DETERMINATION OF CORRECT OPERATION AND BEHAVIOR OF A STRUCTURED AMORPHOUS SURFACE

BY
MALCOLM J. LEAR

A thesis submitted for the degree of MSc(Res)

School of Electronics and Computer Science
University of Essex

December 2015

Supervisors: Dr. Martin Colley
Dr. Graham Clarke
Prof. Vic Callaghan

Abstract

A recurring theme in intelligent environments is the intelligent surface composed of nanoscale processing units (smart dust). Such a surface (iSurface) can be considered an amorphous computer composed of a large array of identical processing units (iCells) each with its own sensor/effectors. An important requirement of such a surface is the need for a fast, reliable method to determine iCell operation, performance and code integrity. Any practical solution must fulfil certain criteria. First the impact on intercellular data communication bandwidth must be kept to a minimum, this is particularly important in high density, high speed iSurface applications such as high resolution video display. Previous work on processor profiling offered a possible solution in the form of metrics derived from profiling. This thesis describes a method developed to create long (>=32 bit) stable, robust metrics using a profiling technique that represents the current operational state of an iCell and thus enabling the quick exchange of diagnostics between iCells along with data traffic. Key requirements in the development of this system were fast acquisition of diagnostic variables, minimal affect on normal operation and the possibility of a hardware implementation which could be completely non intrusive in operation.

The hardware developed fulfilled all these criteria in particular a novel method to create a stable metric that could determine compromised or incorrectly loaded code was developed. The metric of code integrity had both attributes of stability and responsiveness to change, something that has proven difficult to attain before. The uniqueness of the metrics produced by the hardware was also investigated and was determined to be very good and metric bit length was efficiently used. Impact on processor performance was also deemed acceptable at 2.31% and the developed architecture could theoretically be implemented in ‘system on chip’ (SOC) with zero processor overheads.

Publications Arising from this Work

M. J. Lear, "Stable Metrics in Amorphous Computing: An Application to Validate Operation and Monitor Behavior," in Intelligent Environments (IE), 9th International Conference on, 5-8 Aug 2012, pp.204 – 211.

Acknowledgements

I would first like to acknowledge the continuous support of my co supervisor Dr. Graham Clarke in this endeavour, his weekly chats and enthusiastic support is very much appreciated and certainly inspired many ideas on applications of the research technology amongst other things. Secondly this could not have been done without the enthusiastic support of my supervisor Dr. Martin Colley. His practical skills both in hardware and software ensured a guiding light to solutions to any major problems. I would also like acknowledge Prof. Vic Callaghan for listening many times and quite often playing 'devils advocate' to ideas that proved core to the technology.

Lastly many thanks to my family and friends for understanding and allowing me the space and time needed to complete this research.

Lest she thinks she's forgotten, a special thanks to Lena for all her support.

Table of Contents

List of Tables

List of Figures

List of Acronyms and Abbreviations

ASCII	American Standard Code for Information Interchange
CIM	Code Integrity Metric
CPLD	Complex Programmable Logic Device
FPGA	Field Programmable Gate Array
GCC	GNU C Compiler
I^2C	Inter-IC bus
IDE	Integrated Development Environment
JTAG	Joint Test Action Group
LUT	Look Up Table
MCU	Microcontroller Unit
OOCD	Open On Chip Debugger
PCB	Printed Circuit Board
PSDT	Program Structure Development Toolkit
PSM	Program Structure Map
PST	Program Structure Table
SAS	Software Analysis System
SOC	System On Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TAP	Test Access Port
UART	Universal Asynchronous Receiver/Transmitter

Chapter 1: Introduction

People's living space is becoming rich in electronic devices most of which have computational capabilities undreamt of a few years ago; indeed even humble devices such as the bedroom clock usually utilize embedded systems technology. More often such technology will communicate and interact autonomously as protocols are implemented and refined. Of particular interest in the near future are printable electronics and nanotechnology that open up many new possibilities in pervasive computing [1]. Such technology will enable entire surfaces, for example walls to become "intelligent" and "aware" of the environment. Assuming Moore's law is maintained so that in future a particular computational power requires less physical space and lower power consumption then amorphous computing could be utilized to create these intelligent surfaces (iSurface). Typical applications would be full wall audio/visual systems offering such things as immersive education, ambiance and artistic interaction. The iSurface with its potentially enormous processing power, resolution, functionality and inherent pervasive properties could well shape the future of intelligent environments.

1.1 A Structured, Reprogrammable Approach: The iSurface

A practical implementation of the iSurface would most likely be the construction of identical cells (iCells), each with microcontroller (MCU), memory, sensors, effectors and communication hardware. Practical issues with power distribution and communication along with economic issues such as minimizing costs would favour a structured, predefined pattern of iCells as opposed to the more common ad hoc approach. Adopting a structured approach to creating the iSurface also lends itself to practical methods of construction such as printing and stretchy circuits (flexible silicon). Such a structured approach may seem to be at odds with the classic undefined amorphic structure, however faults in use or manufacture would require an adaptive interconnection and data flow method entirely compatible with that needed for an ad hoc arrangement of cells. This structured construction offers the opportunity to operate in various or mixed modes. For example each cell could be programmed in an identical way with the traditional amorphous properties of each cell having no priori knowledge of location or orientation. Other modes could involve the propagation of program code as well as data between nodes either

directed explicitly from an external controller or by some decision making at the cellular level. Certainly the emergent behavior of traditional amorphic components is preserved and maybe even enhanced.

Unlike most “conventional” electronic amorphous computing networks that are preloaded with all required programs, it is intended that the iSurface should be reprogrammable in situ by propagating programs across the surface utilizing the data network. A typical problem with such an approach may well be an error or failure in the reprogramming of particular iCells. It would also be of great benefit that any proposed solution to the issue of array reprogramming be extendable to determine other issues inherent with high density amorphous computing arrays such as sensor, effector, and general hardware performance.

1.1.1 Current Approaches to Node Performance Determination

Whilst much research and development on self assembly and repair topologies has been undertaken, it seems few have addressed the issues of node performance determination, in particular partial degradation of a computing node. Determination of complete node failure is clearly a requirement with the aforementioned repair topologies and consequently there is a real need for a method to monitor node performance and propagate that information across the network array. I need to develop a method that allows adjacent nodes in a network to be aware of minor programming or overall performance issues so that appropriate actions could then be performed if necessary, thus providing a very important component of a robust amorphous computing array.

Although there has been limited research conducted in performance monitoring of nodes within an amorphous computing array, this is not the case with processor performance per se. One of the most common methods involves sampling the processor’s status on a regular basis and deriving a statistical profile of the current program operation. Whilst this may well form the basis of a method to determine operation behavior, its inherent instability due to constant variation in program flow makes it a poor candidate for determining program integrity at run time.

1.1.2 The Proposed Solution

The nature of the statistical profile derived from processor sampling suggests the possibility of reduction to a unique metric that has the advantages of low bandwidth and fast intercellular comparisons to determine odd behavior of a ‘rogue’ iCell.
This assumes:

- The iCells are in close proximity and therefore sensor data is similar.
- The iCells are executing the same program code.

The first requirement can easily be met and becomes less of a problem as density increases and sensor input becomes more homogeneous. The second requirement is however fundamental and a prerequisite to allow intercellular behavioral comparison. If the program code itself could be uniquely described as a metric in much the same way as the dynamic behavior then a very complete description of an iCell nodes health could quickly propagate across the iSurface using very low bandwidth.
A problem with the creation of a metric of code integrity would be to have stability and also be responsive to program change due to errors in programming, memory faults etc. These issues and the fundamental requirement of a method to determine code integrity over and above metrics of iCell behavior present a real challenge and provide the focus of the research.

1.2 Aim of this Work

We believe it should be possible to create a stable profiling metric based on monitoring and determining changes in program structure based on branch opcodes, that would allow the operation of arrays of processing cells to be observed and protected from faulty or unwanted behavior.
Program structure based on branch addresses should remain static and unchanged in memory after programming and therefore be an ideal processor agnostic candidate to provide metrics aimed at determining code integrity. Monitoring activity at these same locations in memory would likewise be ideal in creating behavioral and diagnostic metrics and therefore we consider it important that any system developed to create a metric of code integrity should also be easily expanded to cover behavioral metrics with little extra hardware overheads.

Initial development will be in software at which point evaluation of uniqueness, response to code change, complexity, cost, and System On Chip (SOC) implementation issues will follow.

Work will then move to the implementation in hardware with particular a view on future migration to SOC.

The final stage will be an appraisal of software/hardware implementations, comparisons, conclusions and considerations of future work.

A stable metric of code integrity also puts real meaning into the behavioral and diagnostic metrics, because as with biometrics, it's important to know the animal you are investigating first.

1.3 Thesis Layout

The thesis has a linear layout that closely matches the sequence of research and development. Whilst this order may seem a little odd at times, it does reflect the logical order of work undertaken and reasoning for this sequence will be justified in the various chapters.

Chapter Two reviews related work and presents a brief overview of various approaches to amorphous computing arrays, methods employed to determine health of cells and implementations of fault tolerance systems. This is followed by an introduction of the iSurface and it's more particular fault tolerance requirements and discusses some ideal world requirements for such systems.

Chapter Three introduces the proposed system and discusses the early development and technical specification of the hardware platform to be used.

Chapter Four discusses the use of program structure in fault tolerance, how it can be defined, determined and stored efficiently.

Chapter Five details the development, implementation in software and performance of the Code Integrity Metric (CIM). There is also a brief discussion about how behavioral metrics could be implemented to complement the Code Integrity Metric.

Chapter Six discusses the development of the iCell and implementation in hardware of the Code Integrity Metric.

Chapter Seven looks at development and implementation of the JTAG control system in a Field Programmable Gate Array (FPGA).

Chapter Eight looks at the implementation of the SRAM based Structure Table and I^2C communication sub-systems in the FPGA.

Chapter Nine appraises the hardware implementation and compares performance and operation with the earlier software based method.

Chapter Ten presents conclusions and discusses future work.

Note that parts of Chapters Three to Five were used in the publication "Stable Metrics in Amorphous Computing: An Application to Validate Operation and Monitor Behavior" [2] that arose from this research.

Chapter 2: Amorphous Computing and Fault Tolerance

Traditional amorphous computing as described by D. Coore [3], H. Abelson [4] and R. Nagpal [5] covers a very wide range of sciences and can be best described as the functioning of a multicellular orgasm whether it be biological or electronic in nature. Amorphous computing methods have to a large degree been derived from the swarm behavior of biological organisms that display a coherent emergent intelligence. This would include a swarm of bees, a colony of Ants, the Portuguese Man-of-War or even groups of higher level animals including humans. All these examples to some degree display complex emergent behavior from the cooperation of many individual parts, some of which may not be fully functional. Electronic computing offers a wonderful tool for researchers to allow the simulation of various amorphous computing models and is by far the most common method employed when studying various amorphous cellular systems and network topologies [4]. This has the benefit of almost unhindered flexibility in modelling the entire amorphous system, from the individual cell up to the environment in which it inhabits. The other less common method employs electronic real world hardware using multi-processors interconnected by ad-hoc or structured networks. Natural real world examples seem to have ad-hoc interconnections but the emergence of coherent behavior suggests their dynamics are dependent on rules. For example according to Y. Hu [6] the behavior of individual Ants is based on rules that determine response to pheromone trails left by other Ants giving rise to swarm intelligence [7].

If these natural dynamic network modes could be brought to life in silicon, interesting research in emergent behavior would be forthcoming. This requires a method of propagating individual's actions and responses (determined mostly by sensor/effectors) across the amorphous network. It would also be beneficial if this information exchange could in some way include the health and capabilities of the amorphous cell. For example in nature a member or members of an organism may well loose some functionality, however that does not exclude them from contributing to the colonies emergent behavior. Indeed if that change was a genetic mutation and it benefited the organism in some way it may well be beneficial to ignore the abnormality promote a possible change to the gene pool. A low bandwidth method of inter-cellular information transfer of behavior that intrinsically

includes health and capability offers the possibility of enhanced fault tolerance beyond the current decisive yes/no methods.

2.1 Amorphous Computing Fault Tolerance Methods

Current fault tolerance methods employed tend to vary dependent on processor granularity. For example the interesting work done by J. R. Heath [8] and M. Hartman [9] involved fault tolerance in amorphous networks that have very simple cells based on programmable logic. At the other end of the spectrum there are coarse grained networks based on isolated, traditional von Neumann architectural [10] computing units. Since this work will be based on a coarse grained network utilising standard microcontrollers, further considerations will be confined to that area of fault tolerance.

Considerable work and development of self assembly and repair topologies with respect to amorphous computing has been undertaken, for example by R. Nagpal [11],[12] and L. Clement [13]. However this work is focused on methods to create a functional amorphous computer and how a working network can cope with node failure, not on the actual mode of failure. The most common methods employed to determine correct processor operation within an amorphous network tend to resolve the issue with a decisive yes/no answer. This is typically accomplished by sending test data back and forth between nodes and checking for corruption or more seriously, a total failure of communication. D. Chu considered node failure using such methods [14] and considered the loss of amorphous computer nodes as binary stochastic noise. This work of D. Chu demonstrates that whilst communication between cells may be unimpaired, incorrect operation due to partially corrupted program code or other undetected problem could lead to complete failure of the amorphous computer at relatively low levels of noise. The idea of using processor profiling or some development of it to determine the health of amorphous computer nodes and convey that information in a simplistic form would certainly improve the situation.

Most current profiling techniques are based on two modes of operation, software or hardware. The first and probably most common method employs statistical sampling, discussed by M. A. Jennifer [15] and J. Whaley [16], where the target processor is halted and the current states of various registers are retrieved for analysis. This halting of the processor can be done either under hardware control or

by way of software typically using interrupts initiated by in-built timers. Another approach commonly used called path profiling [17] employs tracing that would typically increment counters at defined points in the program which could then be read under interrupt control, perhaps initiated by a serial port communication. This approach either has the path monitoring code added to the program source code before compilation or the program is modified dynamically at run time by way of branch instructions to profiling code. The merits or otherwise of the methods described so far are discussed by M. Arnold [18] and either rely on specially written in-line code or interrupt subroutines. The use of in-line code is not only intrusive but only allows profiling at fixed points in the program, requiring recompilation or some other method to alter program code to facilitate reconfiguration of the profiling. The intrusive nature of software based profiling was addressed by R. G. Scottow and A. B. T. Hopkins [19] they proposed an interesting refinement that minimised the intrusive in-line code whilst extracting useful profiling information with a processor performance hit of just over 0.005%. Another, but less common technique of profiling involves monitoring the target processors activity using dedicated hardware which eliminates the need for in-line profiling code. A very interesting approach which incorporates path profiling and a novel way of pre-processing the profiling information was developed by H. Zhang, J Ji, X, Zhou, H. Ma [20]. Unfortunately this requires direct access to both the address and data bus, something that is not available on almost all microcontrollers. However the idea it uses for profiling paths (entry to branch) and reducing the profiling information has similarities to the proposed solution outlined in the next chapter. Another method of profiling that may have particular relevance to this work, in that it produced unique digital signatures from data streams was developed by Hewlett-Packard [21].

2.2 Performance Based iCell Fault Tolerance

The computing power required to simulate an amorphous organism can become excessive due to the need to run identical cellular programs and communication and interaction. The idea of studying amorphous computing by employing hardware based computing cells each with dedicated processors solves the problem of processing speed, but issues with reprogramming and adaptive communication methods has problems of it own. The iCell's design goals attempt to solve these problems by employing reprogrammability of processor software by propagation

across a structured amorphous array. This reprogrammability in an inherently failure prone amorphous network raises serious issues of code corruption and therefore precludes the use of software based profiling methods since any in-line profiling code may well in itself be corrupt. As mentioned in the introduction a practical and achievable approach to construct an intelligent surface and one which could be used as a means to evaluate the proposed method of fault tolerance would be structured and regular in nature unlike the traditional ad hoc amorphous network investigated by A. King [22],[23], W. Butera [24] and others, illustrated in Fig. 1. The easiest and simplest two dimensional iSurface would have iCells linked in rows and columns as seen in Fig. 2. Diagonal connections were considered, but for practical reasons a square iCell structure benefits from a symmetrical iCell to iCell interconnect and is sufficient for current work and possible future research into high speed network communication. Such a network also lends itself to practical methods of construction such as printing J. Chang [25] and stretchy circuits (flexible silicon) D. H. Kim [26].

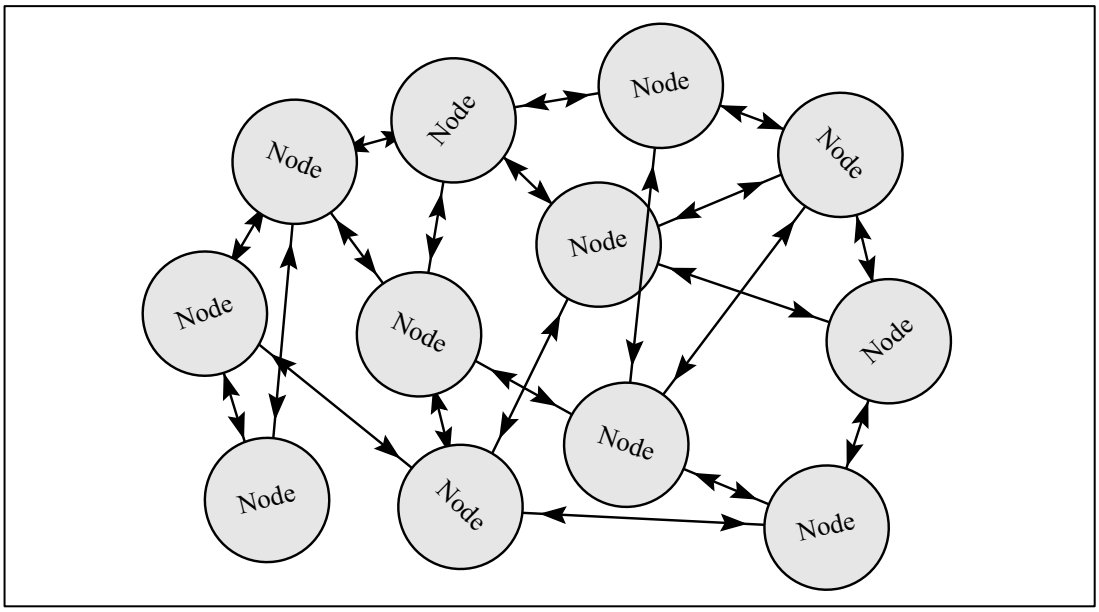


Figure 1: Ad hoc architecture of classic Amorphous Computing.

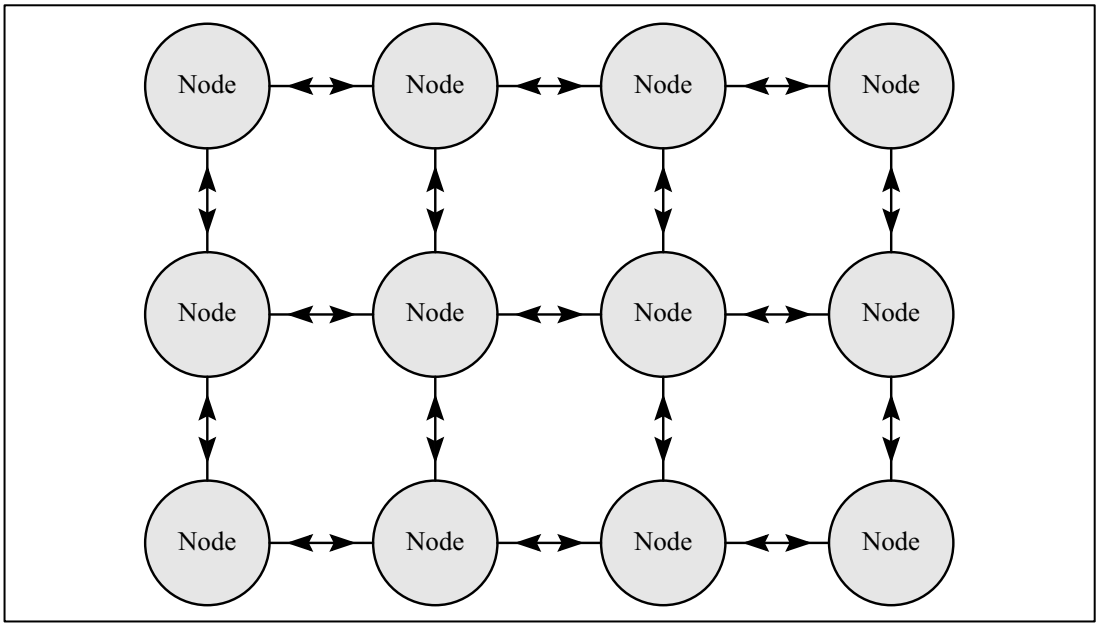


Figure 2: Structured architecture of the iSurface.

Chapter 3: Behavioral Metrics: The Solution

In summary, the iCell profiling requirements are:

- hardware based
- no program code modification
- possible SOC implementation
- simple, low bandwidth method to convey iCell behavior and 'health'

The proposed solution can meet all these targets by use of a variant of hardware sampling and providing iCell behavioral information by way of metrics.

The requirement to identify faulty iCells and abnormal behavior within such an amorphous surface suggested the need to develop a diagnostic system that could run in the background on an iCell, taking little (software based) or no (hardware based) processing time. The idea of producing metrics as an iSurface diagnostic tool addressed several problems, the first of which was the need to propagate the diagnostic data quickly from iCell to iCell. Secondly it would be useful for any diagnostic information to include both behavior and code integrity. The creation of metrics derived from iCell (MCU) profiling seemed to provide a promising solution. The use of metrics to allow adjacent iCells to detect problems, could for example initiate local iCell to iCell repair by means of program mirroring. Metrics based on profiling have been used to create encryption keys, for example ICMetrics, described by Y. Kovalchuk [27] and X. Zhai [28]. However creating stable metrics derived from standard profiling methods in an embedded system is a challenge due to external events causing the execution of rarely used code. Such method are also statistical and accumulative in nature, becoming stable only after long periods of time and therefore would be very unresponsive to serious changes in behavior or executed program code. These issues could be resolved if an alternative profiling method was employed that was inherently unaffected by program flow. This work shall refer to any developed profiling method as iProfiling and derived metrics as iMetrics. Work by V. Callaghan [29] on program analysis based on program branch structure provided more evidence that stable diagnostic metrics could be created.

3.1 Experimental Platform

The experimental setup required to determine the viability and stability of the diagnostic metrics can be broken down into the following areas: (i) the selection of a

suitable hardware target platform; (ii) the selection of a suitable software development environment to run on the host PC; (iii) the selection of suitable communication channels that will allow control and acquire data for subsequent analysis on the host PC. The following subsections describe these areas in more detail.

3.1.1 Hardware Target Platform

Important hardware requirements deemed necessary to analyze program structure and extract metrics included the following features: (i) a flexible interrupt controller with timer, thus allowing periodic sampling of processor states (ii) a counter with sufficient resolution to time processor clock cycles. (iii) Joint Test Action Group (JTAG) interface offers full control including single step operation. (iv) serial port/s offering a channel of communication for control and data acquisition. Previous work and design experience with ARM processor's suggested that the Atmel AT91SAM7S256 [30], indeed an ARM based board developed for teaching purposes using this processor would be the ideal choice to fulfil these requirements. Features of the chosen development target board are:

- AT91SAM7S256 microcontroller
- 64 kBytes of SRAM
- 256 kBytes of FLASH
- 48 MHz clock (typically 1 instruction per clock cycle)
- 2 serial ports offering up to 115200 baud
- JTAG interface.

3.1.2 Software Development Environment

Important features required from the software development environment were: (i) C and assembler language programming. (ii) debug mode utilizing the processors hardware debug module. (iii) hard and soft break points. Previous experience suggested a particular combination of open source software development tools would be ideal. Components used to build the development system include:

- Eclipse [31] is an open source multi language software development environment including an IDE.
- Open On-Chip Debugger (OOCD) [32] is the software interface to the JTAG hardware debugger module.
- GCC C compiler [33].

3.1.3 Control and Data Acquisition

The choice of hardware platform and software development environment offers two possible modes of communication. First there are the serial ports and secondly the JTAG interface. Serial port communication could use a bespoke protocol preferably but not limited to ASCII characters for control and acquisition of data from the target processor connected to a host PC or simply be used with a terminal program such as PuTTY. JTAG offers the possibility to control the processor using OOCD commands via a telnet connection. OOCD has a limited but very useful set of commands offering full control over the target processor and allowing access to memory, program counter, status and other registers.

Chapter 4: Program Structure

4.1 Structure Definition

The most problematic metric to extract is one indicating the integrity of the program code. This metric should ideally be unique to the loaded program whilst remaining stable and unaffected by program flow. A metric based on program structure would fulfil this requirement. Program structure as defined in “SAS-an experimental tool for dynamic program structure acquisition” [29] will be used. In that paper program structure was visualized using ‘structure maps’. An example structure map shown in Fig. 3 represents a calibrated portion of the processors memory as a circle with an arrow indicating the normal sequential incrementation of the program counter. Deviations from the circle caused by branches are depicted as lines with green (dashed) indicating a jump forward in memory and red (solid) a jump backwards. The other important visualization is execution frequency (the frequency of address access) being expressed as variation of intensity of the drawn lines. This visualization of program structure and flow will subsequently be used in this thesis and any future work as required.

4.2 Useful Characteristics for Metrics

When looking at the structure map in Fig. 3 both fixed and dynamic features can be seen. Fixed features that may be used to extract metrics such as code integrity are branch point source and destination addresses. Dynamic features of the program structure that may be used for behavioral diagnostics are the frequencies of processor activity at those same source and destination addresses. Whilst looking at the structure map it is clear that both source and destination addresses are important fixed features that could be utilized to extract metrics unaffected by program flow, however frequencies at destination addresses can be derived from the branch source and therefore metrics based on frequency analysis need only be concerned with branch point memory locations (branch opcodes).

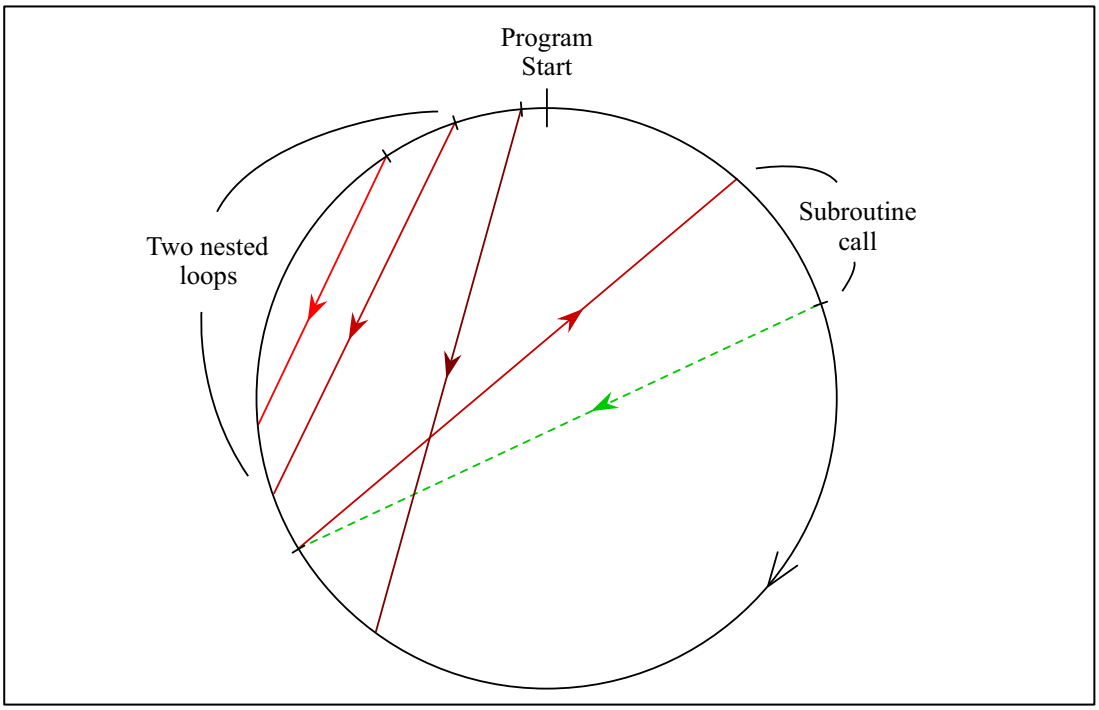


Figure 3: Program Structure Map.

4.3 Considerations

For experimental purposes, a software based system to determine program structure and acquire test data for further analysis was devised. Various options to accomplish this were considered and evaluated. Simple profiling methods such as periodic sampling could be achieved by using the OOCD commands via the JTAG interface. However the JTAG debug module in most processors, including the AT91SAM7S256 are designed primarily to debug software during the development phase and also programming/verification of the device which would make development and testing problematic. The only other dedicated communication channels available were the two serial (UART) ports. Fortunately these ports are reconfigurable, quite fast (115200 baud) and the option of setting interrupts on receive opened up the possibility of an interrupt driven diagnostic and development toolset employing various commands similar to OOCD.

