technische universität
dortmund

Diploma Thesis

A Cross-Layer Analysis of Error Impact on the Instruction Execution in an 8-bit Microprocessor

Ingo Korb

July 16, 2012

Advisors:
Dr. Michael Engel
Prof. Dr. Peter Marwedel

Faculty of Computer Science XII
Embedded Systems (LS12)
Technische Universität Dortmund
http://ls12-www.cs.tu-dortmund.de

Contents

1 Introduction **1**
1.1 Goals . . . 2
1.2 Related Work . . . 2

2 Hardware and Error Model Fundamentals **5**
2.1 Soft Errors . . . 5
2.2 Switch-Level Simulation . . . 6
2.3 Visual6502 . . . 9
2.4 The 6502 microprocessor . . . 10
2.4.1 Programming model . . . 11
2.4.2 Selected implementation details . . . 13

3 Simulation and Analysis Software **15**
3.1 Design . . . 15
3.2 Simulator . . . 15
3.2.1 Original version . . . 15
3.2.2 Choice of Fault Model . . . 17
3.2.3 Implementing Soft Error Simulation . . . 19
3.2.4 Simulation result file size . . . 22
3.3 Analyser . . . 23
3.3.1 Checks in the Analyser . . . 24
3.3.2 Further Ideas . . . 27
3.4 Statistics Aggregation . . . 28

4 Evaluation **29**
4.1 Implementation Bugs . . . 29
4.2 Data Sets . . . 30
4.3 Basic Results . . . 32
4.3.1 No (visible) Effect . . . 32
4.3.2 Deadlocks . . . 36
4.3.3 Simulation Errors . . . 39
4.4 Simple Errors . . . 39
4.5 Interrupts . . . 47
4.6 Control- and Data Flow . . . 50
4.7 Instruction Overwrite . . . 56
4.8 Effect Delay . . . 59
4.9 Alternative Fault Models . . . 61
4.10 Other Uncategorised Runs . . . 61

5 Summary **65**

Appendix **67**
A Three-level simulation issues 67
B 6502 Instruction Set 71

Bibliography **77**

List of Figures **79**

List of Tables **81**

1 Introduction

Soft errors are a type of transient faults in integrated circuits. They can be introduced by an external charged particle impacting on the silicon and have a long history in computing: One of the earliest observations of this problem was in 1978 when dynamic RAMs produced by Intel exhibited increased bit error rates. The problem was later traced back to contaminants in the ceramic packaging of those chips which emitted alpha radiation, occasionally flipping bits in the memory [19]. Although this particular problem was fixed by reducing contaminations, it alerted both the commercial and scientific community to the general problem of random, temporary malfunctions in circuits caused by high-energy particle strikes. Later it was theorised that cosmic rays also contribute to this problem as an unavoidable natural source. This theory was validated when flights at varying altitudes were used to investigate the error probability for static memory and a correlation between the altitude and error rate could be shown [26][25]. Dealing with this "natural source of errors" requires circuits which are designed to be tolerant against particle strikes.

The chance that any given particle strike can cause a soft error is influenced both by the design of the chip as well as the technology it is built with. Previous works have shown that the soft error rate can be expected to increase in future chips with smaller feature sizes and lower operating voltages [9] and the International Technology Roadmap for Semiconductors lists an "acute need for verification techniques than can provide information about a system's reliability in presence of soft errors (...)" in its 2011 report [12].

There are many works that study the effects of soft errors on the operation of microprocessors in varying levels of detail, from simulations of the analog behaviour of partial circuits to high-level simulations that observe the effects of bit-flips in CPU registers or memory on a simulated system or a selection of programs. The computational resources required to run a model of a particular detail level vary greatly depending on the model — generally the closer it is to the hardware the more computations are required. This means that large-scale simulations of a complete CPU are still not feasible below a certain level of modelling. The current work uses a relatively low-level approach based on switch-level simulation, which simplifies the circuit of a chip by assuming that its transistors are ideal voltage-controlled switches. The circuit chosen for the simulation is a complete MOS 6502 CPU which was reverse-engineered

by the Visual6502 project [14], which also provided the base for the simulator used in this work. The 6502 is still small enough that it is feasible to simulate a large number of soft errors on single instructions with current desktop-class machines within a reasonable amount of time. Analysing the effects of soft errors on full program runs would also be within reach of this simulation model/CPU circuit combination, but this work instead focusses on analysis methods to automatically categorise the changes introduced on single instructions. The idea is to determine if it is possible to categorise a large subset of these changes and to characterise instructions based on their susceptibility to soft errors.

1.1 Goals

The goal of this work is to analyse the instruction execution of a simple 8-bit microprocessor in the presence of soft errors and to categorise the changes these errors introduce. The focus is on single instructions so that the results are independent of the program running on the processor and also to determine the vulnerability of each instruction to the various error categories.

1.2 Related Work

The effects of soft errors on microprocessors and related circuits have been studied on many different levels and with different approaches. One of the most direct ones is to expose a chip to radiation and observe the effects. An example of this can be found in [17] where a system with two simultaneously operating MC6809E microprocessors was studied. One CPU was exposed to radiation, the other ran as a reference. Any change in the output signals between the two processors was captured and analysed. When using a simulation instead of actual hardware, the detail level of the simulation can be varied. One example for a full system simulation is the examination of an IBM PC RT in [8]. The paper describes an experiment in which a gate-level representation of the PC RT is simulated with transient single stuck line faults. The faults are injected into a small set of hand-selected nodes during the execution of two test workloads. Changes are detected by comparison with a known-good execution as well as the error-detection mechanisms offered by the PC RT and observed outcome of the simulation is classified into several categories according to the errors that are detected. A similar approach is used in [27] which explores the vulnerability of a PowerPC processor. In that work faults are injected in single bits of latches and the processor state is compared to a known correct trace. The classification of errors is aided by applying a tool that was also

used for verifying the processor during its design which can map the detected differences to microarchitectural features.

Studies of transient errors are not limited to just the CPU, simulated failures in the memory subsystem are also studied. One example is [24] which examines failures in a simulated embedded control system caused by transient faults in its SRAM. The work is based around a full system simulation with an additional bridge between the CPU core and its memory which created failures according to a preprogrammed pattern list. To generate these patterns, a library of SRAM behaviours was generated using SPICE simulations with changing conditions like temperature, supply noise and varying gate threshold voltages. Based on this library a tool creates a suitable pattern of errors for the conditions that are to be simulated.

On the switch-level side the basic algorithm described in [6] has been refined and adapted for the simulation of transient or permanent faults multiple times. The basic digital approach has been refined to include approximations of analog behaviour, for example by using piecewise linear models in [16] to improve the modelling of resistances and capacitances in the circuit. Others have taken advanced versions of switch-level modelling that were already extended to model both an unknown and a high-impedance logic state and added transient fault injection with varying fault strengths on top [15]. Mappings of switch-level simulations to FPGAs have been developed and extended for fault injection [10], increasing the simulation speed by synthesising a specialised simulator on the FPGA that can calculate the circuit transition with increased parallelism.

This work intends to extend the accurate modelling provided by switch-level simulation to a full CPU instead of just a part of it. By developing a fully automated error categorisation combined with a full coverage of tested instructions, soft error injection locations and injection times over a large number of starting conditions the effects of soft errors on a CPU can be examined in more detail than approaches based on higher-level simulations could provide. The selection of single instructions as target for the examination allows for a more detailed categorisation and avoids masking effects that are introduced when full programs instead of single instructions are tested.

2 Hardware and Error Model Fundamentals

This chapter describes the generation of soft errors by particle strikes and give an introduction to switch-level modelling and the 6502 microprocessor.

2.1 Soft Errors

A soft error in a circuit is a transient fault that cannot be traced back to incorrect external inputs, a design mistake in the circuit or a permanent failure. On the chip level such an error can be introduced by particle strikes, in particular alpha particles and neutrons.

Alpha particles consist of two protons and two neutrons without any electrons, so they are identical to an ionised Helium atom (He^{2+}). Alpha particles are easy to shield against: The average distance an alpha particle can travel within air without being absorbed is just a few centimetres, this distance shrinks even more in denser matter. It follows that integrated circuits can only be affected by alpha particle sources that are found within the packaging of the chip or the chip itself because the circuit is shielded against other outside sources. Although these sources are today considered contaminants and are avoided as much as possible, the mechanism by which alpha particles create soft errors is still relevant.

When an alpha particle travels through silicon, it leaves behind a track of electron-hole-pairs [19]. In the case that prompted that paper, the device was a DRAM chip and errors were induced when these separated charges were collected in the capacitor of a DRAM cell. The effects of particle strikes are not limited to DRAMs or even to storage nodes in general — the charge can also generate an impulse on a connection leading to a logic gate which may result in a temporary change of the input of the gate [22]. The strength and duration of the impulse depend both on the energy of the particle as well as the construction of the circuit.

Even if alpha particle sources in packaging could be completely removed, soft errors would not be eliminated completely. Cosmic radiation constantly showers the earth's surface with particles, most of which are protons [22]. These interact with atoms in the atmosphere, creating cascades of secondary particles. Although not all of these secondary particles reach the earth's surface, they are the main source of cosmic radiation-induced neutrons at ground

level. Neutrons are not charged themselves and thus cannot inject charge in a circuit. Instead they can collide with the nucleus of other atoms, creating yet another cascade of particles. Some of these are charged and can then influence an integrated circuit similar to an alpha particle. The chance that a neutron collides with the nucleus of an atom is very small, so the soft error probability per particle is much lower compared to alpha particles. This reduced chance of interaction with other matter also implies that shielding against neutron flux is harder compared to alpha particles. Shielding against either type works because the particles are absorbed within the shielding material and do not reach whatever is shielded by it, but with a reduced interaction rate the effectiveness of the shielding material lessens. Unless a chip designer can convince their customer that a few meters of concrete around the chip for neutron shielding are a desirable feature he will need to design the chip with soft error resilience in mind.

A much more in-depth discussion of particle strikes in integrated circuits can be found in [22] which was used as the basis for this summary.

2.2 Switch-Level Simulation

Switch-level simulation is a modelling approach for MOS logic circuits developed in the early 1980s to overcome limitations of the gate-level simulators used at that time. The following overview is based on the PhD thesis of R. Bryant [6] who was among the first to introduce the idea in his 1980 paper [5].

Before switch-level simulation was introduced, MOS logic designers had the choice between analog and gate-level simulation of their designs. Analog simulation — for example using the SPICE simulator [23] — provides a high-fidelity result using a representation of the actual circuit implemented on the chip as input data. It is extremely computationally intensive and thus could only be used for small circuits and/or a small number of simulations of that circuit. Gate-level simulation abstracts away all the analog details of a circuit by just considering the logic functions the circuit is meant to implement. Such a simulator requires less computational resources, but it cannot handle all of the circuit features available to a logic designer at that time. One example of this is the use of a single MOSFET transistor as a bidirectional switch, another is the use of circuit capacitances as a storage element, for example a DRAM cell. Extensions were developed for gate-level simulators to allow the simulation of circuits with these effects [7], but they required special input from the designer to mark these features or replace them with special elements provided by the simulator. This creates a discrepancy between the circuit that is simulated and the circuit that is actually implemented.

The basic idea of switch-level simulation is to use the same circuit structure in the chip and in the simulation while reducing the computational requirements compared to analog simulation by simplifying the behaviour of the devices and signals in the circuit. Instead of a fully analog MOSFET with a threshold voltage and a linear operating region, the transistors are either on or off, depending on the signal at the gate — a voltage-controlled switch. To model the effects of pull-ups and other features that rely on one signal overriding another, transistors also have a strength from a set of discrete, ordered values. Two strengths are sufficient to model circuits that use pull-ups/pull-downs as their only source of "override-able" signals. Another implicit strength level is introduced by the connections between the transistors. Because each of the connecting sets of traces — usually called nodes in switch-level simulation, following the terminology used in SPICE — has a stray capacitance, it is able to store a small charge. The various capacitances of those nodes (as well as any additional capacitances deliberately designed into the circuit, for example the capacitor of a DRAM cell) are assigned a size from a set of discrete, ordered values. In many circuits a single node size is sufficient for successful simulation. In some cases two or more levels may be needed, for example when the circuit charges a DRAM capacitor (small capacity) from a precharged bus (large capacity). Furthermore the simulation needs to distinguish between two types of nodes: Input nodes, which are externally-provided signals from the simulation environment into the simulated circuit and normal nodes whose state is defined by the network. The set of input nodes includes at least the power and ground supply rails but other input-only signals may be included.

Signals are also simplified to a few discrete values, at minimum 0 and 1 for low and high signal levels. A third signal level X is a common extension which is used as a temporary state for changing input signals, signifying that the signal temporarily moves through an interval of voltages that are neither clearly 0 nor 1. Combined with suitable propagation rules this enables the simulation to uncover hazards and races within the circuit, but it also causes problems when simulating asynchronous designs or circuits that deliberately exploit propagation delays.

Finally time is also simplified for faster simulation. Although it would in principle be feasible to build a switch-level simulator with a continuous time model, the value of doing so is questionable. The exact time a MOSFET in a logic circuit switches is hard to define because a rising gate voltage will put it into linear mode first, with a drain-source resistance depending on the gate voltage. Additionally the timing of the gate voltage function depends on the resistive and capacitive properties of the node the gate is connected to, two details which are mostly abstracted away in a switch-level simulator. A more suitable approach is an unit delay timing model using discrete time steps. In each step the simulator first calculates the entire new state of the circuit based on its current state. When this new state is completely

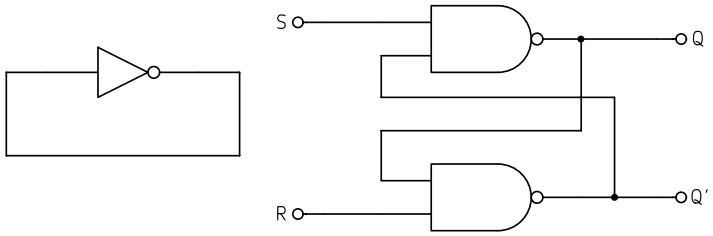


Figure 2.1: Examples of problematic circuits for unit-delay switch level simulation; left: inverter ring; right: two-NAND flipflop

calculated, the simulator switches over to it all at once. This model changes the state of a transistor one simulation cycle after its gate has changed, thus creating a single time unit of delay in its response. Simulation is terminated when the circuit has settled, i.e. when the new state is identical to the one it replaces. Certain circuit features may prevent this termination condition, two simple examples are shown in figure 2.1. Odd-length inverter rings will oscillate infinitely, so the circuit never settles into a stable configuration. Sub-circuits which can only settle on an output value if their components do not have perfectly matched delay times also prevent the simulation from reaching a stable state, even though an actual implementation may work fine. A classic example for this is a two-NAND flipflop that received simultaneous 0-1 transitions on ins set and reset inputs.

This problem can be partially avoided in a pseudo unit delay timing model. Instead of calculating the entire new state of the circuit, the simulator uses an event queue which lists nodes whose state needs to be recalculated. The simulation cycle removes one node from this queue and recalculates the node state based on the current circuit state. Then the circuit is updated and all nodes that may need to be recalculated because they are connected to transistors that changed state are enqueued. The advantage of this model is that it can correctly process some circuits that a unit-delay time model cannot because it effectively serialises simultaneous changes, but the order in which this serialisation occurs is only defined by the simulator implementation. The simulation terminates when the event queue is empty, indicating a new stable state in the circuit.

There are also methods for implementing a switch-level simulation on an FPGA that allow increased simulation speeds by parallelising the simulation. This approach was not used in this work because it is very resource-intensive: An existing project that maps the 6502 net list used for this work onto an FPGA [20] is just about fast enough for replacing an actual 1MHz 6502 with an FPGA, although with some incompatibilities caused by timing differences. The project uses preprocessing of the net list to detect various common structures like gates, latches and muxes and replaces them with native Verilog constructs. Even with these simplifications of the net list, the output still requires 54% of the slices of a Xilinx Spartan

3E-500 FPGA; when they are disabled the synthesis tools report a 552% slice utilisation. For a simulation with soft-error injection these optimisations would need to be disabled and additional logic for injecting the soft errors would increase the circuit size even more. Although there are FPGAs on the market that are large enough for this purpose, the FPGA approach was not pursued any further because a software-based simulation allowed for much faster edit-compile-test cycles compared to the FPGA synthesis time of more than one hour.

2.3 Visual6502

The Visual6502 project was started in 2009 by Greg James, Barry Silverman and Brian Silverman in order to preserve, document and understand historic computer systems, and to present them in a highly visual manner [13].

Their initial target was the MOS6502 processor. A sample of the chip was decapped, photographed under a microscope and the resulting image of the die was then redrawn to generate polygons for each trace in the six layers of the original chip. A net list was extracted from those polygons, allowing for a switch-level simulation of the original chip. These results were first presented at SIGGRAPH 2010 [14]. An interactive simulator (shown in figure **??** showing the current state of each node as an overlay to the layout is available on the project's web site[1] and its full source code as well as the data set that drives it was released as open source.

Unfortunately this method does not recover data about the characteristics of the chip production process beyond the basic geometry. This means that even though the relative sizes of each element are known, resistances and capacitances can only be guessed. Because of this missing data, basic switch-level simulation is the lowest-level simulation that can reasonably be used with this net list, anything that needs more detail about the analog behaviour of the circuit would have to rely on guesses for the missing information.

In the form released by the Visual6502 project and used by this work, the NMOS version of the 6502 consists of 3510 transistors plus 1018 pull-ups, connected in what appear to be 1725 nodes at first glance. Closer inspection of the data set reveals that 21 node numbers never appear in the data set. Additionally there are five nodes that appear only once in the data set, i.e. they are connected only to one pin of one transistor and nothing else. Because all of them are connected in a way that does not influence the rest of the circuit[2] as shown in figure

[1] `http://www.visual6502.org`

[2] These may be remains of the 6501, a variant of the 6502 that was pin-compatible with the Motorola 6800 and which had to be removed from the market after a lawsuit. In die shots it appears that the selection between a 6501 and 6502 was made by changing just the final metal layer, so small bits of apparently-unused circuit are expected.

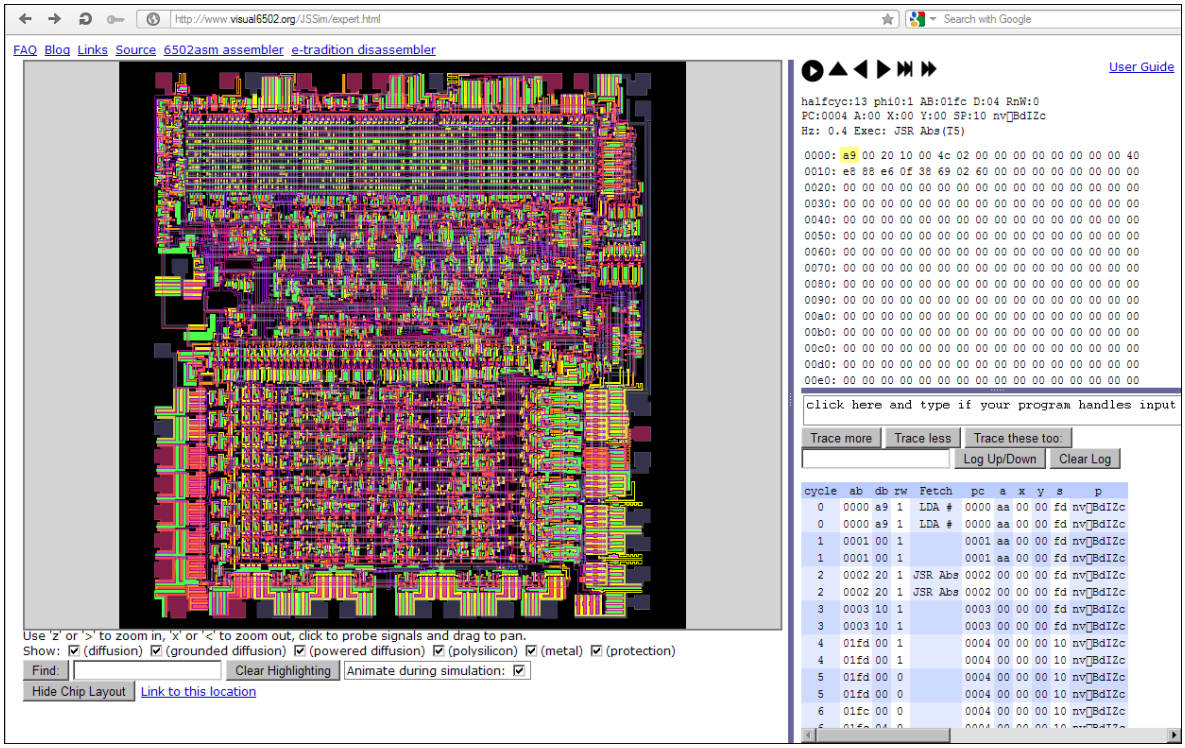


Figure 2.2: Screenshot of the interactive 6502 simulator from the Visual6502 project

2.3 they have also been removed from consideration. Any percentages in this work that refer to the set of nodes are therefore based on a count of 1699 nodes.

