THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

The School of Engineering

ELECTENG423
Computer Systems 3E

Implementation of a Customised Microcontroller as a System on Programmable Chip using FPGA

Developed By : Amal Iddawala (9832356)
Kasun Talwatte (9770802)
Chinthana Jayasena (2285724)
Thusitha Mabotuwana (9790416)

Group Number : 16

Date of Report: 04.10.2004

This document is the sole property of Auckland University and any part of it should not be reproduced or used for any purpose other than what it is intended for without the written consent of the authors and University. All rights reserved

Summary

Exploring the possibilities and then implementing a customised microcontroller as a system on programmable chip using an FPGA was considered the main task of this project. An open source processor called MiCORE was provided for this purpose. The implementation was to be done in two main stages where for the first stage it was required to implement a 2-channel analogue to digital converter scheme where the user could specify online the down sampling ratio and the sampling frequency. Based on the desired down sampling ratio, the average of the sum of each channel's samples was to be calculated and then the two averages were to be squared, added and then the square root calculated. As the second stage of the development process, it was required to introduce a 4-level interrupt structure to the existing MiCORE and this was to be called I-MiCORE.

The implemented system meets all the project requirements and all the hardware coding has been done in VHDL. Quartus II has been used as the main simulation tool and all the synthesis have been done using an EP20K200EFC484-2X, APEX20KE family chip. Timing simulations have been preferred throughout the design process to functional simulations since the original intention was to actually implement the system on an Excalibur NiOS board.

This report outlines the main aspects of the implemented design and discusses the algorithms developed to achieve the required functionality. Various simulation results and other relevant details such as resource usage are also included. Assembly code that was written for the two tasks can be found in the appendices at the end.

Acknowledgement

We would also like to take this opportunity to thank our lecturer Professor Zoran Salcic for giving us help and useful tips with regard to VHDL material and other lecture notes.

Also we would like to thank the two teaching assistants Mr. Ivan Radojevic and Mr. Gary Wang for all the help given throughout the duration of the project, even outside office hours. The kind advice given to us whenever required is also very much appreciated.

Thusitha, Kasun, Amal and Chinthana

Table of Contents

1.0 Introduction 1
2.0 Project Specifications 1
3.0 Architecture of MiCORE 1
4.0 Design Assumptions 3
5.0 Development of the Design 3
6.0 Implementation of the Final Design 4
6.1 Implementation of the Microcomputer without Interrupts 4
6.1.1 ADC Module and Square Root Calculation 4
6.1.2 Modelling of the ADC Module 6
6.1.3 Getting Online Parameters 7
6.1.4 Message Formats and Hardware – Software Integration 8
6.2 Implementation of the Microcomputer with Interrupts 10
6.2.1 Interrupt Design Criteria 10
6.2.2 Interrupt Priority Resolution Algorithm 11
6.2.2.1. Datapath Modifications 13
6.2.2.2. Controlunit Modifications 13
6.2.3 Implementation of I-MiCORE 14
6.3 A Brief Comparison between Polling and Interrupts 16
6.4 Distribution of Tasks among Group Members 16
7.0 Conclusions 16

Appendices
Appendix A: ADC module's output waveforms i
Appendix B: Assembly code written for task1 ii
Appendix C: Modified Lexx and Yacc code to implement RETI iii
Appendix D: Interrupt priority resolution algorithm flowchart iv
Appendix E: Assembly code written for task2 iv

List of Figures

1.0 Introduction 1
2.0 Project Specifications 1
3.0 Architecture of MiCORE 1
4.0 Design Assumptions 3
5.0 Development of the Design 3
6.0 Implementation of the Final Design 4
6.1 Implementation of the Microcomputer without Interrupts 4
6.1.1 ADC Module and Square Root Calculation 4
6.1.2 Modelling of the ADC Module 6
6.1.3 Getting Online Parameters 7
6.1.4 Message Formats and Hardware – Software Integration 8
6.2 Implementation of the Microcomputer with Interrupts 10
6.2.1 Interrupt Design Criteria 10
6.2.2 Interrupt Priority Resolution Algorithm 11
6.2.2.1. Datapath Modifications 13
6.2.2.2. Controlunit Modifications 13
6.2.3 Implementation of I-MiCORE 14
6.3 A Brief Comparison between Polling and Interrupts 16
6.4 Distribution of Tasks among Group Members 16
7.0 Conclusions 16

Appendices
Appendix A: ADC module’s output waveforms i
Appendix B: Assembly code written for task1 ii
Appendix C: Modified Lexx and Yacc code to implement RETI iii
Appendix D: Interrupt priority resolution algorithm flowchart iv
Appendix E: Assembly code written for task2 iv

1.0 Introduction

The main aim of this project was to explore the feasibility and then implement a customized microcontroller as a system on programmable chip using Field Programmable Gate Array (FPGA) as the implementation device. An open source processor called MiCORE was provided as the starting core for customisation where all the modifications were to be made.

The project was to be carried out by 4 students and consisted mainly of two main tasks. Firstly it was required to implement a 2-channel Analogue-to-Digital Converter (ADC) using the existing core without much hardware modifications. User was also to be given the option to vary ADC parameters such as the sampling period and the number of samples taken into account to perform specific calculations (explained later). For the second task it was required to implement the same tasks as for the first one, but using an interrupt structure that was to be introduced to the MiCORE.

This report briefly describes the implementation of various tasks involved in the overall design and is structured as follows: Firstly, a brief introduction is given to the project specifications and the architecture of the existing processor. This is followed by a section that outlines the assumptions made during the design process. Implementation strategies and algorithms developed for the final design are detailed out in the next section along with a description of the task distribution among the group members. Some possible future developments are presented next, and finally conclusions.

2.0 Project Specifications

The main tasks of the project were as follows:

1. On an external request scan digital input lines such as a port, to get online a code for ADC parameters. These parameters were to include the ADC sampling frequency and the down-sampling ratio which would later decide how many samples to add, to perform a specific task.
2. Periodically scan the 2-input lines to get samples into the ADC, based on the user specified sampling frequency.
3. After the required number of samples (specified by the down-sampling ratio in Task 1) were collected, the square root of the sum of squares of the average of the samples was to be calculated and stored in a predetermined memory location. This square root value was also to be sent to an output port in a burst so that this information could be available for other external sources.

These tasks were first to be implemented using the existing core without any major modifications. As the next step, they were to be implemented with a four-level interrupt structure introduced to MiCORE.

3.0 Architecture of MiCORE

The existing processor that was to be modified during the course of the project is called MiCORE and is a RISC-like microprocessor with the following basic features:

- An operating speed of up to 30MHz and can be implemented on Altera NIOS board
- Harvard architecture (separate program and data memory)
- Simple and uniform three machine cycle instruction set
- Pipelined design with theoretically one instruction per clock cycle
- 64K x 32-bit program memory space with configurable internal program memory
- 64K x 16-bit data memory space with configurable internal data memory
- Eight 16-bit registers
- Five addressing modes: immediate, inherent, direct, register indirect and stack
- Two memory-mapped 16-bit I/O ports

The non-pipelined version of the above core was decided to be used in this project mainly due to simplicity, hence the discussions that follow will refer to this core.

As mentioned before, MiCORE has five different addressing modes with the following generic instruction format:

AM = Addressing Mode
OC = Opcode
OP = Operand
Rz = Destination register designator (R0 to R7)
Ry = Source register Y designator (R0 to R7)
Rx = Source register X designator (R0 to R7)

Figure 01: MiCORE instruction format

Table 01 below shows the 4 possible combinations there can be in the *addressing mode* field which will uniquely identify each instruction's addressing mode. Immediate and inherent modes have the same combinations since these modes are used to set or perform operations on internal registers. Various combinations have been used in the *opcode* field and these can be found in the core's VHDL design files.

Addressing Mode	Combination Required in the Instruction's 29-31 bit positions
Immediate and Inherent	00
Stack	01
Direct	10
Register indirect	11

Table 01: Addressing modes and bit combinations required

Execution of MiCORE instructions happens in three stages, namely instruction fetch, instruction decode and instruction execution. Shown in Figure 02 is a flowchart of this execution process.

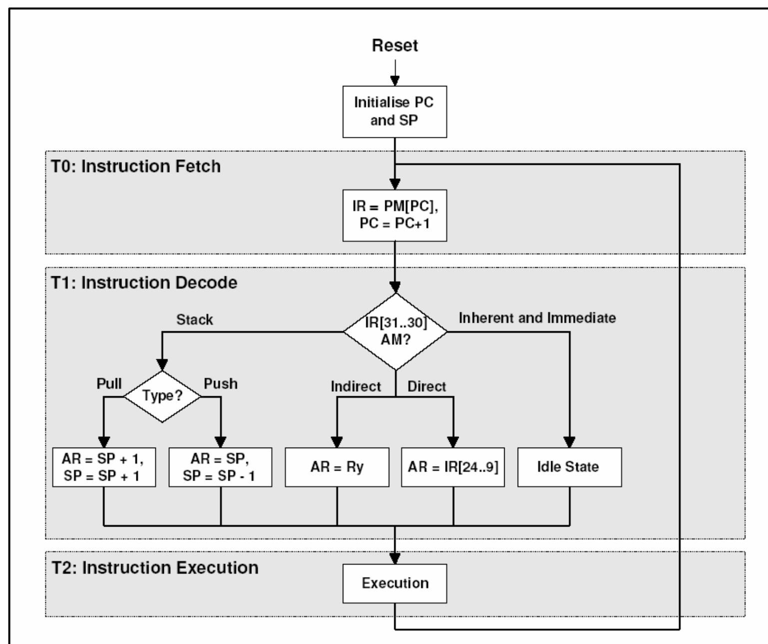


Figure 02: MiCORE instruction execution flowchart

4.0 Design Assumptions

- It was assumed that the provided MiCORE was functioning properly and that the complete instruction set was appropriately implemented to achieve the required functionality.
- The project handout does not specifically say how the ADC module should be implemented and it was decided to implement this in hardware mainly due to speedup and efficiency. Implementing external hardware frees the processor from the ADC data processing and helps in minimising processor overheads, thus allowing more time for the processor to deal with other tasks. During the later stages of the project there was an email sent by Mr. Radojevic and Mr. Wang, the 2 teaching assistants for the course detailing out how the square root algorithm could be implemented in software, but however discussions with the above 2 personnel suggested that a hardware implementation was also acceptable.
- ADC data could not be obtained using real analogue signals, hence the digital samples were assumed to be stored in *mif* files.
- The digital samples were assumed to be 8-bits. This meant that the maximum value each digital sample could hold was 255. Using this assumption the maximum value the square root could have was calculated as follows:

Maximum value of each 8-bit digital word : 255
Number of samples to be added : x ; where x could be 64, 32, 16 or 8.
Average of samples (in worst case scenario) : 255/x
= 255
∴ Maximum value the square root can have :

$$\sqrt{(Max_SampleAvg_source1)^2 + (Max_SampleAvg_source2)^2}$$

$$= \sqrt{(255)^2 + (255)^2}$$

$$= {}_{10}360.6$$

$$= {}_{2}101101000$$

Based on this calculation 9 bits were decided to be sufficient for the square root output.