4.4 Determining Program Structure

Any method of determining program structure by way of branch address location would need to be dynamic due to legitimate reprogramming of some or all of program memory. This requirement and that the Code Integrity Metric should be stable, led to an approach inspired by Popper's scientific method of falsification [34]. The practical application of this method in determining program structure

involves the creation of a metric derived from branch locations in the entire program memory space and then the application of runtime checks to disprove the metric. Once a change in the structure is determined a new metric is created, thus the metric is stable, reflects the current program structure and updates quickly. However, a problem with this approach is that the ideal properties of a program structure metric would preclude the possibility of verifying program structure on-the-fly. For this reason a table dedicated to holding program structure was considered. This Program Structure Table (PST) could then be used both for falsification runtime checks and provide the source for the variable length Code Integrity Metric. It was not deemed important that the scan of program memory to create this table should be particularly non-intrusive (a SOC solution would still require memory access) since this would occur only when a change of program structure had been detected (a significant event), see Fig. 4.

Runtime methods needed to verify executed branch locations (local program structure) against structure information held in the table had to be of a low intrusive nature (ideally non-intrusive if SOC). These requirements led to the idea of verifying the locations of frequently accessed branch addresses in the first instance. Other techniques running at a lower priority could be used to determine the branch locations of rarely executed or dormant code. A Programming Structure Development Toolkit (PSDT) was developed utilizing one of the UART ports with the interrupt controller configured to issue a non-maskable interrupt on reception of characters (commands). The associated interrupt routine performs various operations returning information via the UART if required.

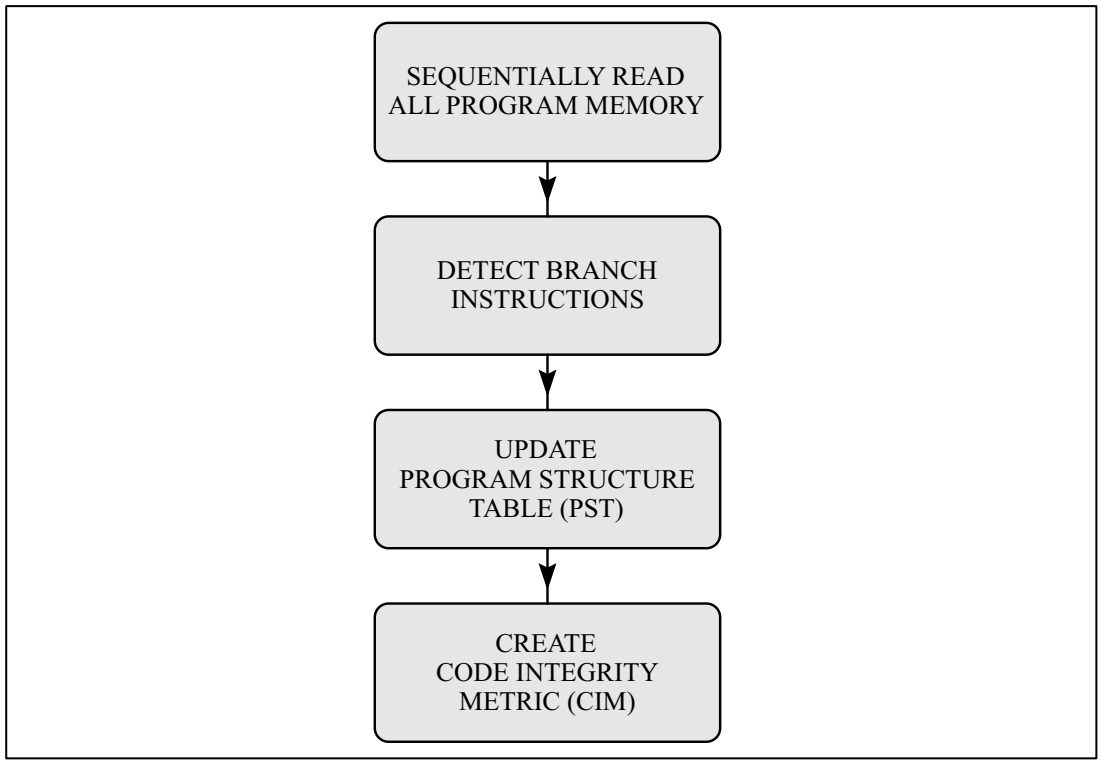


Figure 4: Method to update the Program Structure Table and create the Code Integrity Metric.

4.5 Locating Branch Points in Software

Meeting the requirements with a software solution is difficult, indeed the very nature of a software solution, i.e., it has to run on the processor, ensures it will in some way be intrusive. However, a variation of the industry standard profiling technique called ‘statistical sampling’ offered a way to approach these ideal requirements without too much compromise. A typical implementation of statistical sampling would periodically halt the processor and typically retrieve register contents, program counter and stack pointer for further analysis. The technique used for branch location is an extension of this and has the following sequence of operations:

- Halt the processor.
- Note the program counter.
- Search for the next branch in memory following the program counter address.
- Note the address of located branch.
- Use the program counter and branch address to cross check the Program Structure Table.

In this way an almost statistically random memory location being executed by the processor is investigated and a check of whether a branch point is located within a predetermined number of memory locations in advance of the memory location is performed, see Fig. 5. It can be argued that sampling at a fixed frequency in this way

is deterministic and if processes on the MCU are also periodic at or multiples of the sampling frequency then aliasing of memory locations may skew the results. This is indeed true, however in practice the huge difference between the sampling rate (20 Hz) and MCU clock (48 MHz) along with an asynchronous sampling clock, reduces the problem to such an extent that fixed rate sampling is still the most common method used in processor profiling. Note also that the use of sampling as a way of checking code integrity is not relying on extreme randomness. However future work that may employ the same sampling system to implement behavioral metrics may benefit from refinements such as a pseudo random sampling, perhaps similar to the method investigated by S. McCanne and C. Torek [35].

The first experimental method used to implement the branch discovery system was done by way of an interrupt routine running on the target platform. This technique utilized the MCU's periodic timer to issue system interrupts at a period of 20 milliseconds. The interrupt halts current program execution and retrieves the program counter by way of a modification to the low level interrupt library routine. The program counter is then used as the start point in the search for branch instructions in program memory. A sufficient branch search is then undertaken to cross check a single entry in the Program Structure Table, after which a simple routine to falsify the program structure is executed.

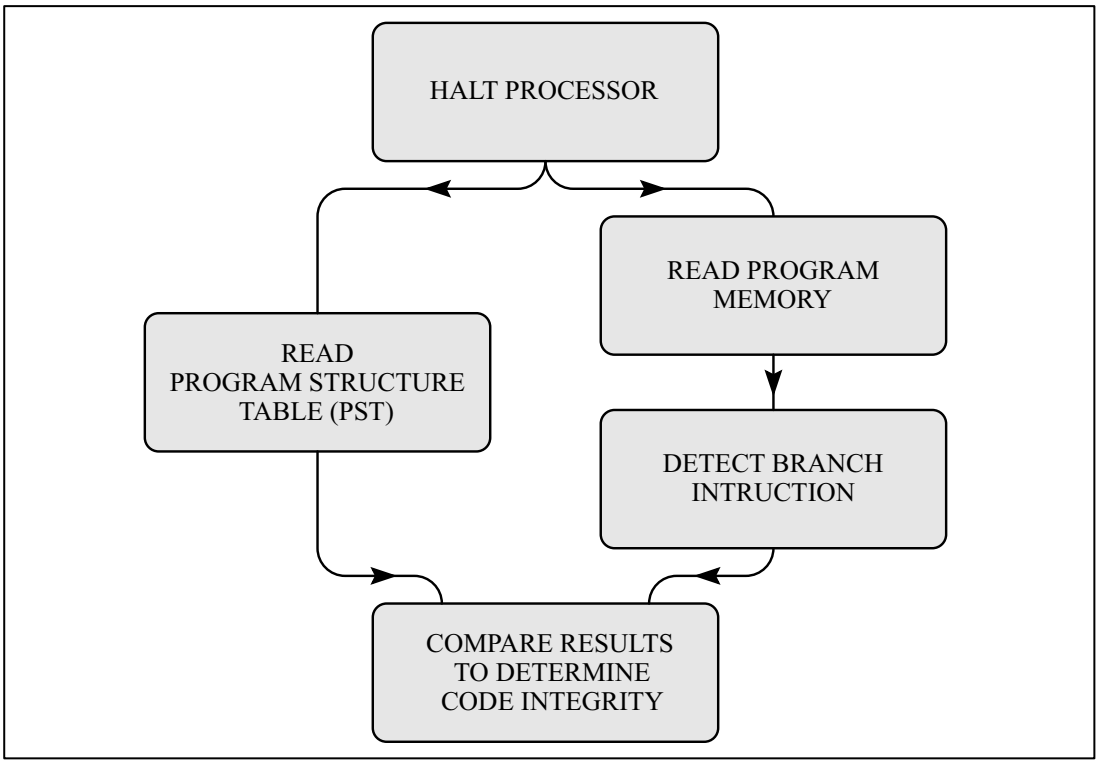


Figure 5: Method to check program code integrity.

This interrupt based solution proved viable and provided evidence that program structure could be determined by way of branch searches within a reasonable time period (<20% of the 20 ms sampling). However since a primary goal of the research was to develop a hardware solution which would utilize the JTAG interface it was decided that although inconvenient, that an OOCD/JTAG solution at this stage of the research would be more productive. The reasoning behind this change was that technical information relating to the JTAG to MCU low level protocol seemed very sparse and almost proprietary to processor manufacturers and further development would likely rely to some degree on protocol reverse engineering.

A diagram of the experimental platform and instrumentation required to proceed with the research can be seen in Fig. 6. It will be noticed that the Open On-Chip Debugger is controlled through a Telnet port. This allows test programs running on the PC to control the target MCU by way of the JTAG interface, thus enabling analysis of the JTAG interface protocol using a logic analyser.

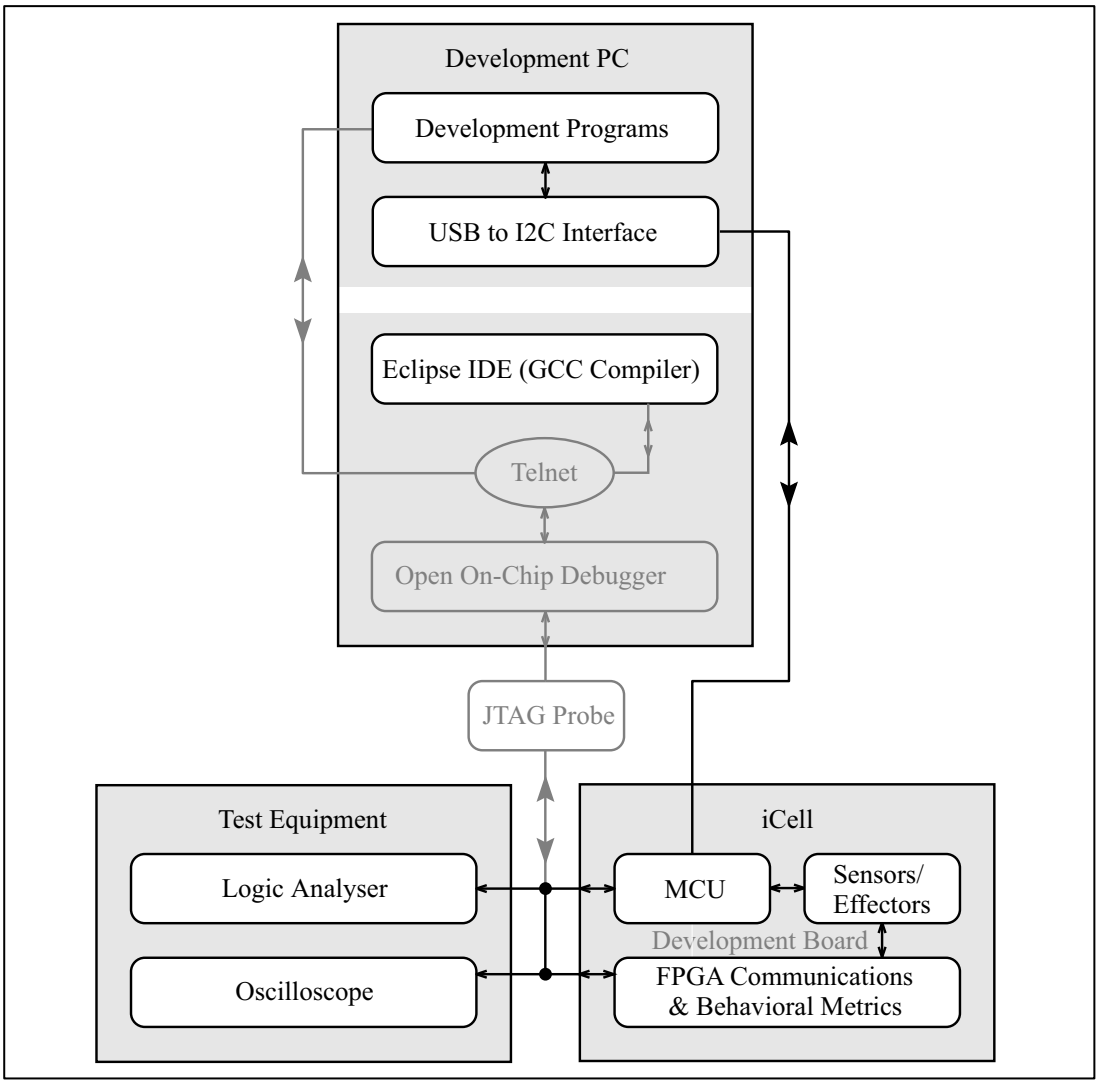


Figure 6: Software based experimental platform and instrumentation.

4.6 Program Structure Table

Ideally the Program Structure Table should have the following properties:

- Can be created quickly in software or SOC.
- Resolution to record all branch locations.
- Format that allows fast comparisons to verify program structure.
- Minimal size to maximize program space in a software implementation and lower costs in a SOC implementation.

The first three properties could be met by storing the presence or absence at a memory location by use of a single bit. This method requires no complex calculations to enable reversible cross checking of table entries and branch locations and allows for high speed operation in software or SOC solutions. Using this method the entire program memory of 64 kB would require a table size of 8 kB (8 bits per byte). Although workable it would severely limit the program space or be an

expensive SOC solution. However in the case of the AT91SAM7S256 processor all branch instructions are placed on even addresses as are all instructions due to its particular 32-bit architecture. Therefore only one bit of the Program Structure Table is required for every two bytes of program memory thus reducing the table requirement to 4 kB. The possibility to reduce the table size even further by skipping N bytes of the program memory without loosing too much program structure accuracy was also explored.

When looking at assembler code it is quite apparent that branch instructions like most other instructions seem to have a somewhat random distribution. To better understand the effects of further Program Structure Table reduction, analysis of branch distribution was performed. Four basic low complexity software routines were employed to ensure representative and comparable results. More specifically the test programs were based on algorithms from the automotive package from the MiBench suite of benchmark algorithms [36], namely: Angle Conversion, Bit Count, Cubic Functions and Random Numbers. A possible approach to obtaining the branch addresses needed for analysis would be to perform a simple parse of the compiled test programs binaries, noting the addresses of valid branch instructions. However this method would result in many false positives due to data areas being parsed as well. The solution employed was to direct the compiler to produce comprehensive listings that included branch addresses. A program was then written to extract the branch memory locations from these listings and then perform the analysis. This was done for all four programs and the results can be seen in Fig. 7. It can be seen immediately that no branches are closer than 12 addresses apart, so a single bit in the Program Structure Table could represent the presence or absence of a branch for every 12 bytes of program memory without loss of resolution. The rather curious similarity between the 4 test programs distribution is due to common library routines used by all 4 programs. This reduction would bring the Program Structure Table size down to 5462 bits or 683 bytes assuming 64 kB of program memory. The sharp rise in the number of branches 16 bytes apart is quite apparent from the graphs. Calculations show that choosing to further reduce the Program Structure Table and increase granularity by assigning 1 bit to 16 memory locations results in an average loss of 10% program structure detail, in other words 10% of the branches in the program memory space would not be represented in the Program Structure Table. Further processing of the data used to produce the branch

distribution graph (Fig. 7) allowed a graphical look at the relationship between the Program Structure Table size (granularity) and loss of program structure detail, see Fig. 8. This was achieved by plotting the percentage of entries already plotted against the total entries in the data set whilst proceeding from the shortest to the longest branch distribution entries. In this way the graph shows the percentage of program structure detail (closer branch distributions) not represented in the Program Structure Table. The effects of a less than optimal Program Structure Table size of 683 bytes are investigated further later in this work, however the optimal size of 683 bytes is perfectly acceptable for use as a means to check code integrity of iCells within the iSurface. It should be noted that locating branches for the building of the Program Structure Table can implement the much more simple approach of parsing and checking the entire program memory space for branch instructions, since false positives found in data areas will not be encountered when checked for falsification at runtime.

The merits of a second table like the first but being based on branch destination addresses will be evaluated if more program structure detail is deemed necessary.

4.7 Summary

A method to define and store ‘program structure’ was developed in software that can later be implemented in hardware. The program structure was defined as locations of branch instructions in program memory. Due to the distribution of the branch instructions, single bit flags could be used to indicate their presence in a Program Structure Table at an optimal 1 bit per 16 bytes of program memory.

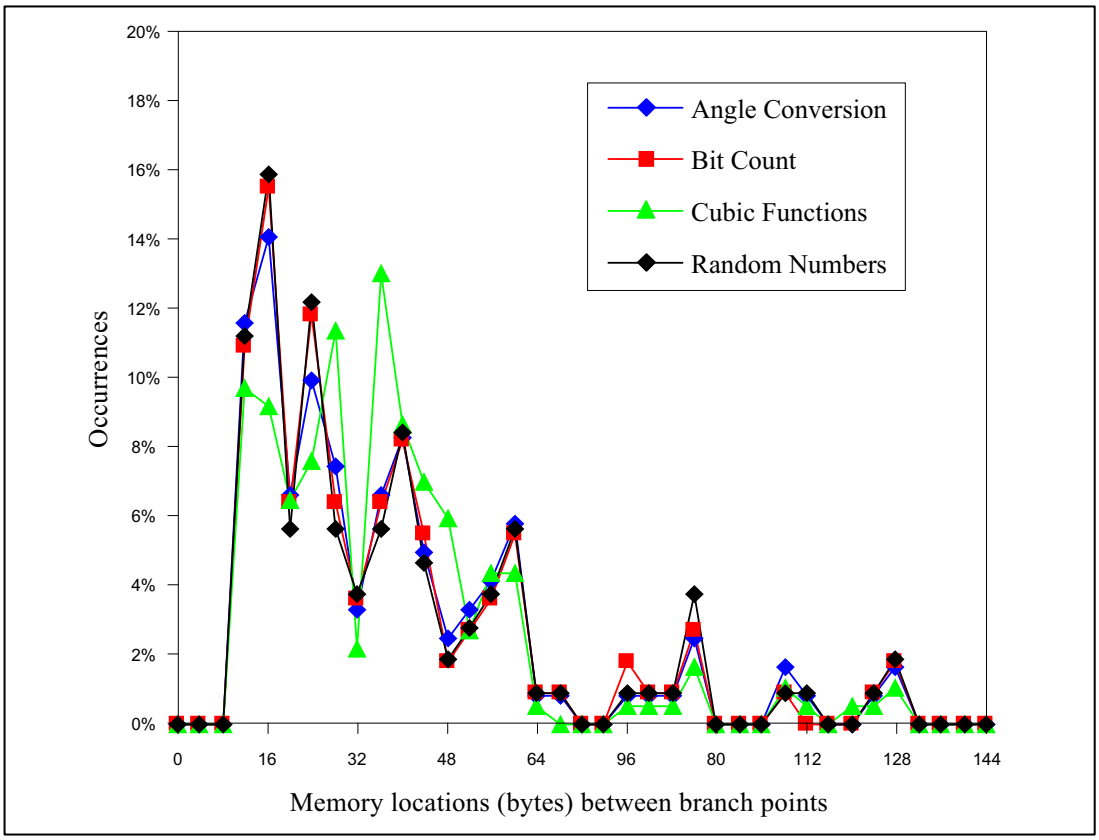


Figure 7: Branch distribution.

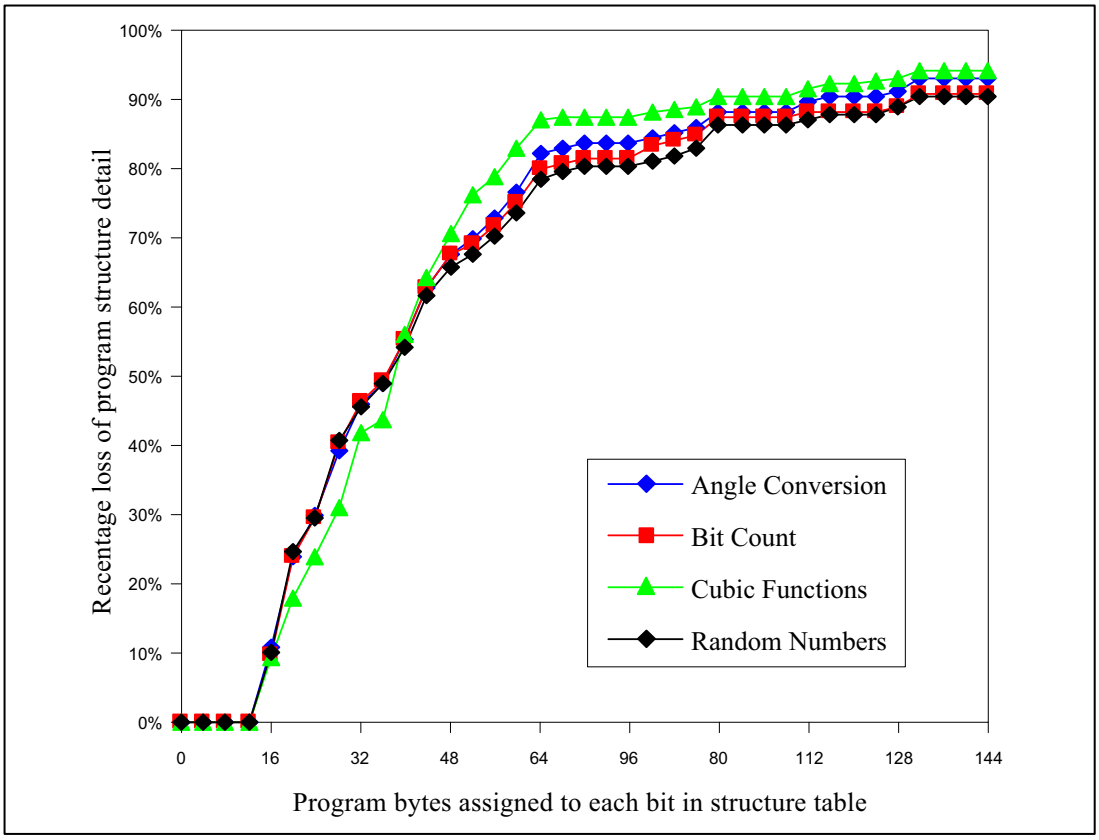


Figure 8: Loss of program structure detail due to program structure granularity.

Chapter 5: Code Integrity Metric

The Code Integrity Metric is derived from the Program Structure Table and should ideally have the following properties:

- Can be created quickly in software or SOC.
- Retains the uniqueness of the Program Structure Table.
- Variable length (bits).

To maintain the uniqueness of the Program Structure Table it was clear that all bits in the Program Structure Table must in some way be used in the creation of the Code Integrity Metric, i.e., any bit change in the table would result in a change of the metric. Whilst there are many possible approaches to meet these goals it was decided to investigate the simplest and most obvious which is to XOR the bits in the Program Structure Table in such a way as to create the new reduced length bit pattern of the Code Integrity Metric. A decision on how best to XOR the bits of the Program Structure Table and maintain the structure information of particular programs was needed since quite similar programs may produce the same Code Integrity Metric. Two simple XOR bits reduction patterns were chosen to be evaluated for the uniqueness quality of various lengths of metric created from the 4 test programs. Table 1 and Table 2 show these 2 patterns of XOR bit reductions from a small 32-bit (4 byte) Program Structure Table to a 4-bit metric. Whilst these examples are of little practical use due to size, this visualization may allow us to deduce properties of both before we perform tests.

If we examine the first method shown in Table 1, it will be noticed that sequential bits in the Program Structure Table are XORed together to create a single bit of the Code Integrity Metric. This sequence length can be calculated using the following formula:

$$\textit{Sequence Length} = \textit{Table Length} / \textit{Metric Length}$$

where ‘Sequence Length’ is the number of sequential bits XORed together, ‘Table Length’ is the total number of bits in the Program Structure Table, and ‘Metric Length’ is the bit length of the Code Integrity Metric. This method which we shall call ‘Type 1’ will reflect the characteristics of the Program Structure Table and hence the position of the branches in the program memory and could therefore be used to identify roughly where in memory the metric doesn’t match with the

expected one. However a disadvantage is that small local changes that you find with minor alterations in code may not show as a change in the Code Integrity Metric. This could be a major issue if the design of the iCell required the independent loading of small programs. The reason being, that a section of program memory could be reprogrammed with legitimate code and that change could be entirely reduced to 1 bit of the Code Integrity Metric. Since that single bit was created by an XOR process, there is a 50:50 chance the variation will not show as a change to the Code Integrity Metric.

The second method shown in Table 2, (Type 2) sources the bits required for XORing across the entire Program Structure Table in a stepwise fashion with each step being the length of the Code Integrity Metric in bits. Such a distribution results in the loss of any direct relationship between the program structure and the Code Integrity Metric, however the uniqueness of the Code Integrity Metric with similar programs should be greater than that using the Type 1 XOR pattern. The uniqueness with both XORing patterns will also be related to the length of the Code Integrity Metric, as clearly the fewer bits used and the granularity becomes coarse, there is less opportunity for the Code Integrity Metric to express unique metrics for different programs. With this in mind, tests were performed to determine the uniqueness of the Code Integrity Metric using the 4 test programs with a range of metric lengths with both types of XOR pattern.

5.1 Uniqueness of the Code Integrity Metric

It was possible to extract the uniqueness data entirely on a PC. The 4 test program were compiled using the Eclipse development environment and the binaries intended for loading into the targets (AT91SAM7S256) SRAM were then used as input files to an analysis program developed and running on a PC.

First the SRAM files were scanned for branch locations using simple binary comparisons at each memory location. Then the Program Structure Table was built using the branch location data. The Program Structure Table was 8192 bits (1024 bytes) in length which works out at 1 bit for every 8 bytes of program memory (64 kB). A conservative size for the Program Structure Table was used to maximize structure detail since the object of these tests is centred on the Code Integrity Metric.

With the Program Structure Tables complete, the various Code Integrity Metrics were created using both Type1 and 2 XOR patterns and bit lengths ranging from 4 to

32. With just 4 test programs it seemed unreasonable to go beyond 32 bits unless the results showed otherwise. Fig. 9 and Fig. 10 show results of program uniqueness utilizing XOR patterns Type 1 and 2 respectively. Note that the resulting Code Integrity Metric value was scaled due to the variable bit length. Maximum shown in the graphs represents the maximum numerical value possible for the various Code Integrity Metric bit lengths. Also note that for clarity matching values have been circled.

When viewing the results of the analysis utilizing the Type 1 XOR pattern (Fig. 9) it will be noticed that as expected the ability of the Code Integrity Metric to differentiate between programs suffers to a greater degree as the granularity becomes coarser compared to that of the Type 2 XOR pattern (Fig. 10). The ability of the Code Integrity Metric based on the Type 2 XOR pattern to differentiate 4 similar programs with a metric length of only 6 bits (64 possible values) seems a good result. The iCells will operate with metric lengths of 128 to 256 bits, so program integrity checks using this system should produce unique metrics for any program.

Table 1: Table to metric XOR Type 1.

32 Bit Table (4 Bytes)															4 Bit Metric	
0	^^	1	^^	2	^^	3	^^	4	^^	5	^^	6	^^	7	=	0
8	^^	9	^^	10	^^	11	^^	12	^^	13	^^	14	^^	15	=	1
16	^^	17	^^	18	^^	19	^^	20	^^	21	^^	22	^^	23	=	2
24	^^	25	^^	26	^^	27	^^	28	^^	29	^^	30	^^	31	=	3

Table 2: Table to metric XOR Type 2.