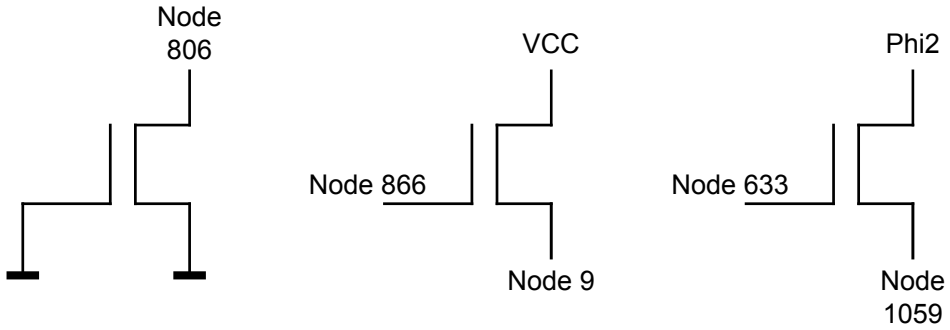


Figure 2.3: Connections of the single-connected nodes in the net list

2.4 The 6502 microprocessor

The 6502 microprocessor was designed by a team of engineers at MOS Technology in 1974-1975 after leaving Motorola with the explicit goal to create a low-cost 8 bit microprocessor

[4]. This goal was reached, the CPU debuted for a price of just 25 USD when other 8-bit microprocessors (e.g. 6800, 8080) were marketed for over 100 USD. The 6502 not only powered the first home computers like the Commodore PET, BBC Micro, Atari 400 and Apple I/II and some quite successful game consoles (Atari 2600, Nintendo Entertainment System and others), but the CPU and its successors were used for various embedded control applications in the following decades. For example some key chain LCD "photo frames" use a 6502-derived CPU internally and 65C02 CPUs (a feature-enhanced CMOS version) are still available commercially as a discrete chip as of mid-2012 [1]. The following sections will give an overview over the programming model of the 6502 and highlight some of its implementation details that are important for the current work.

2.4.1 Programming model

The 6502 is an 8-bit CPU with a 16-bit address bus. It features no segmentation, internal bank switching or separate address spaces, so it can access only 65536 bytes of memory and all I/O devices must be memory-mapped. The chip uses single-operand instructions and a single accumulator (A) plus two index registers (X and Y), all of which are 8 bit wide. Figure 2.4 shows a simplified overview of this internal architecture taken from the MOS manuals, a much more detailed diagram can be found in [11]. There is no way to combine two of these registers into a 16-bit value, so pointers used in indirect memory access must be stored in memory. The instruction set restricts the location of these pointers to the first 256 bytes of memory, the so-called zero page. The stack pointer is also only 8 bits wide, the stack is hardwired to use the second 256 bytes of memory (0x0100-0x01ff). Instructions are encoded in a single byte opcode followed by zero, one or two bytes for their parameter, but only 151 of the 256 possible opcodes are listed in the official documentation [21]. The function of the other 105 opcodes arises from incomplete decoding and ranges from various forms of NOP to combinations of existing instructions to twelve opcodes that cause a complete deadlock of the CPU that is only recoverable by an external reset.

In addition to the mentioned registers the 6502 also has a flag "register" which combines six flag bits within the CPU. One of them is used to enable or disable interrupts, another switches the ALU to decimal mode in which the add and subtract instructions use binary-coded decimals instead of standard binary numbers. The other four flag bits are Negative, Overflow, Zero and Carry. Overflow and Carry are updated in response to addition and subtraction operations to indicate an overflow from bit 7 to 8 (Carry) or bit 6 to bit 7 (Overflow). Overflow can also be set externally by a transition of the SO (Set Overflow) pin which was intended for very low latency event signalling. The Negative and Zero flags are updated by almost every instruction that handles data except for stores which allows for

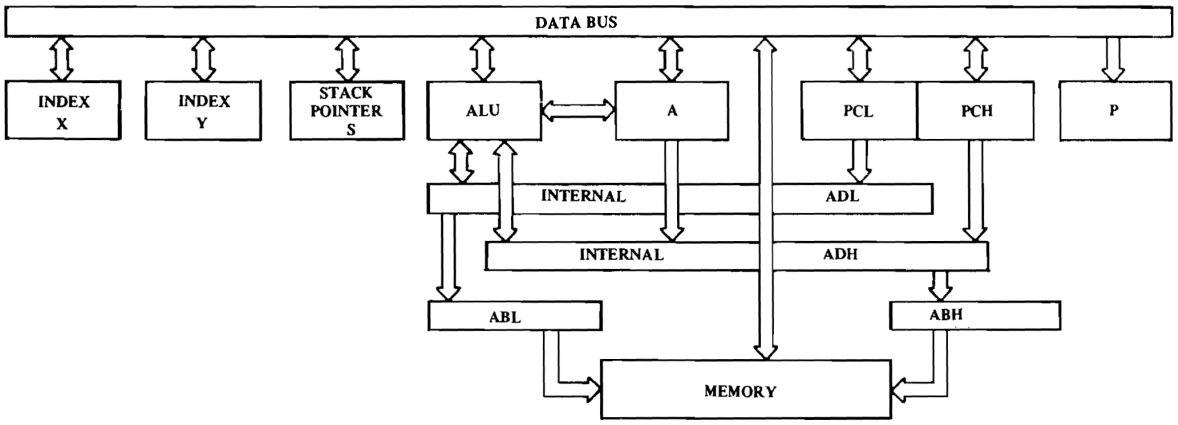


Figure 2.4: Simplified internal architecture of the 6502, from the MOS manual

very efficient loop counters in the X and Y registers when their single-byte increment and decrement instructions are used.

6502 instructions are usually classified into twelve addressing modes which are described in detail in appendix B. The instruction set is not orthogonal – for example the `CMP` instruction (calculates $A - value$ and updates the flags, but discards the result) is available with eight of those addressing modes, but the `BIT` instruction (calculates $A \wedge value$ and updates the flags, but discards the result) is only available with zero-page and absolute addressing. A full description of the instruction set is included in appendix B.

Instructions on the 6502 generally need as many cycles to execute as they need memory accesses, although there are some exceptions. The most important one is that no instruction can complete in less than two cycles. The second exception that is relevant to this work is that address calculations that have an overflow from the low to the high address byte need one additional "penalty" cycle as the ALU can only update one eight-bit value per clock cycle.

There are two kinds of interrupts in the 6502 — maskable (IRQ) and non-maskable (NMI). Both are triggered by their own dedicated signal line and cause the CPU to push the current program counter and the flags register to the stack before continuing execution at the address stored in the vector for the interrupt — 0xfffa/0xfffb for NMI and 0xfffe/ffff for an IRQ. The address stored at 0xfffc/0xfffd is used as the initial program counter after a reset. An IRQ can also be triggered by the `BRK` instruction which can be distinguished from a hardware IRQ by a bit in the flag register value that is pushed to the stack. The `BRK` instruction also differs from a hardware interrupt in the return address on the stack: While a hardware interrupt should resume execution at the interrupted instruction, returning from a `BRK`-induced interrupt resumes two bytes after the `BRK` opcode. The main use of the `BRK` instruction is as a breakpoint for debugging. Although it would be possible to use it as a

syscall instruction, storing the syscall number in the skipped byte, the author is not aware of any system that takes advantage of this.

2.4.2 Selected implementation details

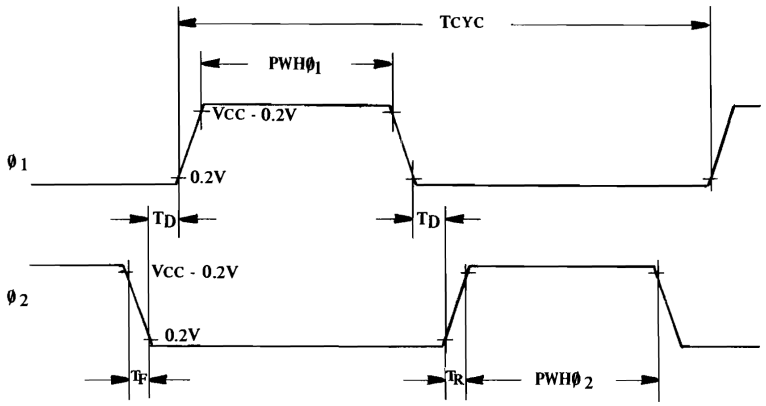
The 6502 decodes instructions by a PLA combined with random control logic, sequenced by a modified one-hot counter called the T-state counter. The twelve instructions that deadlock the CPU arise from the combination of the counter and the PLA: Although the longest instructions are seven cycles long, only six counter stages are implemented, the seventh state is created by dropping the hot bit from the counter. With the documented instructions this is not a problem because the PLA will reset the counter in that state to prepare for the next instruction, but the twelve deadlock-instructions never trigger a counter reset in the PLA. Because interrupts are only handled when the 6502 fetches a new instruction the result is a complete CPU deadlock. The counter can also be frozen by an external signal to generate wait states, but this is limited to memory read cycles.

Interrupt handling in the 6502 is based on the `BRK` instruction. When external hardware has signalled an interrupt and a new instruction is fetched, the fetched opcode is discarded and the instruction register is forced to 0x00 (the opcode of `BRK`) instead. Minor modifications are made to the execution of the instruction to avoid incrementing the program counter, to push a flag register value to the stack that does not indicate the `BRK` instruction and to select the correct vector for the interrupt. Reset is also handled as an interrupt internally, but the stack writes are changed to read cycles instead (the data is discarded).

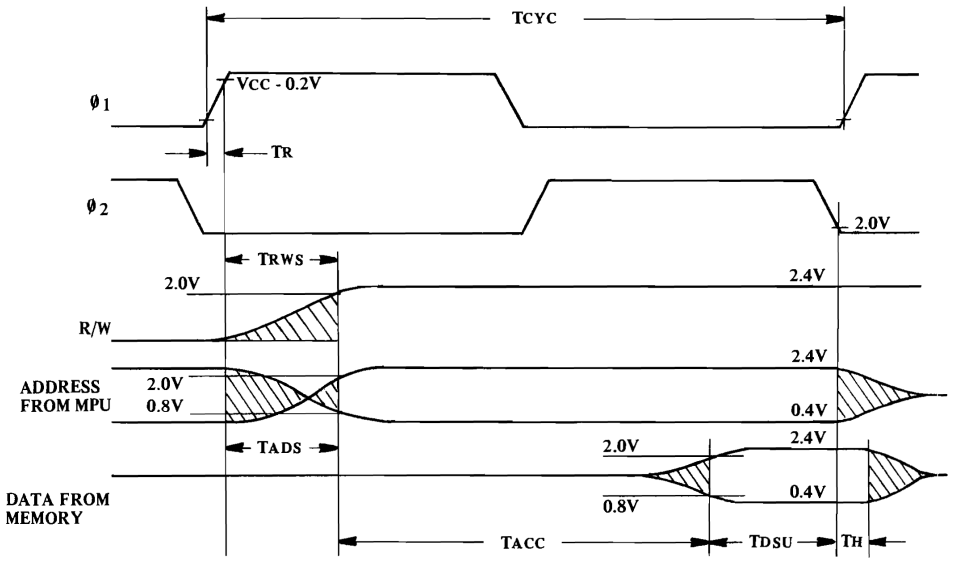
The 6502 generates two internal non-overlapping clock signals called Phi1 and Phi2 from the external clock Phi0. This is the main reason why the chip cannot be successfully simulated in a three-state switch level simulator that uses an X state – a transition of the external clock would first flood both internal clock nets with the X state before switching both clock signals to their final value, thus violating the non-overlapping condition. This causes major problems in the operation of the chip, an example is given in appendix A.

Unlike some of its contemporaries, the 6502 only uses a single ping to signal if a clock cycle is a read or write access. This means that a memory access is made in every clock cycle, even when the value is not used. One example of this is seen in indexed addressing modes that have an overflow from the low to the high address byte, they require an additional cycle to calculate the high byte of the address. During this cycle the externally visible address is the intermediate result, i.e. the correct low-byte but the high byte is not updated yet. The data bus and address bus lines are not multiplexed, but the data bus is only valid while the Phi2 clock signal from the CPU is 1. For the purposes of this work this is equivalent

to the input clock signal Phi0 being high as Phi2 is a twice-inverted version of Phi0 with shortened high-phase and the simulation following a Phi0 state change will not terminate until all resulting changes have been calculated. For read cycles the same Phi2=1 condition applies, the 6502 latches the external data on the falling edge of Phi2. An excerpt from the MOS 6502 hardware manual is shown in figure 2.5 that illustrates the timing of a memory read cycle and the relationship between the Phi1 and Phi2 clock signals.



Two Phase Clock Timing
FIGURE 1.6



Timing for Reading Data from Memory or Peripherals

Figure 2.5: Excerpt from the 6502 hardware manual illustrating the bus timing

3 Simulation and Analysis Software

The following sections will describe the design of the simulation and analysis software used for this work and highlight some of the challenges that were met during their implementation.

3.1 Design

From the beginning it was clear that it would not be sufficient to simulate all instructions just once to get meaningful results — the assumption that the changes introduced by soft errors in the CPU depend only on the instruction and the spatial and temporal location of the errors seemed unjustified and even if it was true this would need to be demonstrated. Therefore it was clear that a large number of simulations with varying conditions would need to be run, requiring a simulator that is easy to automate. Based on the performance characteristics of the existing simulator from the Visual6502 project it was obvious that the simulation itself would take the majority of compute time, so simulation and analysis phases were split into two programs to avoid a rerun of the simulation for every small change in the analysis phase. This also enabled the incremental creation of tests in the analyser tool based on manual observations of the simulation results. When it became obvious that the analyser software had the potential to become a bottleneck while dealing with large data sets, the statistics calculation was split into a third tool. This decoupled the analysis of the simulator output from the aggregation of the analysis result, so statistics for subsets of a single data set could be calculated without reanalysing the entire data set.

3.2 Simulator

3.2.1 Original version

The simulator used for this work is a descendant of the original JavaScript simulator released by the Visual6502 project, ported to C by Michael Steil. It is a modified pseudo-unit-delay simulator that uses only a single node size, two logic levels and six signal strengths. It does

not attempt to speed up simulation by replacing parts of the circuit with equivalent gates. The simulator ignores the notion of input and normal nodes almost completely, the only input nodes are the two supply rails while all other input pins of the chip are handled by changing their virtual pull-up/pull-down resistors. This approach simplifies the simulation as special cases for the bidirectional data pins can be avoided.

The difference between standard pseudo-unit-delay (PUD) simulation and the Visual6502 simulator lies in the event handling. A normal PUD simulator uses a global queue for events and has no notion of simulation cycles except by counting the number of events it has processed. The Visual6502 simulator uses two lists instead of a global queue. One list is the input list, which contains the nodes that need to be processed during the current simulation cycle, the other is the output list which is populated with the nodes that need to be recalculated in the next cycle. Nodes are added to the list when a transistor connected to them has changed state. Initially the input list contains either a single node if an input signal has changed or all nodes if a global recalculation is desired, e.g. during simulation startup. After all nodes in the input list have been processed, the output list is used as the new input list and another simulation cycle is started. Compared to a global event queue this introduction of a simulation cycle count allows for a better escape condition in case of simulation failures, e.g. with a ring of inverters. This cycle count is roughly equivalent to the number of transistors a signal travels through while it is propagated through the circuit. Counting the number of events processed would only give an approximation of the number of affected nodes which may vary depending on which parts of the circuit are active.

Originally the simulator did not check for duplicates in the output list because adding those checks reduces the simulation speed by a few percent. Instead the maximum list size was arbitrarily set to twenty times the number of nodes (34500) as preliminary evaluation showed that both in the normal simulation as well as in soft error injection the maximum list size is 847 with a distribution that is heavily skewed towards smaller numbers (average 12.0, median 5). Unfortunately this limit has proven to be too small when soft error injection is enabled, the problem and its implications will be described later. The number of simulation cycles is also used as an abort condition — if the circuit does not settle into a stable configuration after 500 cycles the simulator terminates. Without soft error injection a stable condition is reached within 10 to 25 cycles, with soft error injection the maximum number of cycles for stabilisation was 68.

To update the state of a node in the input list the simulator first calculates the connected group of that node, the set of nodes that are directly or indirectly connected to it by transistors that are currently turned on. This set is calculated by recursively adding nodes to the set that can be reached by traversing these transistors, stopping the traversal at the supply nodes.

During the recursion the simulator also notes if any pull-ups or pull-downs are encountered or if at least one node in the set currently is in 1 state. This information is enough to immediately calculate the new state of the connected group after recursion is finished. If any of the two supply nodes is in the connected group, its value is the new group value with a priority for VSS (0) as pull-down transistors in NMOS logic are generally stronger than pull-up transistors. If neither of the two supply nodes is connected, pull-ups and pull-downs are considered, with the same preference for pulling low rather than high. Finally charge sharing is simulated by setting the entire connected group to 1 if at least one node in it had a 1 signal on it before and no stronger signal sources override it. If the value of a node in the connected group changes, the state of all transistors whose gate is connected to that node is updated and all nodes on the source/drain of the affected transistor are added to the output list.

Considering the amount of calculations required the speed it achieves is quite high: A single CPU core on an Intel Core i5-2500 CPU (3.3 to 3.7 GHz) runs the full 6502 chip simulation at about 4600 simulated clock cycles per real time second which is approximately 200 times slower than the real chip would run.

3.2.2 Choice of Fault Model

The selection of a fault model for the soft error simulator is not as simple as it may appear at first. The effect of a particle error is just a single pulse, but both spatial and temporal effects must be considered. For simplicity reasons this work assumes that a single particle strike will only affect a single node of the chip, so the target of the soft error can be selected by choosing a single node number. The goal is to provide a full coverage of the chip, so a simple iteration over all node numbers is sufficient. The temporal dimension introduces some complications though: Modelling the pulse generated by the virtual particle strike in a switch-level simulation that does not have an model of time that extends beyond simple causality (event A should be processed before event B) creates additional complications. An obvious way of injecting a soft error would be to change the current state of the target node once and recalculate the chip state afterwards. The equivalent of this value modification in a real circuit would be an infinitesimally short glitch on a line that may persist until other signal sources override it. A real transistor may not propagate such a short glitch though, so a soft error injection method that allows for better control of the error duration is desirable. With the current simulator implementation it would be possible to limit glitches to a number of simulation cycles, but as the number of cycles required for simulating a half-cycle of the master clock varies this would still not allow sufficient precision over the soft error length.

Therefore the current fault model injects the soft error for a full half-cycle of the master clock, it is enabled or disabled while the simulated circuit is in a stable configuration.

Another issue that needs to be considered is the relative strength of the injected signal compared to the other signal sources within the chip. A soft error in a real circuit is attenuated by every device it passes through. If a simulator wants to calculate the new value for a connected group that contains the node with the injected error, it would need to run a shortest-path calculation from that node to every other node in the connected group and apply the soft error only to nodes below a certain distance limit. This approach has a few pitfalls: Nodes beyond that threshold are still part of the connected group and thus need to be updated by this calculation, but it is possible that all nodes beyond the threshold are weaker signal sources than the sources within the soft error range. It is unclear what the correct value should be in that case — if the error value is used, it would propagate beyond its set limits, if the value of the strong source is used it "tunnels" through the area that is influenced by the error. Besides this theoretical argument there is also a practical one: While the Visual6502 data set has complete polygon information for all the nodes in the 6502 that could be used for more accurate path-length modelling, it completely omits the vias between the chip layers. Their position can partially be inferred by comparing the overlap of same-node polygons in different layers, but especially in the case of large drivers there is more than one via per overlap. For these reasons as well as to avoid the additional overhead of path length calculations the chosen fault model always propagates the effects of an injected soft error over the entire connected group that contains the affected node. The simulator considers the error to be stronger than any other signal including both supply nodes, but it does not propagate the error to them. This choice was made to create a worst-case scenario in which all particle strikes are as strong as possible, while still preserving the fact that a particle strike is unlikely to override the level on the global supply nets. A possible further refinement for later work would be to split the global supply nets into localised regions and allow a soft error to override the supply net within that region — it would require major changes in the simulator though, for example pull-ups would need to be annotated with their supply region.

Finally there is the question what a soft error in a node actually does. The real particle strike creates a current pulse, so the signal will be momentarily set to a high level which may flip dynamic storage nodes to a logical 1. This model is certainly easy to implement in a switch-level simulation — just ignore the evaluation result of a connected group that contains a soft error and set it to 1 while the error is active. If you ignore the relation to the real particle for a moment, there are three other possible models:

1. Lock the group to 0 instead of 1.
2. Lock the group to the state it had before the soft error was injected.

3. Lock the group to the inverse of the state it had before the soft error was injected.

The first option does not need much explanation — it may be interesting if a simulator built for NMOS is used to simulate a PMOS circuit.[1] The other two options snapshot the current value of the error node and use that snapshot value either as-is or inverted. If the 6502 was implemented in CMOS logic, the choice of model would be clear. As it is just an NMOS chip, the inversion options seems to be more attractive though under the assumption that it yields results that are closer to a hypothetical CMOS implementation of the 6502[2]. While this assumption has not been verified, limited evaluations have been made with all four error models and the inversion option has produced the highest probability of failures, see chapter 4.9 for details.

3.2.3 Implementing Soft Error Simulation

Implementing the chosen fault models in the existing switch-level simulator requires only minimal modification. When the simulator calculates the connected group it needs to track if the node that was selected for the soft error was added to the connected set. If it was not, everything can proceed just as it would without a soft error. If the affected node was found, the new value is determined according to the selected fault model instead of the standard signal strength hierarchy. As the chosen model keeps a soft error active for a full half-cycle of the master clock, it can be enabled and disabled outside of the simulation loop while the simulated CPU is in a stable configuration. Both enabling and disabling need to trigger a recalculation of the circuit state because the error may have changed the current state of its node.

The ability to selectively turn on and off a soft error within the simulated CPU is not enough though, for a methodical evaluation of the effects of these errors the simulated CPU needs to be connected to a virtual system that feeds it the data it should execute. The original Visual6502 simulator connects the CPU to simulated RAM and allows the user to step through the execution of the data therein; the C port of the simulator comes with an example program that loads the Commodore 64 BASIC interpreter and hooks its input/output calls to the command line window. Both of these approaches are not that helpful for evaluating the effects of soft errors as they are both interactive, so a new system model had to be created.

The intention is to examine the effects of soft errors on single instructions, so instead of a full program only a few instructions need to be executed in each test run. This enables a major simplification of the simulation: Instead of providing a full 64 KiB simulated memory with

[1] As PMOS logic is complementary to NMOS logic this is possible by inverting the logic levels assigned to the supply voltages.