5.0 Development of the Design

The initial stage of the development process was deciding on a sensible breakdown of all the tasks involved and making a rough schedule with estimated time durations (Table 06). It was decided to implement the ADC module first and then test this out completely before moving onto assembly programming and interrupt implementation. Quartus II and/or ModelSim tools were suggested to be used to test the functionality of the modules. In the implemented solution Quartus timing simulations have been preferred over the other since the original intention of the project was to actually implement the design on an Excalibur NIOS board. However ModelSim was used for some of the functional simulations mainly due to its ease of use and fast compilation times. Appropriate test benches were also written for testing purposes and the relevant simulation results are shown in the subsequent sections.

For the ADC module it was deemed appropriate to have the 2 sets of digital samples in *mif* files since actual analogue signals could not be used and converted to digital samples. This was sufficient for simulation purposes but however the decision to implement the ADC module in hardware could not quite be justified during the course of the project due to the large memory overheads. However if implemented on a NIOS board and the output samples of a real ADC were fed into the implemented ADC module instead of getting the samples out of the *mif* files, the ADC module's processing time will be much less and the decision justified.

The next stage of the design process was integrating the ADC module with assembly to achieve the project requirements. The requirement was to poll the external user request and update the ADC parameters accordingly.

Introducing a 4-level interrupt structure to the existing core was considered to be the next and final stage, and also the most challenging. Many possibilities had to be taken into account during this phase of the design which are described in detail in the following sections.

6.0 Implementation of the Final Design

The implemented final solution is the outcome of many preliminary stages where many possible implementation techniques and algorithms were developed and tried out. The aim of this section is to give an outline of the final design solution that has been implemented. It has been structured as follows: Section 5.1 details out the implementation and other algorithms used to implement the first task of the project. It describes how the ADC module has been implemented and how the overall system that includes hardware and software has been integrated to achieve the required functionality. Section 5.2 focuses mainly on the second task and details out how a 4-level interrupt structure has successfully been incorporated into the existing processor. Relevant simulation results are also included.

6.1 Implementation of the Microcomputer without Interrupts

As mentioned previously the main aim of this stage was to implement an ADC module which could get online values for down sampling and sampling frequency, read upto 64 eight-bit digital sample values depending on the desired down sampling ratio, sum them up and then determine the average. After the averages from both the input channels were calculated, a square root calculation was to be performed by taking the squares of these 2 averages and then adding the 2 values up. The final square root value was then to be stored in a predefined location in memory. This subsection outlines how hardware was developed to meet this requirement.

6.1.1 ADC Module and Square Root Calculation

Shown below in Figure 03 is a block diagram of the system developed for the ADC module. A brief description of each of the modules follows.

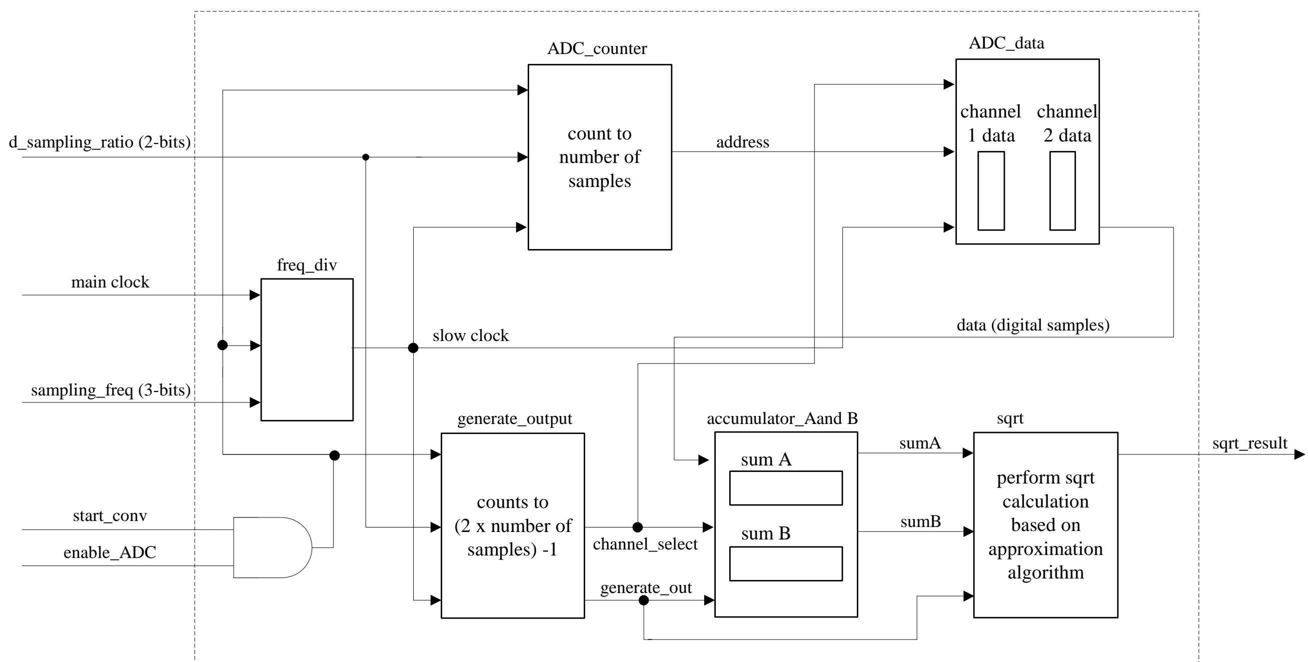


Figure 03: Block diagram and interconnections of the ADC module

Enable_ADC

This input is fed into a simple 2-input AND gate where the 2 inputs are *start_conv* and *enable_adc*. As described in Section 5.1.3, the *start_conv* signal is sent to the ADC module using the memory mapped (memory location $FFFF) I/O port SOR, and there is a possibility that a user could be using this memory location even for other purposes in the assembly code. Therefore it was decided to have the *enable_adc* line which has to be externally pulled high if the ADC was to be used. This prevents the ADC from running unintentionally.

ADC data module

This module contains the 2 *mif* files with 8-bit sampled data. In reality this data could be the outputs from a real ADC, but *mif* files had to be used for this project mainly for simulation purposes.

ADC counter module

As shown above, there are 3 inputs to the ADC counter unit. When the *start_conv* line is triggered, the encoded values on the *DSR* line (or *d_sampling_ratio* input) are read and values are stored into internal registers based on the decoding scheme shown in Table 02.

DSR value	Decoded value
00	64 – 1 = 63
01	32 – 1 = 31
10	16 – 1 = 15
11	8 – 1 = 7

Table 02: Decoding scheme for the ADC counter module

An internal counter is initialised with the decoded value which simply counts down by 1 after reading a sample from each of the 2 *mif* files. The value of the counter has been used as the address into the *mif* files so that any additional hardware is not required. The other output of this unit is a bit that selects between source 1 and source 2. If this bit is set to 0, source 1 is implied and setting this bit high makes the ADC data module select source 2 as the input channel.

Generate output module

When the *start_conv* line is triggered this unit's internal counter is initialised to (2 x decoded value of Table 02) - 1. This number is based on the fact that 2 data values have to be accessed from the ADC data module, which is twice the decoded value. The '-1' term is required as the end of all memory accesses is determined when this internal counter reaches 0. As such, when this counter reaches 0 the output line *generate_out* is set indicating to other modules that the averaging and square root calculations have to be done.

Accumulator_AandB module

This unit has 2 internal registers that can contain a maximum value of 255 x 64 since this is the maximum sum that could be produced based on the assumptions made. As shown in Figure 03, the output of the 'generate_output' module is fed into this module which then outputs *sumA* and *sumB* to the 'sqrt' module.

Sqrt module

The purpose of this module is to average out *sumA* and *sumB* based on the down sampling ratio that was stored during initialisation, and then perform a square root calculation when the *generate_out* signal is triggered by 'generate_output' module. The logic in this module is mainly combinational and produces the *sqrt* output within approximately 25ns (which is less than a clock cycle of a 30MHz main clock) as shown in Figure A.3.

The square root calculation is based on an approximation technique described in one of Professor Salcic's lecture notes [2] which is given by the simplified formula:

$$\sqrt{a^2 + b^2} \cong \max((0.875x + 0.5y, x)$$

, where x, y represent max(|a|, |b|) and min(|a|, |b|) respectively.

A flowchart showing how values a,b can be manipulated in order to implement the above approximation is shown below in Figure 04.

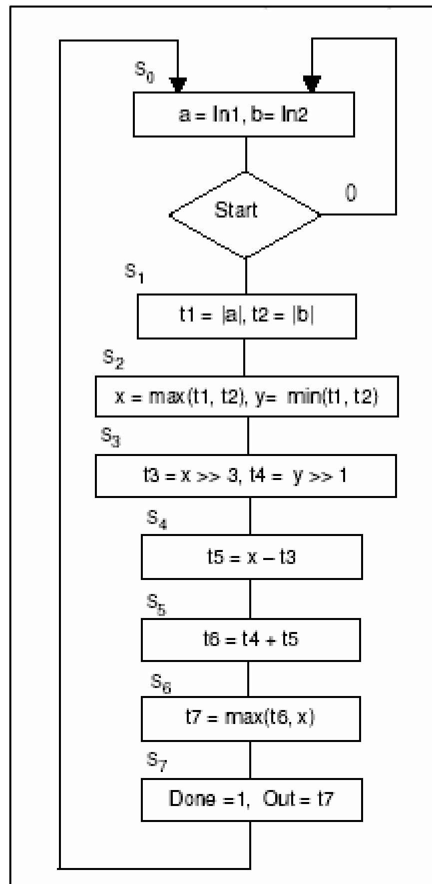


Figure 04: Flowchart showing a square root approximation technique [2]

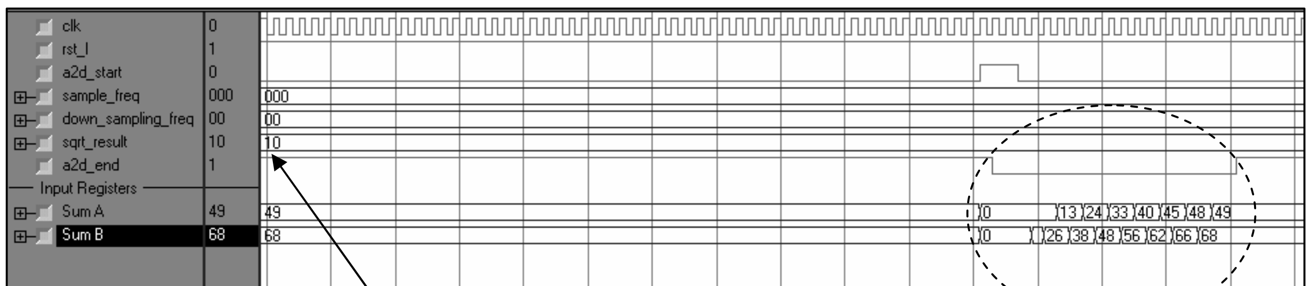
Freq_Div module

As the name implies, this is a simple frequency divider with the *start_conv* signal functioning as a reset. The sampling frequency parameters are read at reset and a predetermined number is loaded into a counter based on this value (for example the counter reload number could be 10 if the sampling frequency parameters are set to '000', 20 if parameters are '001' and so on). This counter decrements at each clock tick of the main clock and produces an output which toggles every time the counter reaches zero, thus producing a different sampling period. The counter reloads itself with the original value at this point.

