2 Introduction

The outcome of this project is to design a simple and low power audio processor that will be implemented on an Altera De2-115 Cyclone IV FPGA. The processor will be based on an embedded microarchitecture designed using a RISC instruction set architecture and will be adapted to work with the on-board audio CODEC. This scope of this project is to design and implement an embedded processor and to explore audio processing techniques. Techniques such as equalisation, synthesis, and compression.

The processor will function as an equaliser which will take a phone input to provide the audio signal to be equalised. The audio will then be put through a parametric equaliser which can control the gain of frequency bands, the centre frequency and the cut off frequencies of the band in real time. This will be controlled by external i/o and the output will be played through a speaker to listen to the new sound.

The processor will also function as a synthesiser. The processor will use audio synthesis techniques such as additive or subtractive synthesis to synthesise audio stored on the FPGA or linear predictive coding to synthesise speech from a microphone. Audio compression may also be added to the processor if time permitting. This would be part of the equaliser and would compress the audio signal before it is passed to the equaliser.

At the time of writing, the first version of a RISC-V processor has been designed and tested which can carry out a small number of instructions. Two versions of an equaliser have been designed for the processor, only one will be implemented and this decision will be made later depending on the capability of the designed processor.

3 Background

3.1 Processor Design

3.1.1 Microarchitecture

When designing a processor on an FPGA there are multiple options for the microarchitecture. The most common architectures in embedded systems are a Reduced Instruction Set Computer (RISC) and a Complex Instruction Set Computer (CISC).

Both RISC and CISC architectures have their uses in modern day computing. The purpose for CISC architectures is to closely model the level of abstraction in high level programming languages. This results in a great number of instructions which may vary in length as one instruction may take a vastly different amount of clock cycles to execute than another instruction [1]. This contrasts greatly with RISC architectures which are designed to execute instructions that do not vary in length and will execute in one clock cycle [2]. This results in the instruction set of a RISC processor being smaller and more concise than that of a CISC processor as a RISC instruction set has fewer redundant instructions which are rarely needed to be executed. For example, the x86-64 CISC architecture has 981 different instructions [3] compared to the 47 base instructions in RISC-V.

This allows a RISC processor to perform better at instruction scheduling as all the instructions take the same time to execute, pipelining is a method of instruction scheduling and can be implemented in a RISC design. Pipelining breaks up the steps of executing an instruction into independent operations, this allows the processor to act in effect as an assembly line which improves processor performance by increasing the number of instructions being executed at one time [4]. RISC processors will also have a simpler design than a CISC processor as there are fewer addressing modes and simpler control signals in the processor itself. This means that RISC is more suited to be part of a simple and low power design than CISC. However, a drawback of this is RISC programs end up being longer on average than a CISC program. This results in more memory being required [5]. Nevertheless, for the purpose of this processor, RISC is more suitable than CISC.

3.1.2 Instruction set Architecture

There are many RISC Instruction Set Architectures (ISA) available. Two common general-purpose architectures are MIPS (Microprocessor without Interlocked Pipeline Stages), and the more novel RISC-V. A crucial aspect of this project is for the processor to be able to carry out complex arithmetic such as convolution and Fast Fourier Transform (FFT) as these are vital parts of digital signal processing techniques. Because of this the chosen instruction set must have multiplication instructions in the base instruction set or in an extension. Both MIPS and RISC-V offer a multiplication extension, however due to RISC-V being an open source and more novel ISA which is designed to be the future is embedded processors, RISC-V is a more suitable ISA for this project.

The programs for MIPS and RISC-V are written by hand in machine code so they are complicated and time consuming to design for, to alleviate this there is a proprietary ISA for Altera FPGAs called NIOS II which is a soft-core architecture designed to allow C programs to be synthesised onto the FPGA. This allows for potentially more complex

programs due to designed for the processor as the program doesn't need to be converted into machine code by hand. However, NIOS II implementations often are more expensive in terms of FPGA logic elements than an ISA such as MIPS or RISC-V, so RISC-V is the most suitable ISA for this project.

3.2 Equalisation

3.2.1 Equalisers

Equalisation is an audio processing technique that is crucial in modern music production. Equalisers are used in various settings that range from radio stations to instrument amplifiers and are used to adjust the amplitude of different parts of the frequency spectrum.