[2] The 65C02 has an extended instruction set.

predetermined values, the memory array is initially set to "unknown". A list of values (called *memvalues*) is provided that is fed to the CPU in sequence for any access to a location that hasn't been accessed yet and the corresponding array location is updated with that value. This drastically reduces the amount of memory values that need to be specified while still keeping a consistent view of memory for the simulated CPU: A memory location that was read or written once will contain the same value on a second access. The simulator repeats the last value of the list if it needs more values than the list contains. Instruction fetches as indicated by the SYNC line of the CPU are an exception to this, they are always fulfilled with a NOP instruction that is not written to the memory array to ensure that the processor receives a valid instruction stream while limiting the memory contents to the user-specified values. Figure attempts to visualise the flow of these values. The column on the left side shows the memvalues list that is used as an input, the middle column is the address bus of the simulated CPU and the right side shows the memory array contents that are updated in response to the CPU access on the same line. The write access updates the contents in the memory value array, so the contents from the previous line are invalidated.

memvalues	CPU memory cycles	memory array
33	R 1000	1000 33
55	R 1001	1000 55
77	R 5533	~~5533 77~~
	W 5533 3b	5533 3b
NOP	R 1002 [opcode]	

Figure 3.1: Illustration of the interaction between the memvalues list, CPU memory accesses and the memory array. The memvalues list is provided by the user, the memory array is calculated dynamically in response to the CPU memory accesses.

Before the instruction under test can be fed to the CPU, the current state of the CPU needs to be initialised to known values. The set of these values plus two memory value lists defines a start condition within the simulator. The simulated CPU is set to such a state by generating a reset pulse followed by a short memory value sequence. The sequence corresponds to a few load instructions that set up the A, X, Y and SP registers as well as the six flag bits followed by a jump instruction to set the program counter to its desired value. At this point a snapshot of the CPU simulator state is taken so the CPU can be restored to this state without simulating the initialisation sequence again.

The whole simulation is structured around the concept of test runs, which start from a selected start condition and run one selected opcode through the CPU, optionally injecting a soft error in one node for one selected half-cycle. Test runs without a soft error are reference runs, test runs with a soft error are injection runs. The reference run has two purposes: It establishes an error-free execution trace of the selected opcode and it is used to measure the number of

cycles the opcode needs to execute with the current start condition. This allows injection runs to be skipped completely if the soft error injection time is so late that it would not affect the tested instruction anymore.

```
[a     b     c    d e f      g       h    i    j    k    l    m     n    o]
 0: 8001: r:69 w:-- R _ phi0:0 PC:8001 A:10 X:00 Y:00 S:00 P:30 IR:69 T:06
 1: 8001: r:01 w:-- R _ phi0:1 PC:8001 A:10 X:00 Y:00 S:00 P:30 IR:69 T:06
 2: 8002: r:01 w:-- R S phi0:0 PC:8002 A:10 X:00 Y:00 S:00 P:30 IR:69 T:01
 3: 8002: r:ea w:-- R S phi0:1 PC:8002 A:10 X:00 Y:00 S:00 P:30 IR:69 T:01
 4: 8003: r:ea w:-- R _ phi0:0 PC:8003 A:11 X:00 Y:00 S:00 P:30 IR:ea T:06
 5: 8003: r:01 w:-- R _ phi0:1 PC:8003 A:11 X:00 Y:00 S:00 P:30 IR:ea T:06
 6: 8003: r:01 w:-- R S phi0:0 PC:8003 A:11 X:00 Y:00 S:00 P:30 IR:ea T:01
 7: 8003: r:ea w:-- R S phi0:1 PC:8003 A:11 X:00 Y:00 S:00 P:30 IR:ea T:01
```

Figure 3.2: Example output of the simulator running an `ADC #$01` instruction

Figure 3.2 shows a truncated example output of the simulator in textual form. Column a is the current half-cycle of the master clock. Column b is the address bus; columns c and d show the data bus, separated into read and write columns for increased clarity. Column e is the Read/Write line of the CPU, column f the sync signal which indicates an instruction fetch. Column g is the input clock signal which directly corresponds to the even/odd state of the half-clock count in column a and is just provided for easier reference. Columns h to l provide a view into the internal registers of the CPU, column m is the flag register (named P to correspond with the MOS manuals). Column n shows the instruction register which holds the opcode of the currently executing instruction, column o is a representation of the T state counter. In the example shown here, an `ADC #$01` instruction was fed to the CPU. The instruction fetch itself is not present in the output because it happens while the last cycle of the preceding instruction is still executing and it was decided that injecting a soft error at that point would not actually influence the instruction under test but the one before it. The trace shows that the contents of the accumulator is initially 0x10 and the `ADC` instruction is supposed to add one to it. The new value 0x11 is not seen in the accumulator until half-cycle four, which is the half-cycle after the next instruction (0xea, a *NOP*) has been fetched. This demonstrates the limited pipelining the 6502 processor performs and shows that only half-cycles zero to three are valid times for soft error injection for this instruction. The simulator uses the first occurrence of an instruction fetch during the reference run to determine the latest time for a valid soft error injection and completely skips injection runs that specify a later time.

The example in figure 3.2 has been truncated because a longer run of `NOP` instruction would not add anything to the explanation. The simulator actually runs until 40 half-cycles are completed or a simulation error prevents it from proceeding. This length was chosen to allow

reliable detection of a deadlocked CPU even when an instruction with the maximum length of seven clock cycles is executed.

The error-free execution provided by the reference run is immediately used in the simulator to distinguish between expected and unexpected memory accesses. When an injection run is simulated, the CPU is fed the memory values from the reference run if the same half-clock cycle in the reference accessed the same memory address. If the address does not match, the memory array from the reference run is consulted and any access to an address that was used during the reference run reuses the value from the memory array. This facilitates a consistent view on memory even if the soft error changes the execution time of an instruction. If the address is not found in the memory array, the second list of memory values (called *failvalues*) is used to determine the value the CPU sees, but the memory array is not updated. Without this second list it would not be clear which entry of the memvalues list should be used at this point in simulation.

The system simulator is based around a loop that processes half-cycles. Initially this loop is entered with the test opcode already on the data bus, but not clocked in. For every one of the 40 half-cycles, the loop first toggles the master clock. If the clock is now in a 1 state, memory accesses by the CPU are handled. Afterwards a the current bus state and the internal registers are logged. If the test run is an injection run, the simulator compares the current half-cycle number with the soft error time and enables or disables the error as required. After all 40 half-cycles are completed, the logs taken during these cycles are added to the output file. Additionally the current value of all nodes within the simulated CPU are read and also added so later analysis can determine if a particular injection run leaves the CPU in exactly the same state as its reference run.

If the switch simulator reports a simulation failure while the master clock signal is toggled, the system simulation is aborted and instead of the normal output a failure record is emitted.

3.2.4 Simulation result file size

A full simulation for a single start condition over all combinations of opcodes, nodes and injection times needs approximately two million test runs. Although convenient for a human, the simulator output presented in figure 3.2 would be too verbose for this. Instead a binary output format was defined that stores a single test run in 530 bytes for the test run data plus 216 bytes for the copy of the final CPU state. This would still result in a file size of approximately 1.5GiB, but fortunately test runs with the same start condition and opcode are very similar to each other. This matches the order in which the test runs are generated because the simulator reuses snapshots taken after the start condition is set up. The situation

is ideal for standard data compression libraries like zlib and liblzma, but it can be improved even further. Most injection runs are expected to be similar to their reference run, so every byte in their state logs is exclusive or-ed with the corresponding byte in the reference state log. The operation can be trivially reversed while reading and is practically free on a modern CPU and it results in state logs which have many bytes that are zero. This is ideal for the entropy compression part of modern compression algorithms and decreases the output file size by approximately 15-40% depending on the compression algorithm. Table 3.1 shows the compressed file sizes and run times for the two implemented compression libraries compared to uncompressed output. The decrease in run time for zlib compression compared to the uncompressed output is probably due to measurement error. liblzma supports higher compression presets than 6e, but even with the maximum compression preset the file size only decreases by about 20% compared to 6e while the RAM usage increases fourfold which would have caused issues on some of the systems used for simulation.

library	preset	size (MiB)	time
uncompressed	-	1457.5	157min. 11s
zlib	9	25.0	156min. 52s
liblzma	6e	2.5	187min. 12s

Table 3.1: compressed file size and simulation time for multiple compression options

To simplify the distribution of simulation results between multiple computers liblzma compression was chosen even though it incurred a 20% penalty in run time.

3.3 Analyser

All the data the simulator generates is not very interesting in itself — another tool is required that analyses the injection runs and attempts to categorise the changes compared to the reference run. The analyser tool reads the simulator output and runs a series of checks on the data, outputting “tags” for any (non-)changes it detects. The analyser was separated from the simulator to avoid rerunning existing simulations whenever a new analysis approach is implemented or a bug is fixed. A single simulator output file that required about 2.5 hours of CPU time to generate can be analysed on the same machine in just 24 seconds. Although a data set is emitted for each analysed injection run, the data size is much less of a concern (about 80 MiB for two million injection runs) and the already-existing file output code with data compression was reused to reduce the size further (about 8 MiB with zlib compression).

3.3.1 Checks in the Analyser

The following sections will describe various checks in the analyser that are used to detect and categorise the effects created by the injected soft error.

Basic Checks

The most basic analysis result is the simulation failure — it is already indicated in the simulator output, but needs to be counted for the final statistics. Another very basic result is an injection run where all logged CPU states are identical to the reference run. There are two variations of this, based on the values of all the nodes in the final CPU state that the simulator provides. If this dump matches the one taken after the reference run, the soft error truly had no effect on the CPU. If there is a difference in the dump, but no difference in the logged CPU states, the soft error is classified as a hidden effect. No attempts have been made to determine if these hidden effects have an effect on subsequent instructions, although they would not qualify as hidden if they had any impact on `NOP` instructions.

Affected States

When none of these cases apply, the analyser searches for the first and last state that differ from their corresponding reference state. The indices of these states are used in later steps to focus the analysis on the states that actually differ. If the soft error introduces a lasting change, the last states of the reference and injection runs may be different even though the "interesting" subset of states is much smaller. To detect this, the analyser assumes that the injection run ends with a series of `NOP` instructions and verifies the correct execution of these instructions starting from the end of the state list. The execution of a `NOP` instruction is considered correct if it does not change any CPU registers, the program counter changes by minus 1 only in the expected half-cycle and all memory accesses are reads at the location in the program counter. The verification is terminated when a state is encountered where the instruction register does not contain a `NOP` opcode or when one of the other conditions do not hold.

CPU Deadlock

CPU deadlocks are another category of errors that are relatively simple to detect. If the injection run ends with four or more consecutive states in which the T-state counter has no active bits, the CPU must be deadlocked. These deadlocks can further be split into three cases

depending on the contents of the instruction register. Some errors may change the content of the IR to an undocumented opcode that is known to deadlock. In other deadlocks the contents of the IR are changed to an opcode that usually does not deadlock the CPU. Finally, a deadlock may occur without changing the IR, for example by resetting the T-state counter bit that was currently active. Obviously no further tests need to be made if a deadlock was detected because the CPU cannot recover by itself[3].

Control Flow and Data Accesses

The same "scan-and-check" approach used to detect deadlocks can also be used for other errors. A change in control flow between the reference and injection run is detected by first scanning both for opcode fetches. The resulting lists may have different length, so the check compares only the first k entries, with k being the minimum of both list lengths. States with the same index on the reference and injection list are compared. If the half-cycle number of the two states does not match, the length of an instruction has changed. If the two states have different program counter values, the control flow of the program has changed. Tags are emitted for both of these results.

Control flow is not completely determined by the initial opcode fetch though, for example a bit flip in the program counter will change the address an instruction parameter is read from while executing the instruction. To detect this, the analyser scans the memory accesses following the opcode fetches in the reference and injection runs. As long as the address bus and program counter are the same, an access is considered to be part of the instruction. Single-byte instructions are an exception to this as the 6502 will always fetch the byte following the opcode as a parameter byte, even when it is not needed. To detect this, the program counter during this access is compared to the program counter at the next instruction fetch. If both values are the same, the instruction must have been a single-byte instruction, so a change on the address bus during this dummy access can be safely ignored.

The remaining memory accesses that are neither opcode fetches nor instruction parameter fetches are assumed to be data accesses. For each instruction in the injection run the addresses of data accesses are compared to the reference run. If the address does not match or if the number of data accesses changes, a tag for a data access change is emitted.

[3] A test for this exists, but it was never triggered.

Interrupts and Reset

One possible cause for control flow changes are interrupts and reset, so there are specific checks for them. The expected value in the instruction register during the injection run can be determined without using data from the reference run. The initial value is known because it is the instruction that is tested, any further values in the IR should correspond to the value read during an instruction fetch. If at any time the value in the IR does not match the expected value, a tag is generated for this change. If the new value is 0x00, it is assumed that an interrupt sequence starts at this point and its execution is verified. The first check is the number of half-cycles where the IR is 0x00. If the value is present only for a single half-cycle, the sequence is not considered to be an interrupt sequence at all based on manual inspection of such cases. If it is shorter than 13 half-cycles, it is considered a shortened interrupt sequence and verification is stopped. Instruction fetches anywhere except at the end also stop verification. The interrupt vector address is compared to the known addresses to determine if the sequence is an NMI, IRQ, Reset or something unknown. If it is a reset, no writes may occur; otherwise exactly three writes to the stack must be found. In those writes the bit indicating a BRK instruction is checked and the return address is compared to the initial program counter to determine if returning from the interrupt will correctly resume the interrupted instruction. Tags are emitted for every issue that is found and the verification is aborted. If the sequence passes all checks, this fact and the vector that was used are noted in two tags. In the case of a reset, the analyser only verifies that the trailing NOP sequence is correct; for other types of interrupts it also checks that the start of the interrupt sequence is also the first difference to the reference run.

Simple Errors

Another interesting issue are register differences between the reference run and the injection run. To analyse this, the concept of a simple error was developed. A simple error is based on checking all the data values in an injection state against the corresponding reference state. If there are differences in the IR, the read/write line or the SYNC signal, the check is immediately aborted. If exactly one of the checked values changes due to the error, the injection run is a simple error if the change persists during all cycles that differ from the reference run. There is one exception to this: If the change is on the address bus, it may cause an access to a different memory location, possibly resulting in a different value. When the simulation is run with the same data in the memory value list and the failed memory value list, the change on the address bus is the only difference in the two simulation runs. For this reason changes on the address bus are not considered simple errors if they can affect a memory operation, i.e. if they exist while Phi0 is 1. It would be possible to refine this

analysis by checking if the current instruction actually uses the value fetched from memory, but this check has not been implemented — in the case of memory-mapped I/O devices an access that is irrelevant to the CPU may still have an effect.

Unusual Memory Writes

As noted before a change on the Read/Write line is never considered a simple error, so it must be handled separately. Multiple checks consider this possibility. Under normal circumstances, the 6502 will never execute more than three consecutive write cycles and it will obviously never issue a write cycle while an opcode is read. The number of writes is another example of a scan-and-check detector that will emit a tag if more than three consecutive memory write cycles are detected. The check for write cycles during an instruction fetch is further refined by generating a list of program counter values encountered in the reference run. If the reference run issues no writes to any of these locations, it can be used to verify the memory writes of the injection run. If any write during the injection run uses an address where the reference has fetched an instruction from, it would overwrite the program stored there (assuming it is in RAM). The check is skipped if the reference already overwrites itself which may occur with certain combinations of memory value lists and initial program counter values. In this case a reduced check verifies that there is no cycle where the CPU indicates both an instruction fetch on the SYNC line and a write access on the Read/Write line.

3.3.2 Further Ideas

There are still many possibilities for additional checks that have not been implemented. One option would be to implement verifiers for every opcode to check if the execution of the instruction itself is affected or if the changes are only due to changes in its input data. In the case of ALU instructions multiple of these verifiers could be used to determine if the behaviour with error corresponds to the error-free behaviour of another ALU instruction. Such a check may result in multiple verifiers matching the same instruction though, so the analysis would need to consider multiple injection runs with differing initial states.

The analyser also outputs a half-cycle number with most of the tags it generates, which could be used to mark the earliest change within an injection run as a possible root cause for later changes. The statistics tool currently does not take advantage of this and not all tags that may be of interest for such an analysis emit this.

3.4 Statistics Aggregation

The analysis tool considers each injection run in isolation. To get from these separate results to per-opcode or per-node aggregate results and to merge the results of separate analysis files, another tool is required. The statistics tool reads one or multiple analyser output files and generates aggregate data from them. The tags in the file can be aggregated either by node or opcode. As the internal behaviour of the 6502 differs greatly between the two internal clock phases, the aggregation can be separated into even and odd error injection cycles. For tests that inject soft errors at all possible half-cycle times, the number of injection runs changes based on the selected instruction and sometimes also on the start condition chosen for the simulation. This means that it would be meaningless to compare the absolute number of affected injection runs between two opcodes, so the output of the statistics tool always includes the total number of runs for any result as well as the percentage derived from these two numbers.

4 Evaluation

The following sections will describe the evaluation process and its results.

Per-opcode probabilites will be presented in the form of opcode matrices with cell background colours corresponding to the probability of the event in question. The colour mapping has been normalised for each table, that is the blue and red ends of the colour scale have been assigned to the minimum and maximum value of the corresponding data. This maximises the colour changes between cells with differing data, but unfortunately also means that the colours cannot be directly compared between two different tables. The scale at the bottom of each table shows the mapping between colours and values in eleven discrete steps — the mapping used in the cells themselves is not limited to these steps but continuous. The matrices only show the mnemonic of each opcode but omit the addressing mode information. A full, uncoloured opcode table can be found in the appendix as table B.2.

Per-node probabilities are presented in a similar way, but use the polygon data provided by the Visual6502 project to visualise the data. Any nodes that are not present in the plotted data at all are shown as 90% transparent black, the others are coloured with normalised colours like the opcode tables. The positive and negative supply polygons have been left out completely to reduce visual clutter.

4.1 Implementation Bugs

Unfortunately two problems were found in the simulation system at a very late stage that did not leave enough time to rerun all CPU simulations. Both are related to failure reports of the switch-level simulation. First, the maximum list size described in section 3.2.1 was not sufficient in rare cases when injection runs were simulated — the event list could actually grow up to 61450 entries. The second problem is that the system simulation only checks if the clock toggle reports a failure, but neither data bus writes nor soft error enable/disable operations did check for failed simulation. While no failure has been observed when setting the data bus, enabling a soft error may result in a simulation failure while the CPU state is

recalculated with the new node value injected by the error. Both problems overlap to some extent, an increased list size would have fixed some of the missed simulation failures.

A partial rerun of the existing simulations with these problems fixed indicates that the chance for misclassification for the first problem is very low (366 of 2,100,886 runs, about 0.017%). The incidence of undetected simulation failures when a soft error is enabled or disabled is slightly higher (5,719 of 2,100,886 runs, about 0.27%). This number only includes test runs where the simulator missed a failure and successfully completed the rest of the test run successfully. As this only affects test runs where the maximum number of simulation cycles was reached it could be argued that the result is still valid because it represents situations where the soft error created an oscillating sub-circuit that was cleared by the next master clock edge. This argument is not completely valid though because the CPU simulator flushes its list of nodes that need recalculation when it aborts and the oscillating sub-circuit may not make it on the list again when the next master clock edge is processed.

4.2 Data Sets

Although it would be nice to run exhaustive simulations with all possible start conditions, this is currently not possible within any reasonable time scale. The input state contains 32 bits for the four CPU registers, a 16 bit program counter, six flag bits and at least eight bits of memory values for a total of at least 62 bits of input state. Considering that a full simulation with just one of these input states for all opcodes, nodes and injection times needs about three hours CPU time, calculating all these possibilities is clearly infeasible. Instead, a few specific data sets have been selected and the evaluation was limited to just the instructions themselves without testing the reaction to additional external signals like interrupts or forced wait states. In all of these data sets the memvalue/failvalue lists consist only of a single byte. Table 4.1 shows an overview over the data sets and the values used in their start conditions. All data sets represent a total of about $4.1 \cdot 10^9$ injected soft errors.

data set	size	PC	A	X	Y	SP	flags	memvalues	failvalues
random	1.120	random							as memvalues
binary	126	one bit set/clear and all bits set/clear							as memvalues
distinct	128	0xf00f	0x33	0x55	0xaa	0xcc	all	0x66	0x66/0x99
	128	0x0ff0	0xcc	0xaa	0x55	0x33	all	0x99	0x99/0x66
nop	65.536	all	0x00	0x33	0xcc	0xaa	0x00	0x55	0x55
	65.536	all	0xff	0xcc	0x33	0x55	0xff	0xaa	0xaa
bugfix	64	random							as memvalues

Table 4.1: Data sets used in the evaluation

In the *random* set, 1.120 start conditions were generated by choosing a random value for each field in the state, except that the same byte was used for both the memvalue and failvalue lists. The values were generated by the rand() function of perl v5.10.1 which is supposed to generate a uniform distribution, the value range for each field was chosen to cover the all values for the field. For each of these states a full simulation with all opcodes, nodes and injection times was run.

The *binary* set consists of 126 start conditions. In all of them the failvalues list is identical to the memvalues list. Of these start conditions, 62 are generated by setting all values in the state to 0 except for a single bit (i.e. all values except one are 0, the remaining one has exactly one 1-bit). A second set of 62 start conditions is generated from these by inverting all bits. The final two start conditions in the set are the all-zero and all-one states. Again the simulation was run with all opcodes, nodes, and injection times.

In an attempt to build a set of start conditions that have the highest possible chance of yielding different results if the operation is modified by a soft error, the *distinct* data set was defined. It has two sub-variants: One of them uses the same value for both memvalues and failvalues, the other uses the inverse of the memvalues byte for failvalues. All values in the start conditions are chosen so that any pair of them (including the two halves of the program counter) have a Hamming distance of four. All possible flag combinations are used for a total of 256 different start conditions.