32 Bit Table (4 Bytes)															4 Bit Metric	
0	^^	4	^^	8	^^	12	^^	16	^^	20	^^	24	^^	28	=	0
1	^^	5	^^	9	^^	13	^^	17	^^	21	^^	25	^^	29	=	1
2	^^	6	^^	10	^^	14	^^	18	^^	22	^^	26	^^	30	=	2
3	^^	7	^^	11	^^	15	^^	19	^^	23	^^	27	^^	31	=	3

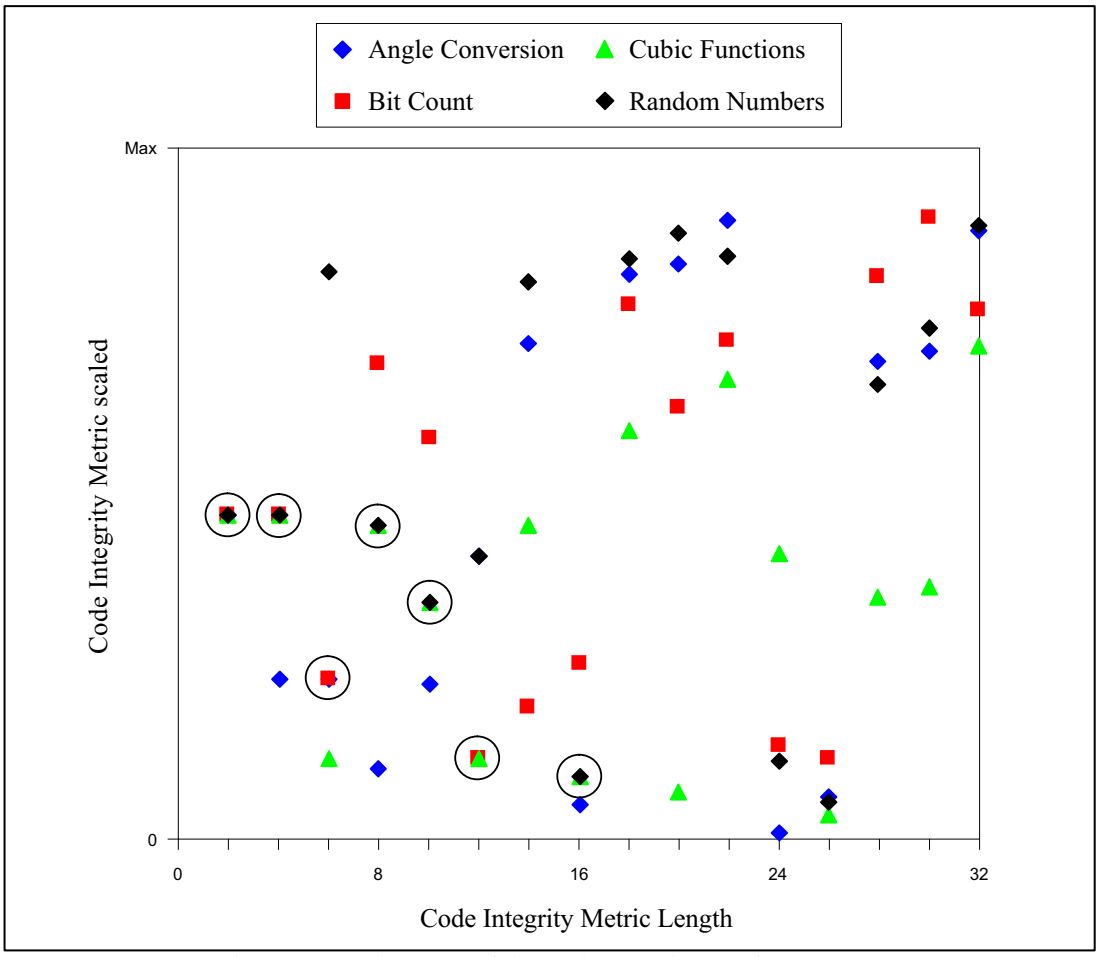


Figure 9: Uniqueness of the Code Integrity Metric (Type 1).

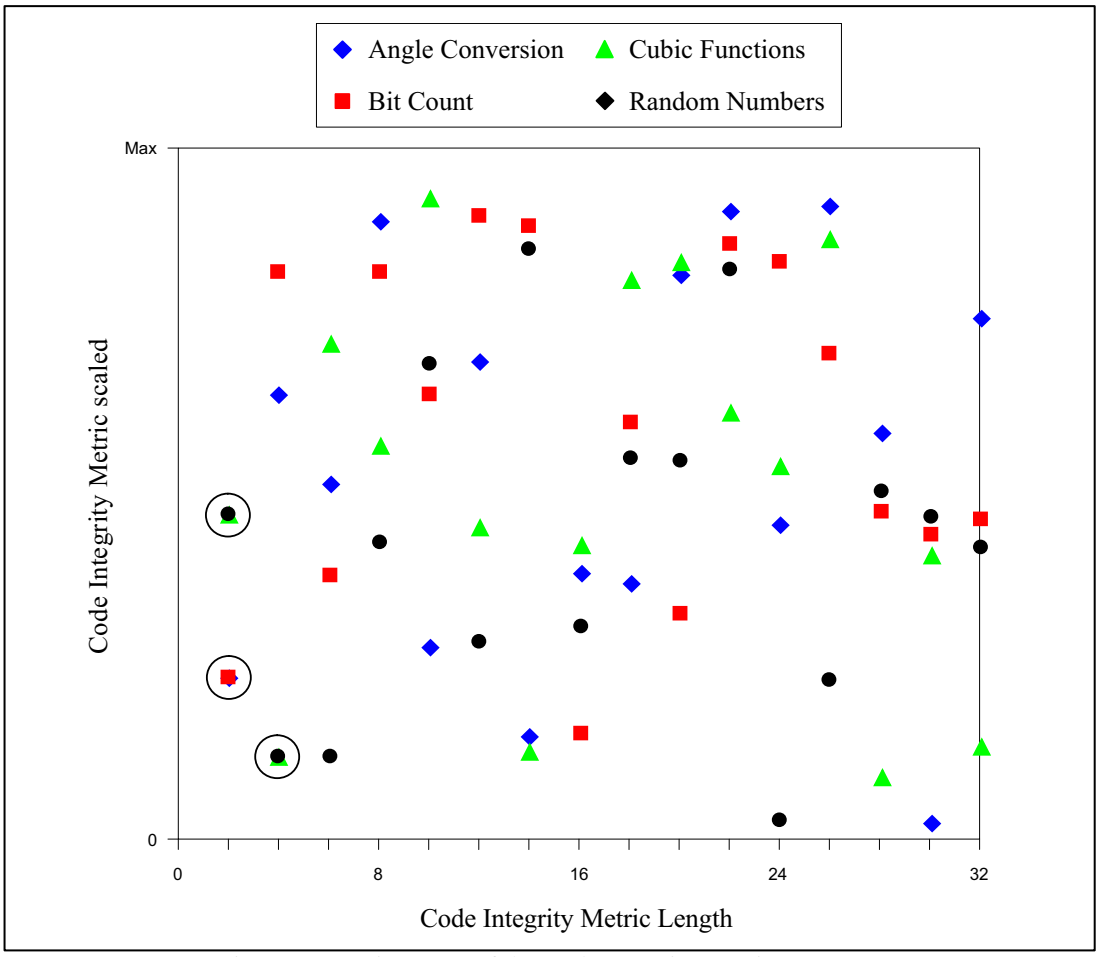


Figure 10: Uniqueness of the Code Integrity Metric (Type 2).

5.2 Falsification of the Program Structure Table

A method of falsification was required that ideally had the following properties:

- Low or no intrusiveness to normal program operation.
- Achievable in software or SOC.
- Targets current program activity.
- Fast operation.

Whilst initial work on the iCell development is based on the AT91SAM7S256 with a communication layer working in a Field-Programmable Gate Array (FPGA), the final design may well be a single chip FPGA soft/hard core implementation. With this in mind, a SOC solution with a zero intrusive nature is an attractive proposition. A practical SOC solution is quite straightforward and would rely on direct access to the program counter the data read from memory during normal operation. Fortunately such access is possible in a totally non intrusive way when using soft cores in an FPGA. Operation would involve comparing the data read from memory checking for presence or absence of branch instructions and cross checking with the

Program Structure Table. Once a mismatch is detected, the program code is falsified and a new scan of program memory is initiated to create a new Program Structure Table and Code Integrity Metric. In conclusion, a SOC solution would meet all ideal requirements.

At this stage of the research a slight enhancement to falsification of code integrity was made. Whilst proceeding with the search for a branch instruction, cross checking with the Program Structure Table entries at each location was introduced. This small change would speed up the average falsification of the program structure with no extra overheads.

5.3 Speed of Falsification

Initial use of the Code Integrity Metric within the iSurface will be a determination of correctly loaded program code in each iCell. Therefore an experiment to determine average time taken for the system to respond to a change of program in SRAM would provide useful information and help system optimization and future development. Speed of falsification of program structure would likely be related to the granularity of the Program Structure Table, so this experiment offered the chance to try all permutations of reloading the SRAM with the test programs and varying the table size (bits per program memory bytes). The results of these tests can be seen in Fig. 12 to Fig. 15.

Each graph shows the results of the 4 test programs replacing the others in program memory. For example, Fig. 12 shows results of the ‘Angle Conversion’ program replacing ‘Bit Count’, ‘Cubic Functions’, and ‘Random Numbers’ in program memory. The granularity of the Program Structure Table is shown along the x-axis as program bytes assigned to each bit. The x-axis shows the average (mean) attempts required to determine that the program is not current and has been replaced by another in program memory. Average speed of falsification ‘time’ taken to falsify the program structure can be calculated by multiplying the average number of attempts by the interrupt timer period. For example, an average number of attempts of 1.5 would take an average time of 30 milliseconds, assuming an interrupt period of 20 milliseconds.

It will be noticed that speed of falsification is more dependent on the replacement programs structure rather than what it replaced which is to be expected since falsification by branch determination will depend on the current structure detail. In

particular ‘Bit Count’ requires more detail (finer granularity) in the Program Structure Table to determine a change of program. Which can be explained by the more simple nature of this program. The binary count program is very short (16 branches in the main program) and therefore there is less opportunity to locate detail variations from the program structure stored in the Program Structure Table. These results provide more evidence that the Program Structure Table requires the detail afforded by the fine granularity of a Program Structure Table of at least 1 bit for every 16 bytes of program memory. It should also be noted that even large programs contain small routines that could be executed for long periods of time, particularly so in small embedded systems such as the iCell. Fig. 11 illustrates the program structure maps for the 4 test programs.

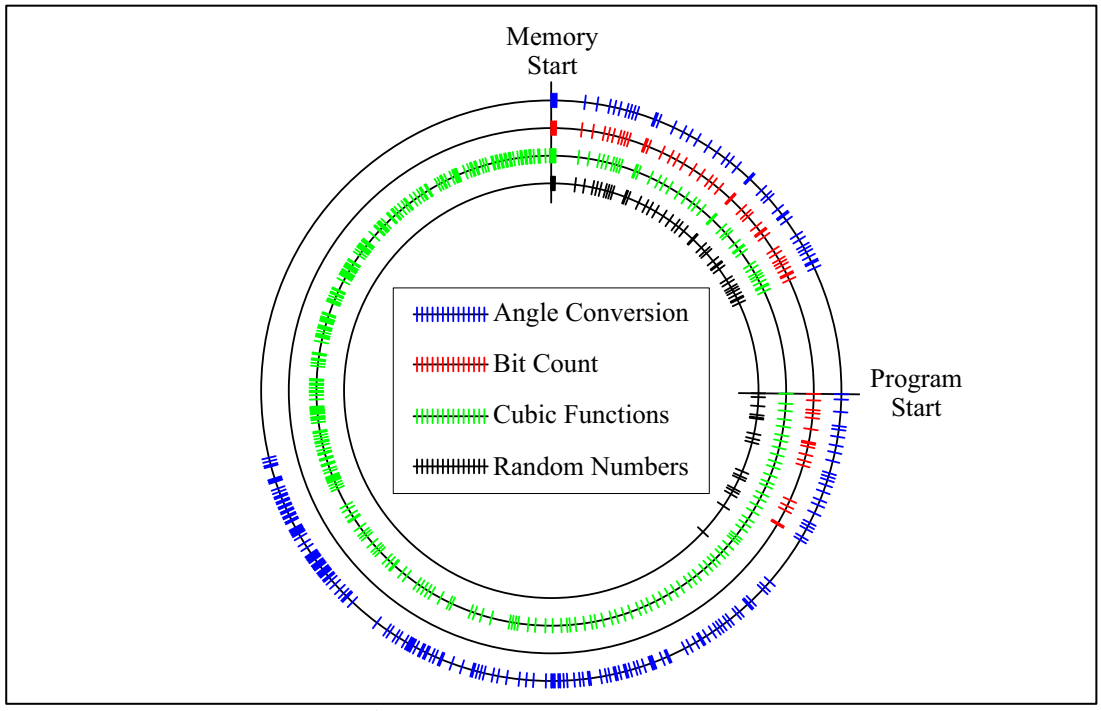


Figure 11: Program structure maps.

These maps only show structure detail relevant to the developed system and as such are a visualisation of the Program Structure Table. Although the test programs are not complex and range from 3 kB to 27 kB in length, to maintain clarity the structure maps were limited to 8 kB requiring the Cubic Functions map to be cut short. It will be noticed that the start of memory is identical for all 4 test programs due to common structures such as vector tables.

This optimal table size of 512 bytes (1 bit for 16 bytes of program locations) for a 64 kB system, works out at 1.28% of the total SRAM space, making a SOC implementation for other uses such as secure communications commercially viable. Another possibility to reduce SOC costs is to dual use the existing JTAG interface pins. In this scenario the JTAG pins can be reconfigured as an external 512 byte serial SRAM interface whilst in run mode, thus a simple redesign of the internal JTAG interface logic could provide a code integrity and behavioral metrics without significant cost penalties to the MCU, whilst retaining 100% electrical and physical compatibility.

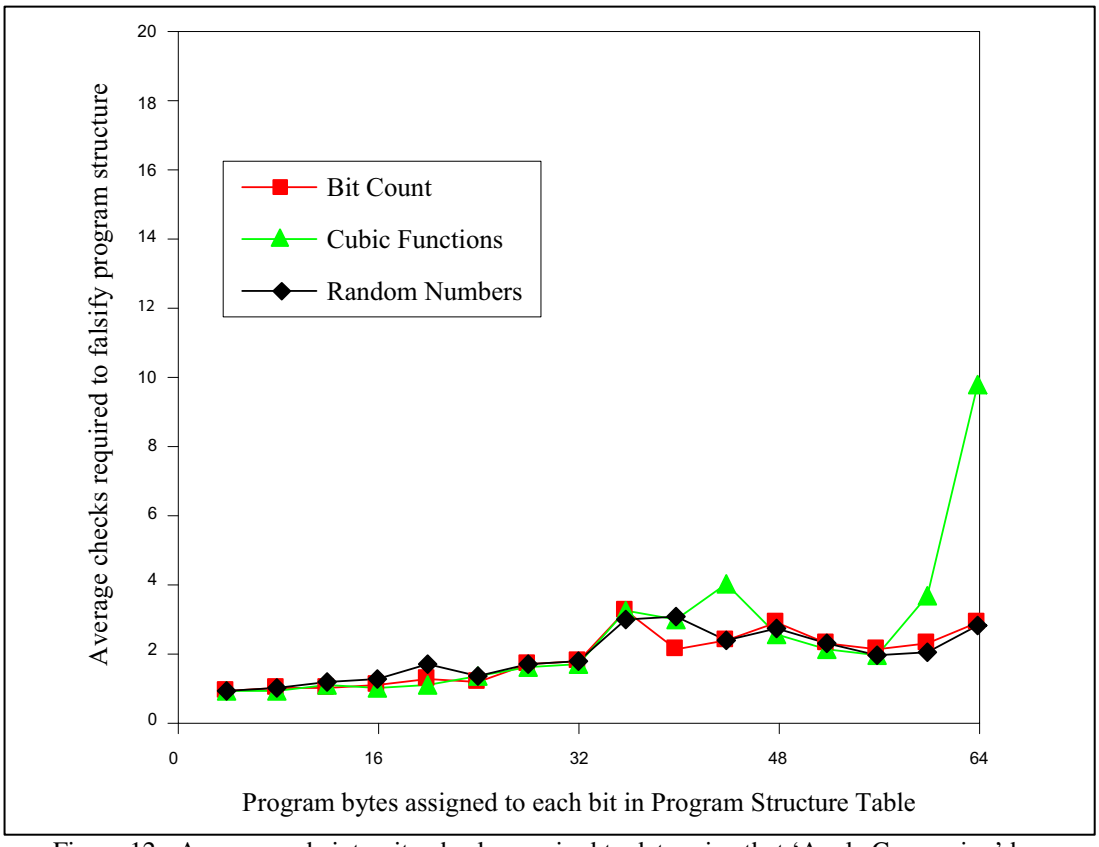


Figure 12: Average code integrity checks required to determine that 'Angle Conversion' has replaced the other 3 test programs in program memory.

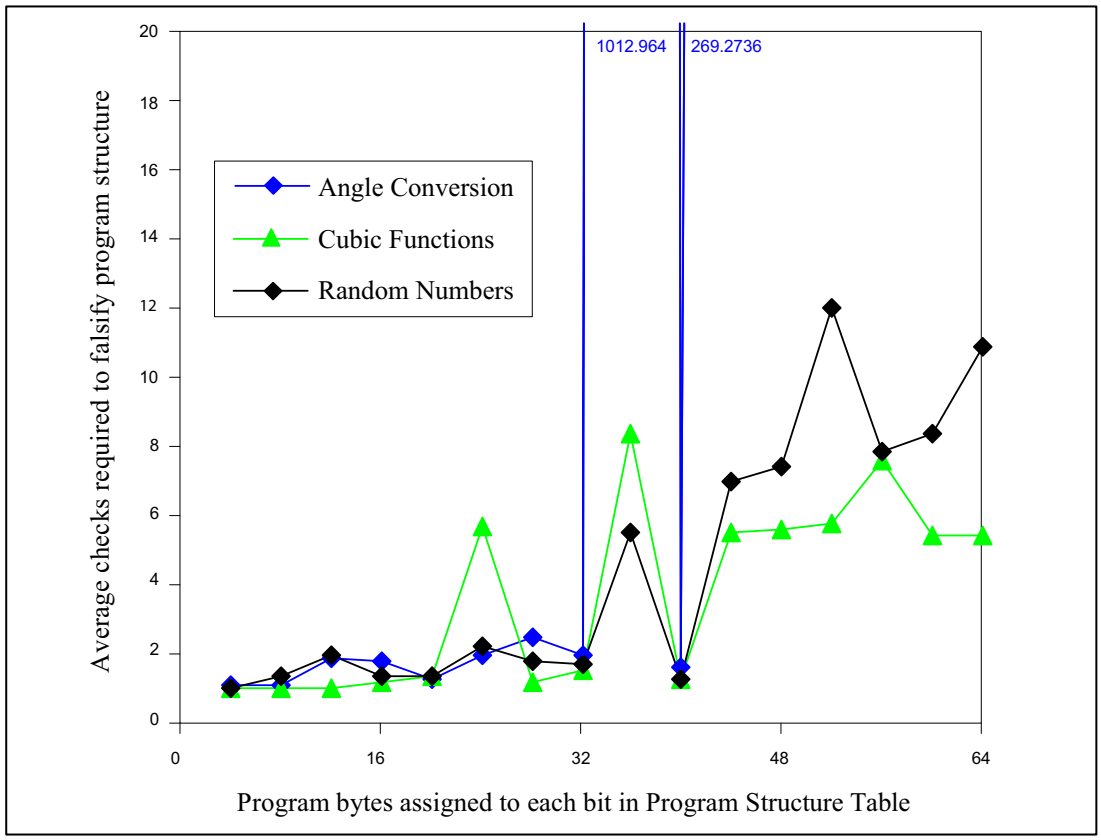


Figure 13: Average code integrity checks required to determine that 'Bit Count' has replaced the other 3 test programs in program memory.

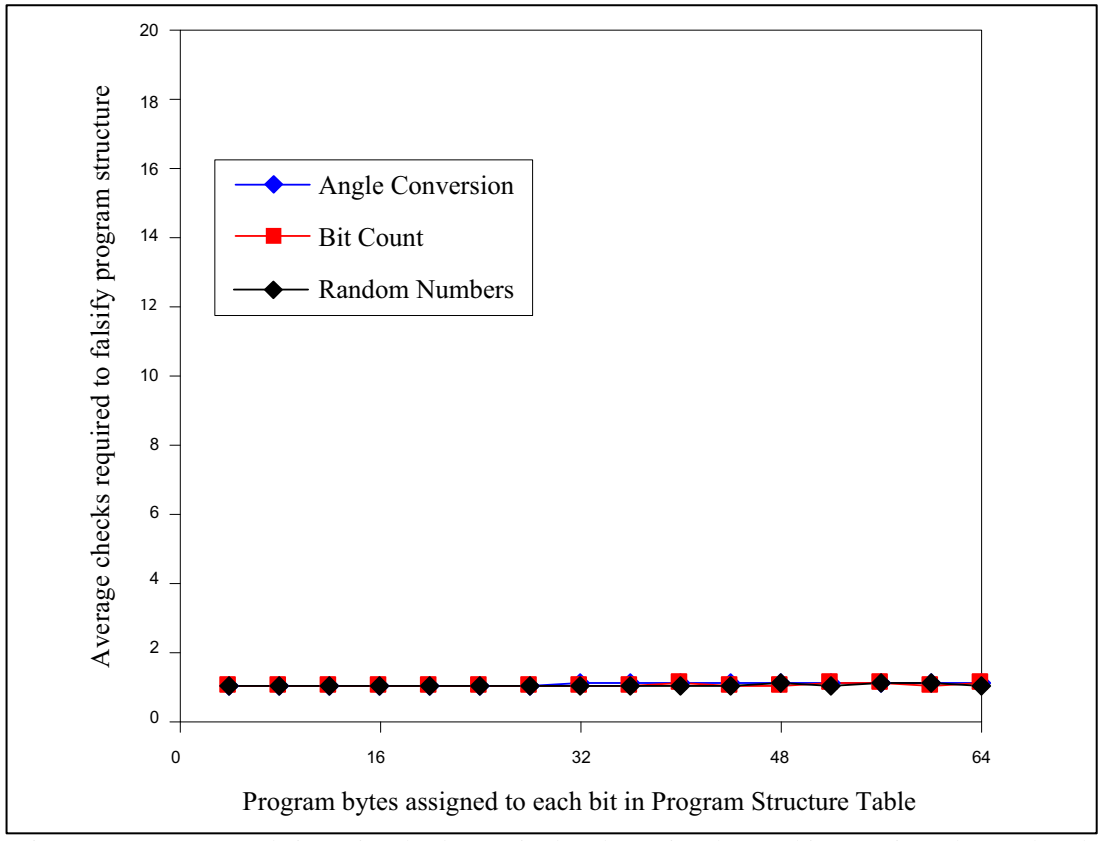


Figure 14: Average code integrity checks required to determine that 'Cubic Functions' has replaced the other 3 test programs in program memory.

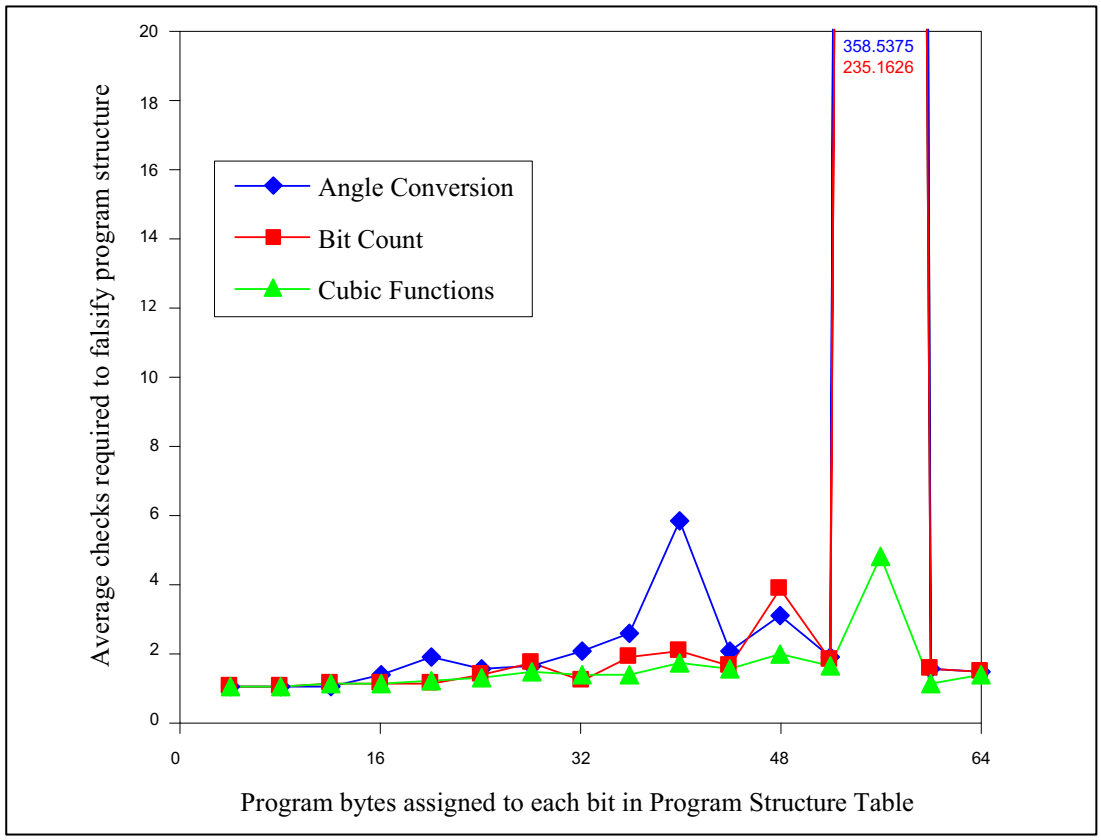


Figure 15: Average code integrity checks to determine that 'Random Numbers' has replaced the other 3 programs in program memory.

5.4 Behavioral Metric

As mentioned earlier in this thesis the primary goal was the implementation of a stable metric of code integrity programmed in an FPGA. However the original idea was complimenting this metric with one of behavior. These two metrics could then be 'spliced' together like RNA and read as a single variable length binary bit pattern to describe the functional nature of the iCell (program running) and its dynamic behavior (what its doing). It would be rewarding although not essential to implement a simplified version of a behavioral metric as 'proof of concept' that the idea of a combined metric of code integrity and behavior has merit. With this goal in mind, determination of the ideal method to create the behavioral metric will be considered 'future work' (see relevant section at the end of this thesis).

There are many possible ways of implementing a metric of dynamic behavior reflecting current activity that compliments the stable metric of code integrity. However it made little sense not to use data already obtained from the implementation of the metric of code integrity which meant the best option was utilizing standard processor profiling techniques such as the commonly used program counter sampling [15],[16]. The profiling technique chosen acquires the

program counter on a regular basis (sampling), thus the address of the current operation code is known. This is repeated at a precise frequency (50 Hz), which is a typical frequency used for processor profiling by way of sampling and also happens to be the rate used by the metric of code integrity. The collected data from such a system can be analyzed to determine approximately how often different memory addresses are accessed over a period of time. In our application a relatively simple variable length metric must be derived from the sampling data so setting a single bit when a short range of addresses is accessed should suffice. After a predefined number of samples are taken the resulting table of bits would then be converted to a metric of behavior. Interestingly Type 1 XOR pattern evaluated earlier in this thesis would likely be an ideal candidate for the table to metric conversion. The reason being is that the state of each resulting bit in the metric would be dependent on the processor's access to a linear continuous section of program memory.

Due to the primary goal of the research being the development of the stable metric of code integrity and time constraints it was decided that only a proof of concept was required. The simple metric of behavior was developed using a software approach utilizing the same embedded ARM platform used in the early stages of the research.

The results were technically very promising but since no analysis of the behavioral metric was undertaken, all that can be concluded is that this approach could be implemented with little overhead in terms of FPGA area over and above the metric of code integrity since it shares many similarities in construct and architecture. Certainly this is a prime candidate for further research. Please see "Future Work", chapter 10.1.

5.5 Summary

A method to create a metric of code integrity derived from the Program Structure Table was developed. A simple XORing of the Program Structure Table (PST) bit pattern was considered a logical method of reduction to create the metric of code integrity, the reason being that any single bit change of the table will alter the resulting metric. Experiments were then performed that determined the optimal XORing method to produce the most unique metric of code integrity for similar running programs. Further tests followed to determine the optimum size for the PST that ensured responsive falsification of altered program code.