6.1.2 Modelling of the ADC Module

The individual modules that were outlined in the previous section were first developed independently and tested in ModelSim using test benches. After satisfactory waveforms were obtained for the functional simulations, timing analysis were performed using Quartus II on a EP20K200EFC484-2X, APEX20KE family chip in order to ensure that the modules could actually be implemented in hardware.

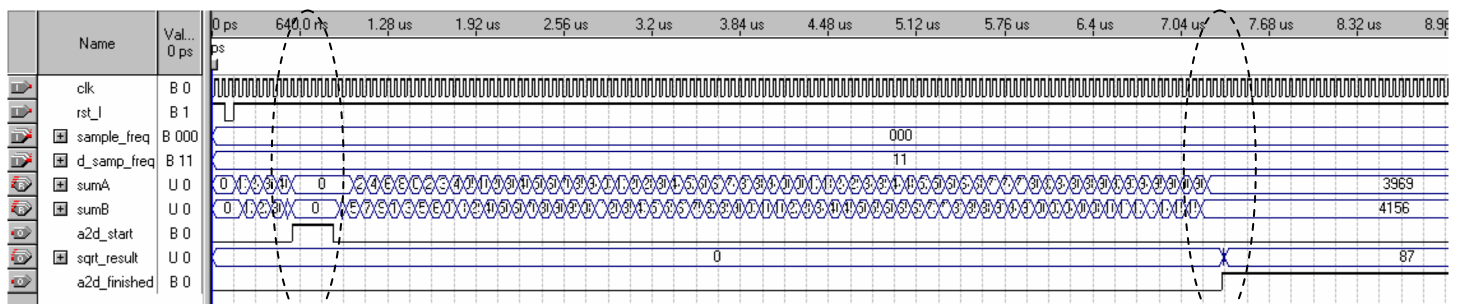
Shown in Figure 05(a) is the functional simulation of the complete ADC module obtained using ModelSim and Figure 05(b) is a waveform of the timing analysis performed in Quartus.



Actual square root should be $\sqrt{(49/8)^2 + (68/8)^2} = 10.47$.
Using approximation algorithm this is approximated to 10.

8 samples taken into account since the down sampling ratio is set to '00'

(a)



Input indicating 'start ADC'

Square root is calculated here

(b)

Figure 05: (a) Functional simulation (b) Timing simulation of the ADC module using ModelSim and Quartus II respectively

As seen in the above waveform, the encoded down sampling ratio specified in this example is '11' which corresponds to 64 samples. The totals of the two set of samples are 3969 and 4156 respectively. The averages are 3969/64 (=62) and 4156/64 (=64.9), and the square root of the sum of squares of these two values is: $\sqrt{62^2 + 64.9^2}$, which is equal to 89.75. The values given is 87 which has a margin of error of approximately 3%.

However if a down sampling ratio of 8 is specified, the calculated error would be approximately 4.5% as shown in the Figure 05(a). Time delays between raising the *ADC_finished* signal and producing the actual square root value on the output bus are shown in the waveforms given in Appendix A.

6.1.3 Getting Online Parameters

Online parameters (down sampling ratio and the sampling frequency) were to be fed into a predetermined location of MiCORE's RAM during runtime based on an external request. To meet this requirement, an external input line was added to MiCORE's existing architecture where the user was expected to send a trigger signal if parameters needed to be updated. Also 2 external pins were provided to specify the desired down sampling ratio and 3 pins were provided to specify the variable sampling frequency. These 6 new inputs to MiCORE were connected to 6 pins of the memory mapped input port SIR, as described in the next section. The encoded values on these lines are later decoded as follows:

Down sampling ratio (2-bits)	Decoded value (i.e. number of samples to add in the square root calculation)
00	64
01	32
10	16
11	8

Table 03: Decoded values for down sampling ratio

Sampling frequency (3-bits)	Decoded value (i.e. number of main clock ticks to count in the 'freq_div' module)
00	10
01	20
10	30
11	40

Table 04: Decoded values for sampling frequency

The locations to store the down sampling ratio and the sampling frequency were arbitrarily chosen to be 0xFFFE and 0xFFFD respectively.

6.1.4 Message Formats and Hardware – Software Integration

The next stage of the development process was to integrate the ADC module with the existing core and use assembly programming to poll the external request. This external request had to be polled in an infinite loop since any major modifications could not be added to the existing MiCORE for the first part of the project.

As discussed previously the ADC parameters and the external request to update parameters are sent from external sources, hence the memory mapped input port SIR was used to input these values into MiCORE. Also the 9-bit output of the ADC module which contains the square root value needed to be sent into MiCORE's RAM so that this information was available to a high end user. It was arbitrarily decided to store this square root value at location $FFFC. The following message format was used for SIR:

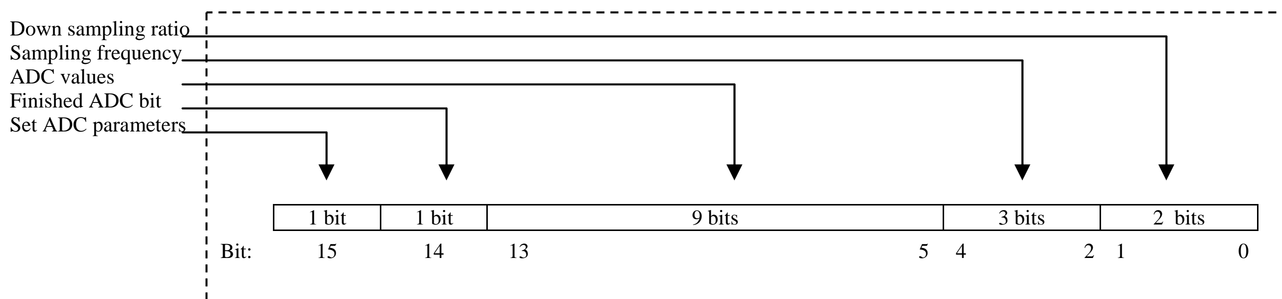


Figure 06: MiCORE input mapping to SIR

Also the user specified ADC parameters needed to be sent into the ADC module along with a start signal indicating to the ADC module to start a conversion. The memory mapped output port SOR was used for this purpose with the following message format.

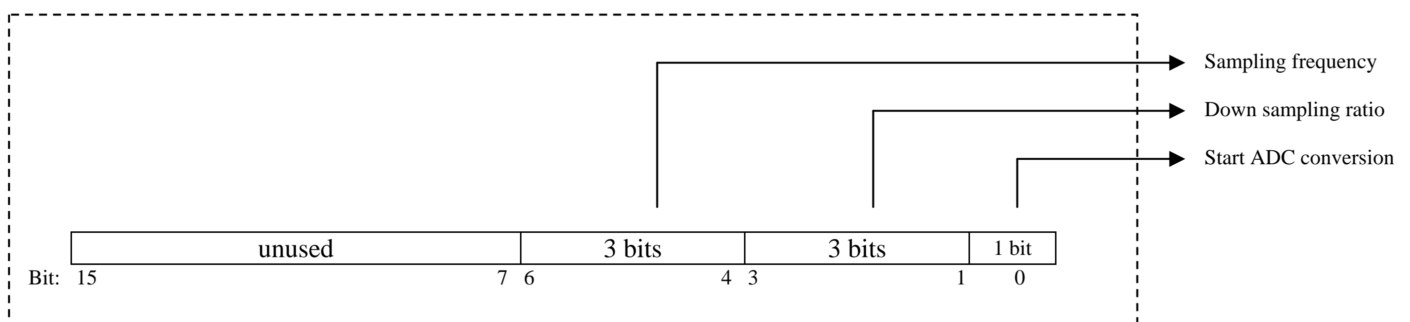


Figure 07: MiCORE output mapping to SOR

The complete system with the above described I/O mappings is shown in Figure 08.

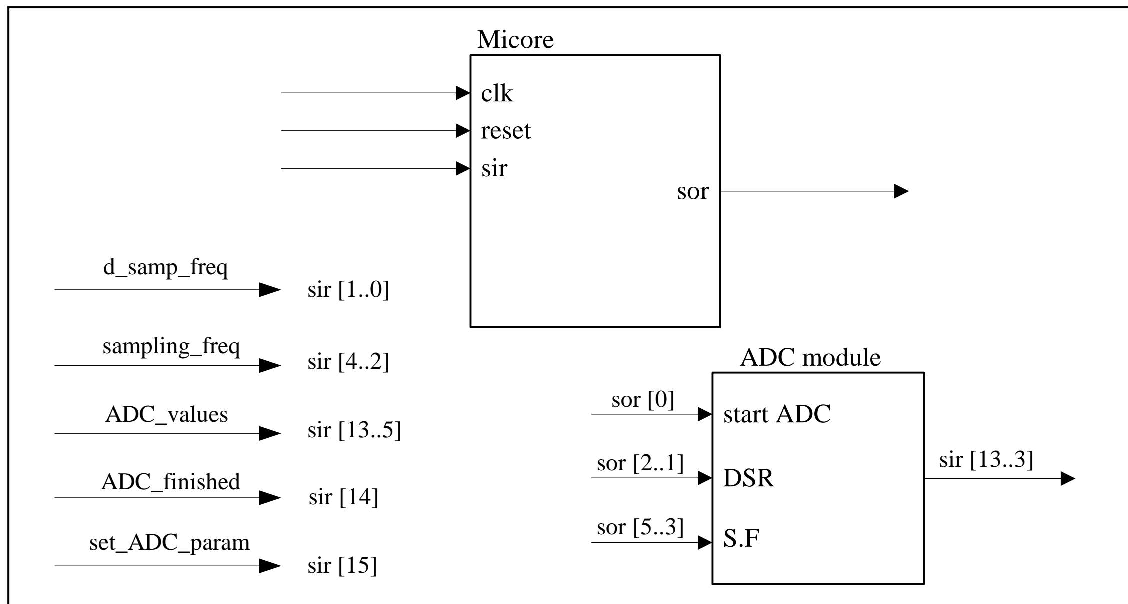


Figure 08: The complete system with I/O mapping

In order for the above system to run continuously while monitoring the I/O lines, assembly code was written and outlined below is the flow of this program. The actual code is included in Appendix B.