Finally the *nop* data set is intended to provide a base line of effects introduced by soft errors. Unlike the other data sets the simulation was only run for the `NOP` opcode with all nodes and all applicable injection times. This greatly reduced the time needed for simulation, so the number of start conditions could be increased. The *nop* data set exhaustively checks all possible PC values combined with two sets of values for the remaining registers, resulting in a combined count of 131.072 start conditions in this data set. Once more the memvalues and failvalues lists were identical.

The *bugfix* data set was generated after fixing the two bugs in the simulator that were noted in section 3.2.3. The start conditions were generated using the same method as the *random* data set.

4.3 Basic Results

4.3.1 No (visible) Effect

The first and very interesting result is that the fraction of injection runs that have no effect on the end result at all is quite high: 52.25% over all data sets (details in table 4.2). The *nop* dataset stands out with a slightly lower percentage of injections runs without any effects, indicating that this percentage may vary depending on the opcode.

data set	no effect	hidden effect	deadlock	simulation failure
random	53.03%	5.09%	0.68%	0.46%
bugfix	52.94%	5.07%	0.64%	0.82%
binary	54.55%	6.24%	0.66%	0.26%
distinct	53.07%	5.05%	0.75%	0.45%
nop	48.98%	12.83%	0.60%	0.34%
total	52.25%	6.84%	0.67%	0.43%

Table 4.2: Basic results for the data sets

As shown in figure 4.1 for the *random* data set (the distribution for the other data sets is almost identical) there is a slight variation between opcodes, but the spread is less than eight percentage points. The seven flag-setting instructions as well as `NOP` that show up in blue in the table have the highest chance of being affected by the injected soft error. This seems curious as those instructions are the only ones that do not need to move any data within the processor, but a possible explanation will be shown later. The effects at the other end of the scale seem to be less clear. The `BIT` instruction with opcode 0x24 (zero-page addressed) has the highest chance of not being affected by soft errors, but its counterpart at 0x2c (absolute addressed) is noticeably lower on the scale than some other instructions. Similar differences seem to apply all of column x4 versus columns xC and x5 versus xD which are all zero-page based vs. absolute-based versions of the same instruction. Using immediate (column x9) or indirect addressing (x1) seems to lower the chance of an unaffected test run slightly. Finally the read-modify-write instructions in columns x6 and xE seem to have a slightly higher chance of not being affected by soft errors than their counterparts that operate on the accumulator (top half of column xA).

The situation changes noticeably when not only effect-free runs are considered but both runs with no effects at all and run with hidden effects. The per-opcode distribution for this is shown in figure 4.2. The spread between opcodes is about eight percentage points again, but the situation has reversed for some opcodes: The eight instructions that seemed to be most vulnerable in the first table are now the least-vulnerable instructions. A likely explanation is that these instructions use the least amount of resources in the CPU and thus have the

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-
0.486	0.493	0.501	0.508	0.515	0.523	0.530	0.537	0.545	0.552	0.559						

Figure 4.1: Per-opcode fraction of no-effect runs for the *random* data set. Please note that in this table a high value corresponds to a low vulnerability to soft errors.

highest chance that a flipped bit in the unused parts remains until the end of the simulation. To test this theory, the *nop* data set was filtered to extract the runs with hidden effects. For each of the runs the difference between the final CPU states of the injection run and its reference was calculated and the affected nodes were counted. Figure 4.3 shows the affected 89 nodes in the processor layout. A spot check of the five nodes that appeared most often has shown that they are all related to the ALU. The three highlighted nodes at the top left are part of the highest bits of the T-state counter which are not used by the `NOP` instruction. It seems likely that an error there would be flushed out without affecting anything when a longer instruction is executed later. The long, low-occurrence lines have not been checked in detail, but based on their position the two horizontal lines at the bottom must be bit seven of two of the internal busses.

In addition to the changes for these eight instructions, some of the ALU operations now stand out: ADC and SBC with an immediate operand (opcodes 0x69 and 0xe9) have the highest chance of being affected by a soft error. There is also a noticeable “pairing” effect visible: For many instructions in the even rows, their twin in the odd row below has a slightly lower chance of error-free execution. The most likely cause is that the ALU is involved in the address calculations for the instructions on the odd rows, increasing the complexity of the instruction and thus the chance that a soft error causes a problem.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-

0.544	0.552	0.560	0.568	0.576	0.584	0.592	0.600	0.608	0.616	0.624

Figure 4.2: Per-opcode fraction of no-effect plus hidden-effect runs for the *random* data set. Please note that in this table a high value corresponds to an assumed low vulnerability.

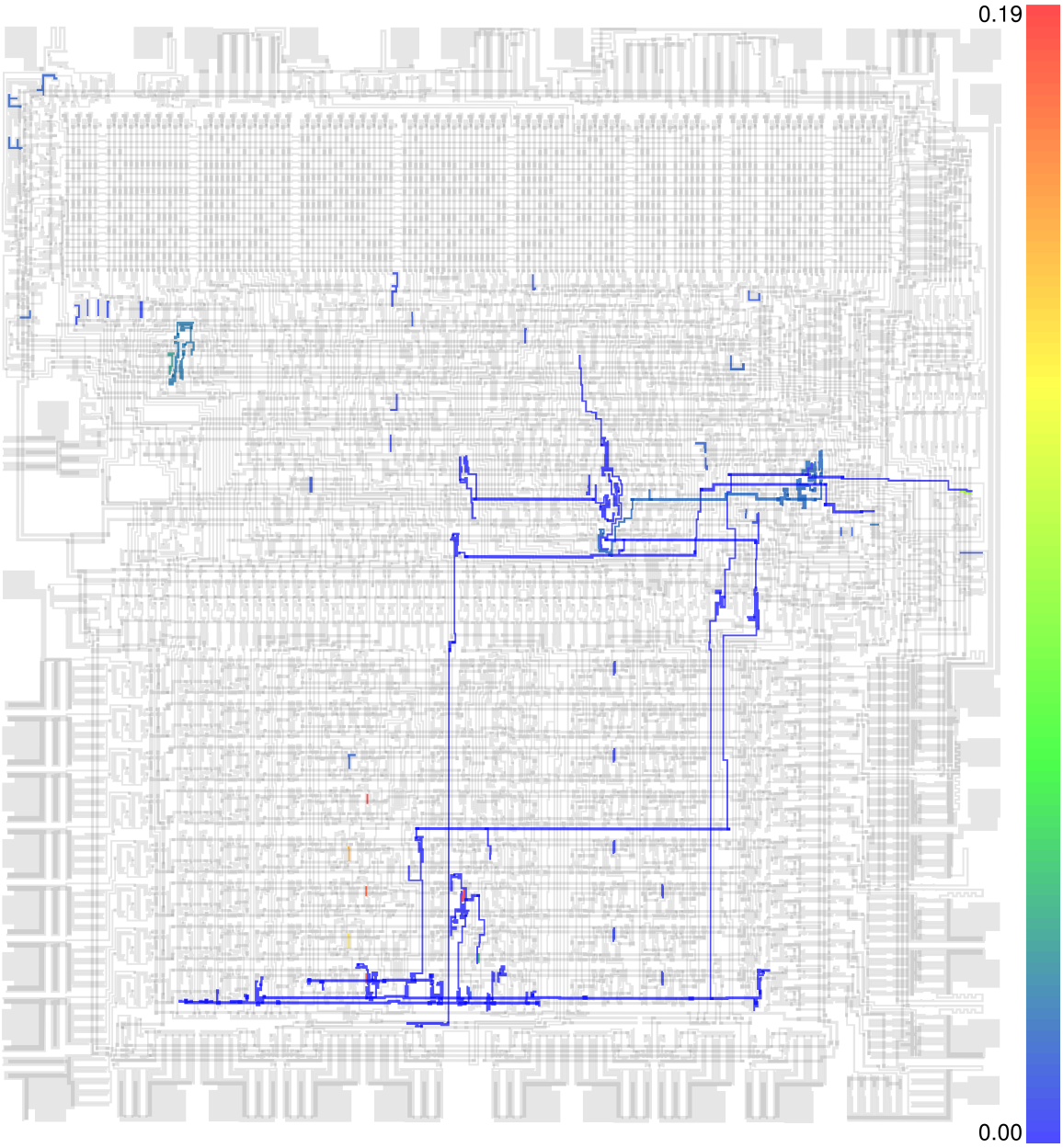


Figure 4.3: Nodes with differences after a hidden-effect run

4.3.2 Deadlocks

Overall the chance for a deadlock due to a soft error is very small, varying between 0.25% and 1.08% for the *random* data set. The range is slightly higher for the *distinct* data set (see table 4.3). Although the *distinct* data set differs from the random data set by the use of distinct memvalue and failvalue lists for half of the set, this difference is not the cause for the different deadlock chances: Calculating the deadlock rates for each subset yields approximately the same result. The *distinct* set can be split another way as it contains two subsets whose start condition values are binary inverses of each other. There is a noticeable difference between these subsets (identified in the table by their program counter value), so the chance of deadlock depends on the contents of one of the four registers, the program counter and/or the memory value — the contents of the flag register can not make a difference here because both subsets were calculated using all possible values.

data set	minimum	average	median	maximum	std. dev.
random	0.25%	0.65%	0.64%	1.08%	0.14%
bugfix	0.22%	0.62%	0.61%	1.08%	0.14%
distinct	0.29%	0.71%	0.70%	1.31%	0.16%
distinct (fail = mem)	0.29%	0.71%	0.70%	1.30%	0.16%
distinct (fail ≠ mem)	0.29%	0.71%	0.70%	1.31%	0.16%
distinct (pc 0xf00f)	0.25%	0.73%	0.74%	1.13%	0.18%
distinct (pc 0x0ff0)	0.32%	0.68%	0.66%	1.48%	0.17%

Table 4.3: Per-opcode deadlock probability statistics

The `JSR` opcode (0x20) was isolated as the opcode that has both the highest absolute probability of deadlock runs for both the *random* and *distinct* data sets (1.08% and 1.31%) as well as the largest difference between the two. To isolate the effects of the six candidate values on the deadlock probability, six tests were run for the `JSR` opcode using the PC 0xf00f start condition with one value inverted per test and all flag bits cleared. The tests were run using the bug-fixed version of the simulator, so the unmodified value set was included again to avoid any false conclusions from mixing simulator versions. Table 4.4 shows the results. It seems unsurprising that the `JSR` instruction is not sensitive to changes in the A, X or Y registers because they are not used by this instruction, but the sensitivities to the stack pointer, program counter and target address (given by the memory value) are interesting. Both the original values as well as the corresponding new ones have the same number of 1-bits, so that is not a factor. A few additional tests with varying values in the program counter were run, but no clear pattern seems to emerge.

To verify the theory that the deadlock probability for `JSR` does not depend on the values in the A, X and Y registers because the instruction does not need these registers, two additional

item	value	probability
(none)		1.10%
A	0xcc	1.10%
X	0xaa	1.10%
Y	0x55	1.10%
SP	0x33	1.06%
PC	0x0ff0	1.50%
memory	0x99	0.94%
PC	0x0000	1.14%
PC	0x0066	1.05%
PC	0x0660	1.52%
PC	0x0990	1.15%
PC	0x6006	1.05%
PC	0x6666	1.07%
PC	0x0f0f	1.12%
PC	0xf0f0	1.16%
PC	0xffff	1.08%

Table 4.4: Probability of deadlock for `JSR` with modified start conditions

opcodes have been tested. The selected opcodes are 0x7d for `ADC abs,X` which uses both the accumulator and X register and 0xee for `INC abs` which increments a value directly in memory and thus does not need any registers except the program counter. The same seven start conditions as in the initial `JSR` investigation were used, the results are listed in table 4.5. The results confirm that the value in the registers that are used by an instruction affect the chance of deadlocks. Memory addresses from the program counter as well as instruction parameters also seem to affect the deadlock rate noticeably, but the exact mechanism is still unclear and has not been investigated further.

item	value	`ADC`	`INC`
(none)		0.70%	0.62%
A	0xcc	0.51%	0.62%
X	0xaa	0.75%	0.62%
Y	0x55	0.70%	0.62%
SP	0x33	0.70%	0.62%
PC	0x0ff0	0.74%	0.59%
memory	0x99	0.33%	0.55%

Table 4.5: Probability of deadlock for `ADC abs,X` and `INC abs` with modified start conditions

The per-opcode distribution of deadlock probability for the *bugfix* data set[1] is shown in figure 4.4. The chance of a deadlock is noticeably higher in the opcodes that access the

[1]The missed simulation failures in the simulator sometimes resulted in deadlocks, changing the data slightly.

stack. Interestingly it does not appear as if there is an increased chance for deadlock in the instructions that differ from the existing deadlock-instructions in only a single bit. There are increased deadlock probabilities in the first column that at first might look like this, but if this were the case the four branch instructions in the upper half should also be affected. A more likely explanation is that the deadlock probability depends on parts of the instruction sequencing that would not occur during error-free operation of the CPU, for example by a bit-flip that changes the IR from a longer instruction to one that would never execute at the current T-state.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	crash	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	crash	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	crash	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	crash	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	crash	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	crash	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	crash	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	crash	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	crash	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	crash	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	crash	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	crash	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-
0.002	0.003	0.004	0.005	0.006	0.007	0.007	0.008	0.009	0.010	0.011						

Figure 4.4: Probability of deadlock per opcode for the *bugfix* data set

A plot of the nodes that cause deadlocks in the *bugfix* data set is shown in figure 4.5. It seems to confirm the theory that the deadlock probability depends on instruction sequencing "hiccups" as the horizontal input lines of the PLA at the top all contribute to some extent. The two highest-probability lines there are IR2 and notIR2, bit 2 of the instruction register and its inverse. The upper T-state bits at the top left are also highlighted a lot, with an internal version of the Ready (wait state) signal as the highest-deadlock-probability signal in the diagram. This signal causes the T-state counter to keep its current value, adding another possibility for modifying the instruction sequencing. The large net that shows up in yellow with a probability of 24.6% is one of the internal clock phases. It has the interesting property that soft errors injected in an even half-cycle never have any effect on the CPU, while soft errors injected in an odd half-cycle will always cause either simulation failure (50.8%) or a

CPU crash (49.2%). This is easily explained though — in an even cycle the signal should normally be 1 and the inversion soft error model resets it to 0. The same state change would normally occur shortly after the next master clock edge, so there is no difference in the simulation. If it is flipped to 1 when it was 0 though, it will enable a lot of pass transistors that should not be active at this time, resulting either in a crash or a simulation error when this creates a feedback loop.

4.3.3 Simulation Errors

For completeness the per-node chance of causing a simulation failure for the *bugfix* data set has been plotted in figure 4.6. As it should be expected both internal clock nets show up in the diagram, although the highest-probability node is the control signal that connects the inverted value of the internal data bus to the B input of the ALU. The feedback path that causes the failure seems obvious, although it is unclear why the failure probability is so high as the pass transistors for the non-inverted path have a completely separate control signal. That line is also visible next to the red polygon, although only with a 16.1% probability of simulator failure. The other two high-probability lines in the ALU are enable signals for pass transistors used for AND and EOR calculations.

The probability of a simulation failure in the *bugfix* data set is between 0.5% and 1.2% per opcode. No opcode diagram has been included because the simulation failure is not a property of the 6502 but the software used for evaluating it. The simulation failures indicate that there is a major malfunction within the chip when they are triggered by a soft error, but without a simulator that is able to continue simulation even in these cases it is unclear what effects they may have on the result. For example the assumed feedback loop on the ALU's B input noted above might not cause anything but a reduced noise margin in practice. This is because the inverter on the ALU input does not need to be very strong while the driver of the internal data bus needs to drive a line that crosses almost the whole chip horizontally.

4.4 Simple Errors

One of the categories defined for the analyser are "simple" errors as explained in section 3.3.1. In this category of error effects, only a single of the logged values is affected, so it almost exclusively deals with register changes. The probabilities for simple errors in the data sets are shown in table 4.6. The upper line for each row shows the probability for a simple error, the lower line the probability for a masked simple error. Both values are not cumulative — if an effect is classified as masked simple error, it is not included in the count of simple errors.

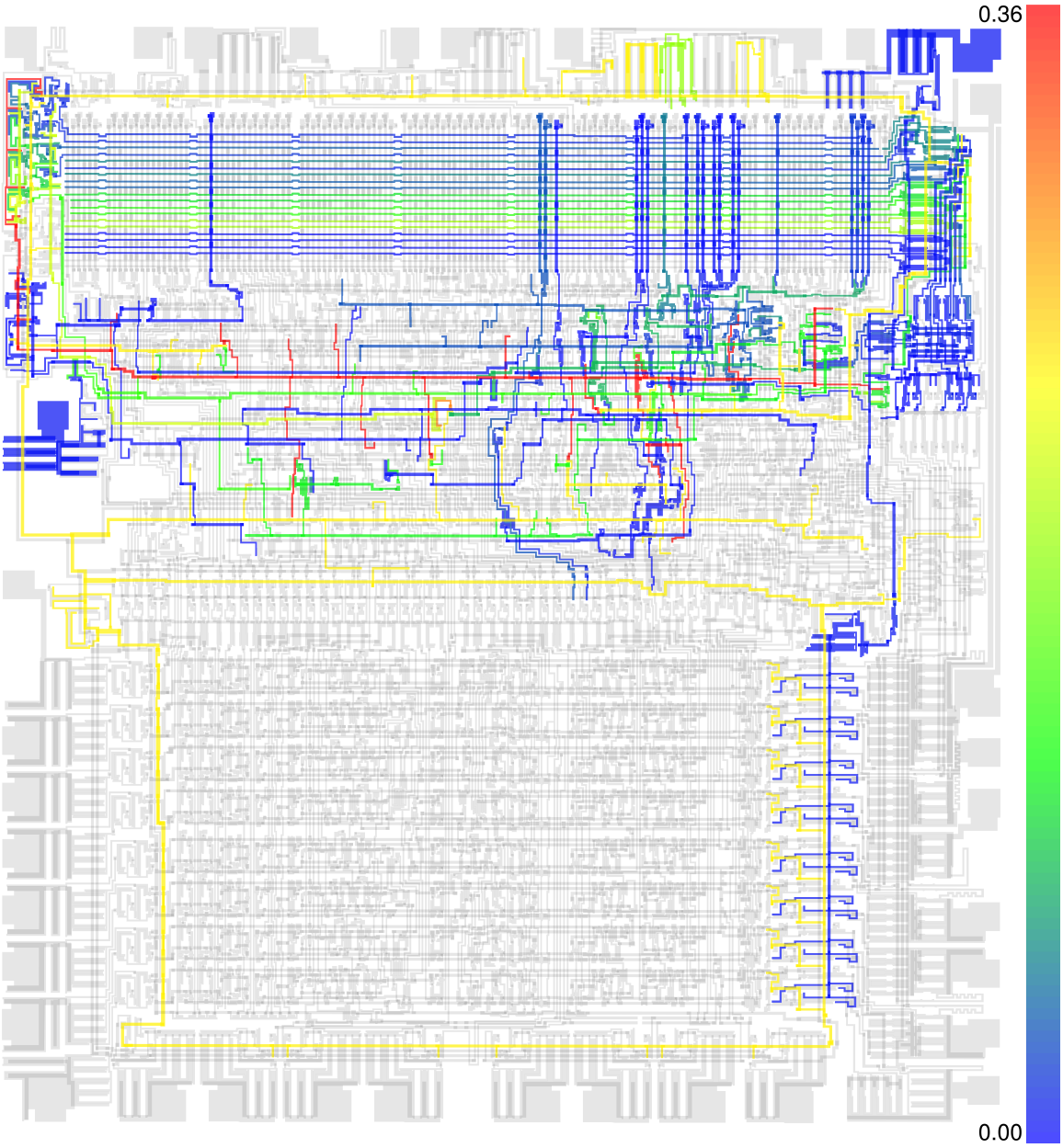


Figure 4.5: Nodes causing deadlocks in the *bugfix* data set

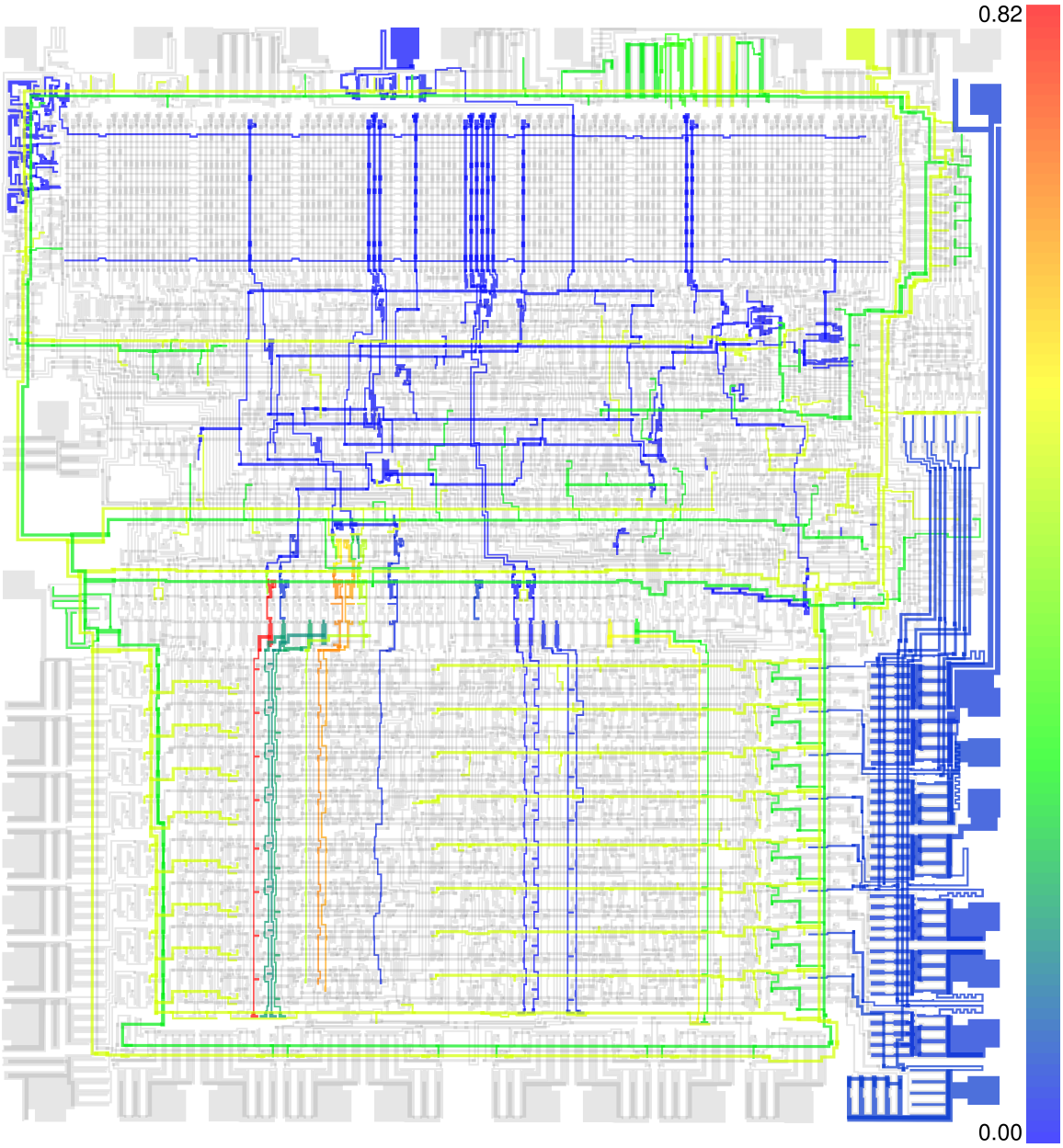


Figure 4.6: Nodes causing simulation failure in the *bugfix* data set

An error is considered masked if it does not affect memory accesses[2] and it does not persist until the end of the test run. This means that during Phi0=1 a single-cycle glitch on the data bus will never be considered masked, even when the instruction that currently executes never uses the value; on the other hand a single-cycle glitch in the program counter may be masked if it happens while the CPU fetches data from another address, so the glitch is not visible on the address bus.