Chapter 6: iCell Hardware Development

6.1 Considerations on the iCell architecture.

Research up to this point has established a method to create a metric of code integrity and briefly outlined a likely method to implement a metric of behaviour. The goal in this second phase of the research is to implement what has been done in software totally in hardware as to be totally transparent to any software running on the processor being analyzed. The importance of a non intrusive method of analysis is primarily to allow any software to run as expected, even poorly written code relying on processor cycle dependent timing loops. The second reason is a possible future use to determine deliberately compromised code and the need to be undetectable by said code. As mentioned in the previous chapters we have kept a mind on a hardware architectural solution whilst developing and evaluating the software based implementations.

A simplified overview of the iCell hardware architecture can be seen in Fig. 16, where it will be noted there are three main sections making up the hardware. The iCell's main processor (MCU) that determines the functionality of the iCell can be seen at the top left. This is connected to the FPGA whose primary functions are high speed inter iCell communications and fault tolerance by way of iMetrics developed in this research. It will be noted that the on-board sensor/effectors are routed to the MCU through the FPGA. This does to a degree future proof the design and allows for preprocessing/protocol conversion of sensor data or the possibility of direct sensor access by the iSurface network using inter iCell communication links. Note that some sensors are connected directly to the MCU and cannot be routed through the FPGA due to the use of the I2C serial bus which is open drain.

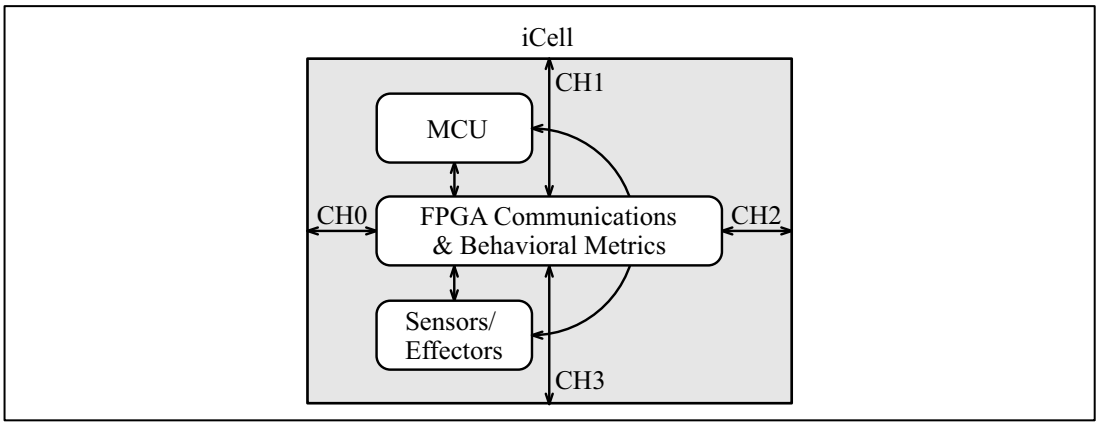


Figure 16: Simplified iCell hardware architecture.

6.1.1 Processor (MCU)

The obvious choice of processor was that which was used previously on the development board during the first phase of this research. Firstly the use of this processor will allow much closer comparisons between the earlier software and the later hardware based methods of creating metrics of behavior. Secondly valuable information and understanding of the JTAG communication of that particular processor could be gleaned by examining the data waveforms of the development board and much knowledge of its operation and nuances had been learnt, thus making the design process that much easier.

MCU specifications for the iCell are:

- AT91SAM7S256 microcontroller
- 64 kBytes of SRAM
- 256 kBytes of FLASH
- 48 MHz clock (typically 1 instruction per clock cycle)
- 2 serial ports offering up to 115200 baud
- JTAG interface

6.1.2 FPGA and SRAM

As with the choice of processor, previous hands on use of the MachXO FPGA's [37], manufactured by Lattice makes use of them here a sensible choice. A very nice feature is they benefit from both the complexity of an FPGA but also have the guaranteed pin to pin timing of a CPLD (Complex Programmable Logic Device). The primary task of the FPGA will be the control of the MCU by way of the JTAG interface. Until work is done in understanding this communication control protocol it is difficult to determine the complexity of the task and evaluate the likely required

number of logic gates. Therefore it was prudent to err on the safe side and select one of the largest FPGA's available in the MachXO range, thus the 1200 LUT (Look Up Table) device was chosen. It should also be noted that a larger pin compatible device was available thus reducing the risk of unexpected logic design complexity.

Other fundamental design decision was to route all general purpose IO (Input/Output) pins on the processor though the FPGA. First of all this approach allowed flexibility on processor pin linkage to sensors and effectors with the option of extra intermediate interface logic and secondly it offered great flexibility to the fast four port iCell communication network design.

FPGA specifications for the iCell are:

- LCMXO1200 FPGA
- 1200 LUT's
- 48 MHz clock.

A minor part of the iCell architecture is the need for non-volatile memory to store the Program Structure Table. Since it was determined in 5.3 Speed of Falsification, that the optimal table size was 4096 bits (512 bytes or 4 Kbit), it was decided that this would be used in the hardware implementation and be the mode of operation when comparing the hardware version of the iCell with the earlier software based system. Although hard coded in the FPGA it could be made selectable at a later date if required. Fortunately a low cost serial SRAM that utilized an SPI (Serial Peripheral Interface) [38] interface was found to be quite suitable with a sequential write mode that would make implementation of the Program Structure Table much easier. Although only 4 Kbit of SRAM memory was required, a popular larger 64 Kbit version of this device was selected due to availability and cost. The additional memory space that this device offered would also allow for future development, supporting perhaps larger tables or more importantly, data space to create metrics of behavior.

SRAM specifications for the iCell are:

- 23K640 Serial SRAM
- 64 kb
- 12 MHz clock.

6.1.3 Sensors and Effectors

A broad range of sensor and effectors were chosen to give the iCell good functionality and allow a wide range of experiments to be carried out including determination of sensor/effector failure or subtle changes in performance by way of metrics. When possible the interfaces were routed through the FPGA to allow hardware data sniffing and augmented functionality. The analogue nature of the ambient light sensor and microphone required a direct connection to the analogue inputs of the processor and the open drain I^2C interface of the 3-axis accelerometer needed direct connections as well.

Sensors/Effectors for the iCell are:

- ambient light sensor
- RGB LED light emitter
- microphone
- loudspeaker
- capacitance proximity sensor
- 3-Axis accelerometer
- temperature sensor

6.1.4 Communications

Whilst the current research is targeting fault tolerance and in particular a method utilizing metrics, it would be diligent to consider inter-cell communication requirements thus current hardware can be used in future work. The intended structure of the iCell architecture of rows and columns was outlined and discussed in the introduction. With this in mind, each iCell would be square in shape requiring 4 communication ports situated on all 4 sides. To maintain maximum bandwidth each port should have at least separate data and clocks both in and out. Also physical simulation of linkage failure requires removable jumpers on serial data lines.

Another useful communication link is a generic bidirectional infrared system that can be used with smart devices, tablets, phones or maybe hand held remote control units.

Communications for the iCell are:

- 4 high speed serial iCell to iCell bidirectional communication links.
- bidirectional infrared communication link.

6.2 Theoretical Hardware Operation

6.2.1 The Advantages of Parallel Operation

Here we consider operations required and how they may be implemented in the FPGA. One huge advantage of a hardware implementation is the inherent parallel nature of logic found in programmable devices such as an FPGA. For example the Code Integrity Metric needs only be determined when the Program Structure Table is created. In the software solution the Program Structure Table was first created by parsing processor memory and then the Code Integrity Metric was created by parsing the Program Structure Table. However the parallel nature of the FPGA means both can be created simultaneously, see Fig. 17. Time saving was not actually the big advantage with this approach, it did however reduce the amount of logic considerably and is a good example of how operationally different the hardware and software solutions proved to be.

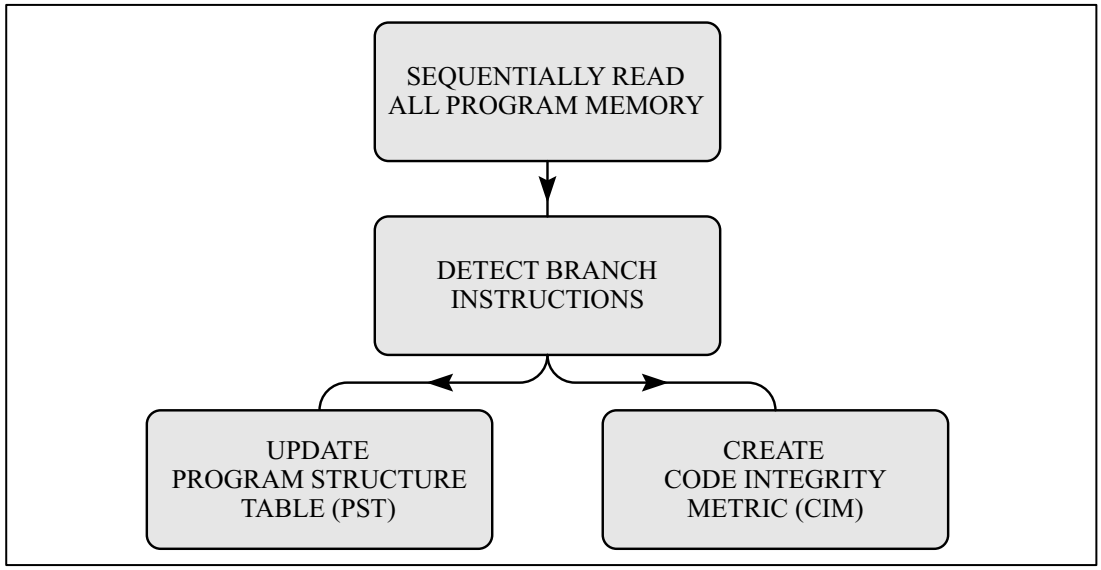


Figure 17: Simultaneous updating of the Structure Table and creation of the Code Integrity Metric.

With parallel solutions that the FPGA could offer in mind, a simplified operational flow diagram was drawn up to aid the design process. The final revision of that flow

diagram can be seen in Fig. 18. Although the FPGA design process hit many problems at the lower subsystem levels the higher level flow diagram proved sound and required minimal changes.

6.2.2 State Counters

It was decided that the best approach to implementing the rather complex FPGA design was to use a hierarchy of state machines. From previous experience this should result in a reduction of logic and result in easier development due to the more manageable modular nature of the subsystems. Referring to the operational flow diagram shown in Fig. 18, it will be noted that each block represents the highest level state machine and each has a hexadecimal number showing the 4-bit state counter.

It was considered that the state counters should if possible use a gray code count [39]. However it was soon determined that it would not be possible in all instances due to state machine flow restrictions and the registered nature of the design would offer no advantage anyway.

All state counters are reset at power-on to known states. This is accomplished by using the MCU's hardware reset output. The MCU has internal circuitry that monitors power supply voltages and issues an internal reset on power-on and pulls an external line low for other devices. It will be noted that there are two possible states at power-on. The first was implemented merely for development use to allow test equipment to be set up ready to trigger and capture signal waveforms and data. The state machine would continually loop in state 'F' waiting for a button to be pressed, at which point it would proceed to state 'C'. State 'C' is the normal power-on state if not in development mode. As with the software based development system, the profiling technique used is activated every 20 ms (50 Hz) and therefore a 50 Hz timing clock was implemented. The state machine stays in state 'C' until a trigger pulse from the timing clock is detected, at which point the state machine counter changes to state '8'.

State '8' halts the processor so memory and register can be accessed. This is done by sending a sequence of commands serially to the MCU's JTAG interface. Whilst this is being done there is plenty of free time to setup the serial SRAM ready for subsequent use. This makes use of the parallel nature of the FPGA so JTAG communication to the MCU and SRAM setup is done simultaneously. SRAM setup

is straightforward and places it into sequential access mode. Also a flag that determines whether subsequent access is read or write, is set to read. Once the processor is halted the state counter changes to '9'.

State '9' in conjunction with the decision making state 'D' forms the key operation of cross-checking the program memory with the Program Structure Table, thus determining code integrity. This operation can be seen in more detail in Fig. 19. It will be noted that because each bit of the Program Structure Table represents 16 bytes of program memory, adjustment to a 16-byte boundary is first performed. State '9' also sets two flags, the first ('Bra') is reset low and the other ('SRAMBra') is determined by the Program Structure Table held in SRAM. It can been seen in Fig. 19 that 16 bytes must be read from program memory for each cross-check of the Program Structure Table therefore state '9' performs a total of 4 reads and sets the 'Bra' flag if a branch instruction is found. State 'D' then evaluates the flags. If both 'Bra' and 'SRAMBra' are low then there is correlation between the Program Structure Table and program memory, but no branch found. In this case the state machine returns to state '9' and performs another check for program integrity on the next 16 bytes of program memory. If however both flags are set high then there is both correlation and the detection of a branch instruction which as with the software development version requires and exit from the check on code integrity. Thus the state machine moves to state 'A' which sends JTAG commands and data to the processor that restores registers to resume operation in a transparent way. Finally a flag mismatch would indicate a change in program memory and loss of program integrity. On detection of this situation the state machine moves to state '3'.

6.2.3 Rebuild Program Structure Table and Create Metric of Code Integrity

State '3' sets the SRAM into sequential write mode ready to accept the Program Structure Table data stream and clearing the Metric of Code Integrity before moving on to state '1'.

State '1' in conjunction with the decision making state '6' forms another key operation of updating the Program Structure Table and in parallel creating the metric of code integrity. This operation can be seen in more detail in Fig. 20. It can been seen that 16 bytes must be read from program memory for each write of the Program Structure Table therefore state '1' performs a total of 4 reads and writes a logic high to the SRAM if a branch instruction is found. As mentioned earlier in this section,

the parallel nature of the programmable logic used in the FPGA allows the Metric of Code Integrity to be created simultaneously with the SRAM write operation. The Code Integrity Metric being quite short can be stored in the FPGA. Although the initial research suggested uniqueness was quite good and divergent with a low Code Integrity Metric bit count it was decided that the Metric of Code Integrity be made variable up to 128 bits to allow possible future use in security research. A single bit is selected from the stored Metric of Code Integrity using a counter following the Type 2 XOR pattern discussed early and seen in Table 2. That selected bit is then XORed with the bit being written to the Program Structure Table. State '6' loops back to continue the operation through the entire processor memory, in this case 64 kB. When the last address is detected the state machine moves to state 'A' which resumes normal processor operation. It must be noted that the time taken to rebuild the Program Structure Table and create the Metric of Code Integrity takes several seconds (4.9 s) and cannot in anyway be considered non-intrusive, however this operation would only occur when there is a change of code in the processor memory which would likely be a highly intrusive event anyway. It should also be noted that a SOC version would only require a single processor 48 MHz clock cycle per 32-bit program memory location to complete this operation thus a scan of 64 kB would take 341.33 µs.

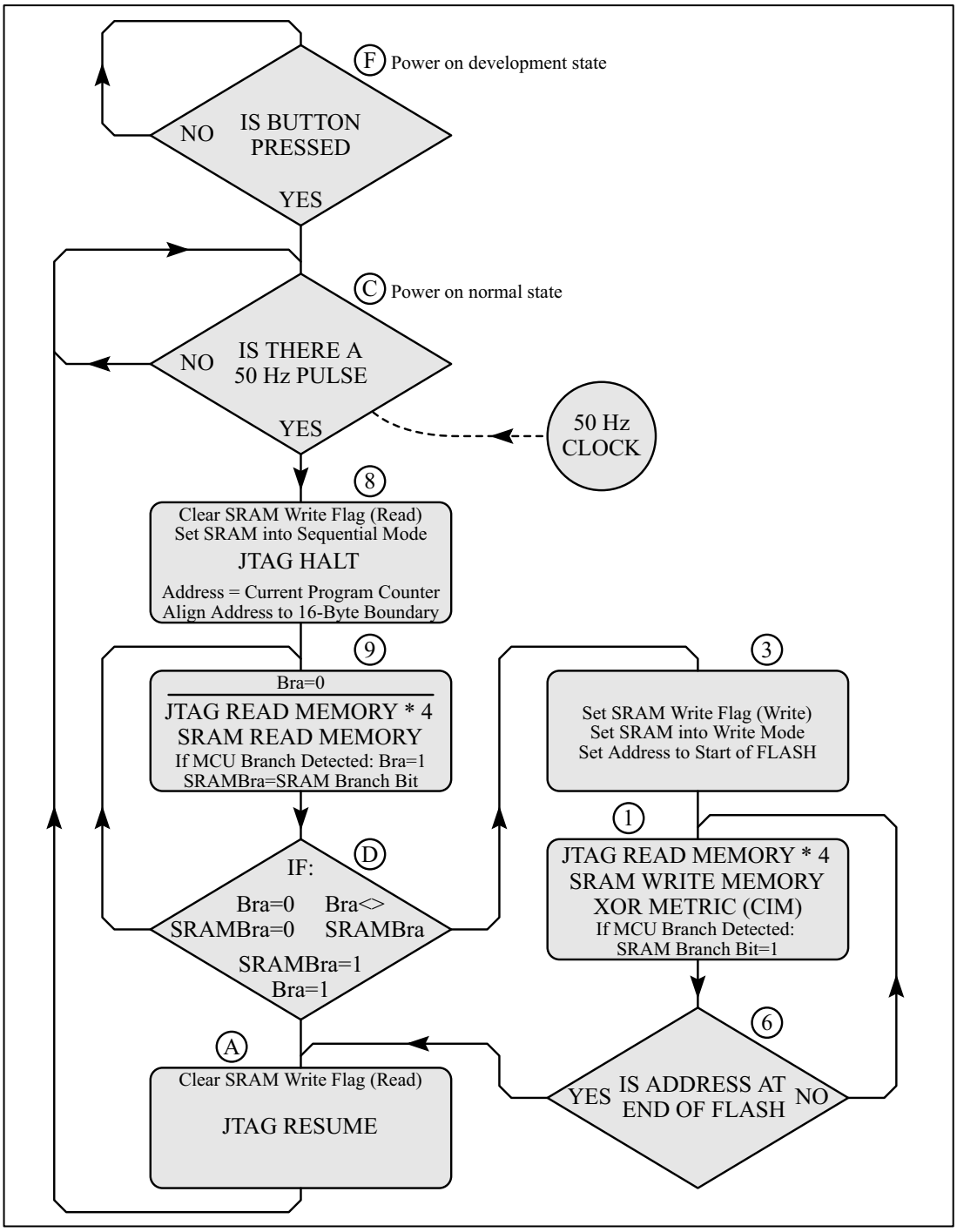


Figure 18: Code Integrity Metric operational flow / high level state machine.

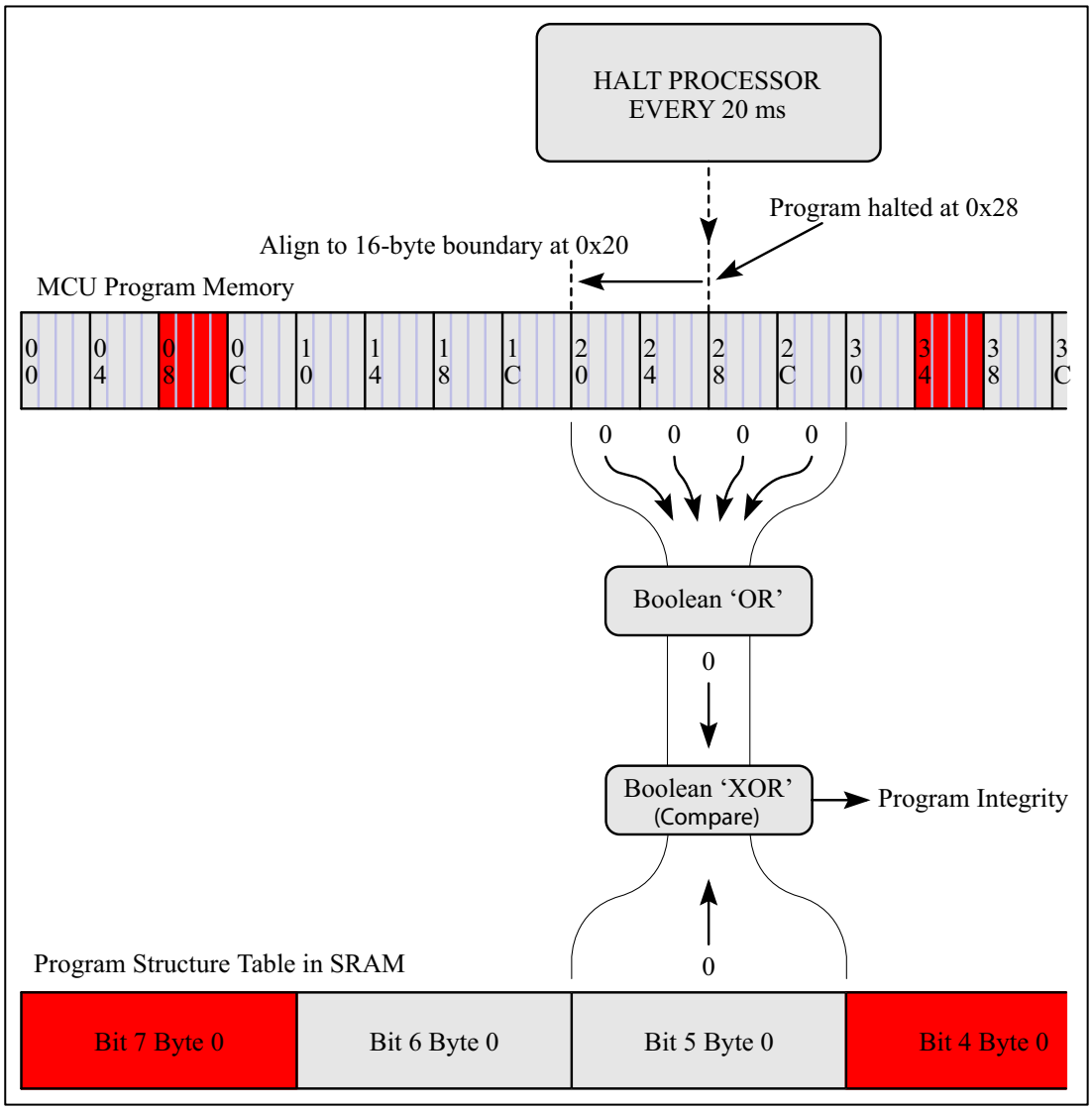


Figure 19: Detailed method to check program code integrity.

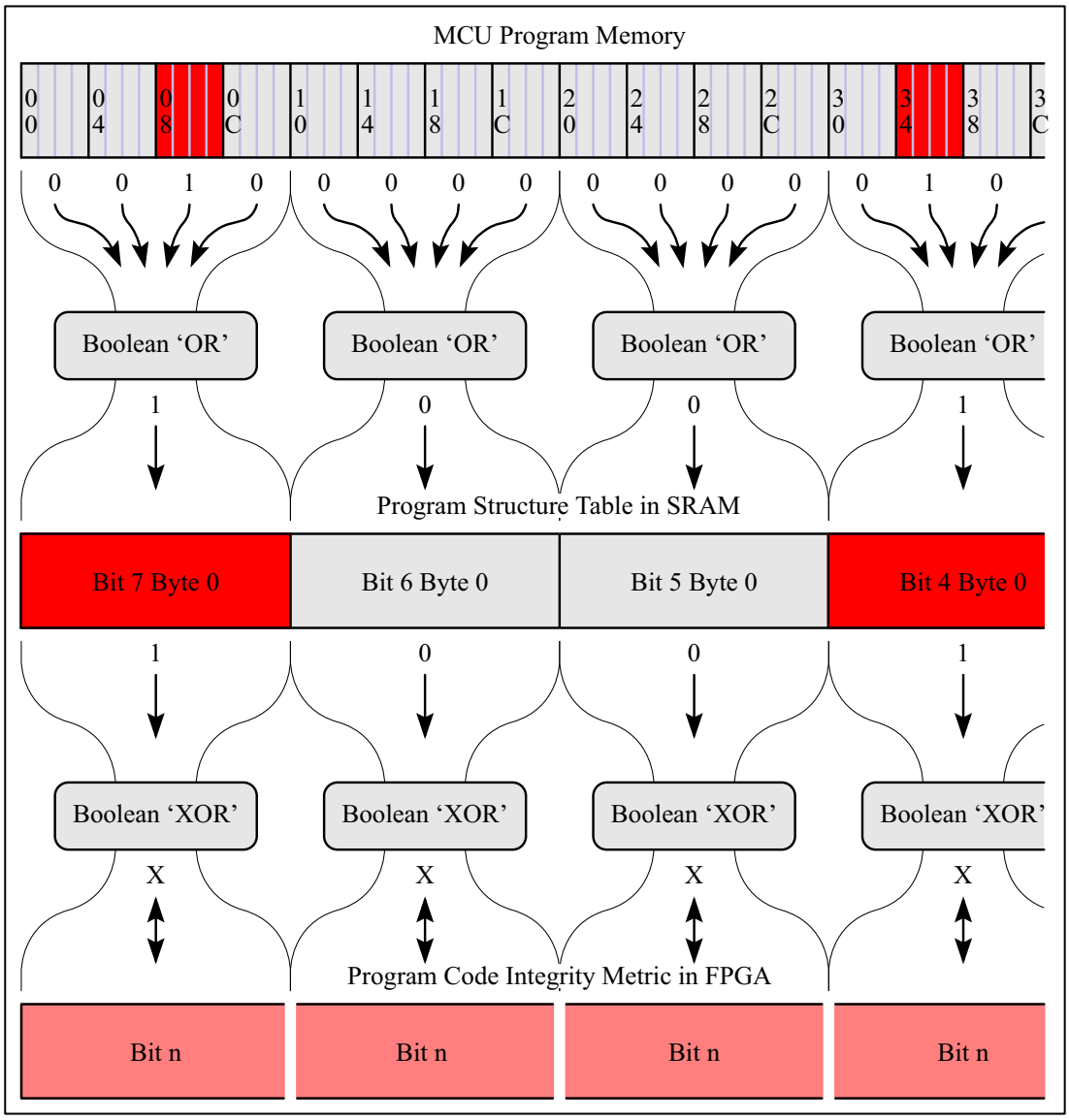
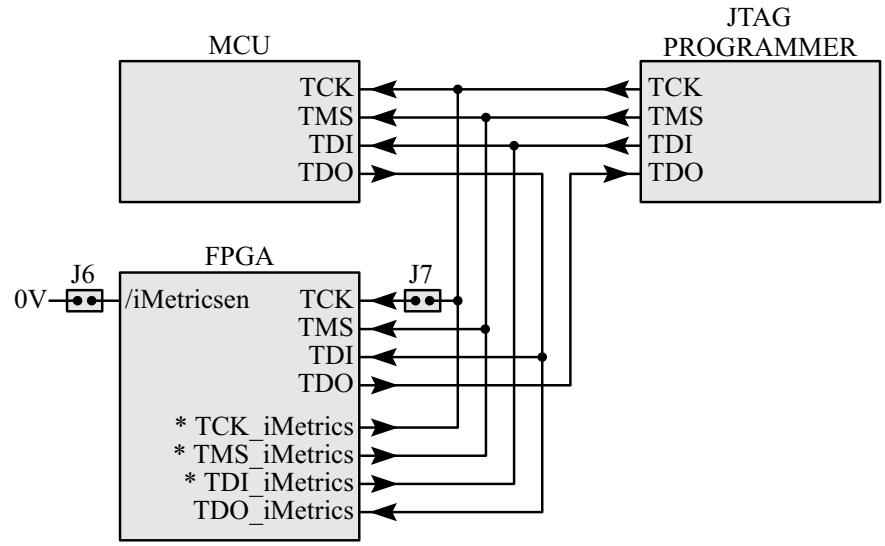


Figure 20: Detailed method to update the Program Structure Table and create the Code Integrity Metric.

6.3 JTAG interface

The JTAG interface to the processor on iCell serves two purposes. The first is a method to debug, reprogram and generally use the interface in for normal purposes. The second is for low intrusive access to the processor that allows dynamic and stable behavioral metrics to be created. Fortunately both are not required simultaneously so an easy method of switching between the modes was required. The method devised can be seen in Fig. 21. In normal JTAG programming mode the single jumper is fitted to J7, thus daisy chaining the JTAG programming through both the processor and FPGA. This way either the processor or the FPGA can be reprogrammed with standard development tools. If the jumper is moved from J7 to

J6, the FPGA takes control and can access the processor. In this mode it is essential that any JTAG programmer interface has been removed.



* Note these 3 FPGA outputs must be programmed to go tristate when /iMetricsen is deasserted. Failure to do this will result in programming lockup, requiring the cutting of PCB tracks.

J6 must be removed when programming and fitted to activate metrics.
J7 must be fitted when programming and removed to allow the metrics system to function.

Figure 21: iCell JTAG interface.