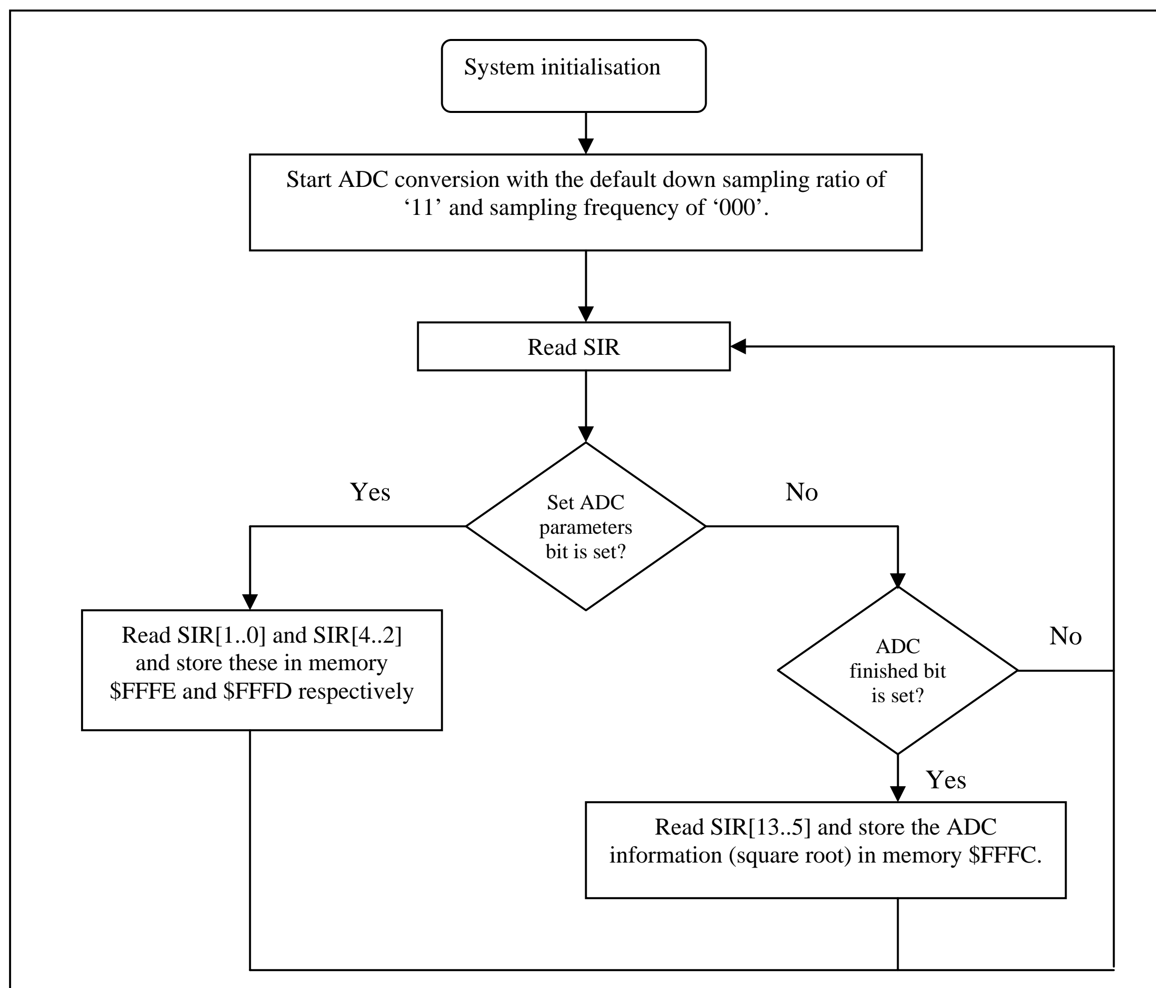


Figure 09: Assembly program flow for the first task

The complete system was simulated and after much testing, error checking and debugging, the following waveform was obtained which shows a timing simulation of the MiCORE with the ADC integrated. It also shows how the values sent on SIR are copied to the corresponding memory locations based on the bit positions shown in Figure 08.

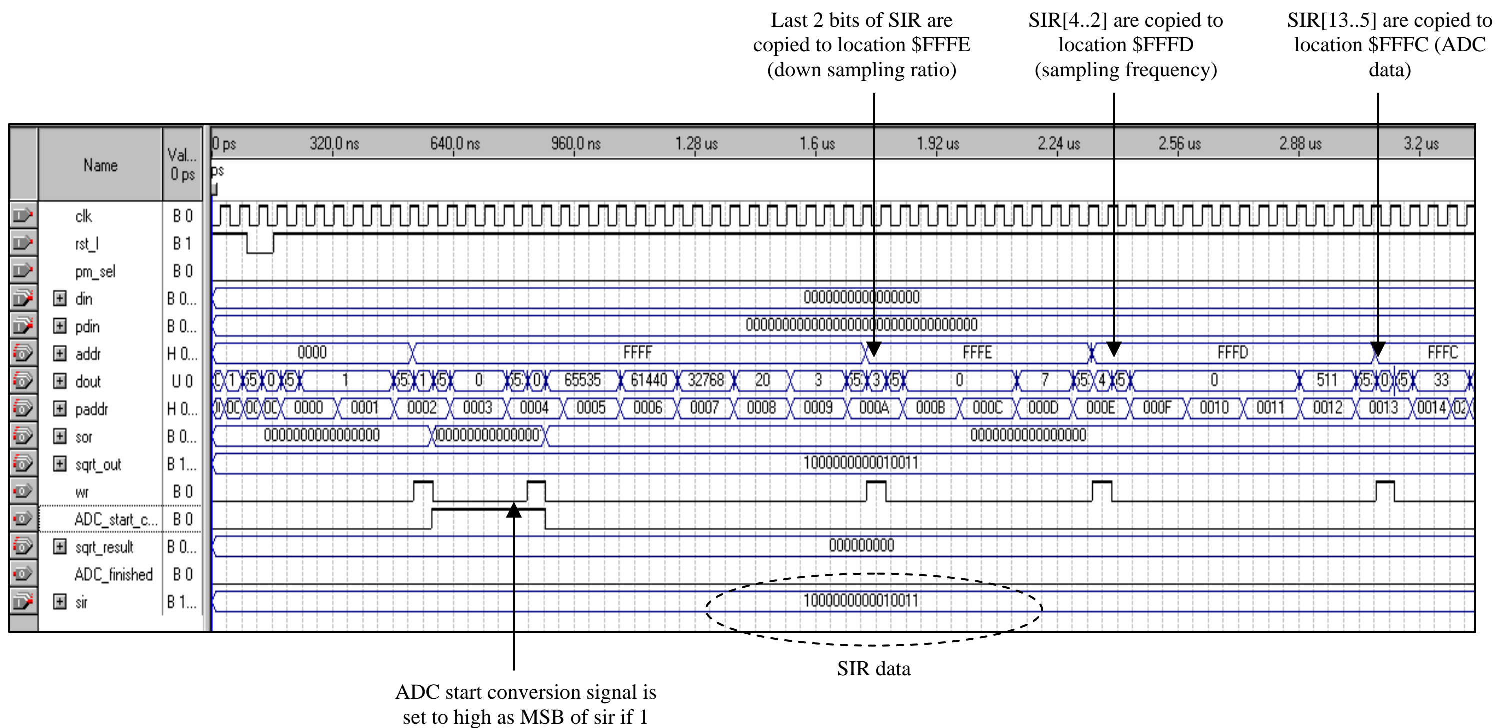


Figure 10: Timing simulation (on a EP20K200EFC484-1X, APEX20KE family chip) of MiCORE with ADC implemented

6.2 Implementation of the Microcomputer with Interrupts

This was the second main task involved in the project and was considered very challenging because a new 4-level interrupt structure had to be implemented in the existing MiCORE. This new design was to be called I-MiCORE and suitable algorithms were to be developed and implemented to achieve the required functionality.

Although 4 interrupts were to be implemented, only 2 were to be triggered during the course of the project. These two interrupts were to be an external interrupt indicating to the processor that ADC parameters need to be updated, and the second was to be an interrupt (with a lower priority than the first) which was to start an A/D conversion. The processing of both the interrupt requests were rather similar to the first task, but needed to be executed within the interrupt service routine (ISR).

6.2.1 Interrupt Design Criteria

The following issues were considered for interrupts during the design phase:

- A 4-level interrupt structure had to be implemented with the external *ADC update* request having the highest priority. The 4 interrupts were named INT0, INT1, INT2 and INT3 with INT0 having the highest priority level. The second interrupt which triggers the ADC was assigned to INT1 while the other 2 were to be implemented but not connected.
- Any interrupt can trigger at any given time.
- While a high priority interrupt is being serviced, the lower priority ones have to be disabled. However these lower priority ones need to be queued and serviced when the high priority interrupt finishes.

- If a high priority interrupt triggers while a low priority one is being serviced, the low priority interrupt should be interrupted and the higher priority one should be serviced. After this finishes, the low priority ones ISR should continue from where it stopped.
- An interrupt should be serviced after the current instruction has been finished executing. This means that although an interrupt can trigger at any given time (during fetch, decode or execution stage), the earliest it will be services is at the end of the execution stage of the current instruction.
- An interrupt can trigger again while the same interrupt is being serviced but this should not trigger the current interrupt execution process. For example if INT1 is currently being serviced and if INT1 triggers again, this request should be ignored and the current INT1 process should run uninterrupted.
- User is expected to push the appropriate registers into stack (within the ISR) if any of the register values are to be preserved. This was considered a valid assumption since in the original MiCORE instruction set the JSR instruction has been implemented with the same assumption. Therefore only the Program Counter (PC) was decided to be pushed into stack when an ISR was to be serviced.

6.2.2 Interrupt Priority Resolution Algorithm

Resolving priority and achieving the above requirements needed a well developed algorithm that took all the possibilities into account. This section details out the algorithm developed for this purpose.

The basic concept behind the developed algorithm is shown below in Figure 11.