data set	total	A	X	Y	SP	Flags	dbus	abus	t	PC
random	9.64%	1.82%	1.45%	1.52%	1.81%	2.18%	0.85%	-	-	-
	6.97%	0.34%	0.05%	0.04%	0.30%	2.02%	0.13%	2.31%	1.32%	0.45%
bugfix	9.62%	1.81%	1.46%	1.53%	1.81%	2.16%	0.85%	-	-	-
	6.94%	0.34%	0.05%	0.04%	0.30%	2.00%	0.13%	2.31%	1.32%	0.45%
distinct	9.61%	1.81%	1.45%	1.52%	1.78%	2.20%	0.85%	-	-	-
	6.96%	0.34%	0.05%	0.05%	0.30%	2.01%	0.13%	2.31%	1.32%	0.45%
binary	8.54%	1.27%	1.32%	1.38%	1.72%	2.17%	0.67%	-	-	-
	7.22%	0.28%	0.04%	0.04%	0.30%	2.21%	0.13%	2.41%	1.39%	0.44%
nop	10.25%	1.58%	1.58%	1.58%	1.91%	3.15%	0.44%	-	-	-
	6.18%	-	-	-	0.26%	1.27%	0.27%	2.73%	1.66%	-

Table 4.6: Probabilities for simple errors in the data sets

The *nop* data set can serve as a baseline here — it changes no registers or flags, so there are no masking effects in the A, X and Y registers. The difference in the stack pointer probability can be accounted for by its slightly more complicated structure, shown in figure 4.7. Compared to the A, X and Y registers it has an additional connection to the low byte of the address bus (unmarked upper transistor) and a second pass transistor (D). The node that is probed to determine the value of the stack pointer in the simulation data is the right side of transistor D. In this circuit configuration it is possible that the probed node is first modified by an invalid PLA output and later reset to its original value because the left inverter still had its original state.

The increased percentage of flag changes in the *nop* data set is also explained by the nature of the `NOP` instruction because it does not modify flags at all while most other data-handling instructions modify at least the N and Z bits. The flag bits also feature dual pass transistors in their storage cells (connected to Phi1 and Phi2) as well as more complex logic, allowing for a similar appear-and-disappear effect as in the stack pointer case.

The absence of unmasked errors in the abus (address bus) and PC columns is by design — if a change in the program counter persists, it will change the value seen on the address bus, so it would no longer be a simple error because two values are changed. A change on

[2] i.e. when Phi0 is 1, the address- and data bus values and the state of the Read/Write line are identical to the reference

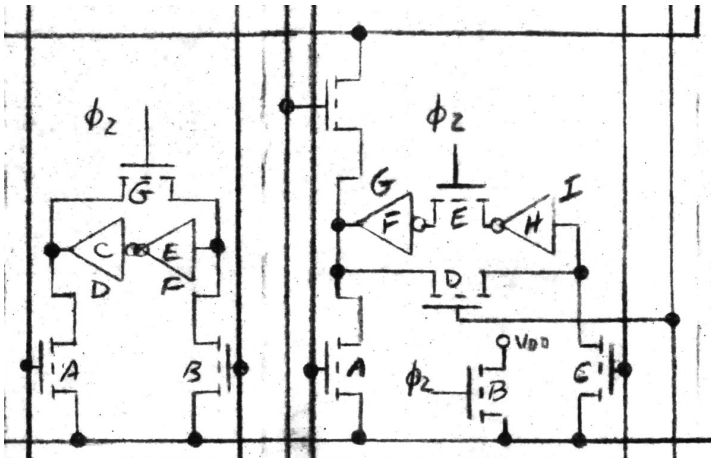


Figure 4.7: Bit 0 of the X register (left) and stack pointer (right), extracted from the 6502 schematics

the address bus during a memory access is also not counted as a simple error as explained in section 3.3.1, so it must be only one half-cycle long. The absence of unmasked errors of the T-state counter (column t) has similar reasons — if the change persists, sooner or later instruction fetches will occur in different half-cycles compared to the reference, so the sync signal does not match and thus more than one value would show a mismatch. The percentage of masked address bus changes is still a bit higher than the other data sets. This seems to be influenced not by the number of data access cycles of the instruction but by the addressing mode. Tests with `CLC`, `DEY`, `LDA` immediate, `LDA` zeropage, `LDA` absolute and `NOP` are shown in table 4.7. `CLC`, `DEY` and `NOP` are all single-byte instructions that complete in two clock cycles and they have the same probability of a masked address bus error. `LDA` immediate completes in the same number of cycles, but does use the byte fetched in the second cycle and shows a different probability. The other two `LDA` instructions show decreased probabilities, but the change is inverted compared to the data access cycles they need. This class of error has not been investigated any further because this error class is defined in a way that should result in no changes in the addresses the CPU actually fetches data from. Even if it triggers, the program execution should not change. A similar argument applies to the masked T state counter changes: If it had an effect on the instruction execution, it would not qualify as a simple error anymore.

A difference between the probability for the X and Y registers can be seen in all data sets that test more than just `NOP`. The X register can be changed by one more opcode than the Y register (`TSX`), but it seems that the difference cannot be accounted for by simply counting the opcodes that affect each register.

instruction	masked abus change probability
CLC	2.80%
DEY	2.80%
LDA zeropage	2.08%
LDA immediate	2.55%
LDA absolute	2.13%
NOP	2.80%

Table 4.7: Probability of a masked address bus change in six selected instructions

The sum of the per-opcode probabilities for simple errors in the four registers (but not flags) are shown in figure 4.8 for the *random* data set. The highest-probability instructions are DEX (0xca) and DEY (0x88) with a slightly lower probability for their incrementing counterparts INX (0xe8) and INY (0xc8). There does not seem to be a clear pattern in general: On the one hand the same opcode can have wildly different probabilities depending on addressing mode, for example LDA is found with both blue-ish and yellow shading in the table (0xb1: 5.49% and 0xa9: 8.43%). On the other hand neither addressing mode nor execution time can be the solely deciding factor as these would create much more consistent patterns in the columns and a small alternating stripe-like variation between adjacent rows. Similarly straight stores into memory (row 8x except 0x81) seem to be more affected by simple errors than stores that use any kind of indexed addressing (0x81 and row 9x), even though the indexed versions have a longer run time which should equate to a larger window of vulnerability. For the stores without indexed addressing the register is connected to the data bus for a longer time relative to the total execution time which may explain at least this difference: While the register is connected to the internal data bus, its contents can be changed if a soft error is currently overriding the value on the data bus. A similar argument could be made about the ALU operations in columns x9 and xA which update the accumulator — the content can be modified both when it is read into the ALU and when the result is written to the register. Such an effect would create different values both before and after the operation when the error hits while the register is read, but simple errors are defined only as an inequality compared to the reference run, not as a value that is changed once and then stays constant. There are examples in the table where the probabilities show the reverse effect to what could be expected though: The TSX instruction copies the stack pointer to the X register and updates the N and Z flag bits and shows a very low probability. On the other hand the TXS instruction — which copies the value the other way around and does not update the flags — shows a higher probability. A theory that errors in TSX may result in changes in both the X register and the flags, thus not qualifying as a simple error was investigated: 7602560 injection runs for each of these opcodes were extracted from the *random* data set and the analysis software was modified to ignore the flags when testing for a simple error. The number of additional

N	V	D	I	Z	C	multiple
13.98%	21.00%	12.11%	14.02%	13.23%	16.42%	9.23%

Table 4.8: Probability of change in a flag bit when a simple (masked) error is detected

injection runs that qualified in these modified conditions was the same for both opcodes: 277. Therefore the difference is caused by a still unknown factor. There also does not seem to be an explanation available for the other register transfer instructions; copying into the accumulator shows an increased simple error probability, but both copying directions update the flag bits and both need to connect the same internal busses. Many other differences remain unexplained too, for example the difference between shifts and rotations and their directions.

Figure 4.9 shows the summed per-opcode probabilities for masked simple errors in the four main registers, also in the *random* data set. This time most of the effects seem to be easier to explain: All instructions that do not write to registers (i.e. all jumps, stores, read-modify-write instructions) show a very low probability of a masked simple error, which is exactly what is to be expected. `PLP` also shows up here as its changes to the flags are excluded from consideration. The ALU instructions that use a register show a difference between `AND` and `ORA` compared to all the other operations (easiest seen in the even rows of columns x9 and xA). The reason for this probably lies in the structure of the ALU implementation as all of these instructions write the updated value to the accumulator in the same cycle of the instruction. The influence of the stack pointer can be seen in `JSR` and `RTS`; the increased probability for masking in `RTI` should be due to the additional stack access and not due to the update of the flag register as changes in it are not listed in this table. All register loads as well as `PLA` have a high chance of masking an erroneous change as they update one register, but curiously the instructions that load the X and Y registers show up with a reduced probability. This probably is caused by the internal bus layout: While the accumulator can be directly connected to the data bus, the X and Y registers are only connected to an internal result bus, increasing the number of nodes where a soft error can influence the register contents while it is updated.

For the flag bits the distribution of changes among the bits were analysed in the *random* data set. Table 4.8 shows the probability for a change in exactly one flag bit given that a simple or masked simple error was found affecting the CPU flags. The difference is calculated only for the first state that differs from the reference run, any subsequent changes are ignored.

The high probability for the V bit is most likely caused by the external pin that can set this bit independent of the executed program. The circuit for this feature uses a synchroniser and edge detector, so it contributes multiple nodes in which a soft error may be injected. The

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-
0.048		0.053		0.058		0.064		0.069		0.074		0.079		0.084		0.089		0.095		0.100

Figure 4.8: Per-opcode probability for a simple error in the *random* data set

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-
0.003		0.004		0.006		0.008		0.010		0.012		0.014		0.015		0.017		0.019		0.021

Figure 4.9: Per-opcode probability for a masked simple error in the *random* data set

values for the other bits seem to fit the expectations based on the features of the instruction set: The D bit can only be affected by explicitly modifying it either with the two set/reset instructions or pulling a flag register value from the stack, so it should be affected less. The N and Z bits are updated by many instructions, although the Z bit has a much higher chance to be cleared rather than set, which should reduce the error probability slightly. The carry bit is only affected by some instructions, but when it is it can not just be changed to a fixed value or restored from the stack as the D bit is, but also in direct response to errors introduced within the ALU. The same is not true for the N and Z flags as they are updated over a separate data path.

4.5 Interrupts

The evaluation specifically excluded the effects of soft errors on external interrupts, but that does not exclude interrupts generated by the errors themselves. The analysis tool attempts to detect these interrupts, verifies if they execute as expected and determines if the interrupt was an IRQ, NMI, Reset or a `BRK` instruction. The distribution of these for the data sets is shown in table 4.9. Once again the *nop* data set seems to indicate a variable probability per opcode and there is an outlier in the percentage of failed interrupt sequences in the *binary* data set. The exact conditions when the analyser considers an interrupt sequence failed are described in section 3.3.1.

data set	IRQ	BRK	NMI	Reset	total	failed
random	0.24%	0.04%	0.61%	0.58%	1.47%	0.01%
bugfix	0.24%	0.04%	0.61%	0.58%	1.47%	0.01%
binary	0.23%	0.05%	0.61%	0.58%	1.48%	0.09%
distinct	0.23%	0.04%	0.61%	0.58%	1.47%	0.00%
nop	0.29%	0.07%	0.62%	0.59%	1.58%	0.00%
total	0.25%	0.05%	0.61%	0.58%	1.49%	0.01%
binary zeroes	0.29%	0.06%	0.61%	0.59%	1.56%	0.18%
binary ones	0.18%	0.04%	0.61%	0.58%	1.41%	0.01%
random I=0	0.29%	0.04%	0.61%	0.58%	1.53%	0.01%
random I=1	0.18%	0.04%	0.61%	0.58%	1.41%	0.01%

Table 4.9: Interrupt probabilities for the data sets

The *binary* data set can be split into all-zero and all-ones subsets which are also shown in the table. The difference is obvious, the all-ones subset shows basically the same behaviour as the other data sets (except for IRQ which will be explained later), while the all-zero subset has a drastically increased percentage of failed interrupts. The detailed analyser output (not shown here) shows that for every data set between 95% and 100% of these failed interrupts

are diagnosed because the number of half-cycles where the IR is zero is too small. Based on this observation it was theorised that the chance for an unsuccessful interrupt is increased when a value of 0x00 is read from memory and unchanged otherwise because the other data sets with a constant memvalue list do not show differences compared to the *random* set. Nine start conditions from the *binary* data set have been isolated to test this theory: The state that has zero in all input values as well as the eight start conditions that have all 0 bits except for one 1 bit in the memory value. The chance for an unsuccessful interrupt for each of these states is presented in table 4.10, clearly showing an increased chance of an interrupt failure when the data bus is 0x00. The most likely explanation is that these cases occur when the instruction register is accidentally updated with the current value on the data bus, changing it to a `BRK` instruction. To verify this the analyser would need to cross-check between start conditions with different memvalue lists to verify that a change in the IR for a given node and injection time combination always corresponds to the current data bus value, but such cross-checks have not been implemented in the current software.

The difference in IRQs between the all-zeros and all-ones subsets can easily be explained by the I-flag. It is almost always 0 in the all-zero subset and almost always 1 in the all-ones subset. The *random* set has been split into two subsets using the state of the I flag, which resulted in two sets with 560 start conditions each. The percentage of generated IRQs for the I=1 case is the same as in the all-ones subset of the *binary* data set, so this difference is caused by the state of the I flag. It should be noted that soft errors were able to trigger an IRQ even when it was disabled by a set I flag, but the much higher percentage of resets and other failures is a more worrying problem than a spurious interrupt at a time where it would not be possible under normal conditions.

memvalue	0x00	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
failure chance	0.20%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%

Table 4.10: Interrupt failure probability depending on memvalue

Finally the differences between opcodes need to be addressed. As there are four types of interrupt which are tracked separately this data is first presented in a single graph in figure 4.10 instead of four separate opcode matrices to allow easier comparison. The x axis is the opcode (undefined opcodes have been left out), the y axis is the probability that a particular interrupts affects the opcode. The plot is for the *bugfix* data set.

At the very left edge the probabilities for `BRK` are a lot lower than those for the other opcodes. This may be caused by two overlapping interrupt sequences from the `BRK` instruction and the error-induced interrupt which have been observed while manually checking the data. They result in various “mixtures” that are not classified as a successful interrupt and thus are not

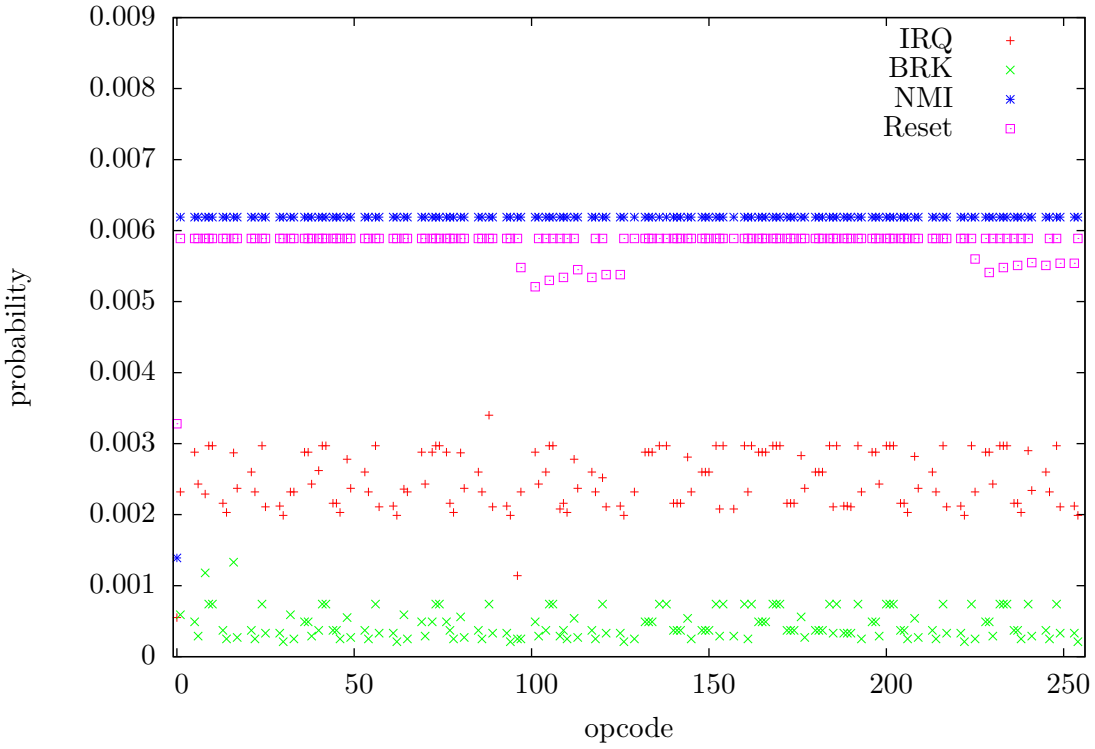


Figure 4.10: Interrupt probabilities per opcode in the *bugfix* data set

counted in the data for the diagram. A very interesting detail is that the probability for an NMI is constant over all opcodes, especially considering that the only difference between an NMI and IRQ is the vector that is fetched by the CPU. The author has currently no explanation for this behaviour. Another open question is the probability for a reset: There are no variations except for 16 opcodes which correspond to all of the versions of `ADC` and `SBC`. The IRQ and BRK values seem to have slight visual similarities especially in the right half of the figure, but the scattered nature of the data makes comparisons hard. They are presented in matrix form in figures 4.11 and 4.12. This time the `BRK` opcode has been omitted from the data to improve the usable dynamic range of the colour scale.

The high probability in the `CLI` instruction in the IRQ table is probably related to the normal function of the instruction: When the I bit is set in the start condition, `CLI` will reset it and thus allow interrupts, with two half-cycles remaining for injection where the I bit is cleared. The reverse seems to be true for `SEI`, which has a noticeable lower probability compared to the other flag-setting instructions in that column. Although there appears to be a pattern related to either the addressing mode or the number of cycles of an opcode, this does not explain all of the variations. An example of this is shown in figure 4.13 where the data was sorted with the minimum cycle count of the instruction as primary key (using 2.5 for branches) and the opcode as the secondary key. Although this results in a visible grouping, the probabilities jump between multiple discrete values. A similar distribution (not shown) was found when the addressing mode was used as the primary sort key. No explanation has been found yet that would account for these variations.

Visually the distribution for a `BRK`-induced soft interrupt seems similar to the distribution for IRQ, although `CLI` and `SEI` now do not stand out — this is to be expected as a soft interrupt is not affected by the state of the I flag. The increased probabilities for opcodes `BPL` and `PHP` might be explained by their binary representations of 0x10 and 0x08 which are both just one bit different from `BRK`, but if this is the cause it is unclear why `ORA` (0x01), `JSR` (0x20) and `RTI` (0x40) are not affected. Since the distribution is mostly similar to the one for IRQ no further attempts for explaining it have been made.