6.4 Physical Design

6.4.1 Circuit Schematics

The physical design of the iCell was dictated by the need to connect several together in a structured array allowing bi-directional communication on all 4 sides. The physical size for these experiments due to the low number of devices would it seem not be particularly important, however future work may well require larger arrays to be constructed, thus a minimal profile is preferred. The 2 main components, both in physical size and functionality are the FPGA and MCU which required a large number of interconnects between them. It was decided that mounting these coaxially on opposite sides of the printed circuit board (PCB) was the best approach to solve this problem and would also minimize the physical profile. A schematic was first devised which incorporated all the desirable sensors and effectors, the JTAG interconnect and inter iCell communication links. The schematics of the MCU section can be seen in Appendices, Fig. 46 and the FPGA, Fig. 47. Fortunately interconnect between the MCU and FPGA offered lots of freedom due to the

reprogrammable nature of the FPGA, thus the schematics were only finalised after the PCB was routed.

An important consideration was power supply voltage and method of distribution across the iSurface. All devices used in the iCell required 3.3V. Diligent design would suggest supplying a somewhat higher voltage across the iSurface and regulate down on each individual iCell to the required 3.3V. However this iCell was constructed at a macro scale purely for practical reasons with components selected that would allow scaling to sub 1 cm^2 size. A smaller scale would preclude the use of an efficient switching regulator and a linear one would cause thermal issues. For this reason the 3.3V supply is routed directly across the iSurface along with the high speed data links. Use of good decoupling and ferrite beads should suffice to provide a reliable power supply.

6.4.2 Printed Circuit Board

The bare PCB can be seen in Fig. 22. The final revision reduced the footprint down to 38mm x 38mm without compromising the desired specification. It will be noted that even with the coaxially mounted MCU and FPGA there is still a good ground plane on the bottom side. Also of note is the use of a reasonable size moving coil loudspeaker (diameter 23mm) mounted on the lower side above the MCU. Although this small size restricts the low frequency response on each individual iCell, larger arrays would enhance performance at lower frequencies with a proportional improvement to the audio output quality. Clearly it would be of benefit to mount the loudspeaker on the top side of the PCB, however that would restrict attachment of test equipment to many of the signal lines. Note that the design does allow for loudspeaker fitment on the top if required.

Although not directly relevant to this research it should be mentioned that Fig. 22 also shows to the right hand side a smaller board that is used for capacitive proximity detection. This sensor relies on capacitance change in a resistor and capacitor (RC) network. This takes advantage of the very high input impedance of the FPGA to measures change in capacitance by timing the charge and discharge times when fed with a slow clock.

6.4.3 Fully Assembled iCell

The completed fully populated iCell can be seen in Fig. 23.

Although only 2 iCells were constructed, Fig. 24 is included to illustrate the way they physically interlock. The example shows what an array of 16 would look like in a basic 4 by 4 configuration. Note that external power and high speed iSurface data would be connected by means of a common multi-connector PCB strip along one of the edges.

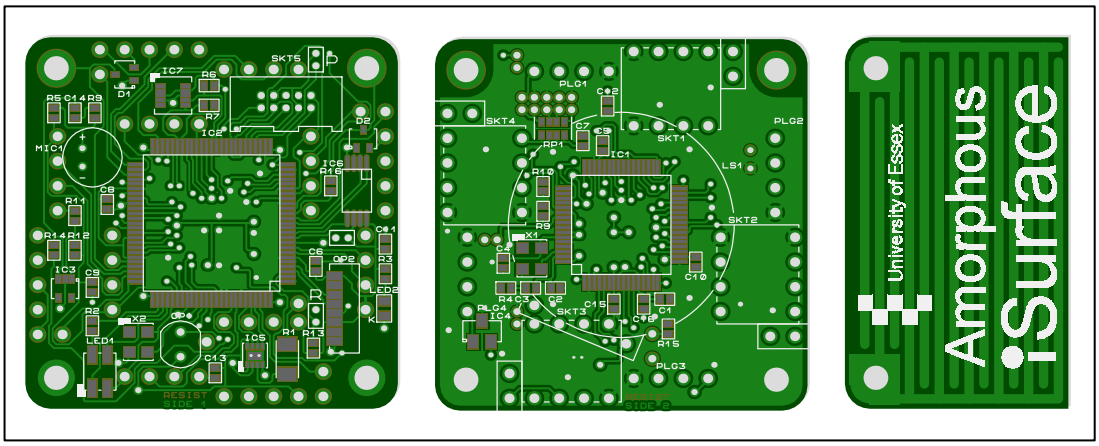


Figure 22: iCell Printed Circuit Board (PCB).

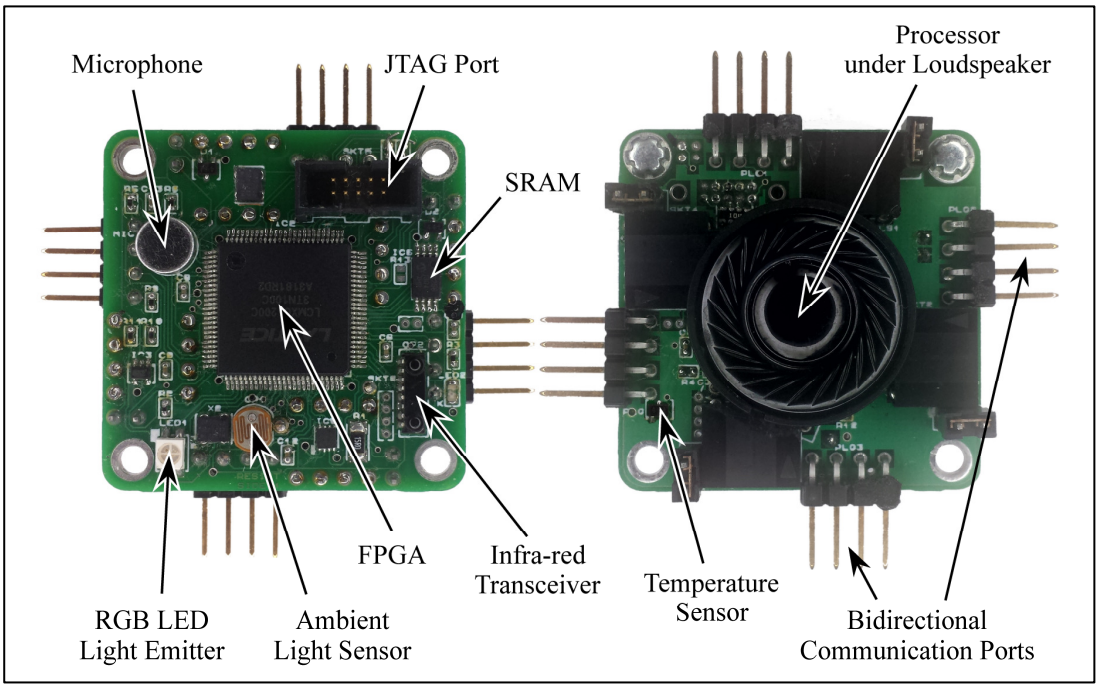


Figure 23: Completed iCell.

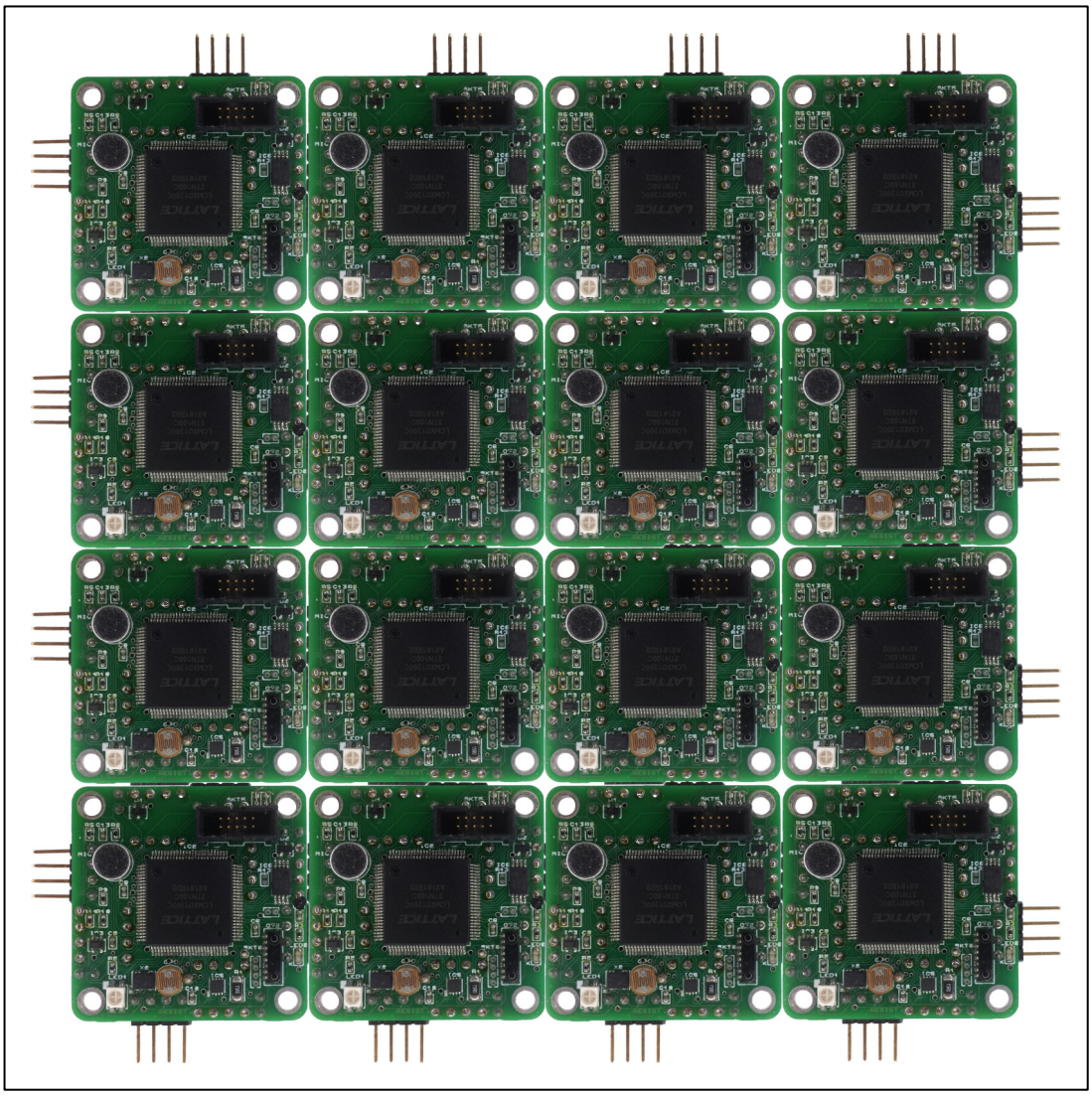


Figure 24: iSurface.

6.5 Summary

With a software based implementation of the metric of code integrity developed and optimized, the hardware design requirements could be finalized. Primary components of the iCell were the MCU, FPGA, SRAM and various sensor/effectors. The MCU provided the functionality of the iCell, whilst the FPGA and SRAM the behavioral metrics. Flexibility on MCU pin usage and functionality was provided by routing the sensor/effector interface lines through the FPGA. The 4 high speed inter iCell communication ports are not required in the current research, however use for FPGA development diagnostics would be possible.

It was decided that the FPGA operation would be based on a hierarchy of state machines, resulting in a complex sequence of logic controlling the MCU by way of the JTAG interface at the lowest levels.

Circuit schematics were drawn up and PCB design files sent off for manufacture.

Chapter 7: Utilizing JTAG to Create Metrics

7.1 JTAG an Overview

A key part of the hardware implementation is the control and access to the MCU by the FPGA using the JTAG interface. Despite extensive searches for previous work in this area, it seems control and profiling analysis by way of dedicated programmable hardware in its entirety is a new concept, so this part of the research proved to be interesting as well as challenging.

7.1.1 The JTAG Serial Bus

The JTAG serial bus or interface is designed both for determining the logic state of pins and for debugging purposes. Debugging involves read/write access to program memory and controlling specially designed debug modules which are usually vendor dependant. The debug module of the AT91SAM7S256 used in the iCell like most allows access to internal registers and memory by way of JTAG loaded machine code instructions placed directly into the core pipeline.

The physical interface is comprised of 5 serial lines, 4 of which are inputs and 1 output. The pins are:

1. TCK (Test Clock)
2. TMS (Test Mode Select)
3. TDI (Test Data In)
4. TDO (Test Data Out)
5. TRST (Test Reset) is optional and is left unconnected in the iCell design.

7.1.2 The TAP State Machine

The AT91SAM7S256 processor uses the ARM7TDMI core and its internal JTAG/debug architecture can be seen in Fig. 25. It will be noticed that both the TCK and TMS signals enter the Test Access Port (TAP) controller. The TAP controller as seen in Fig. 26 is a state machine that uses the TMS signal as the navigation control and TCK as the transition clock. It will be noticed that the controller has both an instruction (IR) and a data register (DR). The instruction register is 4 bits in length and the data register varies dependent on selected use. The instruction register for the ARM7TDMI core has 10 public instructions of which SCAN_N, INTEST and RESTART should be sufficient to implement the required iProfiler functions,

however this isn't certain so for completeness and clarity all instructions can be seen in Table 3.

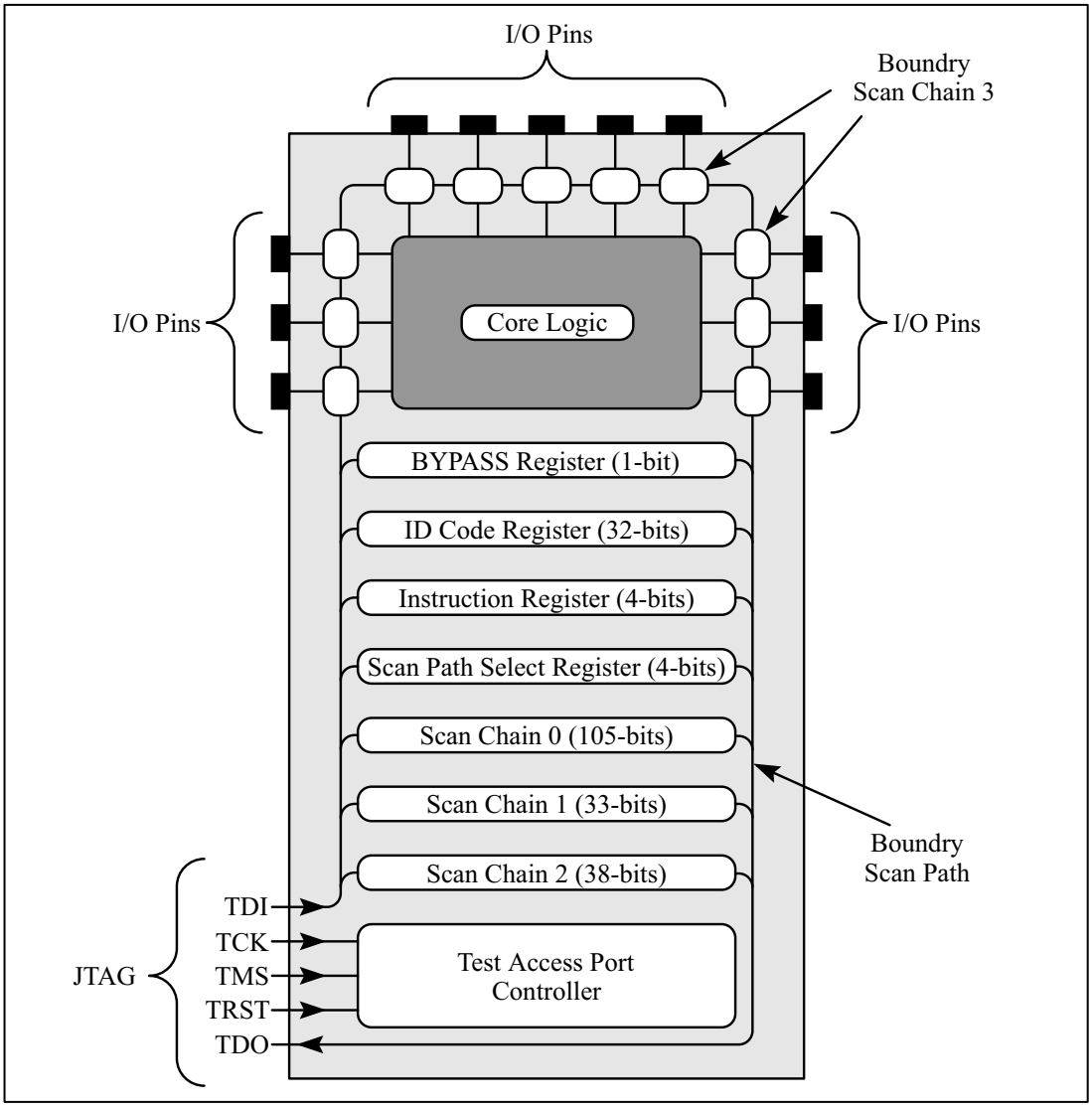


Figure 25: ARM7TDMI JTAG/Debug details.

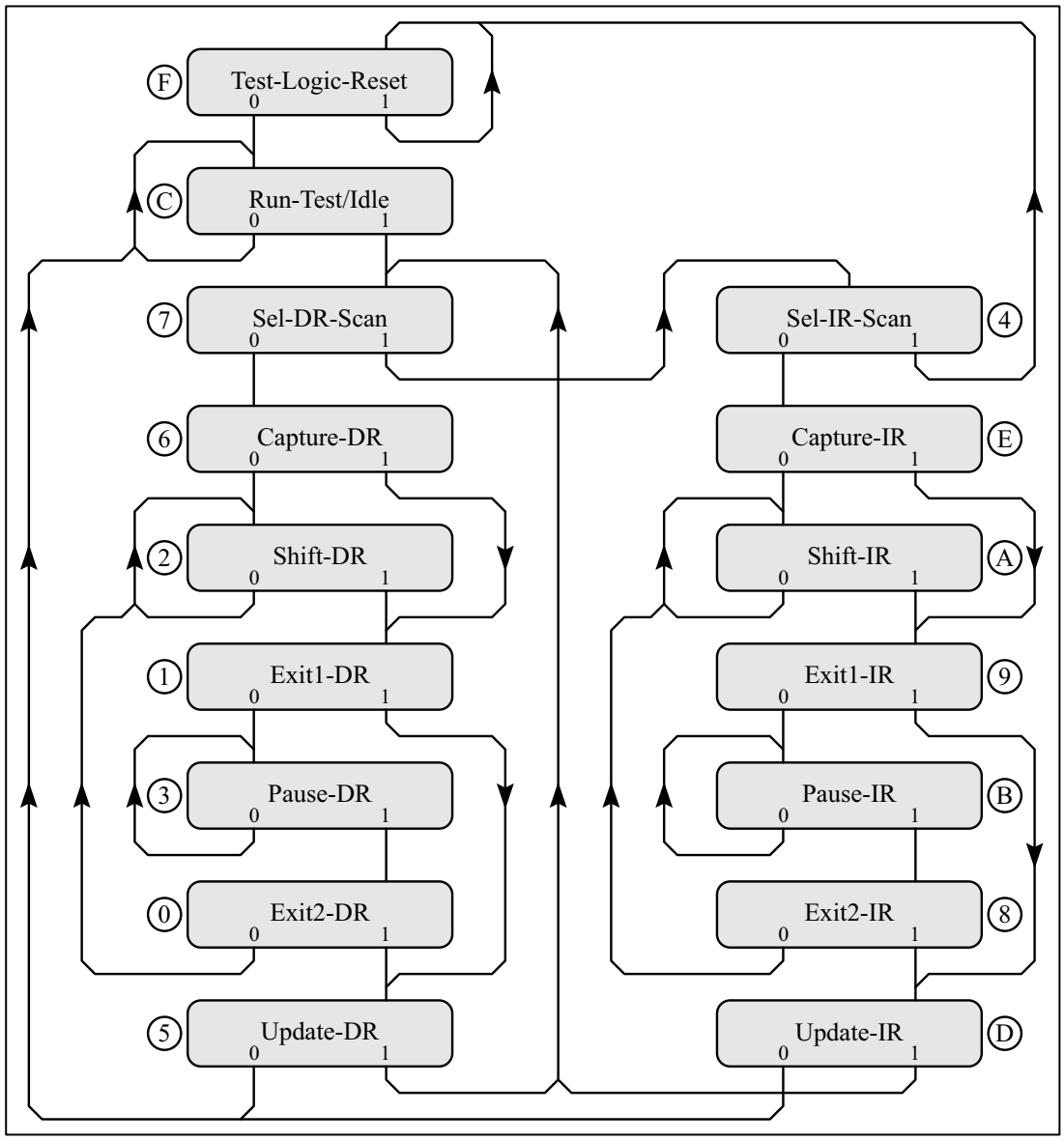


Figure 26: JTAG TAP State machine.

Referring to the TAP state machine shown in Fig. 26, it will be noticed that there are 16 states determined by a 4-bit state counter that is set to hexadecimal 0xF (Test-Logic-Reset) on power-on reset. On each positive going edge of the TCK clock there is a state transition that is dependent on the polarity of TMS (shown at the bottom of each state). For example if TMS stayed high the state would simply loopback and remain in Test-Logic-Reset, however if low it would transition on to state 0xC (Run-Test-Idle).

Table 3: ARM7TDMI Debug Public Instructions.

Instruction	Binary	Hexdecimal
EXTEST	0000	0x0
SCAN_N	0010	0x2
SAMPLE/PRELOAD	0011	0x3
RESTART	0100	0x4
CLAMP	0101	0x5
HIGHZ	0111	0x7
CLAMPZ	1001	0x9
INTEST	1100	0xC
IDCODE	1110	0xE
BYPASS	1111	0xF

7.1.3 Analysis of the Software Based iProfiler JTAG Signals

Although there is sufficient technical information to determine the hierarchy of commands and data, it was decided that the complex nature of the problem could be minimised by a more rigorous analysis of the JTAG transfers between PC and target processor using the software based experimental platform first shown in Fig. 6. The three main JTAG operations required to implement the FPGA based iProfiler are Halt (acquire program counter), read memory and Resume. These operations were initiated by sending commands to OpenOCD by way of a PuTTY telnet terminal. The Halt command halts the processor and returns the complete set of internal registers r0 - r12, sp_svc, lr_svc and pc. These registers values are required to return the processor to its pre-debug state when resuming normal operation since they may well be used or altered whilst in debug mode. The read memory (mdw) command reads a 32-bit value from any location in the address space including SRAM and FLASH. Finally the Resume command restores the registers to their pre-debug state and exits debug mode.

Inspection of the JTAG bus using a 4 channel logic analyser confirmed much of what was gleaned from various sources about the protocol and gave confidence in what was required to develop a working hardware based iProfiler, indeed some sequences of JTAG signals captured by the logic analyzer could form the basis of the initial FPGA design.

7.1.4 TAP and Chain Initialization Preamble

The independence of the FPGA based iProfiler to the target processor was considered important so that serious processor problems such as a system reset or JTAG reconnection whilst 'live' were manageable. To fulfil that aim an initialization preamble was developed that placed the targets TAP controller in a know state at the start of every JTAG sequence of commands. A description of this process here is an ideal opportunity to explain the JTAG communication process at a low level.

Referring to Fig. 27 all 4 JTAG signal lines are shown along with the current state of the TAP controller. It will be noticed that the initial state of the 3 output lines (TCK, TMS and TDI) from the FPGA iProfiler are logic low. The return signal from the target processor happens to be high in this example but depending on the last operation this may not be the condition. The first requirement is to set the TAP controller state machine to a known state. This can be achieved by applying a logic high to the TMS line for at least 5 cycles of the TCK clock. Looking at TAP state machine in Fig. 26 it can be seen that not matter what initial state was present after 5 clock cycles with the TMS line high ensures a return to state 0xF (Test-Logic-Reset). Next the correct scan chain must be selected. As noted in Fig. 25 the processors ARM7TDMI core has 3 scan chains.

Scan chain 0 is 105 bits in length and provides serial access to both the data and address buses. Scan chain 1 is 33 bits in length and is a subset of scan chain 0, providing access to the to the data bus (D[31:0]) and the breakpoint bit (BREAKPT). This can be used to place and execute a processor instruction onto the data bus. Scan chain 2 is 38 bits in length and provides access to the embedded in circuit emulation (EmbeddedICE) macrocell registers. The scan chain places 32-bit values into a set of watchdog registers and comparators to initiate a breakpoint condition. The watchdog register and comparator select address is 5 bits in length plus a read/write bit.

The first action of the iProfiler is to the halt the processor; therefore scan chain 2 needs to be selected. It will be noticed that TMS is used to transition the TAP state machine from 0xF to 0xA which allows loading of the SCAN_N instruction into the 4-bit TAP instruction register. Note the data is mirrored (big-endian) due to the least significant bit being loaded first. Once loaded the state machine is transitioned to 0x2 which allows loading of the scan chain number, in this case 0x2 (big-endian).

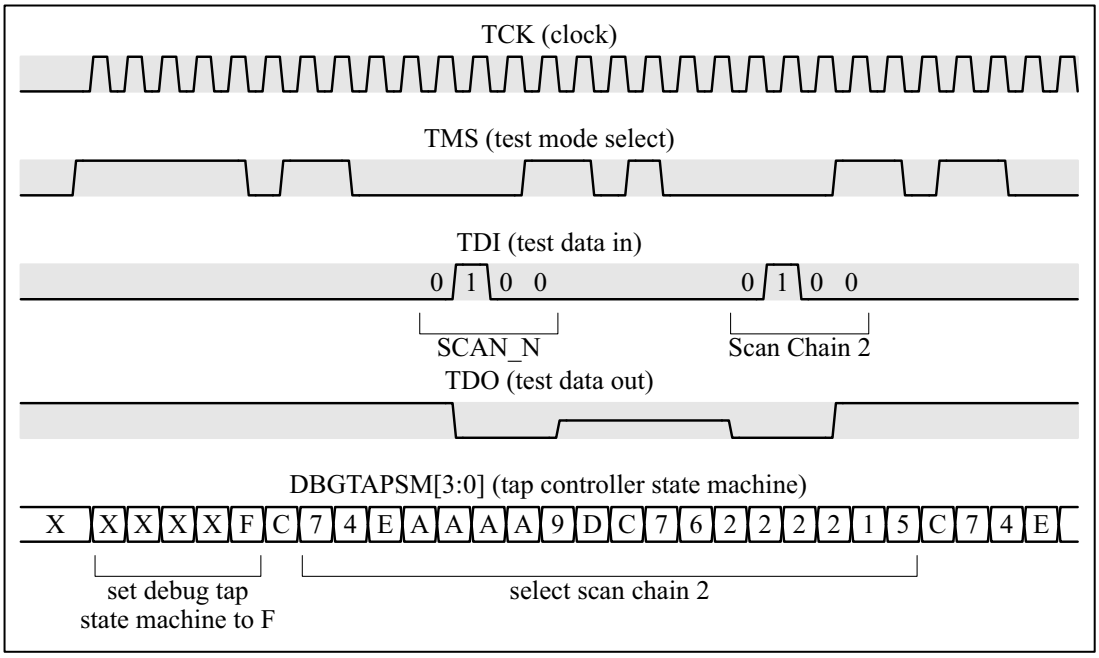


Figure 27: JTAG Communication.

7.1.5 Robust Detection of a Microcontroller on the JTAG Interface

Determination of an active (powered) microcontroller present on the JTAG interface can be achieved be monitoring the TDO JTAG serial line during the TAP and Chain initialization preamble procedure. Referring again to Fig. 27 it will be noticed that the TDO line is logic high up until the point that the FPGA drives the TDI high, at which point the MCU pulls the TDO low. Without a processor connected or a problem with communication due to other hardware faults the TDO would not react in this way. This test during JTAG initialization was used to provide a reliable method to either carry on the JTAG sequence or abort and reattempt at the next sample (20 ms later). This proved to be an incredibly reliable method of determining the status of the JTAG communication during the critical initialization without adding any communication overheads, moreover it crucially does nothing more than attempt to set up the TAP and Chain so any break in communication at this point will be fully recoverable on the next sample.

7.2 Implementing a JTAG Controller in an FPGA

By far the most complex part of the FPGA iProfiler is the JTAG controller. As mentioned earlier in Section 6.2.2 it was decided that the best design strategy was a hierarchy of state machines. Although the top level state machine was already thought out and a top-down design path may seem reasonable, no functionality

would be possible without some sort of real access to the JTAG. Therefore a bottom-up path was chosen. This would allow test JTAG sequences to be developed and checked for accuracy using a logic analyser.