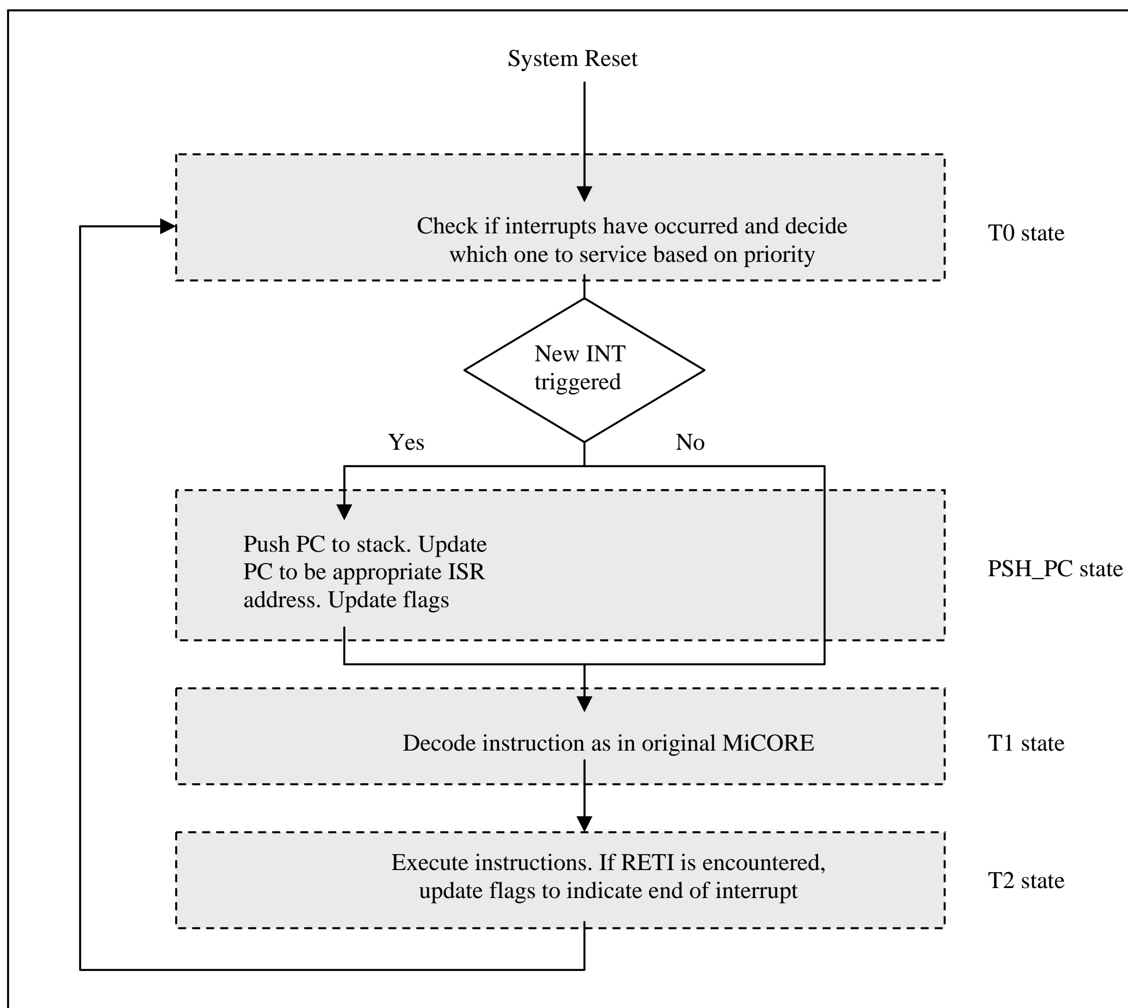


Figure 11: A flowchart showing the basic program flow during interrupts

The RETI (return form interrupt) instruction shown above has been added to the existing instruction set using a similar implementation to the existing instruction RET. The only difference is that the user is expected to use this RETI at the end of an ISR where as RET is to be used at the end of subroutines. The new instruction has been added by slightly modifying the assembler source code written in Lexx and Yacc and this code is given in Appendix C.

Two flag registers were used in the 'datapath' and 'controlunit' modules as shown below.

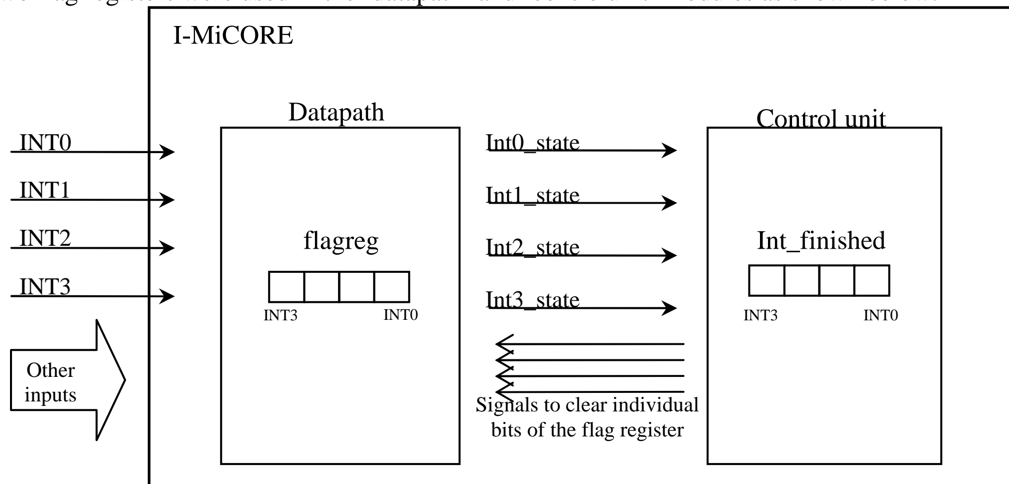


Figure 12: Internal flags and signals used for interrupt priority resolution

The *flag_reg* variable is defined within a process in the 'datapath' which is sensitive to the external interrupts INT0, INT1, INT2 and INT3. The corresponding bits (i.e. bit0 to INT0, bit1 to INT1 etc.) are set if any of the interrupt lines trigger and would remain high until a clear signal is sent by the control unit.

The *INT_finished* variable in the 'controlunit' monitors the INT0_state, INT1_state, INT2_state and INT3_state control signals set by the 'datapath' which are simply the signals with the values of the *flag_reg* variable. The internal variable *INT_finished* is initialised to "1111" and if any of the INT(0-3)_state input signals is high while all the *INT_finished* bits of higher priority interrupts are also high, it indicates that this is an interrupt that should now be processed. Priorities are processed by using sequential statements and this process is described later. The bit checks are done at each T0 (or FETCH) state of the instruction execution process and the next state is set to PSH_PC which is a new state that has been introduced to the original MiCORE. This state was required since the PC has to be pushed into the stack before jumping to the ISR. In the PSH_PC state, PC is written into stack (which takes 1 clock cycle) and the PC reload value is set to the hard-coded address of the ISR by setting the PC_in_sel multiplexer control signal appropriately. This algorithm has been implemented by slightly modifying the 'datapath' and the 'controlunit' of the existing MiCORE, and are discussed below.

6.2.2.1. Datapath Modifications

A new process has been added to the 'datapath' to detect any interrupts that trigger and set the *flag_reg* variable. The respective bit is set if an interrupt triggers and remains high until cleared by the 'controlunit'. Also the PC input selection multiplexer has been extended as shown in Figure 13 to accommodate the ISR addresses of the 4-interrupts. These new ISR address definitions have been added to the "various_constant.vhd" file and can be modified to different locations if required.

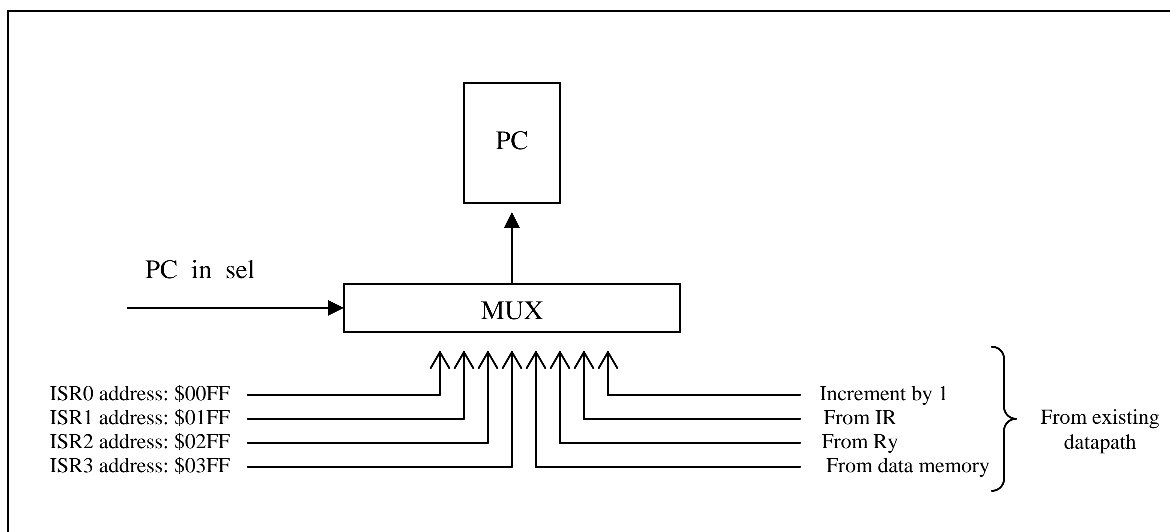


Figure 13: PC input multiplexer extension to accommodate four new addresses

6.2.2.2. Controlunit Modifications

A few modifications have been added to the 'controlunit' and given below is the pseudo code used to implement the algorithm explained in Section 5.2.2. This has been included in the 'opdec' process of the control unit:

```
Initialisation: INT_finished="1111"

State T0 =>
        If: INT_finished(0) ='1' and INT0_state='1'  -- if true: ISR0 should be executed
                Set control signal to push PC into stack, Set next state to be PSH_PC
        Else if: INT_finished(0)='1' and INT_finished(1)='1' and INT1_state='1' --if true: execute ISR1
                Set control signal to push PC into stack, Set next state to be PSH_PC
        Else if: INT_finished(0)='1' and INT_finished(1)='1' and if INT_finished(2)='1'  and INT2_state='1' --
                        if true: execute ISR2
                Set control signal to push PC into stack, Set next state to be PSH_PC
        Else if: INT_finished(0)='1' and INT_finished(1)='1' and if INT_finished(2)='1'  and
                        INT_finished(3)='1' and INT3_state='1' --if true: execute ISR3
        Else:
                Set next state to T1 and process as normal –no interrupt to service

State PSH_PC =>
        Set next state to T1, set control signals to write to stack and make source = PC
        If: INT0_state='1'
                Set INT_finished(0) ='0', Select PC_source = ISR0 address ($00FF)
        Else if: INT1_state='1'
                Set INT_finished(1) ='0', Select PC_source = ISR1 address ($01FF)
        Else if: INT2_state='1'
                Set INT_finished(2) ='0', Select PC_source = ISR2 address ($02FF)
        Else if: INT3_state='1'
                Set INT_finished(3) ='0', Select PC_source = ISR3 address ($03FF)

State T1 => decode instructions.
        No modifications added
```

State T2 => if opcode is RETI – return from interrupt
If: INT0_state='1'
Set INT0 clear signal high
Else if: INT1_state='1'
Set INT1 clear signal high
Else if: INT2_state='1'
Set INT2 clear signal high
Else if: INT3_state='1'
Set INT3 clear signal high

A flowchart of the above algorithm can be found in Appendix D.