4.6 Control- and Data Flow

Technically interrupts are a change of control flow in the program, but these changes can also be introduced by soft errors that do not generate an interrupt. Table 4.11 summarises the probability of control flow changes for the data sets, ignoring control flow changes caused by successful interrupts. Notable outliers are the *binary* and *nop* data sets. The change in the *nop* data set again suggests a dependency of the control flow change probability on the

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-
0.00%		0.02%		0.05%		0.07%		0.09%		0.11%		0.14%		0.16%		0.18%		0.20%		0.34%

Figure 4.11: Per-opcode probability of an IRQ in the *bugfix* data set

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-
0.00%		0.01%		0.02%		0.03%		0.05%		0.06%		0.07%		0.08%		0.09%		0.10%		0.13%

Figure 4.12: Per-opcode probability of a soft interrupt in the *bugfix* data set

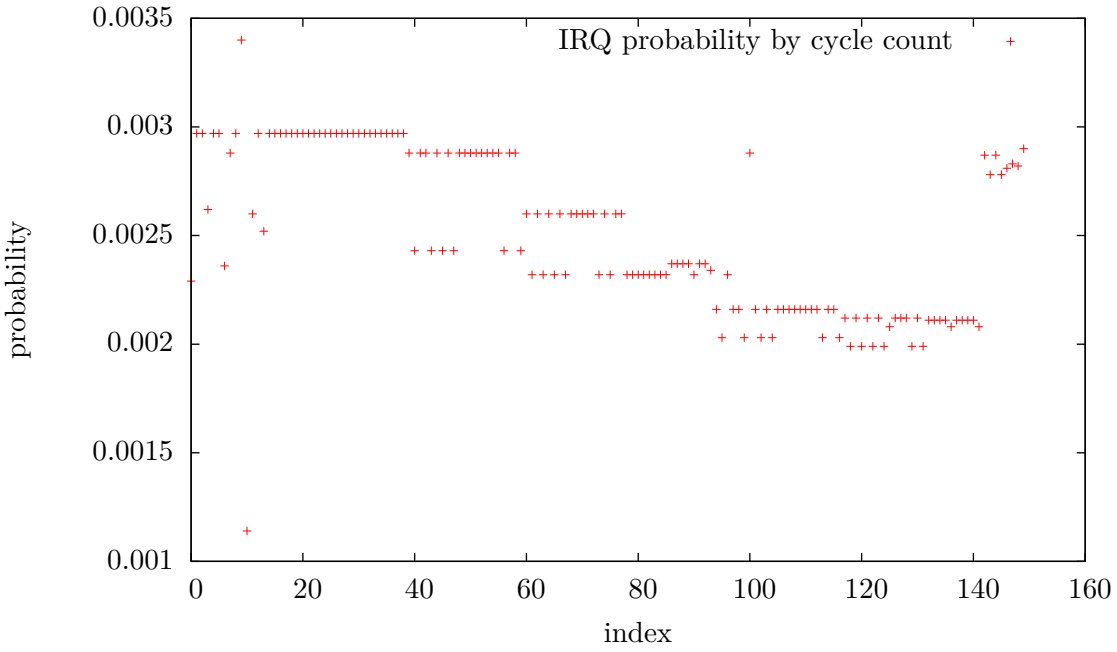


Figure 4.13: Per-opcode IRQ probability sorted by minimum execution time

tested opcode. Figure 4.14 shows the per-opcode probability for the *random* data set. The conditional branches show up with the highest probability in the diagram, but this is to be expected because these instructions are meant to change the control flow depending on one of the CPU flags. If that flag is flipped, the wrong flag is accessed or the comparison logic is affected, all of these instructions will generate a control flow change. On the other hand both `BRK` and `JMP` indirect (opcode 0x6c) shows up with a very low probability. Both of these instructions make multiple memory accesses before fetching the target address, so there are possible injection times before the new value of the program counter is read from memory, realigning the control flow with the reference run. For `BRK` the vector address is predetermined, for `JMP` it depends on the last two data fetches, whose address is determined by the first two data fetches. This means that any address error in one of these four fetches would change the target address, but due to the start conditions of the *random* set the changed address does not matter. This difference can be shown by investigating the control flow change probability for these two opcodes in the *distinct* data set, specifically the subset that uses different memvalue and failvalue lists. In this subset, `BRK` has a 9.96% probability (*random*: 7.61%) for a control flow change, but for `JMP` indirect the probability jumps up to 16.14% (*random*: 8.29%). The increase in probability for `BRK` is most likely caused by interrupt vector fetches from incorrect addresses.

data set	cflow change	data access change	both
random	10.34%	8.60%	1.81%
bugfix	10.27%	8.55%	1.81%
binary	9.47%	7.77%	1.73%
distinct	10.42%	9.02%	2.30%
nop	11.57%	3.18%	1.24%
total	10.56%	7.23%	1.75%
binary-0	9.76%	7.33%	1.81%
binary-1	9.18%	8.19%	1.64%
distinct (fail = mem)	10.27%	8.64%	1.81%
distinct (fail ≠ mem)	10.58%	9.39%	2.80%

Table 4.11: Control flow and data access change probabilities for all data sets

The much higher control flow change probability in the `RTS` instruction compared to the `RTI` instruction also seems simple to explain: Although both instructions execute in the same number of cycles, the `RTI` instruction pops the flag register from the stack while the `RTS` instruction uses the same cycle for a dummy memory access and to increment the program counter. That means the program counter is loaded with the value read from stack one CPU-cycle earlier and thus the window of vulnerability is larger. A similar argument may apply to the two instances of `JMP`: In both the program counter has its new value in the last CPU cycle, just in time to fetch the next instruction and vulnerable to change by a soft error. In the indirect version of the instruction (opcode 0x6c) this window is 20% of the total execution time, but in the non-indirect version (opcode 0x4c) the relative time increases to 33%. For the `JSR` instruction that window is even smaller at 16%, but when the 6502 executes a `JSR` it first fetches the low-byte of the target address before pushing the current program counter to the stack. This opens an additional window with a length of two CPU cycles during which the low-byte of the target address can be modified by a soft error. The reduced probability in column x1 (zero-page indirect instructions) can probably be attributed to the fact that all these instructions are only two bytes long, but have relatively long execution times (five to six CPU cycles). This means that they use the PC incrementer only once during their execution, reducing the vulnerability to errors in that sub-circuit.

The other outlier that needs addressing is the drop in the *binary* data set. As shown in the table splitting the set into the all-zeroes and all-ones subsets did not yield an immediate explanation as both still have a probability below that of the *random* data set. Investigation of each of the start conditions of the *binary* set has shown differences depending on the bits set in the program counter though. Figure 4.15 is generated from 16 start conditions where all values are zero except for a single bit in the program counter plus another all-zero start condition. Clearly set bits in the high byte of the program counter increase the probability

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-

0.076	0.085	0.094	0.102	0.111	0.120	0.129	0.137	0.146	0.155	0.164

Figure 4.14: Per-opcode probability of control flow change in the *random* data set

of a control flow change, although smaller changes in the low eight bits are also visible. This also explains why the all-ones subset of the *binary* data set still showed a lower probability than the other data sets: In most start conditions of that subset the program counter will flip over from 0xffff to 0x0000 after the test instruction is loaded into the CPU, creating a similar situation as in the all-zero subset.

Table 4.11 also shows the probability that a data access changes as a result of a soft error. The reduced value for the *NOP* data set is clearly caused by the fact that NOP instructions normally do not have any data accesses[3] at all — the value in the table is the result of data accesses that were caused by soft errors. Like control flow changes the value does include any successful interrupt sequences. The *distinct* set has a slightly higher percentage here which disappears when only the subset is considered where the same list is used for memvalues and failvalues. A likely cause for this are differences in indirect addressing in which the data accesses depend on the value read from memory. The *binary* data set again shows slightly lower failure probability which stays low when the set is split, but this time with a distinction between both subsets. The all-ones data set should have more data accesses than the all-zeros set though as every indexed access in the all-ones set needs an additional cycle to calculate the address overflow. Although all values presented here are relative to the

[3] A data access was defined as any memory access to an address that is different from the current program counter.

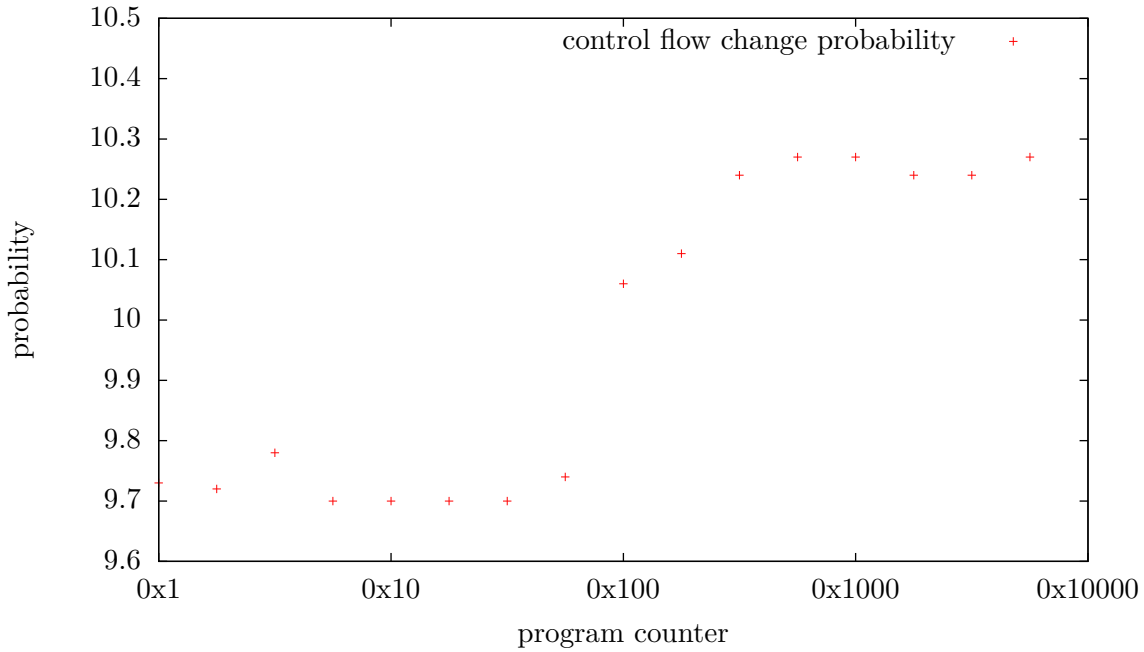


Figure 4.15: Probability of control flow change depending on the initial program counter value

number of injection runs — which means they are effectively compensated for different counts of injection times based on the instruction run time — this only normalises against the total run time of the instruction. The ratio between instruction and data fetches changes towards the data fetches for these cases. This still leaves the question why the *binary* data set shows reduced probabilities in general, so the probability of all sixteen all-zero start conditions is shown in figure 4.16. The number of the bit that is set to 1 is on the x axis, the y axis shows the corresponding probability. There is a curious increase in probability for bit one of the X and Y register as well as the low byte of the program counter and the memory value and a smaller increase for bit zero of the same values, but the reason is unclear. It cannot simply be due to additional data accesses when there is an address overflow in indexed addressing as either the base or the index register must be zero and even their non-zero values are at best 0x80. There is some additional logic that is not connected to all bits of the internal data paths to implement BCD arithmetic, but it cannot be the reason for this difference either — it is not connected to bit zero, but it is present on bits one to seven. Examining the same probabilities for the all-ones data set does not seem to suggest an explanation either. Now the low byte of the program counter stands out as the outlier, probably because this data set guarantees that there will be an overflow within the simulated cycles when the clear bit is in the lowest three bits of the program counter with a chance for a delayed failure during the trailing `NOP` instructions in bits three and four. Instead of a peak there is now a reduction in the memory value and the affected bit has shifted from two to three. The exact causes

data set	instruction overwrite
random	0.60%
bugfix	0.59%
binary	0.99%
distinct	0.59%
nop	1.38%

Table 4.12: Probability of instruction overwrites for all data sets

for data flow changes remain an open question, but it seems clear that the exact values in multiple registers at the time of the soft error can have a large effect on the impact of the error.

4.7 Instruction Overwrite

Some soft errors can cause unusual memory write patterns on the bus of the 6502. One change that can cause either catastrophic results or be of no consequence in itself are erroneous writes that change the program code itself. If the program runs from read-only memory, such writes do not matter at all, although other effects of the same error may still cause problems. If the program runs from RAM though, the instruction stream is modified. The analyser detects writes to memory addresses that match one of the program counter values during the reference run. This check is skipped if the reference run also overwrites one of these memory locations[4]. Table 4.12 shows the probabilities of this error for all data sets. The increased percentage in the *binary* data set are to be expected: As all CPU registers as well as the program counter and memory value are initialised with identical values except for a single bit, the simulated instructions execute in exactly the place where most memory accesses occur. This obviously increases the chance that an accidental write uses the same memory area, raising the percentage for the "instruction overwrite" classification. As the *nop* data set result shows this category varies with the instruction.

The per-opcode probability based on the *random* data set is shown in figure 4.18. `NOP` stands out with the highest probability even though it does not use any memory writes under normal circumstances. The initial expectation was that the two-byte immediate instructions would show an even higher probability as both them and `NOP` do not make any data accesses, but `NOP` accesses only half as many program counter locations in a single test run. On the other hand the repeating values in the program counter and address bus may increase the chance to hit any of the memory locations which would raise `NOP` and the one-byte, two-cycle instructions to the same probability. Neither of these intuitions hold — the instructions

[4]The use of self-modifying code is considered a valid technique among 6502 assembler programmers.

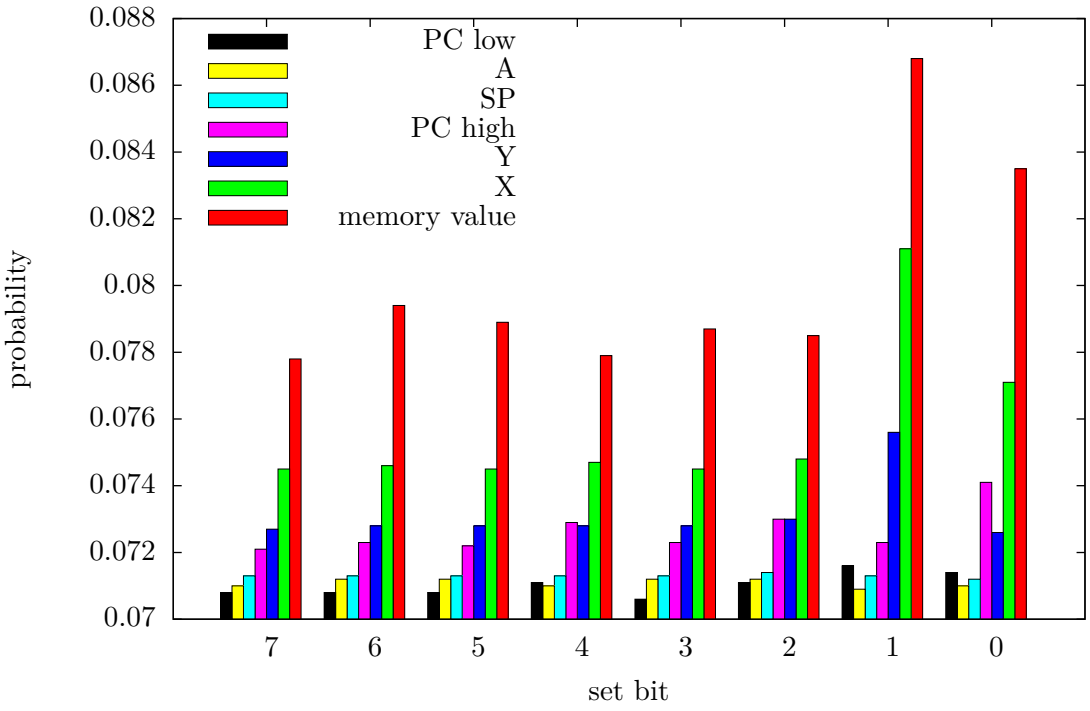


Figure 4.16: Per-bit probability of data access change with one bit set in the start condition

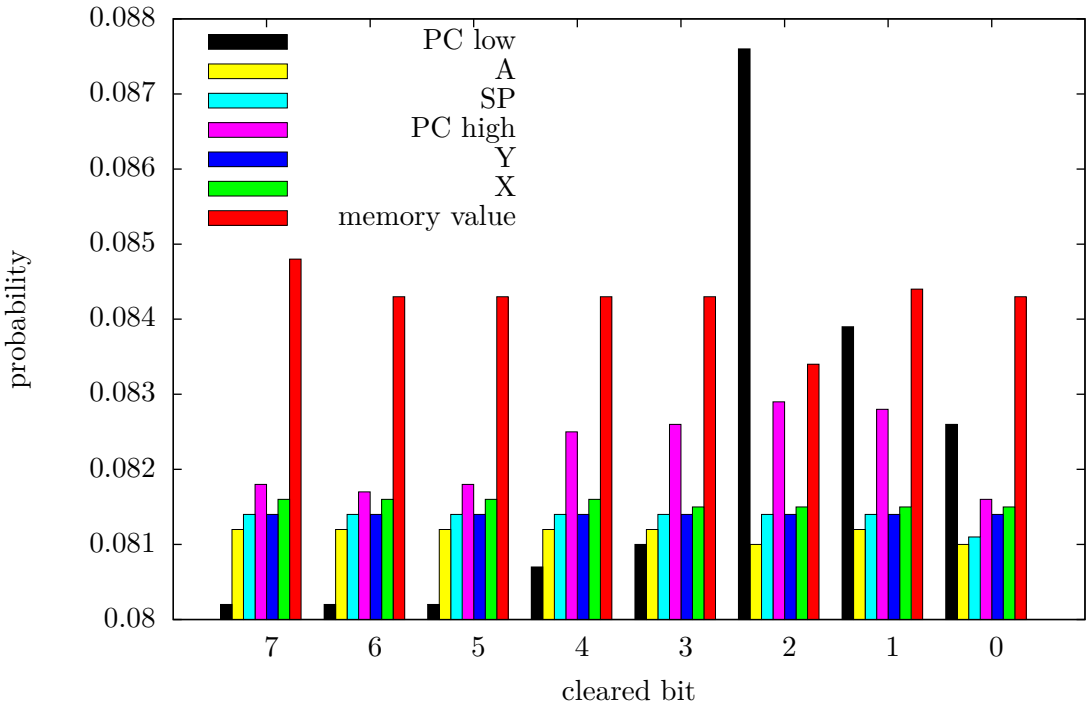


Figure 4.17: Per-bit probability of data access change with one bit cleared in the start condition

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA	-	-	-	ORA	ASL	-	PHP	ORA	ASL	-	-	ORA	ASL	-
1x	BPL	ORA	-	-	-	ORA	ASL	-	CLC	ORA	-	-	-	ORA	ASL	-
2x	JSR	AND	-	-	BIT	AND	ROL	-	PLP	AND	ROL	-	BIT	AND	ROL	-
3x	BMI	AND	-	-	-	AND	ROL	-	SEC	AND	-	-	-	AND	ROL	-
4x	RTI	EOR	-	-	-	EOR	LSR	-	PHA	EOR	LSR	-	JMP	EOR	LSR	-
5x	BVC	EOR	-	-	-	EOR	LSR	-	CLI	EOR	-	-	-	EOR	LSR	-
6x	RTS	ADC	-	-	-	ADC	ROR	-	PLA	ADC	ROR	-	JMP	ADC	ROR	-
7x	BVS	ADC	-	-	-	ADC	ROR	-	SEI	ADC	-	-	-	ADC	ROR	-
8x	-	STA	-	-	STY	STA	STX	-	DEY	-	TXA	-	STY	STA	STX	-
9x	BCC	STA	-	-	STY	STA	STX	-	TYA	STA	TXS	-	-	STA	-	-
Ax	LDY	LDA	LDX	-	LDY	LDA	LDX	-	TAY	LDA	TAX	-	LDY	LDA	LDX	-
Bx	BCS	LDA	-	-	LDY	LDA	LDX	-	CLV	LDA	TSX	-	LDY	LDA	LDX	-
Cx	CPY	CMP	-	-	CPY	CMP	DEC	-	INY	CMP	DEX	-	CPY	CMP	DEC	-
Dx	BNE	CMP	-	-	-	CMP	DEC	-	CLD	CMP	-	-	-	CMP	DEC	-
Ex	CPX	SBC	-	-	CPX	SBC	INC	-	INX	SBC	NOP	-	CPX	SBC	INC	-
Fx	BEQ	SBC	-	-	-	SBC	INC	-	SED	SBC	-	-	-	SBC	INC	-

0.002 0.004 0.005 0.006 0.007 0.008 0.009 0.010 0.011 0.013 0.014

Figure 4.18: Per-opcode probability of instruction overwrites in the *random* data set

with immediate addressing (for example the `LD` immediate family at 0xa0, 0xa2 and 0xa9) show a much reduced probability as do some but not all of the register transfer instructions while the flag-changing instructions are bit more vulnerable than for example the accumulator shifts/rotates. The `BCC` instruction also shows a noticeably raised probability compared to the other conditional branches, the reasons are unclear. There does seem to be a trend towards lower probability for instructions that access the zero page or stack: As they force a fixed value on the upper half of the address bus, the chance to overwrite instructions that are more likely to be outside of these two areas is reduced. Globally there seems to be a higher vulnerability for shorter instructions, so it is possible that the most likely cycle for overwriting an instruction is the opcode fetch and not any of the data access cycles. This would raise the probability shown here as it is calculated relative to the total number of injection runs per opcode, resulting in a varying ratio of opcode fetches to other memory accesses. Unfortunately due to time constraints it was not possible to investigate this phenomenon further. Attempts to characterise the probability of instruction overwrite in relation to the values in the starting conditions seemed to indicate a dependency on the program counter value that extends beyond a simple match between the location of the instructions and the general area in which data accesses should occur due to the starting conditions, but again lack of time prevented a detailed investigation of these relations.