The first important decision was the JTAG clock speed. The earlier software based experiments using OpenOCD had a rather sedate speed of 1 MHz and even then the host PC could not keep up and spurious gaps were apparent in the command and data transfers. Information on the maximum debug clock speed for the ARM7TDMI core was difficult to locate and sometimes contradictory. An expanded and updated version of the AT91SAM7S range of devices was located which gave a maximum TCK speed of 9.8 MHz [30], however specifications found in the Atmel AT91SAM-ICE user guide indicated a maximum emulation speed of 12 MHz. Since this is the official USB JTAG controller from Atmel it was decided that the TCK clock speed for the iProfiler would also be 12 MHz with the option of falling back to 10 MHz should there be reliability problems. The FPGA itself has a master clock that needs to run at least twice that speed to raise and lower the TCK clock at 12 MHz. The diagram, Fig. 28 illustrates critical timing that must be adhered to for reliable JTAG operation. It will be noticed data both from the processor and FPGA changes on the rising edge of the 12 MHz TCK clock and is read on the falling edge. This timing would indicate that an operational 24 MHz clock to the FPGA would suffice, however access to the SRAM based Program Structure Table may require more operations per TCK clock so a 48 MHz clock was initially chosen. The TCK falling to TDO valid (Tbsod) timing shown in Fig. 28 was also not located in official documents, however analysis of the signals whilst using OpenOCD gave a figure of 14 ns, well within the planned sample time of 2 FPGA clock cycles (41.66 ns).

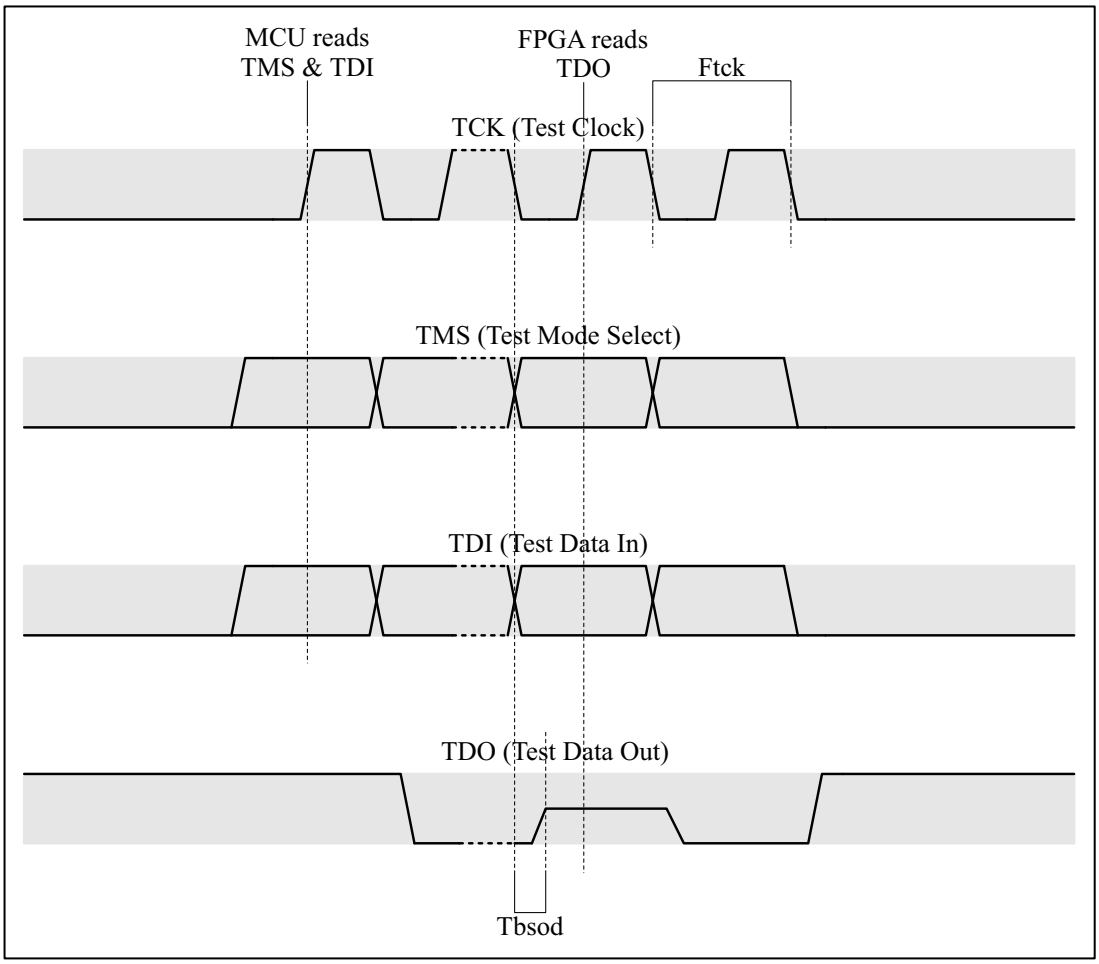


Figure 28: Relevant JTAG data transfer timing.

7.2.1 JTAG Repeated Sequences

When observing the JTAG control signals whilst issuing OpenOCD commands it became apparent that there were many repeated sequences and the idea of coding a state machine at the lowest level to generate those sequences seemed a way a reducing the logistics to manageable level. For example each access the software based iProfiler made to the processor every 20 ms required 3,791 TCK clock cycles with complex defined states for both the TMS and TDI signals. Since the sequences created by the software based iProfiler had more functionality than required it was decided that the entire Halt, Read memory and Resume sequences should be documented in tabular form. This process was completed and a short program was written that could locate and extract repeated instruction sequences. There were 2 main types of sequences. The first and usually smaller sequences controlled the TAP state machine and the others were mostly 32-bit ARM machine instructions to be executed by the processor. It proved convenient that when a 32-bit limit was set to sequence length, the number of unique sequences was also slightly lower that 32.

This meant that a 5-bit binary counter running at 12 MHz could serialize the sequences being sent out of the FPGA and the sequence selector need only be 5-bits as well. Appendices Table 7 lists all sequences required and Appendices Table 8 to Table 10 list the order those sequences are selected to implement the iProfiler's JTAG control system.

Each sequence controls the iProfiler's JTAG TMS and TDO output pins and must be hard coded into the FPGA. This presented a problem since the preferred programming language PALASM had no support for data arrays or any other structures. However it proved fairly straightforward to write a small program to import the sequence data and generate PALASM equations that could be imported to the design files. A significant benefit of coding the sequences into equation code rather than using fixed arrays in FPGA FLASH was the large reduction of logic requirements due to the compilers efficient logic optimisation and minimization.

7.2.2 Halt and Read MCU Registers Operation

The first operation to be implemented in the FPGA serves albeit with minor modifications much the same function to the software based OpenOCD Halt command. The first deviation from the OpenOCD implementation is the addition of the preamble discussed in Section 7.1.4.

A fundamental requirement with OpenOCD and the iProfiler is the restoration of all processor program registers when issuing the Resume command. In the case of OpenOCD this is accomplished by having the Halt command provide a complete processor register dump that are saved so the Resume command can restore them and allow the processor to continue seamlessly from where normal operation was halted. Whilst OpenOCD has to cope with the possibility of any or all registers being modified during debug mode that is not the case with the iProfiler which has a defined operation that requires the use of only the program counter, registers r0 and r1. This allows a simplified Halt sequence which can be considerably shorter than the OpenOCD implementation.

For clarity the sequences were grouped and referred to as iProfiler instructions, the order of which to halt the processor can be seen in Table 4. It will be noticed the table also includes the clock cycles for each instruction. With the total cycle count being 1,190 and each clock cycle at 12 MHz having a period of 83.33 ns results in a total period of 99.166 µs. It will be noticed that the rather curious processor

instruction MOV r8,r8 is to be seen. The ARM processor is a Reduced Instruction Set Computing (RISC) processor and saves an opcode by making use of the MOV instruction to perform a No Operation (NOP) by moving a register to itself, in this case r8, but any register could be used for this purpose. No Operations in this case are required to push instructions into the ARM7TDMI 3-stage pipeline.

Fig. 29 shows a timeline of important events within the 99 µs period. Note that JTAG controller operations are shown in blue. The first operation highlighted is the initialization of the SRAM into sequential access mode. This does not play a part in halting the processor, but is a separate initialization done in parallel and shown due to being within the iProfiler's halt timeframe. Next is the reading of the processor registers r0 and r1, followed by the program counter. Note that due to pipelining the program counter value is 24 bytes ahead of the instruction currently being processed. When the processor is instructed to resume operation the pipelining look-ahead buffer is cleared and the program counter address must reflect this, which means 24 must be subtracted before it is restored. This adjusted value is also used by the iProfiler to search for branches in the quest to falsify code integrity. This adjustment is highlighted in green and like the SRAM initialization is also performed in parallel.

Table 4: iProfiler Halt detailed order of instructions.

Clock Cycles @ 12 MHz	JTAG Instructions
7	Set TAP State C
32	Scan Chain 2, INTEST
43	Write 0xFFFFFFFF to Watchpoint 0 Address Mask
43	Write 0xFFFFFFFF to Watchpoint 0 Data Mask
43	Write 0x00000100 to Watchpoint 0 Control Value
43	Write 0x000000F7 to Watchpoint 0 Control Mask
43	Read Debug Status
45	Read Debug Comms Control Register
43	Write 0x00000005 to Debug Control
39	Write 0x00000000 to Watchpoint 0 Control Value
38	Scan Chain 1, INTEST
43	STM r0, r1, sp_svc, lr_svc, pc
43	MOV r8 r8
43	MOV r8 r8
43	Return r0
43	Return r1
43	Return spsr
43	Return spsr
43	Return pc + 24
43	MRS Rd CPSR
43	STR
43	MOV r8 r8
43	MOV r8 r8
43	Return spsr
43	MRS Rd SPSR
43	STR
43	MOV r8 r8
43	MOV r8 r8
40	Return spsr

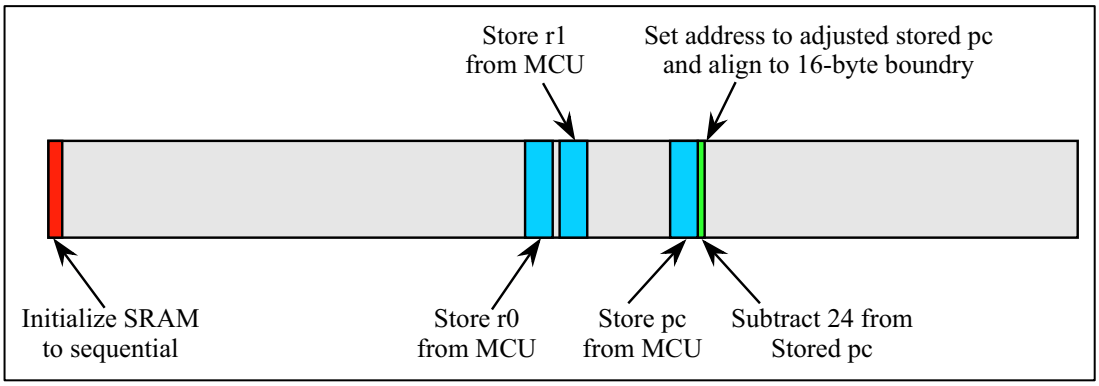


Figure 29: Halt operation timeline.

7.2.3 Read MCU Memory Operation

The iProfiler instructions to perform a processor memory read can be seen in Table 5. Of note is the 'Set Address' and 'Return Data' where a 32-bit memory address is sent and the 32-bit read data is returned. The entire operation has a total cycle count of 895 which results in a total period of 74.5833 µs.

Fig. 30 and Fig. 31 both illustrate the timeline of the iProfiler memory read and various other operations performed in parallel. The memory read operation highlighted in blue is identical in both instances and as mentioned in the diagram is repeated 4 times to determined whether a branch instruction is present in within a 16-byte block.

7.2.3.1 State '9' Parallel Operations

The secondary parallel operations shown in Fig. 30 are performed when the high level code integrity state machine seen in Fig. 18 is at state '9'. The first is the clearing of the single bit 'Bra' flag. As noted on the diagram this is only done once at start of the 4 repeated memory reads.

Next, highlighted in red is the reading the Program Structure Table held in the serial SRAM ready for comparison in state 'D' of the code integrity state machine. Immediately after the return from the processor of the 32-bit data from the memory read, a comparison is made to determine the presence of a branch instruction which if present will cause the 'Bra' flag to be set.

Finally the memory read address in incremented by 4, ready for the next read operation.

Table 5: iProfiler Read Memory order of instructions.

Clock Cycles @ 12 MHz	JTAG Instructions
46	STM r0
43	MOV r8 r8
43	MOV r8 r8
43	Set Address
43	MOV r8 r8
43	MOV r8 r8
43	MOV r8 r8
43	LDM r1
42	Scan Chain 2, INTEST
43	Read Debug Status
41	Read Debug Comms Control Register
38	Scan Chain 1, INTEST
43	STM r1
43	MOV r8 r8
43	MOV r8 r8
43	Return Data
43	MRS Rd CPSR
43	STR
43	MOV r8 r8
43	MOV r8 r8
40	Return spsr

7.2.3.2 State '1' Parallel Operations

This memory read operation is performed whilst reading the entire processor memory for the purpose of Rebuilding the Program Structure Table and creating a new Code Integrity Metric. Apart from reading of processor program memory the other parallel operations involve SRAM write access. As with the parallel operations described for state '9', the 'Bra' flag is again cleared at the start and set if a branch is detected. Referring to Fig. 31, it can be seen highlighted in pink that the SRAM is placed into write mode. This is actually initiated on exit from state '3' but executes in state '1'. The SRAM's serial data input (SI) is then set to reflect the state of the 'Bra' flag ready to be written into the SRAM which is then completed by the strobing of the SRAM clock pin with a high going single pulse. Note that as shown in the diagram, the SRAM write only occurs at the end of the 4 repeated operations.

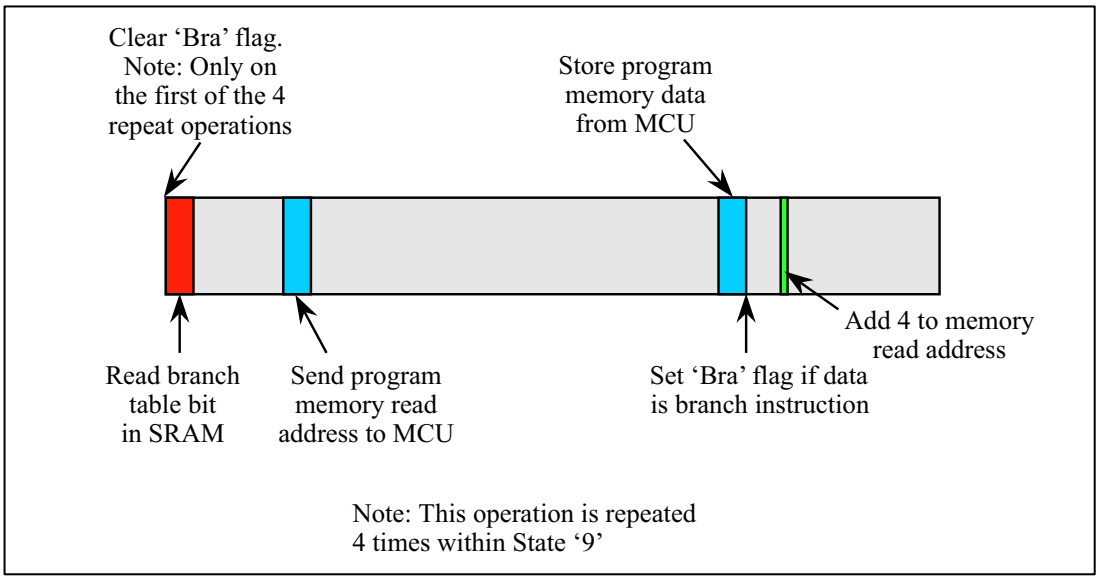


Figure 30: Read MCU and read SRAM operation timeline.

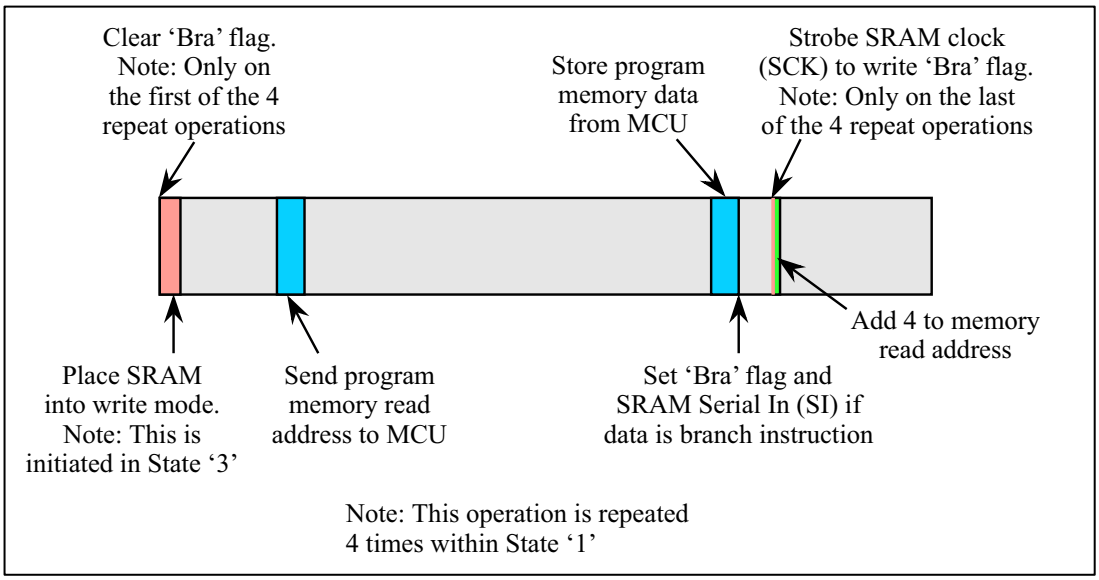


Figure 31: Read MCU and write SRAM operation timeline.

7.2.4 Restore MCU Registers and Resume Operation

The last iProfiler instructions take the target processor out of debug mode and resumes normal operation. The instructions can be seen in Table 5 and the timeline in Fig. 32. The iProfiler resume restores registers r0 and r1, both of which were used in the memory read operation. Also restored is the already adjusted program counter. The final debug command 'RESTART' returns the processor to normal operation. The entire cycle count for resume operation is 764 which results in a total period of 63.66 µs.

Table 6: iProfiler Resume order of instructions.

Clock Cycles @ 12 MHz	JTAG Instructions
46	STM r0 r1
43	MOV r8 r8
43	MOV r8 r8
43	Restore r0
43	Restore r1
43	MOV r8 r8
43	STM pc
43	MOV r8 r8
43	MOV r8 r8
43	Restore pc
43	MOV r8 r8
43	MOV r8 r8
43	MOV r8 r8
43	MOV r8 r8
43	MOV r8 r8
43	B
32	Scan Chain 2, INTEST
40	Write 0x00000000 to Debug Control
10	RESTART

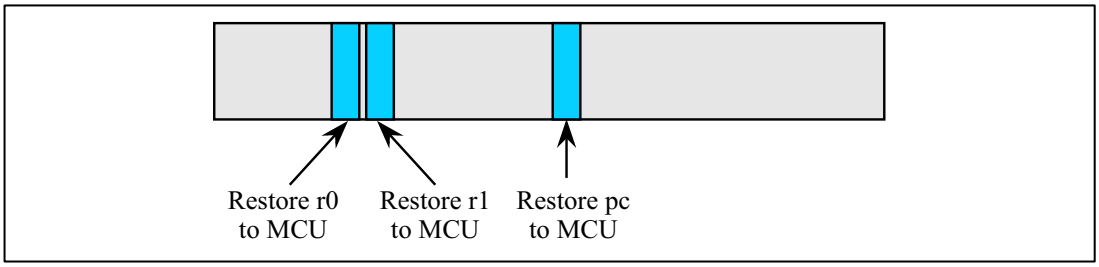


Figure 32: Resume operation timeline.

7.2.5 Complete JTAG Instruction Sequence

In normal operation the iProfiler would on boot up determine very quickly a mismatch between the Program Structure Table and the program structure found in the target processor. This would result in a program memory scan to rebuild the Program Structure Table and create a new Code Integrity Metric. Once settled all three iProfiler operations (Halt, Memory Read and Resume) would be executed sequentially every 20 ms. The total clock cycles would be Halt cycles (1,190) + Read cycles (895 * 4 * n) + Resume cycles (773), where ‘n’ is the average number of read cycles required to locate a branch within program memory and verify program structure. Referring back to section 5.3, a typical value of n with the

optimal 1 bit per 16 bytes structure table is around 1.6 which results in a total clock cycle count of 7691. With a continuous JTAG TCK clock frequency of 12 MHz (83.33 ns), this results in a total iProfiler period of 0.641 ms every 20 ms or 3.3204% of the target processors average runtime. Note the hardware iProfiler has different modes of operation including an option that fixes the value of n to 1 (one read cycle). Although operating in such a single read cycle mode would certainly take more samples to identify a potential problem with code integrity, it does have the advantage of a fixed penalty on processor performance. This mode of operation and others will be investigated and discussed later in this thesis.

The design of the iProfiler's JTAG controller was a significantly more complex problem than first imagined and utilized most of the LCMXO1200 FPGA's logic resources. For this reason the device was removed and upgraded to the larger, pin compatible LCMXO2280. Final utilization for this upgraded device used 1,893 of the available 2,280 look up tables (LUTs), leaving 17% free space for future development and work.

7.3 Summary

The first and most difficult requirement to implement the metric of code integrity in hardware was a full understanding of the JTAG low level communication protocol. Due to lack of comprehensive information and in particular useful documentation on the ARM7TDMI debug interface, it was decided that a degree of reverse engineering using a logic analyzer would be the best approach. Using the information gleaned from this signal analysis coupled with the available documentation resulted in an optimized method to perform the all the required functions needed to create the iMetrics in hardware. These functions were: halting of the MCU and retrieving the program counter address, reading data from program memory space, then finally resuming normal MCU operation.

Chapter 8: SRAM Program Structure Table and I^2C Communications

8.1 Using Serial SRAM to Implement the Program Structure Table

The selection process for the 64 Kbit 23K640 Serial SRAM was discussed in Section 6.1.2 and here we take the opportunity to look at how it functions as the Program Structure Table in more detail.

8.1.1 SRAM Transfer Timing

Fig. 33 illustrates the relevant transfer timing when using a serial clock (SCK) with a 50:50 mark/space ratio.

To ease the logic design process it would be useful if the device could operate at the same clock rate as the JTAG, detailed in the previous chapter. In fact referring to both transfer diagrams it will be seen that the clocking and relative transfer sequences are almost identical. The chosen memory chip has a maximum clock speed rating of 20 MHz, so 12 MHz operation is well within its limits. As with the JTAG, data is read on the rising clock edge and data out made valid on the falling, thus the FPGA's 48 MHz clock is more than adequate to implement the SRAM control interface. The only other possible timing issue is output valid from clock low, indicated in the diagram as 'Tv' which in the case of the selected chip is 32 ns. With the FPGA running at 12 MHz the worst case period between the falling (request data) and rising edge (read data) will be 41.66 ns, resulting in a good 9.66 ns safety margin.

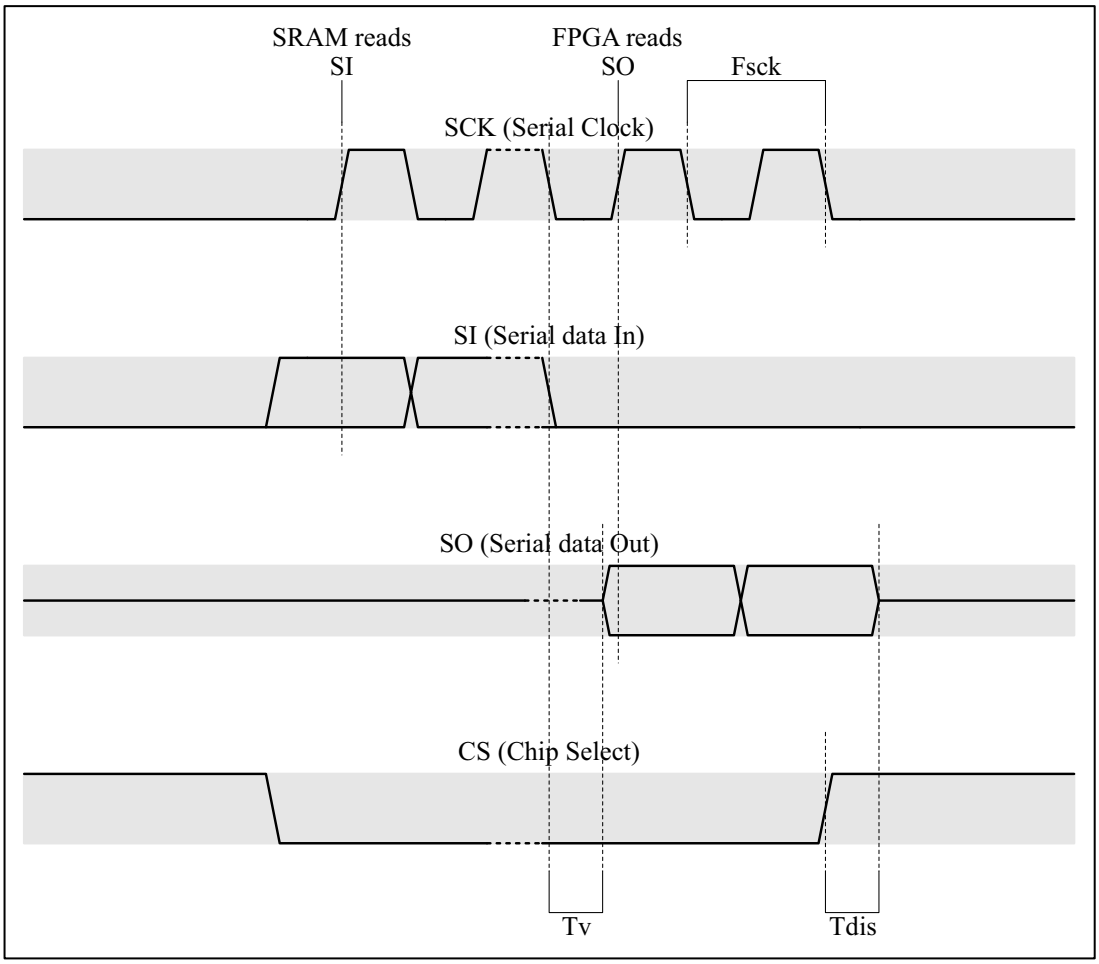


Figure 33: Relevant SRAM data transfer timing.

8.1.2 SRAM Initialization

The serial SRAM as used in the iProfiler requires well defined modes of read/write access. The device is only written to when the Program Structure Table is being rebuilt. In this mode, data is written sequentially through the entire process without interruption. Fortunately the chip has a sequential mode of operation that only requires the data being set on the serial data input pin followed by a strobing of the clock pin. This however is not the default mode when powered up, so initialization is needed.

Fortunately initialization in the selected device is fairly straightforward with the sequence needed to achieve this illustrated in Fig. 34. When the chip select (CS) goes active low the SRAM expects an 8-bit instruction. The instruction ‘Write STATUS register’ is sent, followed by the 0x81 which sets the device into sequential mode. Finally the chip select is deselected.

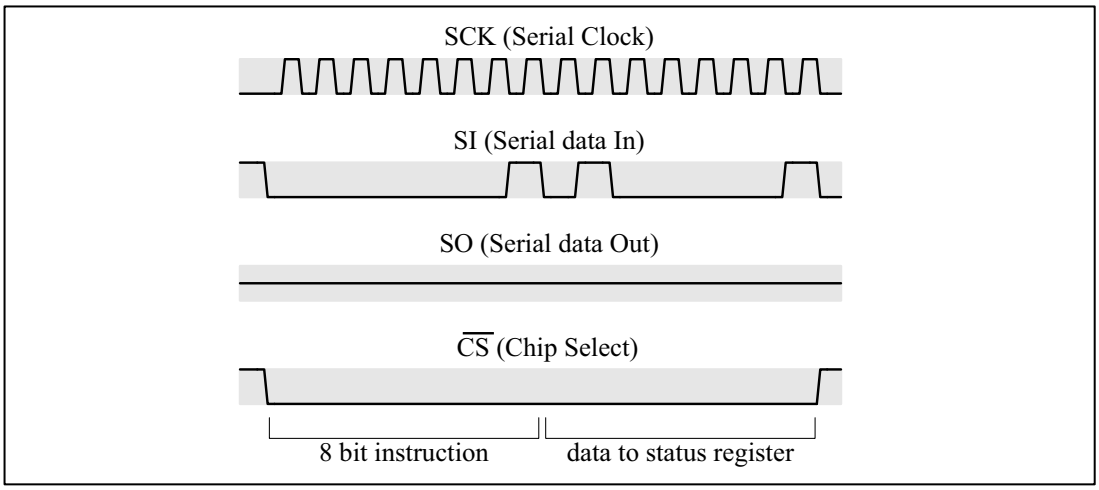


Figure 34: SRAM Sequential mode operation.