Equalisers are comprised of banks of filters and there are two main types of equalisers, the graphic equaliser, and the parametric equaliser. Graphic equalisers are comprised of several filters with fixed bandwidth where only the amplitude (gain) of each filter can be changed. Whereas parametric equalisers are usually 3 or 4 filters which has changeable bandwidth, cut-off frequency and gain this allows for greater precision over the frequency range.

3.2.2 Filters

There are two options for the filters used in a digital equaliser, the Infinite Impulse Response (IIR) filter, and the Finite Impulse Response (FIR) filter. The two types of filters have similar mathematical definitions however they differ in their response. As in the name an IIR has infinite response, this is shown by the infinite sum in equation 1 and an FIR has finite response, shown in equation 2.

$$y(n) = \sum_{k=0}^{\infty} h(k)x(n-k)$$

Equation 2: IIR Equation

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

Equation 1: FIR Equation

Because of this FIR and IIR filters perform and are implemented differently, in general to achieve the same filtering performance an IIR filter requires a lower filter order [6] this is due to FIR filters having a linear phase response. This is beneficial in an embedded processor as having a lower filter order lowers the amount of memory needed to store coefficients. On the other hand, FIR filters are designed using convolution and an IIR filters are designed using recursion and they rely on previous values this results in FIR filters being simpler to implement and more stable when compared to an IIR filter of the same performance [7].

Another consideration must be made regarding the calculation of the filter coefficients, the value of 'h' in Equation 1 and 2. In a parametric equaliser the filter properties are dynamically variable, this means that the processor must be able to calculate the new coefficients whenever the cut-off frequency or bandwidth is changed, the coefficients of an IIR can be calculated using methods such as impulse invariance or bilinear transformation. In a graphic equaliser the coefficients are fixed so can be generated in a

program such as MATLAB and then stored in the processor's memory. This would result in the parametric equaliser using less space in memory as the coefficients are calculated dynamically.

Taking this all into account, a parametric equaliser made of 3-4 IIR filters would be most suited for this project as it is a more precise equaliser than a graphic equaliser and using 3-4 IIR filters would result in less memory being used as less coefficients are required compared to a FIR graphic equaliser. However due to the increased calculation complexity of using IIR filters and calculating the coefficients dynamically this may not be feasible. Due to this, a FIR graphic equaliser with 10 bands should be implemented instead, however this has a cost in terms of memory usage and equalisation precision although the performance of each individual filter may be greater. This is a decision that will be made at a later stage in the project.

3.3 Synthesis

3.3.1 Speech Synthesis

Linear Predictive Coding (LPC) is a common method to synthesise speech in digital systems. LPC works by trying to estimate resonance in speech signals. Coefficients can be calculated which correspond to the resonance and spectral envelope of the speech signal. These coefficients can then be used in digital filters to recreate the speech synthetically [8]. The maths behind a linear predictor is very similar to that of an FIR filter shown in equation 2, this is shown in equation 3. This shows that speech can be predicted as a product of past speech samples. To implement this the filter coefficients will need to be calculated dynamically using techniques such as Gauss-Jordan Elimination or the Levinson-Durbin algorithm. In a similar vain to the parametric equaliser outlined earlier this may be unfeasible to calculate onboard the processor.

$$s(n) = \sum_{k=1}^{p} a_k s[n-k]$$

Equation 3: Predicted Speech Signal (Sourced from Lecture 6: LPC [9])

3.3.2 Additive and Subtractive Synthesis

Additive and subtractive synthesis are two techniques that are available to synthesise audio signals. As the name suggests additive synthesis works by adding together sine waves to create a tone and subtractive synthesis works by subtracting or filtering out parts of an audio signal to create a new signal. Both techniques would require audio signals to be stored on the processor to be able to be synthesised. To implement this the MIDI protocol may need to be implemented on the processor to obtain the musical information of the audio signal for use in synthesis [10].

3.3.3 Synthesiser Design

The final design of the synthesiser will be decided upon later due to concerns about processor capabilities and synthesis algorithm complexity.

4 Report on Technical Progress

4.1 Processor Implementation

4.1.1 Version 1 Design

At the point of writing, a 2-stage pipelined RISC-V processor has been designed to carry out the base instruction set RV32I [11] and has been inspired by the architecture shown in Figure 1 and from the lecture notes of ELEC6234 [12]. The first version of this has been implemented using System Verilog and tested using Modelsim. Figure 1 shows a RISC-V computer architecture with pipelining registers as shown as the shaded blocks, these allow the processor to hold the instructions until they are scheduled to be executed.