4.8 Effect Delay

During the investigation of instruction overwrites an interesting effect was noticed in some of the affected injection runs. An example of this effect is shown in figure 4.19. In this run a `NOP` instruction was tested and the error was injected at half-cycle 3. The execution continues without any visible effects until half-cycle 6, during which the memory bus switches to a write. The CPU reads 0xfc instead of the actual memory value and executes it as an instruction, an undocumented opcode that works as a `NOP` with absolute addressing. Subsequent opcode fetches are not affected and the execution continues normally after opcode 0xfc has completed.

```
 0: 0001: r:ea w:-- R _ phi0:0 PC:0001 A:00 X:00 Y:00 S:00 P:30 IR:ea T:06
 1: 0001: r:00 w:-- R _ phi0:1 PC:0001 A:00 X:00 Y:00 S:00 P:30 IR:ea T:06
 2: 0001: r:00 w:-- R S phi0:0 PC:0001 A:00 X:00 Y:00 S:00 P:30 IR:ea T:01
 3: 0001: r:ea w:-- R S phi0:1 PC:0001 A:00 X:00 Y:00 S:00 P:30 IR:ea T:01
 4: 0002: r:ea w:-- R _ phi0:0 PC:0002 A:00 X:00 Y:00 S:00 P:30 IR:ea T:06
 5: 0002: r:00 w:-- R _ phi0:1 PC:0002 A:00 X:00 Y:00 S:00 P:30 IR:ea T:06
 6: 0002: r:-- w:00 w S phi0:0 PC:0002 A:00 X:00 Y:00 S:00 P:b0 IR:ea T:01
 7: 0002: r:-- w:fc w S phi0:1 PC:0002 A:00 X:00 Y:00 S:00 P:b0 IR:ea T:01
 8: 0003: r:00 w:-- R _ phi0:0 PC:0003 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:04
 9: 0003: r:00 w:-- R _ phi0:1 PC:0003 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:04
10: 0004: r:00 w:-- R _ phi0:0 PC:0004 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:08
11: 0004: r:00 w:-- R _ phi0:1 PC:0004 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:08
12: 0000: r:00 w:-- R _ phi0:0 PC:0005 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:02
13: 0000: r:00 w:-- R _ phi0:1 PC:0005 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:02
14: 0005: r:00 w:-- R S phi0:0 PC:0005 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:01
15: 0005: r:ea w:-- R S phi0:1 PC:0005 A:00 X:00 Y:00 S:00 P:b0 IR:fc T:01
16: 0006: r:ea w:-- R _ phi0:0 PC:0006 A:00 X:00 Y:00 S:00 P:b0 IR:ea T:06
17: 0006: r:00 w:-- R _ phi0:1 PC:0006 A:00 X:00 Y:00 S:00 P:b0 IR:ea T:06
18: 0006: r:00 w:-- R S phi0:0 PC:0006 A:00 X:00 Y:00 S:00 P:b0 IR:ea T:01
19: 0006: r:ea w:-- R S phi0:1 PC:0006 A:00 X:00 Y:00 S:00 P:b0 IR:ea T:01
```

Figure 4.19: Delayed effect during error injection in half-cycle 3 at node 504

This delay between the injection of the soft error and its effect prompted a short analysis of the distribution of this delay which is shown in figure 4.20 (please note the logarithmic y scale). Unfortunately the existing analyser output does not include an indication of the length of the tested instruction, so the histogram only shows the difference in half-cycles between error injection and the first visible effect, ignoring if this effect still appears within the tested instruction or if it appears in a subsequent instruction.

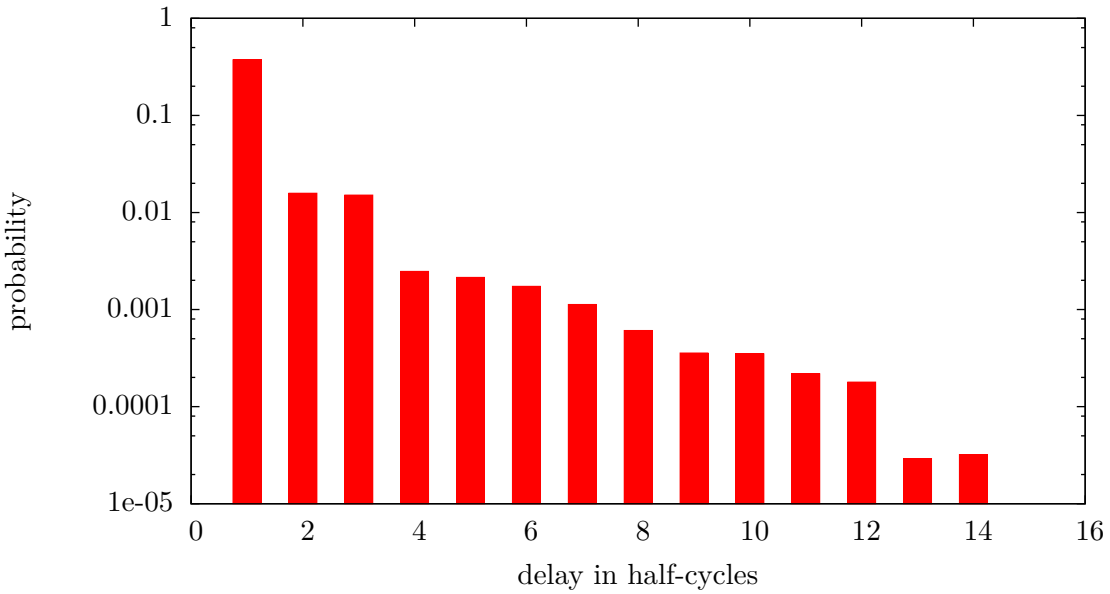


Figure 4.20: Histogram of effect delay in half-cycles for the *random* data set

Due to the way the simulation works, there can be no delay of zero half-cycles as the error is injected after the current CPU state has been sent to the output file. Most of the errors (37.4% of all test runs) do appear immediately though and the probability mostly decreases with increasing delay with a slight increase at the very end. It is unfortunate that no data is currently available to calculate the delay between the end of the tested instruction and the appearance of the error as a delay of 14 cycles could also correspond to an error that injected at half-cycle 0 of a seven clock-cycle instruction which appears in the first cycle of the next instruction. As the example trace in figure 4.19 shows an error may only manifest in a later instruction, so additional tests that isolate these errors and test the outcome of various instructions at that point remain as an open topic for future work.

model	no effect	hidden effect	simulator failure	interrupts	simple error
lock 0	81.41%	1.56%	0.13%	0.68%	6.89%
lock 1	73.03%	3.08%	0.41%	0.88%	9.90%
lock invert	56.25%	4.54%	0.44%	1.56%	15.79%
lock current	98.19%	0.10%	0.10%	none	1.00%
instant invert	90.92%	2.83%	0.03%	0.42%	2.56%

Table 4.13: Probabilities under different fault models

4.9 Alternative Fault Models

As noted in section 3.2.2 there were multiple choices available for the fault model implementation. The current work has chosen to use the *invert* model which locks the node affected by the soft error to its inverted pre-fault state while the error is active. Limited simulations have been made with the other three available models and their results are summarised in table 4.13. All runs in the table were made with the bug-fixed version of the simulator and use a start condition with all values set to zero. As expected the *invert* model has proven to be the one that produces the most effects on the simulated CPU. The column for "simple error" includes both masked and unmasked simple errors.

In addition to these four models a fifth version was tested that does not lock the error node to any value but simply inverts its current value, corresponding to a soft error with instantaneous duration (listed in the table as *instant invert*). Since the value of the node is not locked, it is possible that it is immediately flipped back by stronger signal sources in the rest of the circuit, resulting in a high probability that a given injection run does not experience any changes at all.

4.10 Other Uncategorised Runs

The various methods implemented for the evaluation categorised a total of 72.43% of all injection runs of the *bugfix* data set. This value excludes the runs with hidden effect which may add another 5.07% to that percentage. For an additional 17.01% of all runs a control flow and/or data access error could be diagnosed even though they were not completely categorised. Among these uncategorised results are many that exhibit arbitrarily weird CPU behaviour, one example is shown in figure 4.21.

This example would be tagged both as a control flow as well as a data access change, but these tags do not tell the whole story. The tested instruction is `JSR`, but during the injection a bit flips in the instruction register at half-cycle 3 and the opcode changes to `BMI`. The

```
 0: 1001: r:20 w:-- R _ phi0:0 PC:1001 A:00 X:00 Y:00 S:00 P:30 IR:20 T:04
 1: 1001: r:33 w:-- R _ phi0:1 PC:1001 A:00 X:00 Y:00 S:00 P:30 IR:20 T:04
 2: 0100: r:33 w:-- R _ phi0:0 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:08
 3: 0100: r:33 w:-- R _ phi0:1 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:08
 4: 0100: r:-- w:33 W _ phi0:0 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:10
 5: 0100: r:-- w:10 W _ phi0:1 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:10
 6: 01ff: r:-- w:33 W _ phi0:0 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:20
 7: 01ff: r:-- w:02 W _ phi0:1 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:20
 8: 1002: r:33 w:-- R _ phi0:0 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:02
 9: 1002: r:33 w:-- R _ phi0:1 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:02
10: 3333: r:33 w:-- R S phi0:0 PC:3333 A:00 X:00 Y:00 S:fe P:30 IR:20 T:01
11: 3333: r:ea w:-- R S phi0:1 PC:3333 A:00 X:00 Y:00 S:fe P:30 IR:20 T:01
---------------------------------------------------------------------------
 0: 1001: r:20 w:-- R _ phi0:0 PC:1001 A:00 X:00 Y:00 S:00 P:30 IR:20 T:04
 1: 1001: r:33 w:-- R _ phi0:1 PC:1001 A:00 X:00 Y:00 S:00 P:30 IR:20 T:04
 2: 0100: r:33 w:-- R _ phi0:0 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:20 T:08
 3: 0100: r:33 w:-- R _ phi0:1 PC:1002 A:00 X:00 Y:00 S:33 P:30 IR:30 T:08
 4: 0100: r:33 w:-- R _ phi0:0 PC:1000 A:00 X:00 Y:00 S:33 P:30 IR:30 T:02
 5: 0100: r:33 w:-- R _ phi0:1 PC:1000 A:00 X:00 Y:00 S:33 P:30 IR:30 T:02
 6: 0f00: r:33 w:-- R S phi0:0 PC:0f00 A:00 X:00 Y:00 S:33 P:30 IR:30 T:01
 7: 0f00: r:ea w:-- R S phi0:1 PC:0f00 A:00 X:00 Y:00 S:33 P:30 IR:30 T:01
 8: 0f01: r:ea w:-- R _ phi0:0 PC:0f01 A:00 X:00 Y:00 S:33 P:30 IR:ea T:06
 9: 0f01: r:33 w:-- R _ phi0:1 PC:0f01 A:00 X:00 Y:00 S:33 P:30 IR:ea T:06
10: 0f01: r:33 w:-- R S phi0:0 PC:0f01 A:00 X:00 Y:00 S:33 P:30 IR:ea T:01
11: 0f01: r:ea w:-- R S phi0:1 PC:0f01 A:00 X:00 Y:00 S:33 P:30 IR:ea T:01
```

Figure 4.21: Example of an uncategorised test run; reference at the top

following states do not match the behaviour of that instruction either, the final result is that execution resumes at 0x0f00 which is nowhere near the intended target address of 0x3333 and the contents of the stack pointer have changed to 0x33 instead. Although it would be possible to define an algorithm that could categorise this example as a combination of a modified stack pointer, a control flow change and a missing return address on the stack, the problem at this point are diminishing returns. The example above is just one of many where the error result is obvious to a human but not that easy to describe algorithmically; other examples are less obvious, especially when a partial interrupt sequence combines with something else. Still, successful categorisation of more than 72% of all errors (or 95% if hidden, control flow and data access errors are accepted as fully categorised) is a pleasingly high success rate.

5 Summary

The goal of this work was to analyse the instruction execution of a small microprocessor in the presence of soft errors and to categorise the changes caused by the errors. To achieve a higher accuracy a relatively low-level simulation approach was used in the form of switch-level modelling. The choice of CPU allowed for an exhaustive coverage of single instructions as well as temporal and spatial soft error locations. A combination of specialised simulation and analysis software was presented that allowed automated simulation of this CPU with simulated soft errors and attempted to categorise the changes introduced by these errors.

This categorisation was achieved by comparing an error-free reference simulation to the error-affected simulation as well as verification of known traits of the simulated instruction sequence to separate error-affected from error-free parts of the simulation results even when it ceased to match the reference simulation. The fault model used for injecting soft errors was chosen to maximise the amount of changes created by the errors. The choice was successfully validated by comparing the results of the analysis software on a small set of simulations created with each of the available fault models. Five major categories of error effects were defined with a varying number of sub-categories as well as three additional tests that give an indication about the effect of the simulated error without resulting in a full categorisation.

The evaluation was based of five data sets created using different selections of start conditions for the simulated CPU. More than 75% of the simulated soft errors could be successfully categorised, for another 15-18% a non-categorising indication about the effects of the error could be assigned. The most surprising result was that even with the choice of a worst-case fault model more than 50% of all simulated soft errors did not influence the operation of the CPU at all. The other major categories are CPU deadlocks, interrupts (including reset), single-register changes and simulation failures. The additional tests indicate changes in control flow, data accesses and write accesses to the memory locations of the instructions when none of the major categories could be assigned.

For most of these the per-opcode probability of failure is examined as well as the dependency on the starting conditions of the simulation. In some cases changes in probability between opcodes or dependency on the start conditions could be successfully isolated and sometimes explained based on features of the CPU implementation, in others no reasonable pattern

linking the error probability to opcodes, start conditions or CPU features could be found. In addition to the categorisation of errors a very limited examination of the delay between the injection of a soft error and its first visible influence on the CPU behaviour was undertaken which shows that the delay is limited even though there seem to be cases where a complete instruction is executed before the effects of the error become visible.

There are many options for future research. It would be very interesting to examine the changes in susceptibility to soft errors between the fully hand-designed NMOS 6502 and later CMOS versions of the chip or to versions that are partially or completely created using CAD tools — for example the newer 8500 HMOS variant appears to a completely redesigned implementation using CAD layout tools [2]. Another option could be to extend a similar analysis to a different microprocessor: For example the Cortex-M0 ARM CPU is claimed to have between 32.000 and 50.000 transistors depending on the source, which is about one order of magnitude larger than the 6502 that was investigated in this work. Using an actual cluster instead of a small set of machines that happened to be available for the simulation it should be possible to perform a similar evaluation for this current-generation CPU.

Even without switching targets there still are interesting questions within the 6502 that remain unanswered. One example are externally-triggered interrupts: In many applications the processor would become a lot less useful if it suddenly would be unable to successfully process external interrupts due to a soft error. Other open questions include the exact mechanisms of some of the errors already investigated here; there were a few categories where the per-opcode distribution could only be described as “weird” with no explanation readily available.

One topic that spans both these directions is the possibility of reducing the susceptibility to errors of the 6502 by introducing modifications to the circuit or by adding external hardware which attempts to detect errors that are visible externally — a kind of extended watch dog circuit that does not trigger just based on time but also CPU bus conditions that do not appear in normal operation. A similar approach with closer coupling to the CPU has been proposed in the form of checker cores [3] which verify the correct execution of instructions that are about to be retired from a multi-issue out-of-order core.

Appendix

A Three-level simulation issues

The goal of three-level simulation with an additional X state is to set every signal that would change in response to an external input to the X state before it takes on its new state. allows the detection of races and other instabilities within the simulated circuit [18], but this advantage turns into a problem when the circuit was intentionally designed to rely on switching delays for its operation. The main problem source in the 6502 is the clock generator which creates two non-overlapping clock phases Phi1 and Phi2 from the external clock signal. Using three-level simulation both clock phases first switch to X before they settle on their final value. This appendix presents an example of a subcircuit of the 6502 that will operate incorrectly in these circumstances.

Figure A.1 shows the circuit of the upper four bits of the T-state counter. It is basically a resetable shift register built in dynamic logic that shifts the signal received from $T1$ out to $\overline{T2}$...$\overline{T5}$ while the notRdy0 signal is 0 or keeps it current state when it is 1. For the purposes of the following discussion both the reset signal and notRdy0 can be assumed to be statically 0.

Three-level simulation is appliciable to both gate-level and switch-level simulation, which allows the following discussion to ignore switch-level details within the gates of the circuit diagram. For gates, the X state is an extension of the usual boolean logic truth tables where an X on the input results in an X on the output if the output of the gate is different if the X was a 0 or 1 instead. The extended truth tables for the three types of gates in the circuit are shown in table A.1.

For the pass transistors, simulation rules from switch-level simulation are used: If any transistors receive an X on their gate, two simulation passes must be made. In the first pass the X'ed transistors only propagate 0 and X values, in the second pass these transistors only propagate 1 and X values. Any node whose calculated value differs between those two passes is set to X [7].

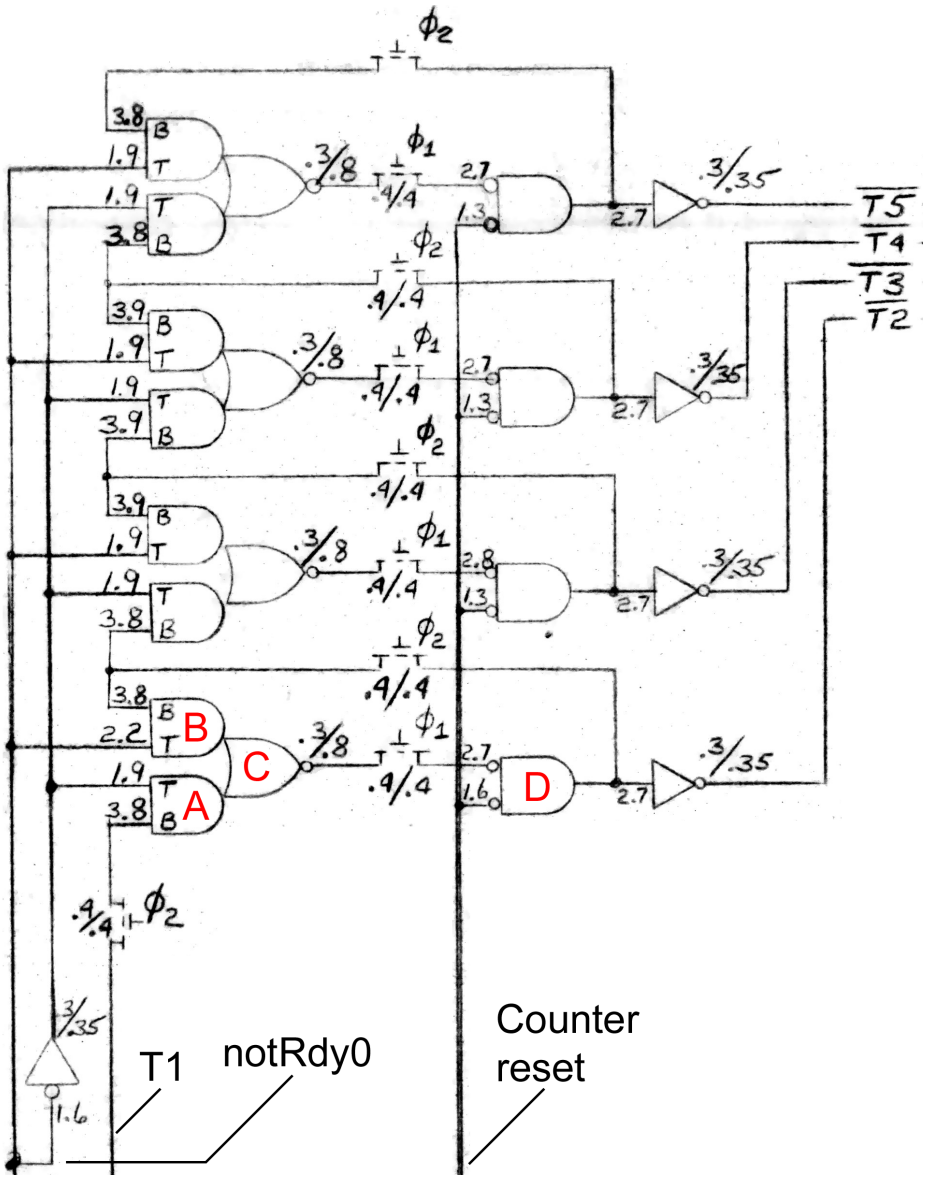


Figure A.1: T-state counter bits T2 to T5, from the 6502 schematics

a	b	NOT a	a AND b	a NOR b
0	0	1	0	1
0	1	1	0	0
0	X	1	0	X
1	0	0	0	0
1	1	0	1	0
1	X	0	X	0
X	0	X	0	X
X	1	X	X	0
X	X	X	X	X

Table A.1: extended truth tables for AND, NOR and NOT gates

The initial state for this example is the moment just before a 1 value on $T1$ is passed into the shifter. $T2$, $T3$, $T4$ and $T5$ are all 0, so their inverted outputs are 1. Phi1 is 1, so Phi2 must be 0 and because T1 has just changed to 1, the node between the Phi2 pass transistor and the B input of gate A is still 0. As mentioned above both notRdy0 and the reset line will be kept at 0 all the time. The output of gate C follows from these conditions and is 1.

Phi1	Phi2	T1	Cout	T2	T3	T4	T5
1	0	1	1	0	0	0	0

When the next edge of the master clock arrives, it first transitions to X which will set both Phi2 and Phi1 to X. Based on the switch-level propagation rules the $T1$ pass transistor will pass the 1 value on T1 in the "1 or X" pass, but not in the "0 or X" pass. This yields different results for the node connected to the B input of gate A in both passes, so this node must be X now. The T input of this gate is 1, so its output is X according to the extended truth tables. Gate C receives a 0 on its other input (notRdy is 0, so gate B must output 0), so it also outputs an X which is passed by the Phi1 pass transistor. Gate D calculates X NOR 0, so $T2$ and its inverse $\overline{T2}$ are now both X. Since this X value is also passed by the Phi2 pass transistor that connects the $T2$ signal to the input of the next stage, the inputs for $T3$ are the same as for $T2$ and it follows that all of $T2$ to $T5$ must switch to X.

