given wire and is known as the critical current density, E_{aEM} is the activation energy needed for electromigration, k is the Boltzmann's constant, and T is the absolute temperature in Kelvin. n and E_{aEM} are assumed to be 1.1 and 0.9 respectively for copper interconnects in the RAMP model.

The important point to note is the relationship with temperature. As the temperature increases the mean time to failure reduces. The relationship with the reciprocal of the absolute temperature is exponential.

Stress Migration

Another failure mechanism in interconnects is *stress migration*. First note that in a modern interconnect its contacts are made of different materials, and they have different thermal expansion rates. Because of this some stresses tend to accumulate in the body of the interconnect, and this causes a migration of metal atoms. The subsequent aging and failure mechanism is similar to electromigration. Its MTTF is given by the following equation.

$$MTTF_{sm} \propto \mid T_0 - T \mid^{-n} e^{\frac{E_{aSM}}{kT}} \tag{12.16}$$

Here, T_0 is a very high baseline temperature (typically 500 K), n and E_{aSM} are constants (2.5 and 0.9, respectively, for copper interconnects in the RAMP model), and the connotation of k and T is the same as that for the equation for electromigration. As the temperature increases, $\mid T_0 - T \mid^{-n}$ increases. However, there is a reverse effect where $e^{\frac{E_{aEM}}{kT}}$ decreases. The latter compensates for the former, and thus the overall MTTF reduces. Here also note the dependence with temperature.

Time-dependent Dielectric Breakdown

As the dielectric layer gets thinner because of miniaturization, the tunneling current through it has an increasing impact. It ultimately changes its chemical structure till it ultimately fails and forms a conductive path. The MTTF is inversely proportional to the voltage, V, and a complex function of the temperature, T.

$$MTTF_{tddb} \propto \frac{1}{V}^{a-bT} e^{\frac{X + \frac{Y}{T} + ZT}{kT}} \tag{12.17}$$

The RAMP model uses the following values for the constants: $a = 78$, $b = -0.081$, $X = 0.759\,eV$, $Y = -66.8\,eVK$, $Z = -8.37 \times 10^{-4}\,eV/K$ (eV stands for electron volts).

Thermal Cycling

The chip goes through many thermal cycles. Some of these represent major events such as powering up or powering down the processor (low frequency cycles), and many of these represent minor events that correspond to variations in the temperature because of the variation in the dynamic power consumption (high frequency cycles). The low frequency cycles have a greater effect on the solder joints in the packaging. They alternately expand and contract; this constant thermal cycling causes metal fatigue and the joints fail over time. The resulting MTTF is given by the Coffin-Manson equation [Coffin Jr, 1954, Manson, 1953].

$$MTTF_{tc} \propto \left(\frac{1}{T - T_{ambient}}\right)^q \tag{12.18}$$

T is the average temperature of the packaging, $T_{ambient}$ is the ambient temperature, and q is a material dependent parameter (assumed to be 2.35). Again note the polynomial dependence with temperature.

12.6.3 Failure Rate of the Entire System

Estimating the failure rate of the entire system is difficult because we need to understand how the failures are distributed over time. In general, we assume that the processor is a series-failure system, which means that if one component fails the entire processor is deemed to have failed. There are several models that have been considered for representing failure rates across time. There are studies that have used the exponentially distributed failure rate model or the log-normal failure rate model. This is a complex function of the technology and the usage pattern.

One reason why the academic community has particularly not been very interested in modeling hard errors over time is because most failures in an electronic system follow a bath-tub curve as shown in Figure 12.9. This curve has three components. At the outset, the failure rate is high because processors have many manufacturing defects whose effects are further exacerbated during the burn-in phase (see Section 12.4.1). Once these defects are detected, the processors are discarded. This explains the elevated failure rates in the first phase of the processors' operation, which is basically post-silicon validation. Subsequently, once the processors are released to the market (second phase) they have a period of relatively stable operation. During this time hard errors are not particularly concerning because in most cases processors become obsolete before they actually start developing permanent faults. Towards the end (third phase) of the lifetime of a processor – typically 7 to 10 years after it is first put to use – some processors start developing faults, and then they get finally decommissioned. Sadly, most processors become obsolete before they reach this stage; hence, for most customers hard errors are not particularly concerning.

12.6.4 Methods to Reduce or Tolerate Hard Errors

The silver bullet for reducing the probability of developing such permanent faults is to reduce the operating temperature. As we have seen, most of the failure rates are exponentially dependent on linear functions of the temperature, and thus any temperature management technique will help reduce the incidence of such hard errors.