6.2.3 Implementation of I-MiCORE

According to the project specifications, INT0 of the above algorithm was to be the input indicating that ADC parameters needed to be updated, and was to have the highest priority. The second interrupt was to be a signal that required the ADC module to perform a complete 1-shot ADC sweep and produce the square root output. Using the above algorithm, an interrupt structure has successfully been implemented in VHDL and shown below are some of the timing simulation results obtained using Quartus II. Brief descriptions of the waveforms are also included.

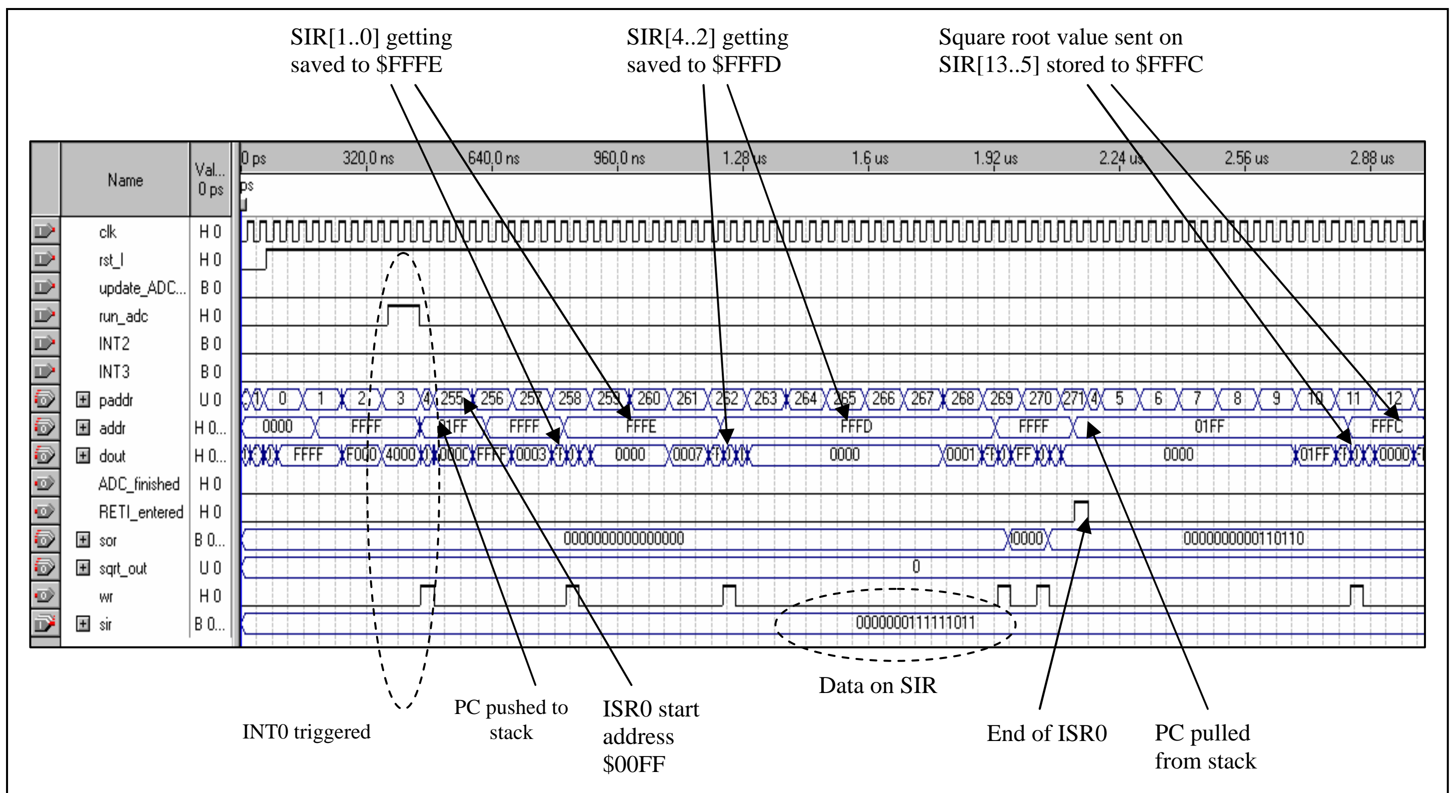


Figure 14: Output waveforms when INT0 is triggered

Shown above in Figure 14 is the input-output waveform when INT0 is triggered. As indicated by the arrows, PC is pushed into the stack and the interrupt is serviced as described in previous sections. The waveform also shows how the ADC parameters are extracted from SIR and stored into their repective memory locations. The assembly code given in Appendix E was used to obtain the above waveform.

The same code used to obtain the above waveform was used to simulate the functionality of INT1, but with ISR1 which starts at $01FF. This interrupt routine simply reads the down sampling ratio and the sampling frequency from their respective memory locations and write to SOR along with a start conversion trigger. Shown below in Figure 15 is a waveform depicting this scenario.

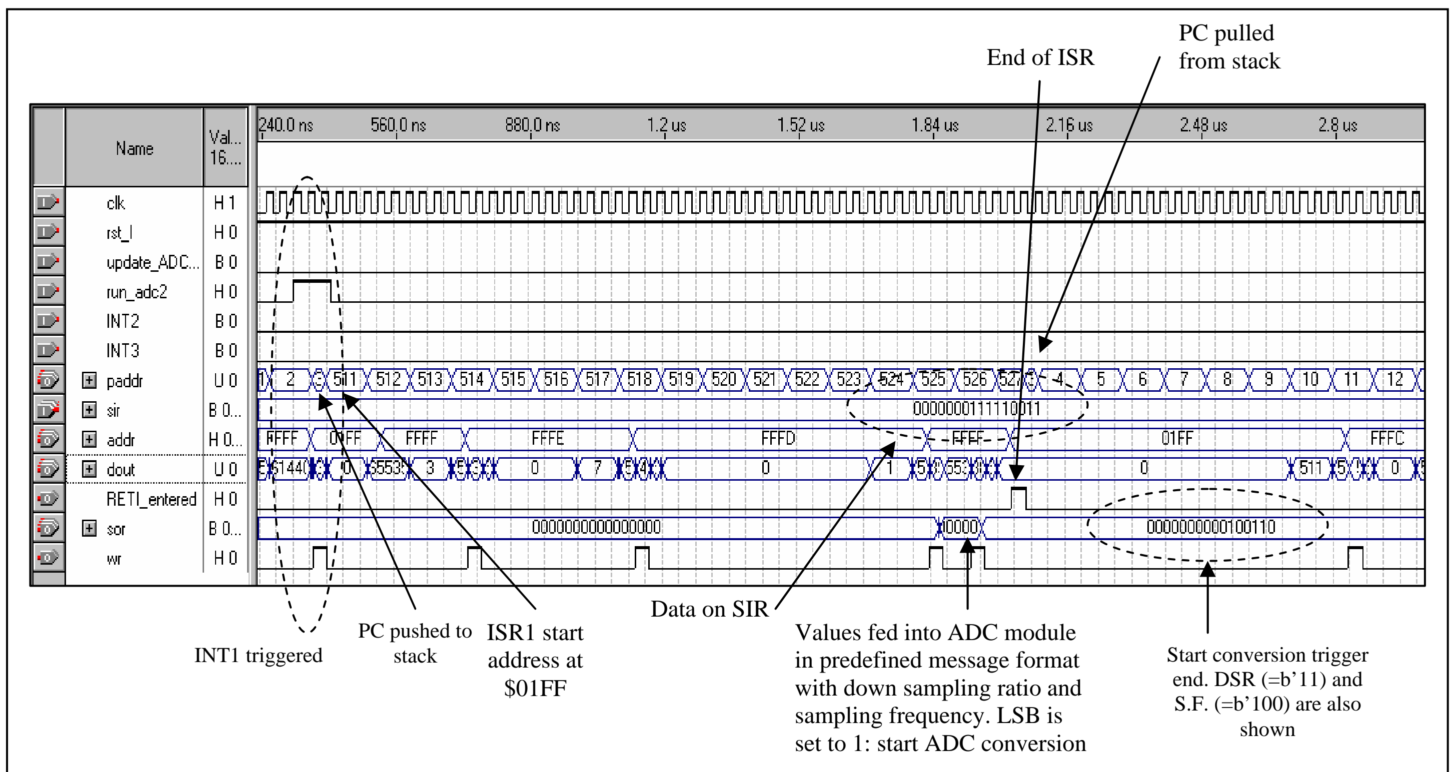


Figure 15: Output waveforms when INT1 is triggered

Figure 16 shows what happens if a higher priority interrupt triggers while a lower priority one is being serviced. As expected, the higher priority one interrupts the execution of the lower priority one and at the end of this higher priority interrupt's ISR, the lower priority ISR resumes before giving control to the main loop. Other 2 interrupts also have been implemented in a similar way.