Phi1	Phi2	T1	Cout	T2	T3	T4	T5
X	X	1	X	X	X	X	X

The summary of the new state is rather simple: The $T1$ input signal is 1, notRdy0 and reset are 0, all other signals are X. We can now set the two clock phases to their new values, Phi1 to 0 and Phi2 to 1. The $T1$ pass transistor will pass the 1 value to the lower input of gate A, which will switch to an 1 output as result. Gate C now has 0 and 1 on its inputs, so it switches to 0. The value is not propagated yet because Phi1 is 0 and the remainder of the circuit keeps its current X state.

Phi1	Phi2	T1	Cout	T2	T3	T4	T5
0	1	1	0	X	X	X	X

As there is now nothing more to do in this part of the circuit, we can clock in the next edge of the master clock. Both Phi1 and Phi2 first switch to X. At this point we must assume that $T1$ also changes its state[1], so it also moves to an X value for now. The B input of gate A will switch to X because of that and the output of the gate responds with X. The output of B is still 0 because of the 0 on notRdy0, so gate C calculates 0 NOR X, resulting in X. This is also the current value of the upper input node of gate D, so it doesn't really matter that the pass transistor connecting those two currently has an X state on its gate and the remainder of the circuit stays in X state.

Phi1	Phi2	T1	Cout	T2	T3	T4	T5
X	X	X	X	X	X	X	X

We can now switch to the final input under consideration: Phi1 and Phi2 move from the X state to Phi1 1 and Phi2 0 and T1 switches to 0. The output value of gate C could now propagate to the input of gate D, but as we saw the output has just switched to X again, so gate D again sees no changes and $T2$ to $T5$ all stay in the X state.

Phi1	Phi2	T1	Cout	T2	T3	T4	T5
1	0	0	X	X	X	X	X

As we can see the clock phases are now the same as in the initial state, so the shift register should have copied the input signal to its first output. Instead of this the outputs are now all in the X state, resulting in a completely unusable input to the PLA and a processor that would not work. Even if the assumption that $T1$ changes back is wrong the outputs would be 1XXX instead of the expected value 1000, which would also result in unusable data from the PLA and a nonworking CPU.

The demonstrated issue could be fixed by removing the internal clock generation logic and driving the Phi1 and Phi2 nets with explicit non-overlapping clock signals generated by the simulator, effectively changing the 6502 into a 6501. This option was not pursued however because it is not known if this change would be sufficient to use three-level simulation with the 6502. There is at least one place in the schematics where the delay created by two inverters and a NOR gate are used to build a pulse stretcher (see figure A.2), so it is reasonable to assume that there are more delay-dependant subcircuits in the 6502 that would have to be eliminated.

[1]This can easily be seen in simulations, for example using the interactive JavaScript simulator at `http://www.visual6502.org`. The T1 signal is node 862, the other side of the pass transistor is node 537, Phi2 is cclk.

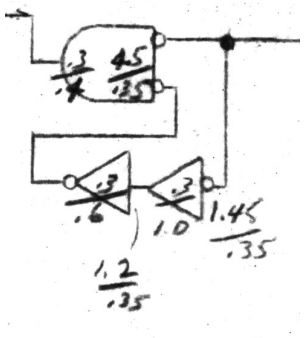


Figure A.2: A pulse-stretcher found in the 6502 schematics

B 6502 Instruction Set

This chapter will give a brief overview over the 6502 instruction set — many more in-depth references are available, for example the original MOS manuals [21]. The full opcode matrix (ignoring undocumented opcodes) is shown in table B.2. The upper line in each cell shows the mnemonic, the lower one the addressing mode (if any). The twelve addressing modes shown in the table are:

(none)
: Implicit: single-byte instructions without parameter

immed
: Immediate: a single parameter byte whose value is directly used, e.g. as operand in addition

abs
: Absolute: a two-byte parameter that gives a memory address, used as the target address in jumps, the memory location for loads/stores or the address of the operand of an ALU instruction

abs,X/Y
: Absolute,X/Y: same as Absolute, but the content of the X or Y register is added to the address before it is used

zeropg
: Zero-page: same as absolute, but with just a single byte parameter that limits the available addresses to 0x00-0xff

zero,X/Y
: Zero-page,X/Y: same as Absolute,X/Y, but with the same limits as the Zero-page addressing mode

(abs)
: Absolute Indirect: This mode is only used for a single instruction, jump indirect. Two

parameter bytes are interpreted as a memory address from where another two bytes are fetched which are the new program counter value.

relative

This mode is only used for conditional branches. A single byte parameter is interpreted as a signed 8-bit offset which is added to the current program counter to obtain the address where execution should continue.

(zero,X)

Indirect X: The current contents of the X register are added to the single parameter byte modulo 256, the result is used as the base address of a two-byte pointer that points to the address where the final value is located.

(zero),Y

Indirect Y: The single parameter byte is used as the base address of a two-byte pointer in the zero page. The Y register is added to the value read from this pointer (this time with correct overflow from the low to the high address byte) and the result is the address where the final value is located.

The following list summarises the function of each of the mnemonics in alphabetical order. The descriptions assume that the associated addressing mode has already been resolved and the calculated address (for branches, stores and read/modify/write instructions) or value at that address (for anything that only reads from memory) is available.

`ADC` ADd with Carry: Add the memory value, the current contents of the A register and the C bit from the flags register, put the result in the A register. Update N, Z, V and C flags. If the D flag bit is set, adjust the result for BCD arithmetic before updating the A register.

`AND` AND accumulator: Calculate the bit-wise *and* of the memory value and the current contents of the A register and put the result in the A register, update N and Z flags.

`ASL` Arithmetic Shift Left: Shift the value left by one bit, filling the new bit 0 with 0 and storing the new bit 8 in the C flag. Updates the N and Z bit. The implicit version of this instruction shifts the A register, the others modify the value in memory.

`BCC` Branch if Carry Clear: Continue execution at the target address if the C bit is 0; do nothing if it is 1.

`BCS` Branch if Carry Set: Continue execution at the target address if the C bit is 1; do nothing if it is 0.

`BEQ` Branch if EQual: Continue execution at the target address if the Z bit is 1; do nothing if it is 0.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA (zero,X)	crash	-	-	ORA zeropg	ASL zeropg	-	PHP	ORA immed	ASL	-	-	ORA abs	ASL abs	-
1x	BPL relative	ORA (zero),Y	crash	-	-	ORA zero,X	ASL zero,X	-	CLC	ORA abs,Y	-	-	-	ORA abs,X	ASL abs,X	-
2x	JSR abs	AND (zero,X)	crash	-	BIT zeropg	AND zeropg	ROL zeropg	-	PLP	AND immed	ROL	-	BIT abs	AND abs	ROL abs	-
3x	BMI relative	AND (zero),Y	crash	-	-	AND zero,X	ROL zero,X	-	SEC	AND abs,Y	-	-	-	AND abs,X	ROL abs,X	-
4x	RTI	EOR (zero,X)	crash	-	-	EOR zeropg	LSR zeropg	-	PHA	EOR immed	LSR	-	JMP abs	EOR abs	LSR abs	-
5x	BVC relative	EOR (zero),Y	crash	-	-	EOR zero,X	LSR zero,X	-	CLI	EOR abs,Y	-	-	-	EOR abs,X	LSR abs,X	-
6x	RTS	ADC (zero,X)	crash	-	-	ADC zeropg	ROR zeropg	-	PLA	ADC immed	ROR	-	JMP (abs)	ADC abs	ROR abs	-
7x	BVS relative	ADC (zero),Y	crash	-	-	ADC zero,X	ROR zero,X	-	SEI	ADC abs,Y	-	-	-	ADC abs,X	ROR abs,X	-
8x	-	STA (zero,X)	-	-	STY zeropg	STA zeropg	STX zeropg	-	DEY	-	TXA	-	STY abs	STA abs	STX abs	-
9x	BCC relative	STA (zero),Y	crash	-	STY zero,X	STA zero,X	STX zero,Y	-	TYA	STA abs,Y	TXS	-	-	STA abs,X	-	-
Ax	LDY immed	LDA (zero,X)	LDX immed	-	LDY zeropg	LDA zeropg	LDX zeropg	-	TAY	LDA immed	TAX	-	LDY abs	LDA abs	LDX abs	-
Bx	BCS relative	LDA (zero),Y	crash	-	LDY zero,X	LDA zero,X	LDX zero,Y	-	CLV	LDA abs,Y	TSX	-	LDY abs,X	LDA abs,X	LDX abs,Y	-
Cx	CPY immed	CMP (zero,X)	-	-	CPY zeropg	CMP zeropg	DEC zeropg	-	INY	CMP immed	DEX	-	CPY abs	CMP abs	DEC abs	-
Dx	BNE relative	CMP (zero),Y	crash	-	-	CMP zero,X	DEC zero,X	-	CLD	CMP abs,Y	-	-	-	CMP abs,X	DEC abs,X	-
Ex	CPX immed	SBC (zero,X)	-	-	CPX zeropg	SBC zeropg	INC zeropg	-	INX	SBC immed	NOP	-	CPX abs	SBC abs	INC abs	-
Fx	BEQ relative	SBC (zero),Y	crash	-	-	SBC zero,X	INC zero,X	-	SED	SBC abs,Y	-	-	-	SBC abs,X	INC abs,X	-

Table B.2: 6502 opcode matrix

BIT BIt Test: Calculate the bit-wise *and* of the memory value and the current contents of the A register. Update the Z flag based on the result, copy bit 7 of the memory value into the N flag and bit 6 into the V flag.

BMI Branch if MInus: Continue execution at the target address if the N bit is 1; do nothing if it is 0.

BNE Branch if Not Equal: Continue execution at the target address if the Z bit is 0; do nothing if it is 1.

BPL Branch if PLus: Continue execution at the target address if the N bit is 0; do nothing if it is 1.

BRK BReaK: execute a soft interrupt. The value of the program counter plus two as well as the current flag register with the B bit set is pushed to the stack. Execution continues at the address found in the IRQ vector (0xfffc/fffd) with the I flag bit set.

BVC Branch if oVerflow Clear: Continue execution at the target address if the V bit is 0; do nothing if it is 1.

BVS Branch if oVerflow Set: Continue execution at the target address if the V bit is 1; do nothing if it is 0.

CLC CLear Carry: Set the C flag to 0.

CLD CLear Decimal: Set the D flag to 0 (binary arithmetic).

CLI CLear Interrupt: Set the I flag to 0 (enables IRQ).

CLV CLear oVerflow: Set the V flag to 0.

CMP ComPare with accumulator: Subtract the memory value from the A register and set the N, Z and C flags based on the result, but discard the calculated value.

CPX ComPare with X: Subtract the memory value from the X register and set the N, Z and C flags based on the result, but discard the calculated value.

CPY ComPare with Y: Subtract the memory value from the Y register and set the N, Z and C flags based on the result, but discard the calculated value.

crash Instructions marked “crash” in the table will always deadlock an NMOS 6502 even in the absence of soft errors.

DEC DECrement: Subtract 1 from the value and store it at its original address, update the N and Z flags.

DEX DEcrement X: Subtract 1 from the X register and store the result in the X register, update the N and Z flags.

DEY DEcrement Y: Subtract 1 from the Y register and store the result in the Y register, update the N and Z flags.

EOR Exclusive OR accumulator: Calculate the bit-wise *exclusive-or* of the memory value and the current contents of the A register and put the result in the A register, update N and Z flags.

`INC` INCrement: Add 1 to the value and store it at its original address, update the N and Z flags.

`INX` INcrement X: Add 1 to the X register and store the result in the X register, update the N and Z flags.

`INY` INcrement Y: Add 1 to the Y register and store the result in the Y register, update the N and Z flags.

`JMP` JuMP: Resume execution at the address specified as instruction parameter (0x4c) or at the address stored at the address specified at the parameter (0x6c).

`JSR` Jump to SubRoutine: Push the program counter plus two on the stack, jump to the address given as parameter.

`LDA` LoaD Accumulator: Move the value read from memory into the A register, update N and Z flags.

`LDX` LoaD X: Move the value read from memory into the X register, update N and Z flags.

`LDY` LoaD Y: Move the value read from memory into the Y register, update N and Z flags.

`LSR` Logical Shift Right: Shift the value right by one bit, filling the new bit 7 with 0 and storing the previous bit 0 in the C flag. Updates the N and Z bit. The implicit version of this instruction shifts the A register, the others modify the value in memory.

`NOP` No OPeration: Do nothing.

`ORA` OR Accumulator: Calculate the bit-wise *or* of the memory value and the current contents of the A register and put the result in the A register, update N and Z flags.

`PHA` PusH A: Push the A register on the stack.

`PHP` PusH P: Push the flag register on the stack.

`PLA` PuLl A: Pop the A register from the stack.

`PLP` PuLl P: Pop the flag register from the stack.

`ROL` ROtate Left: Shift the value left by one bit, filling the new bit 0 with the previous contents of the C flag and storing the new bit 8 in the C flag. Updates the N and Z bit. The implicit version of this instruction shifts the A register, the others modify the value in memory.

`ROR` ROtate Right: Shift the value right by one bit, filling the new bit 7 with the previous contents of the C flag and storing the previous bit 0 in the C flag. Updates the N and Z bit. The implicit version of this instruction shifts the A register, the others modify the value in memory.

`RTI` ReTurn from Interrupt: Restore the flags and the program counter from the stack.

`RTS` ReTurn from Subroutine: Restore the program counter from the stack, add 1 before resuming execution.

`SBC` SuBtract with Carry: Take the current contents of the A register, subtract the memory value and the inverse of the C bit; put the result in the A register. Update N, Z, V and

C flags. If the D flag bit is set, adjust the result for BCD arithmetic before updating the A register.

SEC SEt Carry: Set the C flag to 1.

SED SEt Decimal: Set the D flag to 1 (BCD arithmetic).

SEI SEt Interrupt: Set the I flag to 1 (disables IRQ).

STA STore Accumulator: Store the contents of the A register into memory.

STX STore X: Store the contents of the X register into memory.

STY STore Y: Store the contents of the Y register into memory.

TAX Transfer A to X: Set the X register to the value in the A register, update N and Z flags.

TAY Transfer A to Y: Set the Y register to the value in the A register, update N and Z flags.

TSX Transfer S to X: Set the X register to the value in the stack pointer, update N and Z flags.

TXA Transfer X to A: Set the A register to the value in the X register, update N and Z flags.

TXS Transfer X to S: Set the stack pointer to the value in the X register (no flag updates).

TYA Transfer Y to A: Set the A register to the value in the Y register, update N and Z flags.

Most opcodes that handle data update the N and Z flag bits. The N bit is set to bit 7 of the data value, the Z bit is set if the data value is 0 and cleared otherwise.

Bibliography

[1] 65c02 product page at wdc. `http://www.westerndesigncenter.com/wdc/w65c02s-chip.cfm`.

[2] Posting to cbm-hackers mailing list. `http://www.softwolves.com/arkiv/cbm-hackers/17/17707.html`.

[3] Todd M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[4] Brian Bagnall. *Commodore: a company on the edge*. Variant Press, 2nd edition, 2010.

[5] Randal E. Bryant. An algorithm for MOS logic simulation. *Lambda, the Magazine of VLSI Design*, pages 46–53, 1980.

[6] Randal E. Bryant. A switch-level simulation model for integrated logic circuits. Technical report, Cambridge, MA, USA, 1981.

[7] Randal E. Bryant. A survey of switch-level algorithms. *IEEE Des. Test*, 4(4):26–40, July 1987.

[8] E.W. Czeck and D.P. Siewiorek. Effects of transient gate-level faults on program behavior. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 236 –243, jun 1990.

[9] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 5B.4.1 –5B.4.7, april 2011.

[10] Alireza Ejlali and Seyed Ghassem Miremadi. FPGA-based fault injection into switch-level models. *Microprocessors and Microsystems*, 28(5-6):317–327, 2004.

[11] Donald F. Hanson. A VHDL conversion tool for logic equations with embedded D latches. In *Proceedings of the 1995 workshop on Computer architecture education*, WCAE-1 '95, New York, NY, USA, 1995. ACM.

[12] International Roadmap Committee. International Technology Roadmap for Semiconductors, 2011 Edition, Design. Technical report, ITRC, 2011.

[13] Greg James, Barry Silverman, and Brian Silverman. Visual6502. `http://www.visual6502.org`.

[14] Greg James, Barry Silverman, and Brian Silverman. Visualizing a classic CPU in action: the 6502. In *SIGGRAPH Talks*. ACM, 2010.

[15] P.K. Kalkat, R. Sedaghat, J.M. Chikhe, and R. Javaheri. Soft error injection using advanced switch-level models for combinational logic in nanometer technologies. In *Microelectronics (ICM), 2009 International Conference on*, pages 332 –335, dec. 2009.

[16] Russell Kao. Piecewise linear models for switch-level simulation. Technical report, Stanford, CA, USA, 1992.

[17] Johan Karlsson, Ulf Gunneflo, and Jan Torin. The effects of heavy-ion induced single event upsets in the MC6809E microprocessor. In *Fehlertolerierende Rechensysteme / Fault-Tolerant Computing Systems, Automatisierungssysteme, Methoden, Anwendungen / Automation Systems, Methods, Applications; 4. Internationale GI/ITG/GMA-Fachtagung*, pages 296–307, London, UK, UK, 1989. Springer-Verlag.

[18] Peter Marwedel. *Synthese und Simulation von VLSI-Systemen.* Carl-Hanser, 1993. ISBN: 978-3446167467.

[19] T.C. May and M.H. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2 – 9, jan 1979.

[20] Peter Monta. FPGA-netlist-tools. `https://github.com/pmonta/FPGA-netlist-tools`.

[21] MOS Technology, Inc. *MCS6500 Microcomputer Family Programming Manual*, 1976.

[22] Shubu Mukherjee. *Architecture Design for Soft Errors.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[23] L. W. Nagel and D.O. Pederson. SPICE (simulation program with integrated circuit emphasis). Technical Report Memorandum No. ERL-M382, University of California, Berkeley, Apr. 1973.

[24] Yohei Nakata, Yasuhiro Ito, Yasuo Sugure, Shigeru Oho, Yusuke Takeuchi, Shunsuke Okumura, Hiroshi Kawaguchi, and Masahiko Yoshimoto. Model-based fault injection for failure effect analysis – evaluation of dependable SRAM for vehicle control units. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 91–96, Washington, DC, USA, 2011. IEEE Computer Society.

[25] E. Normand and T.J. Baker. Altitude and latitude variations in avionics SEU and atmospheric neutron flux. *Nuclear Science, IEEE Transactions on*, 40(6):1484 –1490, dec 1993.

[26] A. Taber and E. Normand. Single event upset in avionics. *Nuclear Science, IEEE Transactions on*, 40(2):120 –126, apr 1993.

[27] Catherine Trammell. Exploring the microarchitectural behavior of an industrial processor in the presence of transient faults. Master's thesis, Cornell University, 2009.

List of Figures

2.1 Examples of problematic circuits for unit-delay switch level simulation 8
2.2 Screenshot of the interactive 6502 simulator from the Visual6502 project . . . 10
2.3 Connections of the single-connected nodes in the net list 10
2.4 Simplified internal architecture of the 6502, from the MOS manual 12
2.5 Excerpt from the 6502 hardware manual illustrating the bus timing 14

3.1 Illustration of the interaction between the memvalues list, CPU memory accesses and the memory array 20
3.2 Example output of the simulator running an `ADC #$01` instruction 21

4.1 Per-opcode fraction of no-effect runs for the *random* data set 33
4.2 Per-opcode fraction of no-effect plus hidden-effect runs for the *random* data set 34
4.3 Nodes with differences after a hidden-effect run 35
4.4 Probability of deadlock per opcode for the *bugfix* data set 38
4.5 Nodes causing deadlocks in the *bugfix* data set 40
4.6 Nodes causing simulation failure in the *bugfix* data set 41
4.7 Bit 0 of the X register (left) and stack pointer (right), extracted from the 6502 schematics 43
4.8 Per-opcode probability for a simple error in the *random* data set 46
4.9 Per-opcode probability for a masked simple error in the *random* data set . . . 46
4.10 Interrupt probabilities per opcode in the *bugfix* data set 49
4.11 Per-opcode probability of an IRQ in the *bugfix* data set 51
4.12 Per-opcode probability of a soft interrupt in the *bugfix* data set 51
4.13 Per-opcode IRQ probability sorted by minimum execution time 52
4.14 Per-opcode probability of control flow change in the *random* data set 54
4.15 Probability of control flow change depending on the initial program counter value 55
4.16 Per-bit probability of data access change with one bit set in the start condition 57
4.17 Per-bit probability of data access change with one bit cleared in the start condition 57
4.18 Per-opcode probability of instruction overwrites in the *random* data set . . . 58
4.19 Delayed effect during error injection in half-cycle 3 at node 504 59
4.20 Histogram of effect delay in half-cycles for the *random* data set 60
4.21 Example of an uncategorised test run; reference at the top 62

A.1 T-state counter bits T2 to T5, from the 6502 schematics 68
A.2 A pulse-stretcher found in the 6502 schematics 71

List of Tables

3.1 compressed file size and simulation time for multiple compression options . . 23

4.1 Data sets used in the evaluation 30
4.2 Basic results for the data sets 32
4.3 Per-opcode deadlock probability statistics 36
4.4 Probability of deadlock for `JSR` with modified start conditions 37
4.5 Probability of deadlock for `ADC abs,X` and `INC abs` with modified start conditions 37
4.6 Probabilities for simple errors in the data sets 42
4.7 Probability of a masked address bus change in six selected instructions 44
4.8 Probability of change in a flag bit when a simple (masked) error is detected . 45
4.9 Interrupt probabilities for the data sets 47
4.10 Interrupt failure probability depending on memvalue 48
4.11 Control flow and data access change probabilities for all data sets 53
4.12 Probability of instruction overwrites for all data sets 56
4.13 Probabilities under different fault models 61

A.1 extended truth tables for AND, NOR and NOT gates 69
B.2 6502 opcode matrix 73