In addition, there are many solutions at the level of the fabrication process to reduce the effects of failure mechanisms such as electromigration. For example, we can coat copper wires with materials such as TaN (Tantalum nitride). Such protective coatings stop the diffusion of copper atoms to nearby structures.

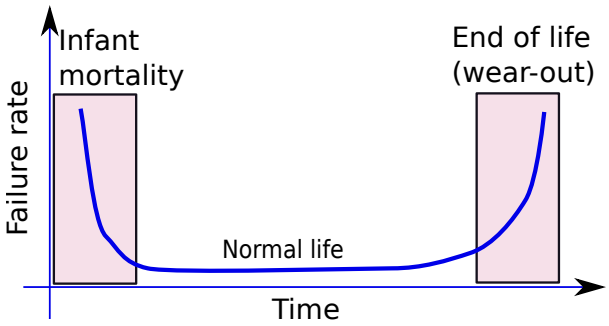

Figure 12.9: Bathtub curve for failures

At the architectural level, most modern processors do not have the level of redundancy that is needed to tolerate such hard errors simply because processors are expected to get obsolete before they develop such problems. The only exception is the memory system, where we always keep a set of spare rows. Whenever, we have a permanent fault in a row we remap it to a spare row.

12.7 Summary and Further Reading

12.7.1 Summary

Summary 11

1. *The three most important terms while studying processor reliability are fault, error, and failure.*
   (a) *A fault is defined as a defect in the system, which might or might not manifest into an erroneous internal state or output. It can either be permanent or transient.*
   (b) *An error is an incorrect internal state.*
   (c) *A failure is an externally visible event where the output or the behavior of the system deviates from its specifications.*
2. *We typically quantify the error rate using the following metrics.*

   FIT *A FIT is defined as one failure per billion hours.*

   MTTF *It is the meantime to failure, assuming that the system cannot be repaired.*

   MTBF *If the system can be repaired, this metric represents the mean time between failures.*

   MTTR *This is the mean time to repair the system.*
3. *We can divide the set of faults into three types: transient, congenital, and ageing related faults. Transient faults are ephemeral in nature, whereas congenital faults are present in the processor right after it is fabricated and packaged, and faults related to ageing develop gradually over time.*
4. *Soft errors that are caused by particle strikes lead to transient faults. Earlier the main sources of soft errors were uranium and thorium impurities, and unstable isotopes such as ^{210}Pb and ^{10}B in the materials used to fabricate the chip. Nowadays, the primary sources of soft errors are impacts from alpha particles and neutrons.*

5. *Soft errors are caused because the particle strike displaces charge causing a current pulse. If this pulse propagates to the latches then it can flip the value of the bit that needs to be stored. It is dependent on the velocity, angle of incidence, and the critical charge Q_{crit}. Q_{crit} is a function of the transistor's characteristics and the output capacitance.*

6. *There are three masking mechanisms for soft errors in circuits: electrical masking (pulse gets attenuated), logical masking (the bit being flipped does not matter), and timing window masking (the pulse does not reach the latch near the relevant clock edge). The process of redesigning circuits to reduce the probability of bit flips due to particle strikes is known as radiation hardening.*

7. *At the architectural level we define two terms: TVF (timing vulnerability factor) and AVF (architectural vulnerability factor). The TVF indicates the probability of the functional unit being turned on, and the AVF is the probability that the soft error leads to an erroneous output.*

8. *Another source of transient errors is the fluctuation in the voltage because of the varying impedance of the power grid (inductive noise). When the variation of the on-chip power requirement matches the resonant frequency, the supply voltage fluctuations are the highest, and this can lead to faults. We thus need to ensure that the resultant voltage fluctuations are limited by throttling activity, or by inserting dummy instructions.*

9. *While trying to debug a processor, we need to ensure that its behavior is deterministic such that the observed errors are reproducible. There are inherent sources of non-determinism in processors such as clock gating in CPUs, scrubbing in memories, jitter in transmission, and delays due to inductive noise or soft errors.*

10. *Design faults (bugs in the RTL) slip into production silicon in spite of extensive verification and post-silicon validation. Such bugs can be classified into three classes: non-critical (defects in performance monitors and debug registers), critical-concurrent (several signals need to be enabled at the same time), and critical-complex (time-dependent relationships such as signal A needs to be enabled k cycles after signal B). A* signal *is defined as an event that can be tapped from a functional unit, and it often has a Boolean value.*