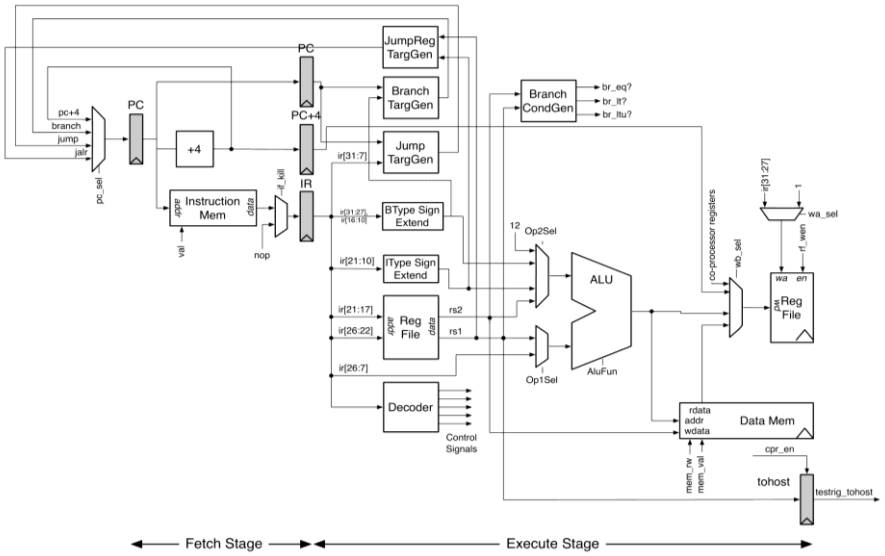

Figure 1: 2 – Stage Pipeline RISC-V Architecture (Sourced from CS152 Resources [13])

The current architecture of the processor is shown in Figure 2 and consists of 32 ‘x’ 32-bit registers which store the result of arithmetic calculations and include ‘x0’ which is a register that is always set to 0. The value of the registers can be accessed by inputting the desired register address in one of the two register read inputs, or data can be read to the registers if the ‘regw’ control signal is high and the chosen register address is inputted at the write input.

The Arithmetic Logic Unit (ALU) carries out all the arithmetic required in the processor. The ALU takes one input as an output of the registers and the other is either an output of the registers or an immediate value in the instruction. This is controlled by an ‘imm’ signal. The output of the ALU depends on a control signal called ‘ALUOp’ this decides what operation the ALU must carry out, i.e., addition or subtraction. The output of the ALU is then fed into the register write input to store the result of the calculation, in this version of the processor the ALU output is also used as the out port for testing.

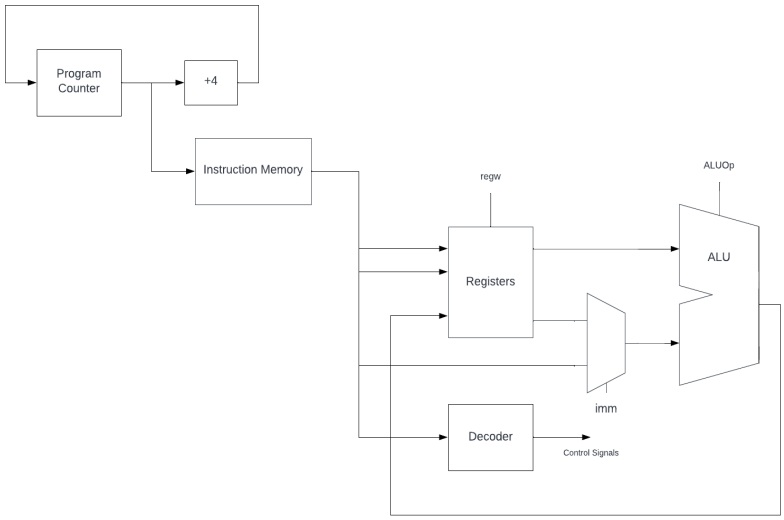

Figure 2: Version 1 Architecture

The processor includes an instruction memory which stores the machine code program, which is written in hexadecimal and converted to binary, that will be executed in memory, the program counter in the processor denotes the current instruction in the instruction memory that is being executed, once this is executed the program counter increments by 4 bytes to move to the next 32-bit instruction. Lastly the processor includes a decoder which controls the control signals that tell the processor what instruction needs to be executed.