8.1.3 SRAM Write Access Operation

As described in the previous section, the SRAM writing requirements of the hardware based iProfiler are quite straightforward in that only a simple clock pulse is required for every bit written into memory. However although the device is initialized into sequential mode, it still requires an instruction to start the process.

This procedure can be seen in Fig. 35. Again the chip select is made active and the 8-bit instruction to initiate a serial write is clocked in followed by the 16-bit start address, which in this case is zero. The SRAM is now set up to allow sequential writes, which can continue indefinitely whilst the chip select is held low.

The subsequent writing operation is illustrated in Fig. 36, where it can be seen that the data on the serial input (SI) is made valid immediately after the branch detection which is followed by a single high going clock 3.54166 µs later. This continues until the Program Structure Table has been rebuilt at which point the chip select is raised to terminate the sequential write operation.

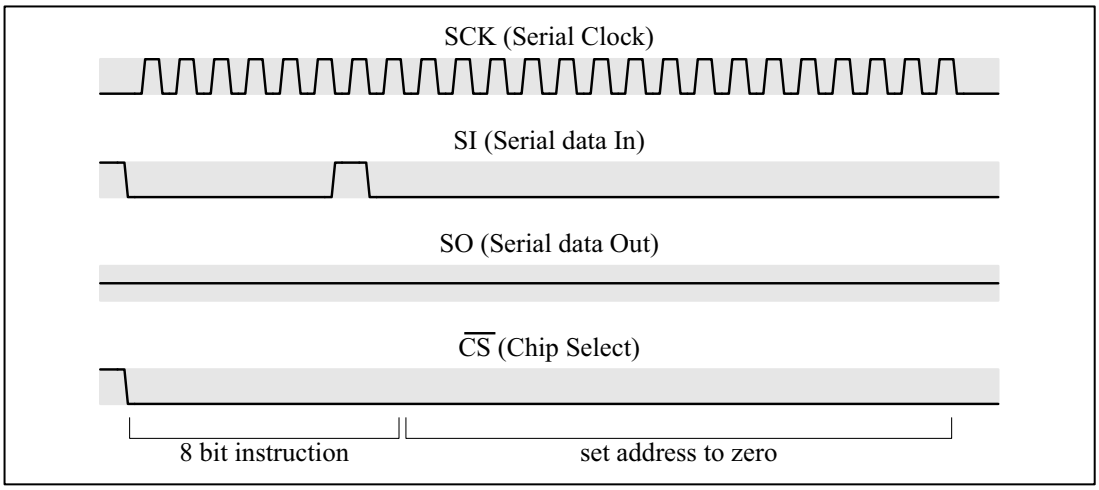


Figure 35: SRAM Write mode operation.

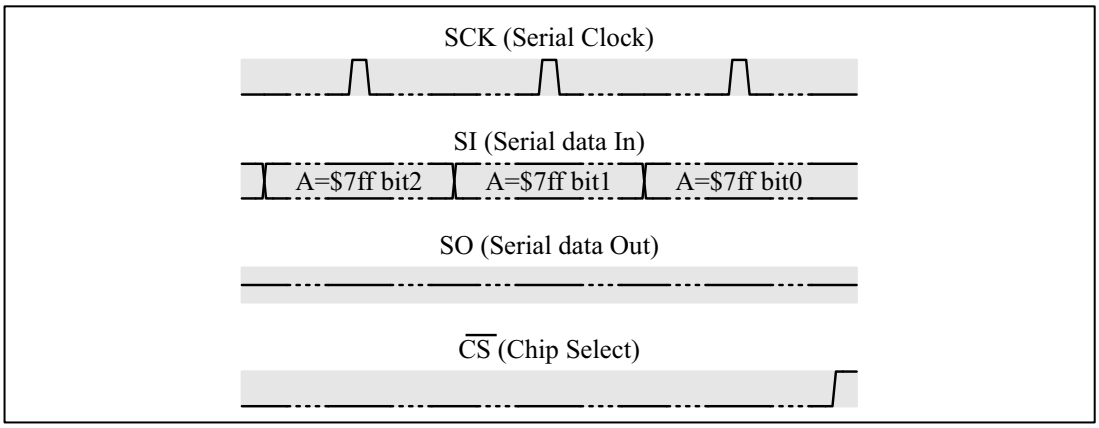


Figure 36: SRAM Write bit operation.

8.1.4 SRAM Read Access Operation

Whilst single bits can be written into the SRAM, the situation is slightly different when reading. Although each read requires only 1 bit to determine program code integrity, it will be noticed in Fig. 37 that a whole byte must be read out. First the byte read instruction is sent (0x03) followed by the address location. The 8-bit data is then read out and the lower 3 bits of the Program Structure Table selects the required bit to cross check with program structure found in processor memory.

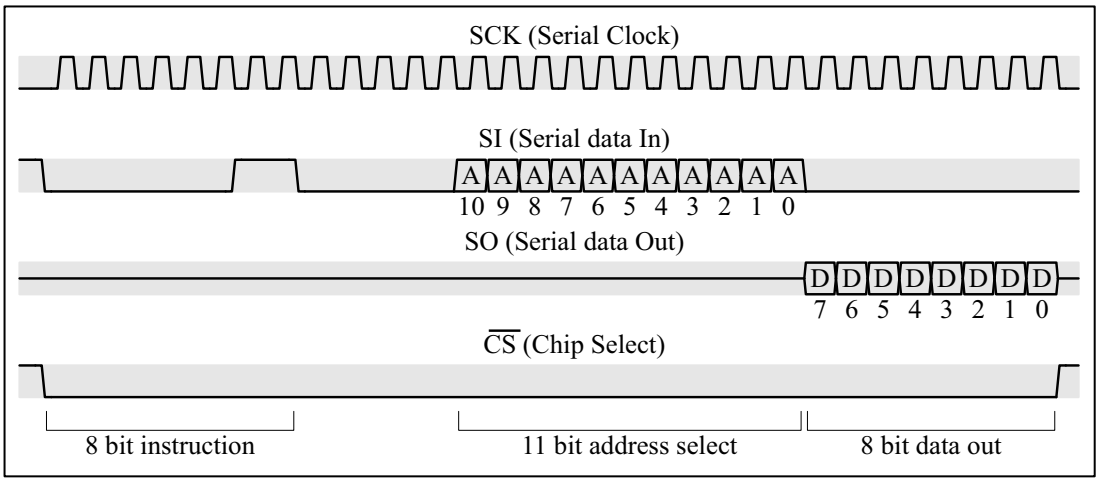


Figure 37: SRAM Read byte operation.

8.2 Reading Metrics

8.2.1 Interface Options

The intended use of behavioral metrics in amorphous computing is manyfold and could well spawn new avenues of research. However the hardware aims of the current work require only the creation of the Code Integrity Metric for performance comparison with the initial software implementation. Having said that, it would be nice to implement an interface that allowed control over the metric parameters and could allow the iCell processor to access it own metrics. It would also be preferable that the interface was not obvious should the iCell software not require access. Note the metrics may well stay in the domain of the FPGA in some implementations, since inter-iCell communication links are also at that level.

There are three common interfaces that could be used to implement such an interface. The first is serial RS232 (Recommended Standard-232 [40]. This is a rather simple standard with a long history dating back 1962. Most microcomputers support it and although PC's don't usually have a built in port, USB adapters are available. The main problem with RS232 is synchronization and quite often buffer requirements which make implementation in an FPGA troublesome. Also it would rob the processor of a potentially valuable communication port, since only one device can be connected at a time. The next possibility is SPI [38], a form of which is used by the Program Structure Table SRAM. This serial interface has the advantage of supporting several devices, thus does not place limitations on the processor. However SPI devices are daisy chained, and any data has to traverse and pass though all devices, requiring software modification to access other devices even

if metric data is not required at the processor level. Lastly there is the I^2C [38] serial bus option. This bus like SPI is serial, but differs in that devices are connected in parallel rather than in series. Individual device selection is accomplished by giving each device type a unique identifier in the form of a 7-bit address. For control and access to the iProfiler's metrics this interface type seems ideal due to its transparency when connected, assuming device address clash is avoided.

8.2.2 I^2C Metrics Interface

Fortunately the I^2C protocol allows flexibility on data length which meant no compromises were necessary on the rather straight forward interface. The need to modify or control the creation of the Code Integrity Metric is entirely optional and depending on application, parameters could be hard coded into the FPGA or SOC implementation. However various modes of operation are required to complete a comprehensive analysis and performance comparisons with the software implementation which will require a minimal number of control settings. The first of these would be the setting of the metric length. The ability to initiate a metric update on demand would also be useful as a diagnostic and development aid.

The control registers functionality could be accommodated in less than 8 bits thus a simple 8-bit I^2C write cycle would suffice with bits [2:0] selecting the metric length and bit 3 initiating a metric update cycle when high. The 8 possible metric lengths are: 2,4,8,16,32,64,128 and 24. Note that the 24-bit selection was only implemented for purposes of comparison with the earlier software implementation.

The I^2C multiple-byte read is perfectly suited to read back variable length metrics.

It should be noted that the LCMXO2280 FPGA has no native I^2C as some other chips do, requiring additional work to design the interface in its entirety. This however has the advantage of a no-compromise solution to the interface.

8.2.3 I^2C Transfer Timing

The relevant timing can be seen in Fig. 38. Although more complex than the JTAG or SRAM interfaces the bus is quite slow and raises few problems in terms of critical timing. The I^2C serial bus has a clock (SCL) and data wire (SDA). The clock frequency has a standard operational mode of 100 kHz and a fast mode of 400 kHz. The specification [41] was enhanced in 1998 to allow bit rates up to 3.4 Mbits/s, however few if any microprocessors support such speeds so they can be ignored.

The clock timing period (Fscl) at 400 kHz would be 2.5 µs or 120 FPGA clock cycles at 48 MHz. Although clock speed isn't an issue, the I^2C bus has some idiosyncrasies that need to be considered. It will be noticed in Fig. 38, that data changes only whilst the clock is low (ignoring start/stop conditions). This is to allow data to be read by master or slave while clock is high. Clearly synchronization is important and clock edges are normally used to capture data, but here the original specification was unclear. Previous experience has revealed that most implementations tend to clock data on the rising edge and change on the falling. More importantly this mode of operation does not violate the timing diagram, where Tsu;dat must be greater than 10 ns and Thd;dat can be 0 ns.

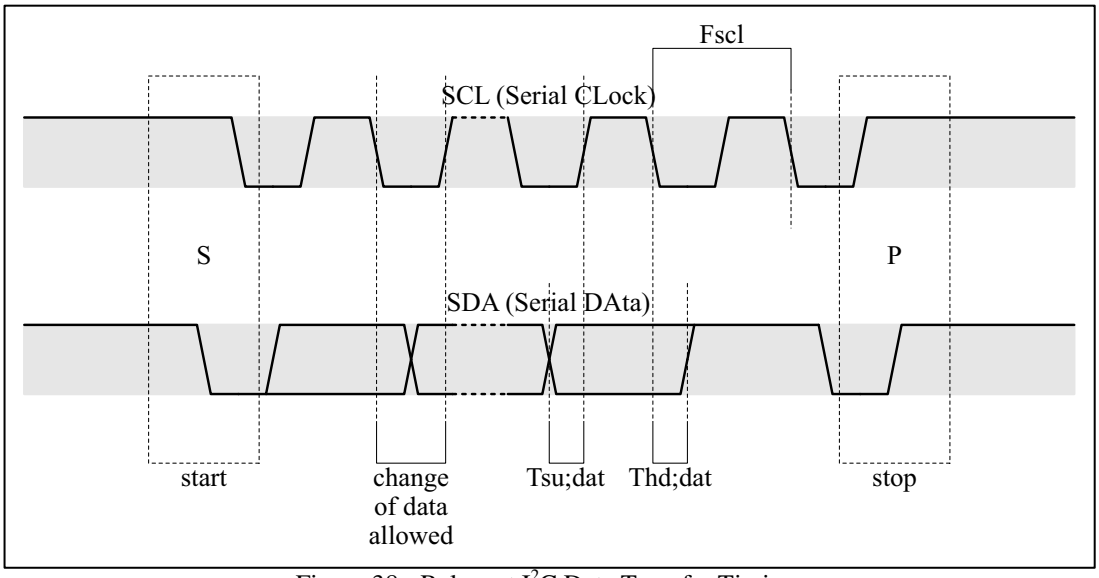


Figure 38: Relevant I^2C Data Transfer Timing.

8.2.4 I^2C Protocol

A typical I^2C read/write cycle is illustrated in Fig. 39. An important feature of the I^2C serial bus is that any device, be it master or slave can only pull the clock or data lines low and resistors must be used to pull the signals high. The iProfiler is a slave device so the clock will be provided by the master which could well be the iCell processor itself or for development purposes some sort of adapter to allow capturing of the created metrics by a PC or some other monitoring system.

The start condition is identified by a falling edge on the data line whilst clock is allowed to remain high and stop on a rising edge. Both of these conditions are controlled by the master.

The master follows the start condition with a 7-bit slave address and a read/write bit which indicates a write cycle if low. A responding slave that matches the address then pulls the data low to indicate acknowledgement. The slave address chosen for the iProfiler is 0x11 (read) and 0x10 (write).

This is followed by single or multiple data bytes, also with acknowledgement bits.

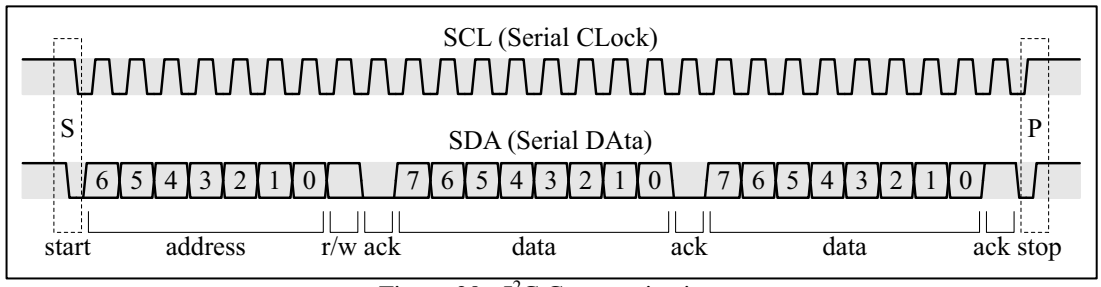


Figure 39: I²C Communication.

8.2.5 I²C Read Access Operation

The iProfiler's variable length metric requires a read multiple operation which can be seen in Fig. 40 where the transmissions from master to slave are shaded. The multiple read is actually identical to the single byte read with the number of bytes being transferred determined by acknowledgments from the master. For each byte sent back from the slave device the master responds with an acknowledgment as seen in the diagram Fig. 40. An acknowledgment indicates that the master expects another byte of data and its absence is a request to terminate the read operation. An early implementation had the termination of the read multiple operation determined by the stop condition, however it became apparent that the absence of acknowledge was the correct way to do this and the stop condition in a read access is most likely superfluous.

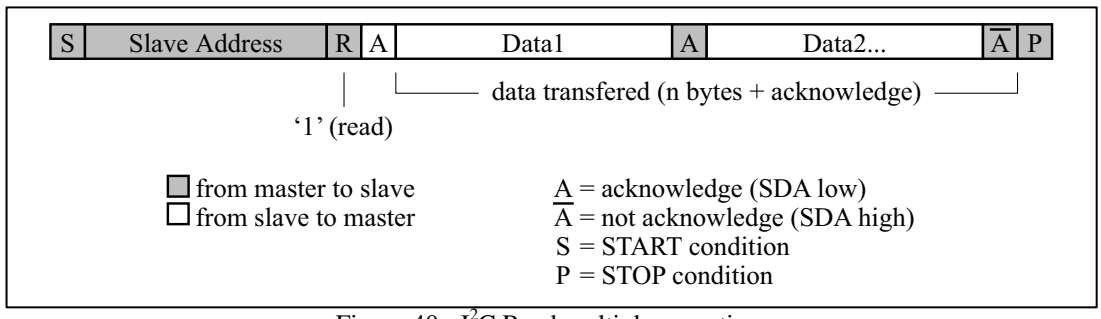


Figure 40: I²C Read multiple operation.

8.2.6 I²C Write Access Operation

The I²C write byte operation is fairly straightforward and can be seen in Fig. 41. As with the read operation, the master first sends the slave address followed by the

read/write bit set low (0x10). If the sent address matches that of the slave, a response in the form of an acknowledgment in sent back to the master. This is followed by a single data byte from the master which is also acknowledged. Finally the master sends the stop condition. Although the stop condition could well be ignored, it is used to reset the iProfiler's I^2C transfer state machine to cope with any communication errors.

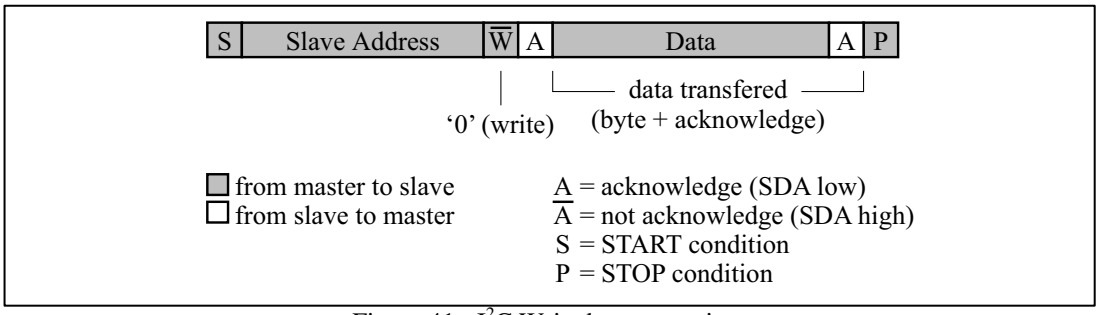


Figure 41: I^2C Write byte operation.

Chapter 9: FPGA iProfiler Operation and Performance

9.1 Initial iProfiler Operational Checks

Throughout the development of the iProfiler rigorous checks at each stage of the design process were instigated to ensure operation was as expected. This verification process was vindicated by the reliable operation of the final design. Such checks included forcing the hierarchy of state machines into unused modes and ensuring safe recovery back to normal operation. Although this is unlikely to happen during normal operation, it does provide some protection against a noisy or fluctuating power supply. This current work conveniently left the iCell's network communication links unused, thus providing a useful way to break out and instrument the internal operational states of the iProfiler in the FPGA.

The hardware based iProfiler, although difficult to implement proved very robust in operation and easy to control and obtain a stable but responsive metric of code integrity. The iProfiler never failed in operation over a period greater than 6 month and always recovered from deliberate electrical interruptions to its power and target processor JTAG link connections. Also such actions never caused a non-recoverable system failure although sometimes it would result in a recoverable reset of the target processor. Power cycling the iProfiler or the target processor together or alone always resulted in correct operation. Performance was excellent, predictable and fulfilled all the design goals.

Unfortunately due to the unexpected complexity of the JTAG control interface, the chosen FPGA was too low on space to implement both metric of code integrity and behavior in the same device running simultaneously. However all low level functions to implement a basic behavioral metric in FPGA already exist in the current design and have been proven to operate as specified.

9.2 Comparison with Software Implementation

Once the I^2C communication interface was finalised, further checks could be performed at a higher level. The first was confirming that the hardware iProfiler was producing the same metrics of code integrity as the earlier software based version. When this was performed with all 4 test programs at metric lengths of 2, 4, 8, 16, 32, 64, and 128, total correlation was found. This rather simple test verified much of the iProfiler's hardware design, including the following systems/sub-systems:

1. Halt the target processor (JTAG).
2. Read program memory of the target processor (JTAG).
3. Resume operation of the target processor (JTAG).
4. Create the Code Integrity Metric.

Correct operation of these systems was already determined by careful observation of internal state machines and external signals lines during development however conformation was welcome.

To perform a more in depth analysis and comparison of the iProfiler to the earlier software based system, some instrumentation features were added to the FPGA design. This involved incorporating a test mode selected by one of the iCells unused communication ports. Referring back to Fig. 18, the 50 Hz timer was disabled and triggering was instead initiated by an I^2C write to the iProfiler control register. Also during this mode all write operations to the Program Structure Table held in SRAM were disabled and finally the I^2C returned the SRAMBra and Bra flags rather than the Code Integrity Metric. With these minor changes, the speed of falsification tests could be duplicated in hardware, thus verifying operational replication of the earlier software based iProfiler. The analysis involves loading one of the test programs into the target processor and running the iProfiler in normal mode, thus the Program Structure Table reflects the structure of the loaded test program. Then the iProfiler is placed into its new test mode and a second test program is loaded into the target processor. In this mode the target processor's program structure does not match the Program Structure Table stored in the SRAM and due to the disabling of the write access to the SRAM, this situation is maintained. A small data gathering program on a PC then continually triggers the iProfiler and determined whether the processors resident running program has been falsified by checking the Bra and SRAMBra flags. In this way average falsifications checks required to determine a program code change can be calculated and compared with the original data gathered in the earlier work detailed in Chapter Five The data matched to greater than 3 decimal places and displayed convergence, thus verifying the hardware was operating in the same manner as the software version.

9.3 Modes of Operation

The existing operation of the iProfiler searches in processor memory for the next branch instruction from the current program counter location and along the way

cross checks the encountered program structure with that recorded in the Program Structure Table held in SRAM. Whilst it was felt this was a fairly optimal method to verify structure in terms of response time it was by no means certain and for this reason consideration of other operating modes was required. An alternative method is to fix the number of entries cross checked in the Program Structure Table, again starting at the current program counter location.

Due to the rather more time consuming hardware setup needed for data collection it was decided to return to the software version to determine the optimal method of program structure verification. The software method also has the advantage of returning the number of program structure entries read whilst running in the dynamic mode used so far, which will allow computation of the average iProfiler samples required to falsify code integrity.

Fig. 42 illustrates average falsification times with a range of Program Structure Table entry checks per iProfiler sample. This graph was the result of all 4 test programs being replaced by each other in program memory with 500,000 samples in each configuration. It will be noticed that there is a significant drop in average iProfiler samples required to falsify code integrity when more than one Program Structure Table entries are cross checked, with two entries showing the greatest improvement. However the longer sample period would offset this gain to some degree so this must be taken into account to determine the optimal solution. The average period of processor time in milliseconds to falsify the exchanged program code was computed and can be seen in Fig. 43. It will be noticed that the longer sample periods required to check multiple structure table entries outweighs the gain of the reduced samples seen in Fig. 42, thus a fixed length iProfiler sample utilizing a single Program Structure Table entry check would seem to be the optimal solution. This method also has the advantage of a relatively easy fit into a SOC solution due to processor instructions needed for comparison with the Program Structure Table being held in the instruction pipeline. The ARM7TDMI core has a 3 instruction pipeline, which would require either a core change to 4 instructions or the more practical solution of adopting 12 bytes per structure table entry.

Comparison with the dynamic variable period branch search method would be useful so a single plot of the average entries and period to falsification is shown in red. Although the average period of 0.7736 ms is faster than the 0.8607 ms found with the fixed single entry check, the advantage of a constant, deterministic performance

hit on the target processor would outweigh the slight gain of 11.26% in falsification speed. Processor speed degradation with the single entry check performed every 20 ms (50 Hz) is 2.3096%.

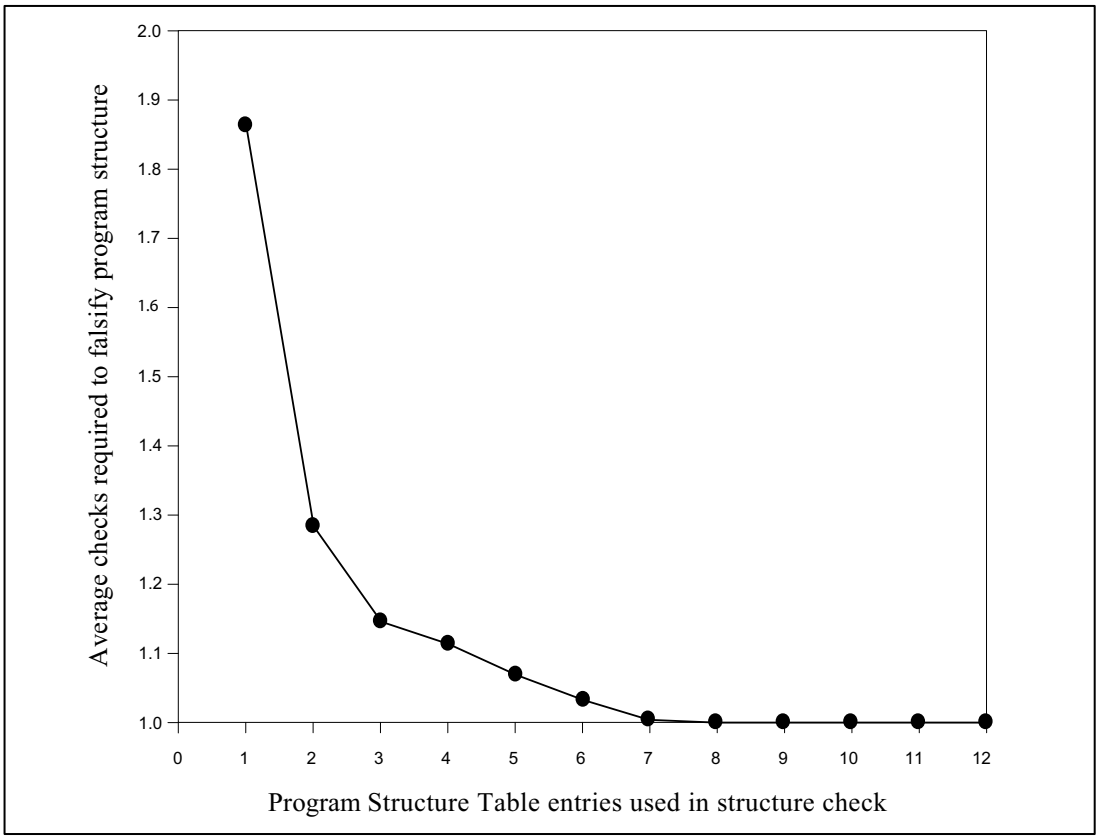


Figure 42: Average checks required to falsify program structure dependent on Program Structure Table entries used.

The iProfiler sampling frequency is directly related to the processor performance hit and the average period of falsification. This relationship based on the test program falsification results can be seen in Fig. 44 and were calculated using the following formulas.

*Sampling Frequency = 1 / (Average Period to Falsify * Average Samples to Falsify)*

Where the average samples to falsify using the single entry check is 1.8634251.

*Processor Hit = Sampling Frequency * Period Per Active Sample*

Where the period per active sample using the single entry check is 0.4619166 ms.

Perhaps in some situations a slower response time to determine code integrity would be quite adequate. For example an average response time of 10 seconds would result in an iProfiler sample rate of 0.186342 Hz and an almost unperceivable performance slow down of 0.008607% which is acceptable in most real-time systems.

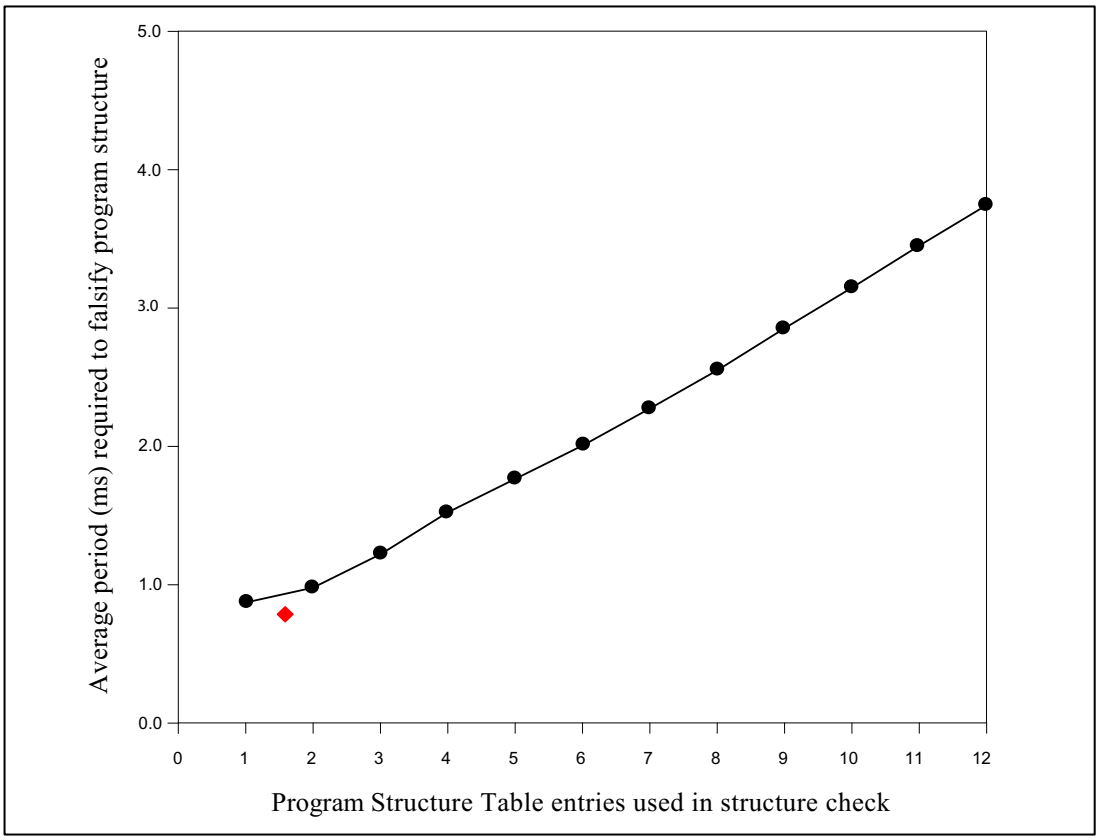


Figure 43: Average processor period in ms required to determine change in structure dependent on Program Structure Table entries used.