11. *We can use tapped signals to characterize design faults and also detect them as soon as they happen.*

12. *Another set of congenital faults are because of parameter variation, which has three components: process variation, supply voltage variation, and temperature variation.*

    (a) *Process variation is defined as the deviations in transistors' parameters such as the channel length and threshold voltage because of errors induced due to random effects (dopant density fluctuations and line edge roughness) and systematic effects (CMP and photolithographic effects).*

    (b) *The supply voltage and temperature can vary due to inductive noise and due to a change in the dynamic power dissipation respectively.*

    (c) *All the sources of parameter variation can affect the switching speed and leakage power of transistors. The former can cause timing faults.*

13. *Process variation has three components: die to die variation, intra-die systematic variation and intra-die random variation.*

14. *We model systematic variation as a spherically correlated multivariate normal distribution, and we model random variation as a univariate normal distribution.*

15. *The four techniques to address photolithography based systematic variation are optical proximity correction, off-axis illumination, sub-resolution assist features and phase shift masking. For addressing random variation we perform statistical timing analysis and set our timing margins accordingly.*

16. *Two prominent ageing mechanisms are negative bias temperature instability (NBTI) and hot carrier injection (HCI). The former is seen in PMOS transistors when positive charges accumulate at the gate-channel interface. The latter is seen in NMOS transistors where charged electrons change the chemical structure of the interface and lead to charge trapping in the dielectric layer. Both increase the threshold voltage and change the electrical properties of the transistor.*

17. *The prominent mechanisms for inducing hard errors are electromigration, stress migration, time-dependent dielectric breakdown and thermal cycling. All of them are strongly dependent on temperature.*

12.7.2 Further Reading

The original model for soft errors was created by Hazucha and Svensson [Hazucha and Svensson, 2000]. For a slightly better understanding of soft errors, readers can refer to [Baumann, 2005,Wang and Agrawal, 2008]. For an even deeper and holistic treatment of this area, the most comprehensive reference is the book by Mukherjee [Mukherjee, 2011].

For inductive noise some seminal ideas are pipeline damping [Powell and Vijaykumar, 2003a] and muffling [Powell and Vijaykumar, 2003b]. There are similar papers for GPUs [Leng et al., 2014, Leng et al., 2015] as well.

For non-determinism one of the seminal papers is by your author [Sarangi et al., 2006a]. This work has been extended by Chen et al. [Chen et al., 2012]. Similarly, for design faults the first impactful work in the area was also by your author [Sarangi et al., 2006b]. Subsequently, the work was extended to consider signals in a real processor by Constantinides [Constantinides et al., 2008] and Chandran et al. proposed a method to capture critical-complex faults by proposing a theoretical framework [Chandran et al., 2017].

Your author had the good fortune to be involved in creating the first publicly available toolkit to simulate parameter variation. It is the Varius toolkit [Sarangi et al., 2008]. For work on the circuits side, readers can refer to the work by Kuhn et al. [Kuhn et al., 2011], where they propose techniques to measure the degree of variation, and present a detailed characterization of the results. There are many architectural techniques to deal with process variation. Mittal's survey [Mittal, 2016a] is a comprehensive reference in this space.

One of the early papers in the area of ageing was Facelift [Tiwari and Torrellas, 2008] that introduced NBTI and HCI. However, the models and mechanisms have subsequently changed. A more recent reference in this area is a paper by Mahapatra et al. [Mahapatra and Parihar, 2018b]. For hard errors the two classic references are the papers by Srinivasan [Srinivasan et al., 2004, Srinivasan et al., 2005].

Exercises

Ex. 1 — Let us consider a processor that has a large physical register file (let's say 150 entries). The designers considered it a very expensive idea to have ECC (error correcting code) protection for each entry because of area constraints. However, they later realized that they would like to have some

protection against soft errors for register operands as well. The protection mechanism need not cover all the accesses, but covering 60-70% of the accesses is desirable. How can we ensure this?

Ex. 2 — Assume that a bit has been flipped because of a soft error. How can we recover the state if the error has propagated to any of the following components?

1.Logic in the same functional unit.

2.Other stages in the pipeline.

3.L1 cache.

4.L2 cache.

Ex. 3 — The dynamic instruction verification architecture (DIVA) is a checker architecture in which a small checker processor is used to verify the execution of the master processor. Implement the DIVA architecture using the Tejas architectural simulator and quantify the loss in performance for the SPEC benchmarks.

Ex. 4 — Extend the Varius toolkit to model hard errors and ageing.