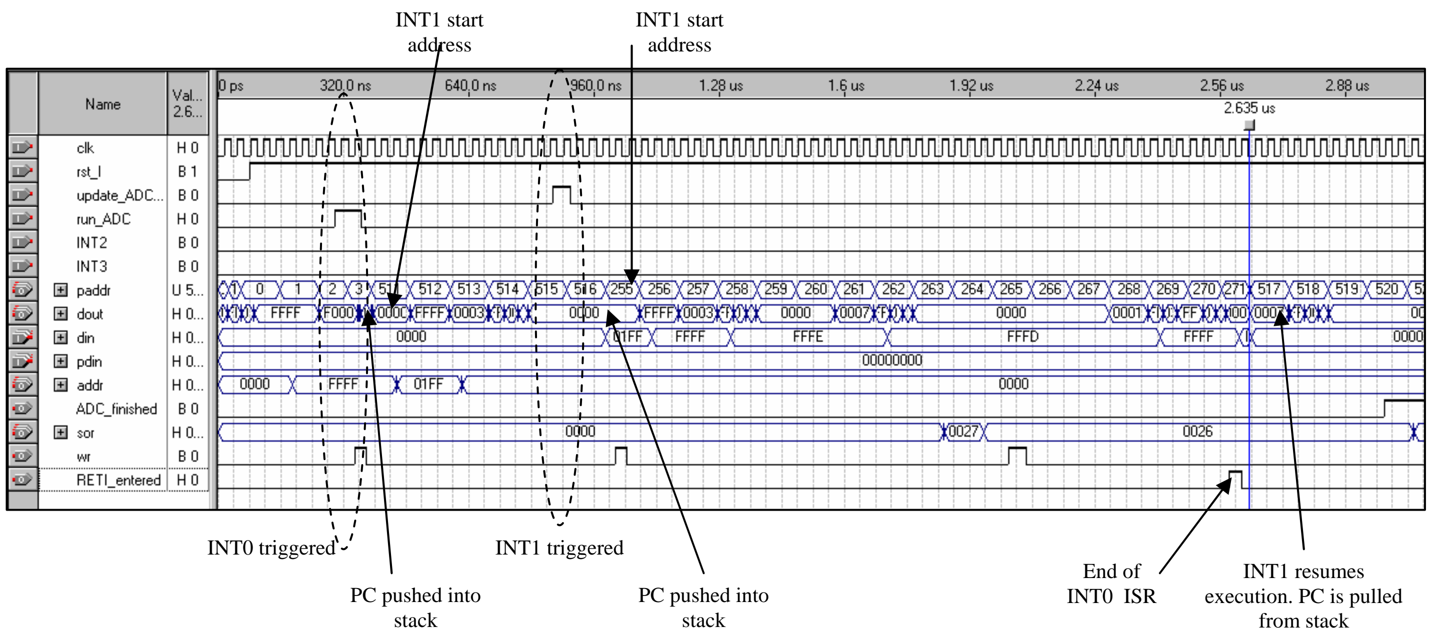


Figure 16: Waveform showing priority resolution

6.3 A Brief Comparison between Polling and Interrupts

The worst case scenario with polling occurs when the 'set ADC parameters' bit is set as soon as instruction that checks this condition finishes execution. In the Assembly code given in Appendix E, the number of instructions executed if this condition is not true is 6. This means that if the bit is set as soon as the test condition has finished, it'll have to wait for 5 instructions to be executed before detected. Execution of 5 instructions takes 15 clock cycles, hence the delay is 15 times the period.

However with interrupts the worst case scenario occurs if an interrupt triggers at the end of the T0 state as soon a new instruction has been fetched. This means that the worst case delay is the time taken for 3 states to be executed. Since each state takes one clock cycle, the worst case delay with interrupts is only 3 times the period, which is much less than the delay with polling.

The following information about MiCORE and I-MiCORE were obtained using the reports generated by the synthesiser.

	MiCORE	I-MiCORE
Speed	34.65 MHz	30.92 MHz
Logic cells	993	846
Registers	191	194
Memory bits	50688	50690

Table 05: Resources utilised by MiCORE and I-MiCORE

6.4 Distribution of Tasks among Group Members

	Task Breakdown	**Contribution of each Team member**			
		Amal	**Chinthana**	**Kasun**	**Thusitha**
1	Background research and understanding	Major	Major	Major	Major
2	Implementation of ADC modules and sub modules	Average	Major	Average	Average
3	Software algorithm to enable ADC computation	Minimal	Average	Average	Major
4	Integration of ADC with existing MiCORE	Average	Major	Major	Major
5	Assembly programming	Average	Minimal	Major	Average
6	Implementation of Interrupts	Major	Average	Average	Major
5	Integration with the existing MiCORE to design I-MiCORE	Major	Average	Major	Major
7	Generating various waveforms and synthesis reports	Major	Minimal	Major	Average
8	Testing and analysing of waveforms	Average	Minimal	Major	Major
9	Report	Average	Minimal	Average	Major

Table 06: Table showing distribution of tasks among group members

7.0 Conclusions

The implemented final solution meets all the required project specifications. User is allowed to update the down sampling ratio and the sampling frequency of the ADC module while preserving the current architecture of MiCORE. Also the an ADC conversion can be started by signalling on SIR, the memory mapped input port.

The second task of the project has also been successfully implemented by introducing a 4-level interrupt structure to MiCORE, to from I-MiCORE. The first interrupt, which also has the highest priority is the external user request where the user is allowed to update the ADC parameters. Second interrupt source is used to start an ADC conversion while interrupts 3 and 4 are left without any specific inputs. However these have also been implemented and can be triggered if required.

ModelSim and Quartus II have been used as design tools and all modules have successfully been compiled and synthesised. Timing analysis have been performed and the maximum frequency the system can be run at was determined to be 35.87MHz. The resource usage on a EP20K200EFC484-2X, APEX20KE family chip was estimated to be 1001 logic cells, 192 registers and 50688 memory bits.

Bibliography

- *EVITA-VHDL tutorials by Aldec.* Retrieved 1 August, 2004 from http://www.aldec.com/downloads/
- Salcic, Z., *FLIX - A CUSTOM CONFIGURABLE MICROCOMPUTER.,* Kluwer Acedemic, 1998
- Salcic Z, Hui D., *MiCORE – A Customisable Microprocessor Core.* 2004

Appendix A: ADC module's output waveforms

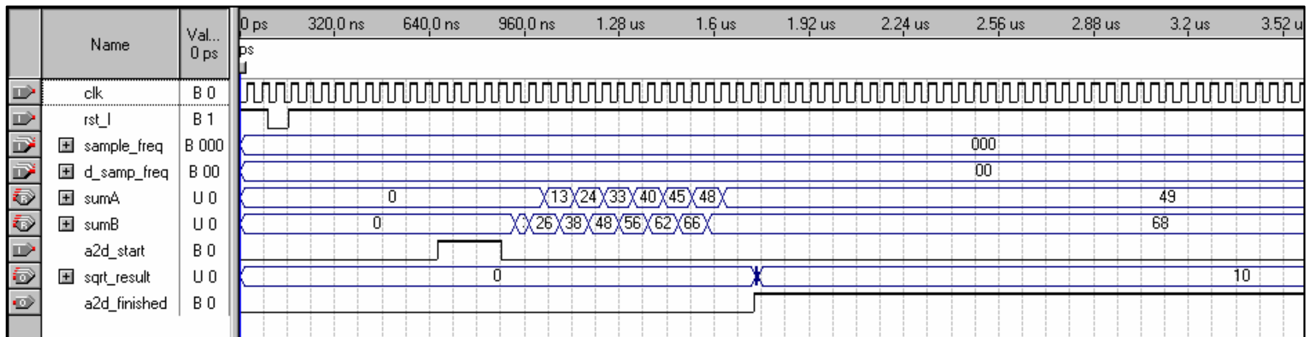


Figure A.1: Waveform showing ADC output using 8 samples

As seen in the above waveform, the encoded down sampling ratio specified in this example is '00' which corresponds to 64 samples. The totals of the two set of samples are 49 and 68 respectively. The averages are 49/8 (=6.125) and 68/8 (=8.5), and the square root of the sum of squares of these two values is: $\sqrt{6.125^2 + 8.5^2}$, which is equal to 10.47. The values given is 10 which has a margin of error of approximately 4.55%.

If a down sampling ratio of '01' (or 16 samples) is used, the error reduces to approximately 4% as shown in the waveform below.

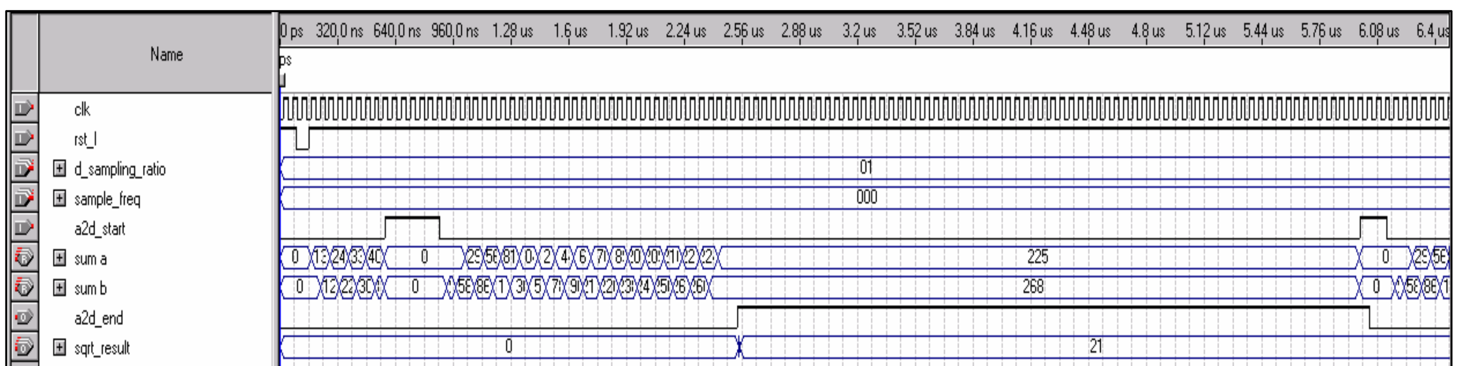


Figure A.2: Waveform showing ADC output using 16 samples

The delay between raising the *ADC_finished* signal and presenting the square-root to the output lines was also measured and as shown in the following waveform, was measured to be only around 20ns.