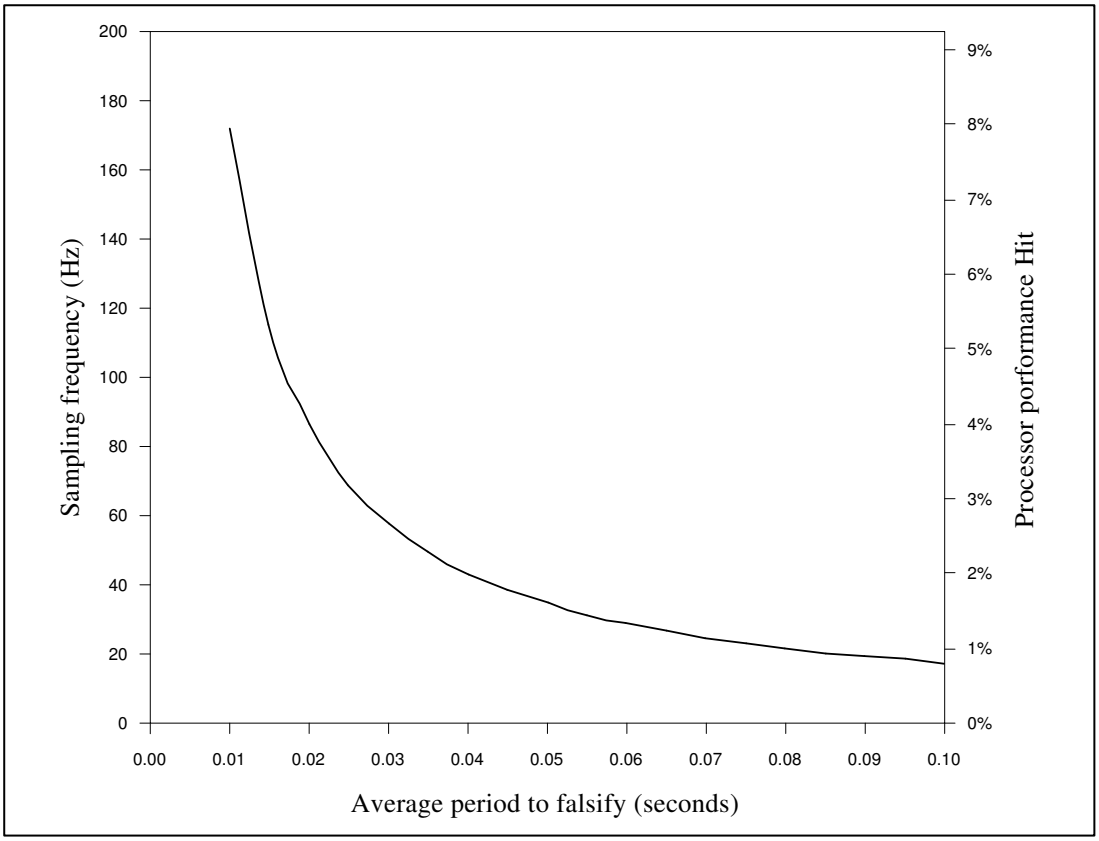


Figure 44: Processor performance hit and iProfiler sampling frequency determined by the average period to falsify.

Since the fixed program structure entry check and other modes of operation were considered worthy of investigation early in the hardware development, the FPGA design already had a working implementation. This mode of operation requires an alteration of the Code Integrity Metric state machine previously seen in Fig. 18. This modification for single entry check can be seen here in Fig. 45, where it will be noted it's actually a simplification of the original design and confined to state 'D'.

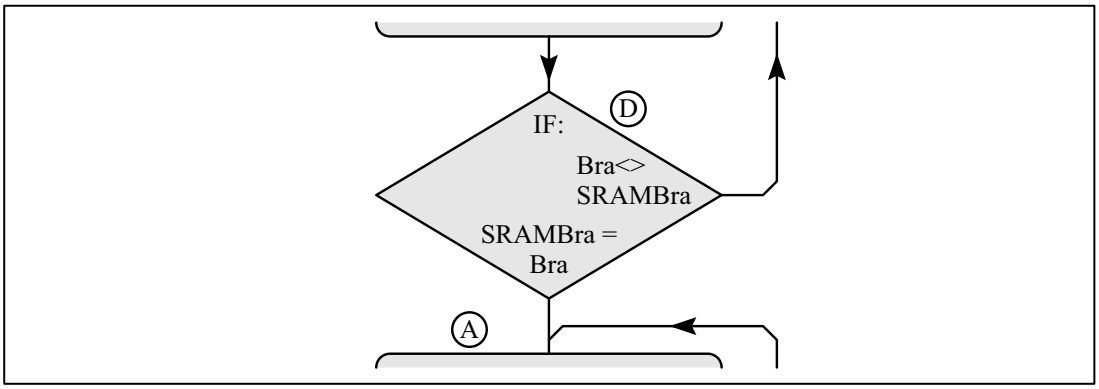


Figure 45: Change of the Code Integrity Metric state machine to implement a simpler single entry check of the Program Structure Table.

Chapter 10: Conclusions

A requirement a large amorphous computing array such as the iSurface for quick inter-iCell diagnostics is clear and I argue that this can be fulfilled by the use of stable metrics based on program structure. Further, behavioral metrics can only have any real meaning when a metric based on program code is available. Traditional methods to produce such metrics rely on the idea of accumulating data during run time using various profiling techniques. However, creating something stable derived from the inherently dynamic process of program activity is a serious problem and likely unresolvable. The idea of turning this process 'on its head' and using similar profiling techniques to disprove a metric created by program structure has been shown to possess the ideal properties of responsiveness to code change, stability and the possibility of a low cost SOC implementation.

Although security was not the intended application for the Code Integrity Metric, it does offer a way to remotely monitor a single processor installation. One scenario would be the installation of an iProfiler dongle being plugged into the monitored processor's JTAG socket. Such a dongle would either continuously stream the CIM or send an alarm on a change in CIM over an encrypted link, for example GPRS. Behavioral metrics could also be used to trip an alarm if straying beyond defined parameters.

Further work on behavioral metrics is already underway and will be addressed in a follow-up paper to "Stable Metrics in Amorphous Computing: An Application to Validate Operation and Monitor Behavior" [2].

A stable metric of code integrity also puts real meaning into the behavioral and diagnostic metrics, because as with biometrics, it's important to know the animal you are investigating first.

10.1 Future Work

The work done so far has been based on the determination of incorrect loaded program code in its entirety and not on the smaller differences caused by corruption or external compromisation. In this respect it would be important to look at possible benefits of fixed or variable multiple structure table entry checks to determine such small changes.

Whilst sampling data in the current system it was noted that sometimes aliasing seemed to occur, in other words the sampling period matched the time taken for the target processor's running code to perform a set of operations and return to the same address in memory. This is a fairly well known profiling problem when sampling and can be solved by various methods including the randomization of the sampling period. It would be interesting to investigate the improvement such enhancements would have when employed in the iProfiler system.

Initial preliminary work on the creation of a behavioral metric to complement the Code Integrity Metric showed great promise and further development will certainly be pursued in this direction.

The original research plan was to develop a method that harnessed the two layer FPGA/MCU infrastructure to solve the problem of fast inter-iCell data communication, thus allowing directed, parallel propagation across the iSurface. This however raised the issue of fault tolerance in such a topology which resulted in the work presented in this thesis. To continue back on track with the original concept, along with a developed method of fault tolerance would seem a logical next step.

To conclude, the technology developed in this work has potential far beyond the initial research goals, particularly in security of embedded systems and offers many opportunities for further research.

Bibliography

[1] M. Satyanarayanan, “Pervasive Computing Vision and Challenges,” IEEE Personal Comm., vol. 6, no. 8, Aug. 2001, pp. 10–17.

[2] M. J. Lear, "Stable Metrics in Amorphous Computing: An Application to Validate Operation and Monitor Behavior," in Intelligent Environments (IE), 9th International Conference on, 5-8 Aug 2012, pp.204 – 211.

[3] D. Coore, “Introduction to Amorphous Computing,” *Lecture Notes in Computer Science*, Springer Verlag: Berlin, 2005, pp. 99-109.

[4] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, J. Knight, T. F. R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss, “Amorphous Computing,” in *Communications of the ACM*, vol. 43, no. 5, 2000, pp. 74-82.

[5] R. Nagpal and M. Mamei, “Engineering Amorphous Computing Systems,” in *Methodologies and Software Engineering for Agent Systems, The Agent-Oriented Software Engineering Handbook*, Kluwer Academic Publishing, New York (NY), 2004, pp. 303-320.

[6] Y. Hu, “Swarm Intelligence,” 2012. Available: http://guava.physics.uiuc.edu/~nigel/courses/569/Essays_Fall2012/Files/Hu.pdf

[7] M. G. Hinchey, R. Sterritt, and C. Rouff, “Swarms and swarm intelligence,” in *Computer*, vol. 40, no. 4, pp. 111–113, Apr. 2007.

[8] J. R. Heath, P. J. Kuekes, G. S. Snider, R. S. Williams, “A Defect- Tolerant Computer Architecture: Opportunities for Nanotechnology,” in *Science*, Jun 1998, pp.1716-1721.

[9] M. Hartmann, F. Eskelund, P. C. Haddow, J. F. Miller, “Evolving Fault Tolerance on an Unreliable Technology platform,” *Proc. Conf. on Genetic and Evolutionary Computation (GECCO)*, New York, Jul 2002, pp. 171–177.

[10] J. vonNeumann, “The Fiist Draft of a Report on EDVAC,” 1947, repnnted in Annals of the History of Computing, vol 15, no 4, 1993.

[11] R. Nagpal, A. Kondacs, C. Chang, “Programming Methodology for Biologically-Inpired Self-Assembling Systems,” in *Computational Synthesis : From Basic Building Blocks to High Level Fuctionality: Paper from the AAAI Spring Symp*, AAAI Press, 2003, pp. 173-180.

[12] R. Nagpal, “Programmable Self-Assembly using Biologically-Inspired Multiagent Control,” *1st Int.l conference on Autonomous Agents and Multiagent Systems*, Bologna, Italy, Jul 2002.

[13] L. Clement, R. Nagpal, “Self-Assembly and Self-Repairing Topologies,” in *Proceedings of the Workshop on Adaptability in Multi-Agent Systems*, RoboCup Australian Open, Jan 2003.

[14] D. Chu, D. J. Barnes, S. Perkins, “Amorphous Computing in the Presence of Stochastic Disturbances,” in *Biosystems*, vol. 125, Nov 2014, pp. 32-42.

[15] M. A. Jennifer, et al, ”Continuous Profiling: Where Have All the Cycles Gone?,” in *ACM*, 1997, pp. 357-390.

[16] J. Whaley, “A Portable Sampling-Based Profiler for Java Virtual Machines.,” in *ACM 2000 Java Grande Conference*, June 2000.

[17] T. Ball, and J. R. Larus, “Efficient path profiling,” in Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris, 1996, pp.46-57.

[18] M. Arnold and D. Grove, “Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines,” in *International Symposium on Code Generation and Optimization*, March 2005.

[19] R. G. Scottow and A. B. T. Hopkins, “Instrumentation of Real-Time Embedded System for Performance Analysis,” in *Proc. IEEE IMTC*, Sorrento, Italy, Apr. 2006, pp. 1307–1310.

[20] H. Zhang, J. Ji, X. Zhou, H. Ma and C. Wang. “Design and Implementation of a Configurable Hardware Profiler Supporting Path Profiling and Sampling,” *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC 2009)*, Zhangjiajie, China, Oct 2009, pp.325-330.

[21] R. A. Frohwerk, "Signature analysis: A new digital field service method," Hewlett-Packard J., May 1977, pp.2-8.

[22] A. King, V. Callaghan, G. Clarke, “Using An Amorphous Computer For Visual Display Applications In Intelligent Environments,” *IET 4th International Conference on Intelligent Environments*, Seattle, WA, USA, Jul 2008.

[23] A. King, “Deus Ex Machina: Engineering Emergence in an Amorphous Computer,” *International Workshop on Intelligent Environments*, Colchester, UK, Jun 2005.

[24] W. Butera, “Programming a Paintable Computer,” PhD Thesis, MIT Media Lab, Feb 2002.

[25] J. Chang, T. Ge, E. Sanchez-Sinencio, "Challenges of Printed Electronics on Flexible Substrates," in *Circuits and Systems (MWSCAS)*, IEEE 55th International Midwest Symposium on, 5-8 Aug 2012, pp.582,585.

[26] D. H. Kim, et al, “Epidermal Electronics,” in *Science*, vol 333, Aug 2012, pp. 838-843.

[27] Y. Kovalchuk, G. Howells, and K. D. McDonald-Maier, “Overview of ICMetric Technology – Security Infrastructure for Autonomous and Intelligent Healthcare System,” in *International Journal of u- and e- Service, Science and Technology*, vol. 4, no. 3, Sep 2011, pp. 49-60.

[28] X. Zhai, K. Appiah, S. Ehsan, H. Hu, D. Gu, K. McDonald-Maier, W. M. Cheung, and G. Howells, “Application of ICmetrics for Embedded System Security,” in *Emerging Security Technologies (EST)*, 2013 Fourth International Conference on, 2013, pp. 89–92.

[29] V. Callaghan and K. Barker, “SAS-an Experimental Tool for Dynamic Program Structure Acquisition and Analysis,” *Journal of Microcomputer Applications*, vol. 5, 1982, pp. 209–223.

[30] *AT91SAM ARM-based Flash MCU (doc6175.pdf)*, Atmel Corporation – Microcontrollers, San Jose, CA, USA, 2012.

[31] *Eclipse Official Website:* http://www.eclipse.org

[32] *Online OpenOCD User’s Guide:* http://openocd.sourceforge.net/doc/html/index.html#

[33] *GCC Official Website:* http://gcc.gnu.org/

[34] K. R. Popper, "Science as Falsification," *Conjectures and Refutations*, Routledge and Keagan Paul: London, 1963, pp. 30-39.

[35] S. McCanne, C. Torek, "A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling," in *Proc. of the Winter USENIX Conf*, San Diago, CA, USA, Jan 1993, pp 387-394.

[36] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proceedings of the International Workshop on Workload Characterization*, 2001, pp. 3-14.

[37] *Lattice MachXO Family Data Sheet:* http://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/MachXO23/MachXOFamilyDataSheet.pdf

[38] F. Leens, "An Introduction to SPI and I2C Protocol," *IEEE Instrumentation and Measurement magazine*, February 2009.

[39] I. Zinovik, Y. Cherbiryak and D. Kroening, "Computing Binary Combinatorial Gray Codes Via Exhaustive Search with Sat Solvers," in *IEEE Transactions on Information Theory*, 2008, pp.1819-1823.

[40] *EIA standard RS-232-C: Interface between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange,* Washington: Electronic Industries Association. Engineering Dept, 1969.

[41] *The I2C-bus specification and user manual*, Rev. 6 – 4, NXP Semiconductors N.V., San Jose, CA, USA, 2014.

Appendices

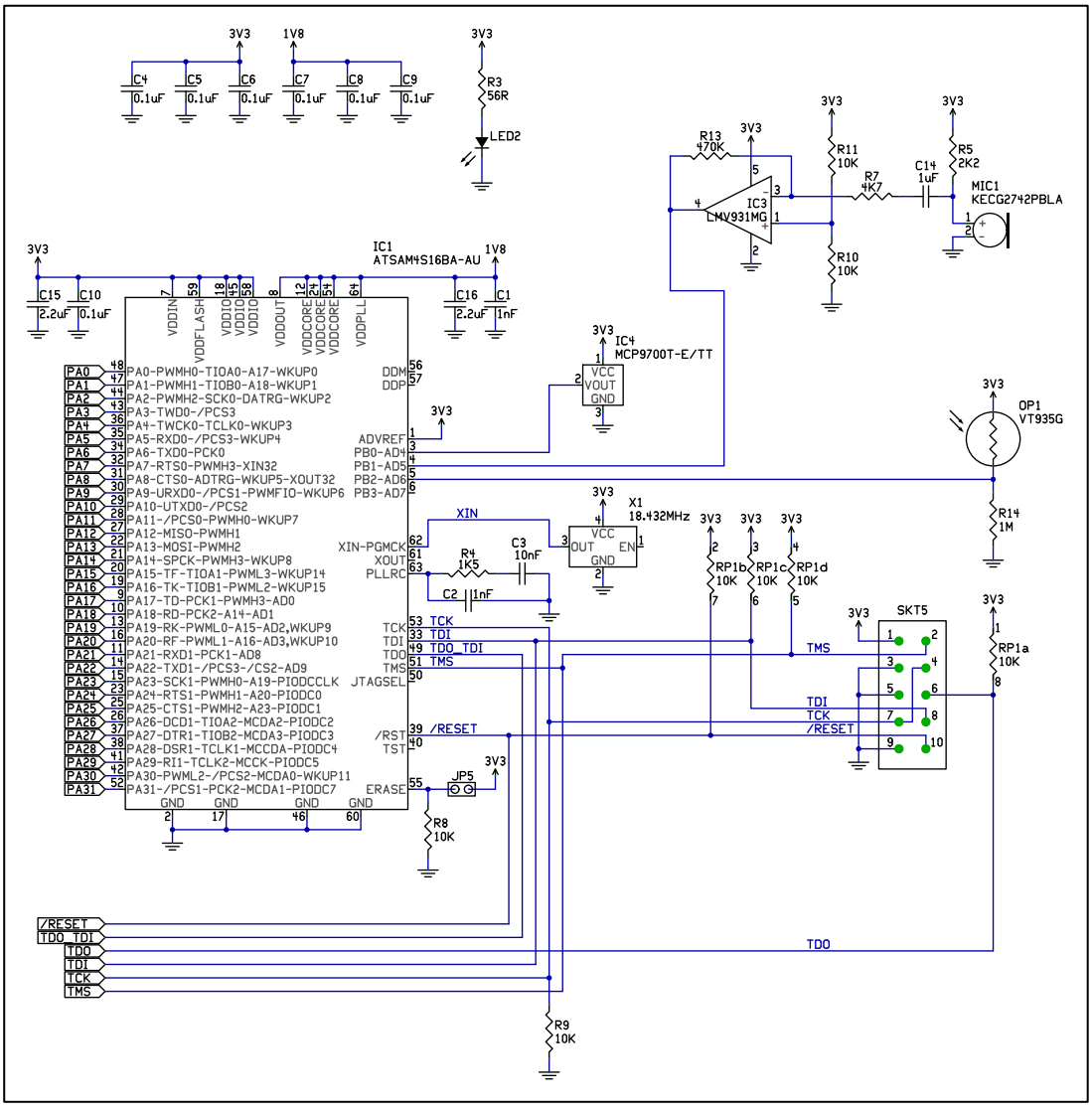


Figure 46: iCell schematic (MCU section).

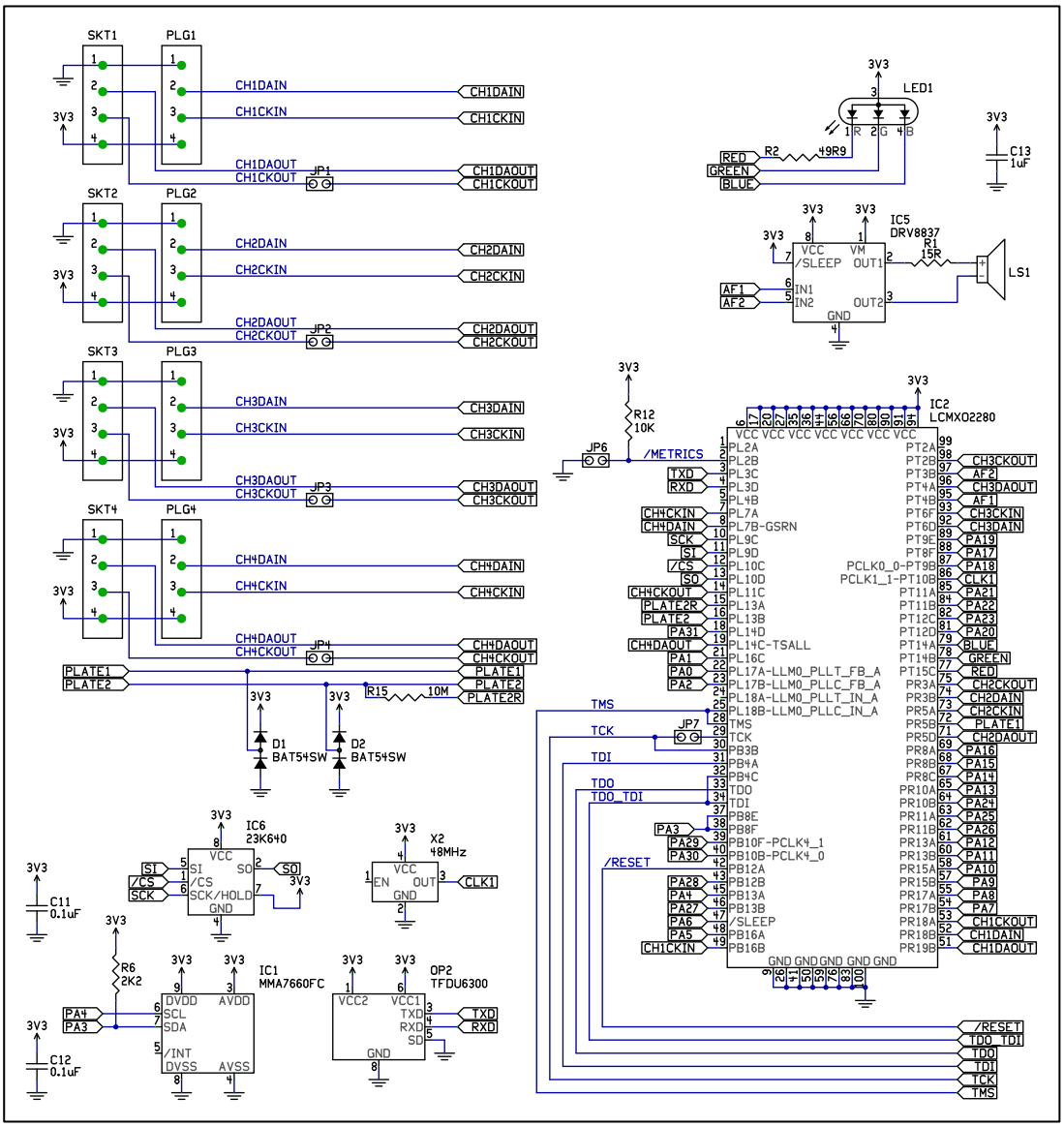


Figure 47: iCell schematic (FPGA section).

Table 7: FPGA JTAG sequences.

No	Length	Sequence
0	3	TAP State 3-2-2
1	3	Reserved
2	4	Exit DR Enter DR
3	4	Reserved
4	6	Exit DR Enter DR
5	6	Reserved
6	7	Write Watchpoint 0 Control Value
7	7	Set TAP State C
8	8	Write Debug Control
9	8	Exit DR Enter DR
10	9	Read Debug Comms Control Register
11	9	Reserved
12	10	RESTART
13	10	SCAN_N
14	10	Reserved
15	10	Reserved
16	11	Read Debug Status
17	11	Write Debug Control
18	11	Write Watchpoint 0 Address Mask
19	11	Exit DR Enter DR
20	11	Exit DR Exit DR Enter DR
21	11	Exit DR Enter DR
22	11	Write Watchpoint 0 Data Mask
23	11	Exit DR Enter DR
24	32	0x00000000
25	32	Return spsr
26	32	0x00000100
27	32	0x00000009
28	32	0x00000009
29	32	0x00000009
30	32	0x00000005
31	32	Reserved
32	32	Scan Chain 2, INTEST
33	32	Scan Chain 2, INTEST
34	32	Scan Chain 1, INTEST

35	32	Reserved
36	32	Reserved
37	32	Reserved
38	32	Reserved
39	32	Reserved
40	32	0x000000F7
41	32	MRS
42	32	MRS Rd SPSR
43	32	MOV r8 r8
44	32	STR
45	32	STM r1
46	32	STM r0, r1, sp_svc, pc
47	32	STM r0
48	32	STM pc
49	32	STM r0 r1
50	32	LDM r1
51	32	B
52	32	0xFFFFFFFF
53	32	Reserved
54	32	Reserved
55	32	Reserved
56	32	Return r0
57	32	Return r1
58	32	Return pc + 24
59	32	Return Data
60	32	Restore r0
61	32	Restore r1
62	32	Restore pc - 24
63	32	Set Address

Table 8: FPGA JTAG Halt sequences.

No	Length	Sequence
7	7	Set TAP State C
33	32	Scan Chain 2, INTEST
52	32	0xFFFFFFFF
18	11	Write Watchpoint 0 Address Mask
52	32	0xFFFFFFFF

22	11	Write Watchpoint 0 Data Mask
26	32	0x00000100
6	7	Write Watchpoint 0 Control Value
2	4	Exit DR Enter DR
40	32	0x000000F7
6	7	Write Watchpoint 0 Control Mask
2	4	Exit DR Enter DR
29	32	0x00000009
16	11	Read Debug Status
27	32	0x00000009
10	9	Read Debug Comms Control Register
2	4	Exit DR Enter DR
30	32	0x00000005
17	11	Write Debug Control
24	32	0x00000000
6	7	Write Watchpoint 0 Control Value
34	32	Scan Chain 1, INTEST
4	6	Exit DR Enter DR
46	32	STM r0, r1, sp_svc, lr_svc, pc
23	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
56	32	Return r0
19	11	Exit DR Enter DR
57	32	Return r1
19	11	Exit DR Enter DR
25	32	Return spsr
19	11	Exit DR Enter DR
25	32	Return spsr
19	11	Exit DR Enter DR
58	32	Return pc + 24
19	11	Exit DR Enter DR
41	32	MRS Rd CPSR
19	11	Exit DR Enter DR
44	32	STR

19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
25	32	Return spsr
19	11	Exit DR Enter DR
42	32	MRS Rd SPSR
19	11	Exit DR Enter DR
44	32	STR
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
25	32	Return spsr
9	8	Exit DR Enter DR

Table 9: FPGA JTAG Read memory sequences.

No	Length	Sequence
0	3	TAP State 3-2-2
47	32	STM r0
23	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
63	32	Set Address
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
21	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
50	32	LDM r1
20	11	Exit DR Exit DR Enter DR

13	10	SCAN_N
33	32	Scan Chain 2, INTEST
28	32	0x00000009
16	11	Read Debug Status
27	32	0x00000009
10	9	Read Debug Comms Control Register
34	32	Scan Chain 1, INTEST
4	6	Exit DR Enter DR
45	32	STM r1
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
59	32	Return Data
19	11	Exit DR Enter DR
41	32	MRS Rd CPSR
19	11	Exit DR Enter DR
44	32	STR
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
25	32	Return spsr
9	8	Exit DR Enter DR

Table 10: FPGA JTAG Resume sequences.

No	Length	Sequence
0	3	TAP State 3-2-2
49	32	STM r0 r1
23	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
60	32	Restore r0

19	11	Exit DR Enter DR
61	32	Restore r1
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
48	32	STM pc
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
62	32	Restore pc - 24
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
43	32	MOV r8 r8
21	11	Exit DR Enter DR
43	32	MOV r8 r8
19	11	Exit DR Enter DR
51	32	B
20	11	Exit DR Exit DR Enter DR
32	32	Scan Chain 2, INTEST
24	32	0x00000000
8	8	Write Debug Control
12	10	RESTART