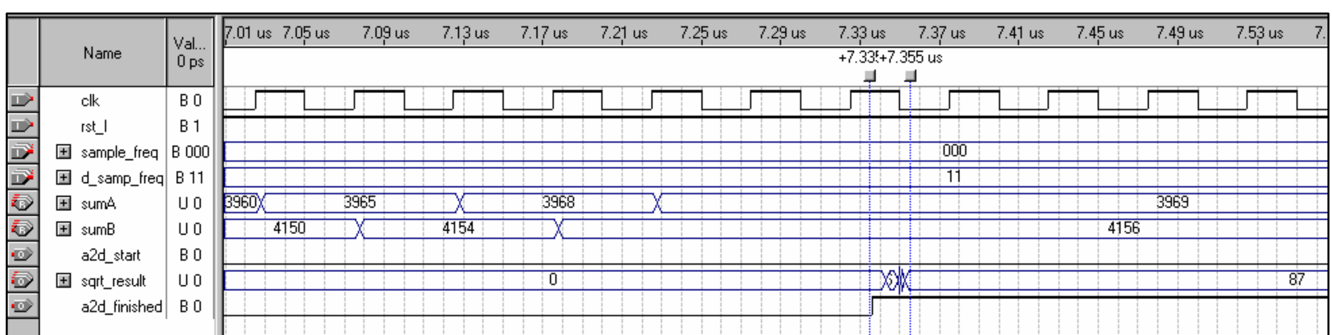


Figure A.3: Waveform showing delay between raising the *ADC_finished* signal and presenting the square-root to the output

Appendix B: Assembly code written for task1

```
; Date modified : 01/10/2004
; Polling code for the ADC module

              &ORG $0
              LDR R1 #1                ; set the start bit (user data)
              STR R1 $FFFF             ; store at SOR

START         LDR R1 #0                ; set the start bit to zero to create a transition
              STR R1 $FFFF             ; store at SOR
              LDR R0 $FFFF             ; load SIR to R0
              CLF CZVN                 ; clear the zero flag
              AND R1 R0 #32768         ; test if set ADC parameter bit is set
              SZ BRANCH1               ; branch to the location if the condition is not correct
              AND R1 R0 #3             ; get the first two bits (d_samp_freq)
              STR R1 $FFFE             ; store d_samp_freq at the given location
              RSH R1 R0                ; shift right 2 times
              RSH R1 R1                ;
              AND R1 R1 #7             ; get the first three bits (sampling frequency)
              STR R1 $FFFD             ; sotre sampling frequency at the given location
              RSH R1 R0                ; shift 3 times
              RSH R1 R1                ;
              RSH R1 R1                ;
              AND R1 R1 #511                   ;
              STR R1 $FFFC             ; storing adc values
              JMP END                          ; go to the end once in the body has been
executed
                                       ;
BRANCH1              AND R1 R0 #16384          ; compare to test if set finished adc is set
              SZ END                   ; branch to the location if the statement is not correct
              LDR R0 $FFFE             ; load DS ratio
              LSH R1 R0                ; shift one to the right
              OR R1 R1 #1              ; shift to get the DS ratio and add it to R2
              LDR R0 $FFFD             ; load sampling frequency
              LSH R0 R0                ; shift 3 to the right
              LSH R0 R0                ;
              LSH R0 R0                ;
              OR R2 R0 R1              ; add to R2
              AND R3 R2 #$FFFE         ; to make a transition load the first bit with zero
              STR R2 $FFFF             ; write to SOR
              STR R3 $FFFF             ; set to 0 to make a transition
                                       ;
END           JMP START                ; loop back to the main program
              &END                     ; this is the end of the program
```

Appendix C: Modified Lexx and Yacc code to implement RETI

mc2bb.y modifications:

--Add the following lines after RET
const unsigned long RETICODE = 0X04000000; --at the beginning

```
| RETIt                     { fprintf(codefile, "\t\t%X\t:%08X;\t%%RETI%%\n", --towards
the end
                                                    location_counter,
                                                ST_AM + RETICODE);
                                          location_counter++;
                                         }
```

mc2fb.txt modifications:

--Add the following line between RET and NOOP at the beginning
RETI/{SEPARATOR}   return RETIt;

Controlunit modifications:

```
when T2 =>
                    next_state <= T0;
                    if opcode(10 downto 9) = stack then
                            case opcode(8 downto 4) is
                                    when pul =>
                                            -- Rz <- dm
                                            rf_in_sel <= seldm2rf;
                                            ld_rf <= '1';
                                    when psh =>
                                            -- dm <- Rx
                                            dm_in_sel <= selrx2dm;
                                            we <= '1';
                                    when ret =>
                                            -- pc <- dm
                                            pc_in_sel <= seldm2pc;
                                            ld_pc <= '1';
                                    when jsr =>
                                            -- dm <- pc
                                            dm_in_sel <= selpc2dm;
                                            we <= '1';
                                            -- pc <- ir[24..9]
                                            pc_in_sel <= selir2pc;
                                            ld_pc <= '1';
                                    when reti => --basic instruction implementation.
Other INT control signals need to be added here.
                                            -- pc <- dm
                                            pc_in_sel <= seldm2pc;
                                            ld_pc <= '1';
                                    when others =>
                                            -- should be invalid instruction code
                            end case;

                    elsif opcode(10 downto 9) = direct then
```

opcodes.vhd modifications:

Simply add a new stack type opcode. The last 4 digits can be any combination that is not already being used. The chosen opcode for this assignment was:

```
constant reti: bit_5 := "00010";
```

Appendix D: Interrupt priority resolution algorithm flowchart

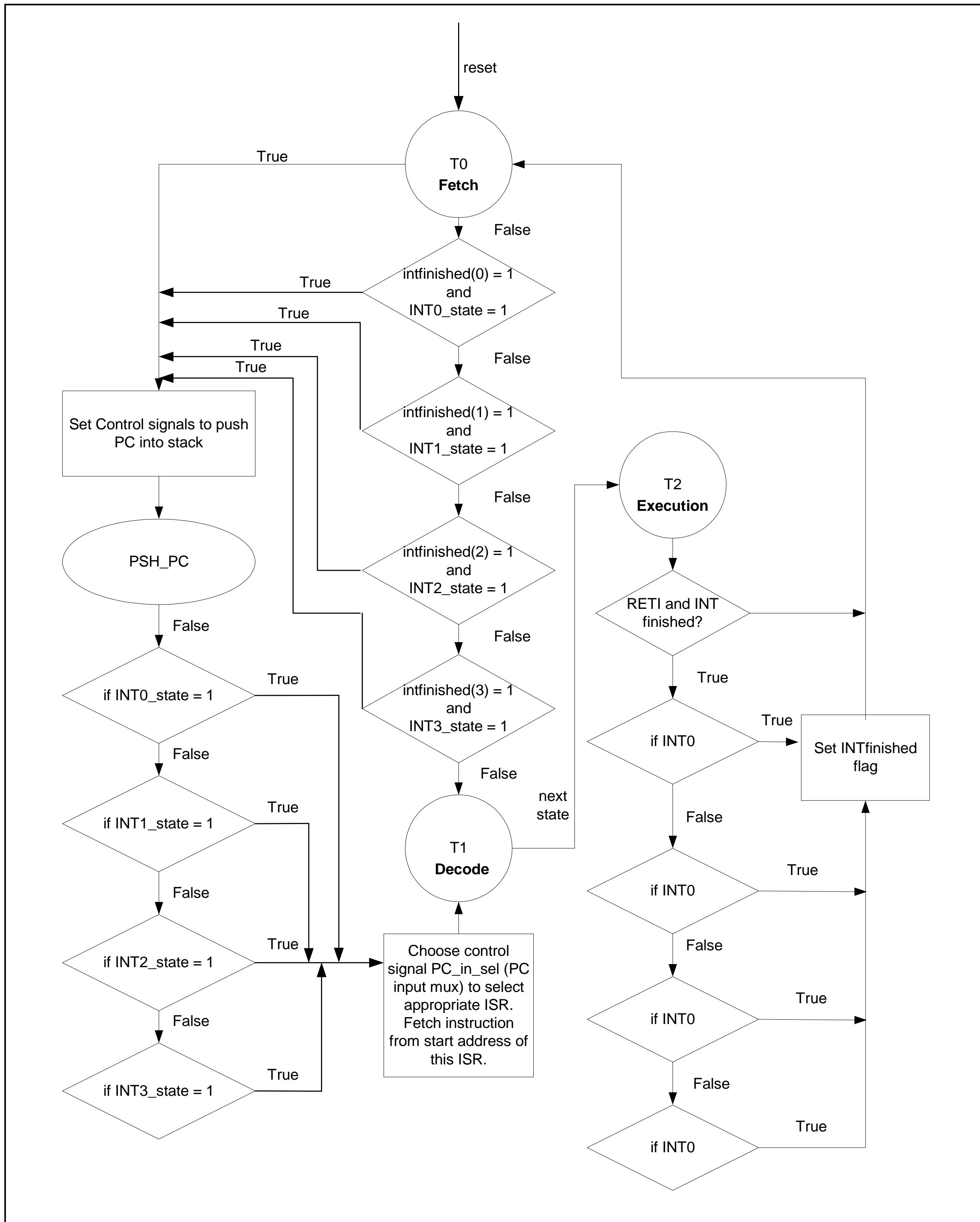


Figure D.1: Flowchart showing how multiple interrupts are handled based on priority

Appendix E: Assembly code written for task2

```
; Last Updated Date : 01/10/2004
; Assembly code to test the interrupt handling of iMicore
; First ISR (Interrupt Sub-Routine) services the updating of the ADC parameters : Down sampling frequency and
; sampling frequency.
; Second ISR (Interrupt Sub-Routine)  services ADC execution with ADC start bit

        &ORG $0                 ; start of program
START   LDR R0 $FFFF            ; read SIR and store in to register R0
        CLF CZVN                ; clear status flag register
        AND R1 R0 #16384        ; poll for ADC_finished bit to be set
        SZ START                ; loop until the bit is set
        RSH R1 R0               ; shift the register to get the 9 bits
        RSH R1 R1               ;
        RSH R1 R1               ;
        RSH R1 R1               ;
        RSH R1 R1               ;
        AND R1 R1 #511          ; bit operation to get 9 bits
        STR R1 $FFFC            ; store the 9 bits in the given location
        JMP START               ; go to start

; Uses the following SIR format :
;------------------------------------------------------------------------------------------
;        1     |     1     |     9     |         3          |          2          | bits
;set ADC param | finished ADC | ADC value | sampling frequency | down sampling ratio |
;------------------------------------------------------------------------------------------
;

        &ORG $00FF              ; Location of First ISR
                                ;
        LDR R0 $FFFF            ; read sir and store in to register R0
        AND R1 R0 #3            ; get the down sampling ratio
        STR R1 $FFFE            ; and store it in the memory
        LSH R1 R1               ; shift left to account for the zero start bit
        RSH R2 R0               ; shift right by 2 bits
        RSH R2 R2               ;
        AND R2 R2 #7            ; get the sampling frequency
        STR R2 $FFFD            ; and store it in the memory
        LSH R2 R2               ;
        LSH R2 R2               ;
        LSH R2 R2               ;
        OR R3 R1 R2             ; Write down sampling ration and sampling frequency to SOR
        STR R3 $FFFF            ;
        RETI                    ; return to main loop after servicing ISR
                                ;
        &END                    ; end of program

        ;
==================================================================================

        &ORG $11FF              ; Location of Second ISR
                                ;
        LDR R0 $FFFE            ; read register R0 with down sampling ration stored at the memory location
        LSH R1 R0               ; shift left and store in R1
        LDR R0 $FFFD            ; load R0 with sampling frequency
        LSH R0 R0               ; shift left by 3 bits
        LSH R0 R0               ;
        LSH R0 R0               ;
        OR R3 R1 R0             ; concat dsr and sampling frequency and store it in R2 register
        OR R2 R2 #1             ; replace the first bit with 1
        STR R2 $FFFF            ; write to SOR thus toggeling the first bit
        STR R3 $FFFF            ; first bit is 0
        RETI                    ; return to main loop after servicing ISR
        &END                    ; end of program
```