So far, a limited number of instructions have been implemented including register ALU operations such as ADD, SUB AND and OR and ALU immediate operations such as ADDI, ANDI, and ORI. The register ALU operations perform the selected operation on the data stored in two registers, the immediate ALU operations perform the selected operation on the data in one register and the 12-bit immediate data stored in the instruction itself.

4.1.2 Version 1 Testing

An assembly language program has been written to test these functions, this was written by hand using the RISC-V instruction set and tests the operation of the implemented instructions. The test program is shown in Figure 3, which shows the 8-digit hexadecimal values which represent each instruction of the program. Next to these are the assembly language instructions represented by the hexadecimal values and their register functions. As the load instructions have not been implemented ADDI has been used instead. This instruction adds the immediate value to the 0 register and is stored in the targeted register this effectively acts as a load operation. The expected outcomes of these instructions is shown in Table 1.

```
00500093  // ADDI x1, x0, 5;   // load 5 in x1
02B00113 // ADDI x2, x0, 43; // load 43 in x2
002081B3 // ADD x3 x1, x2;    // x3 = x1 + x2
0020E233 // OR x4, x1, x2;      // x4 = x1 OR x2
401102B3 // SUB x5, x2, x1;   // x5 = x2 - x1
```

Figure 3: Hexadecimal Test Program

Instruction	*Expected Result*
ADDI x1, x0, 5	*x1 = 5*
ADDI x2, x0, 43	*x2 = 43*
ADD x3 x1, x2	*x3 = 48*
OR x4, x1, x2	*x4 = 47*
SUB x5, x2, x1	*x5 = 38*

Table 1: Table of Instruction Results

The processor and test program have been tested in Modelsim to verify the operation of the first version of the processor. The output wave waveform of the test is shown in Figure 4 and shows that the processor outputs the expected value as shown in Table 1 at the outport (ALU output).

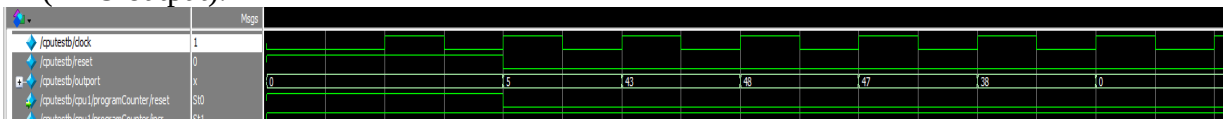

Figure 4: Modelsim Output of the Test Program and Processor

4.2 Equaliser Design

Two versions of an equaliser have been designed, one is a 10-band graphic equaliser, and the other is a 3-band parametric equaliser, the final design will be chosen later. The graphic equaliser has been designed as a bank of 10 FIR bandpass filters. This equaliser has been designed to use the 1/1 octave bands which results in 10 filters with centre frequencies shown in Table 2. The frequency bands are spaced in this way to allow one to hear the difference in amplitude between the bands more distinctly. As the coefficients of this equaliser are fixed, they will be calculated in MATLAB and stored in the processor's memory.

Lower Band (Hz)	*Centre Frequency (Hz)*	*Upper Band (Hz)*
22	*31.5*	*44*
44	*63*	*88*
88	*125*	*177*
177	*250*	*355*
355	*500*	*710*
710	*1000*	*1420*
1420	*2000*	*2840*
2840	*4000*	*5680*
5680	*8000*	*11360*
11360	*16000*	*22720*

Table 2: 1/1 Octave Frequency Bands

The parametric equaliser has been designed as 3 IIR band pass filters, as the bandwidth and centre frequencies of each filter are not fixed the coefficients must be calculated on the processor as outlined previously. The IIR filters have been designed as biquad filters, these are second order filters containing two poles and two zeros. Both the parametric and graphic equalisers are designed to be controlled by external circuity such as potentiometers and will take an input from a phone source and will output to a speaker or oscilloscope..

5 Plan of remaining work

So far, the first version of the processor has been implemented. The next stage of processor design will be to implement the second version of the processor, this will include branch instructions and load and store instructions. This includes implementing a branch generation module and a RAM module. The rest of the base instruction set needs to be implemented which will be the rest of the ALU instructions and jump instructions. Once completed 2 stage pipelining will be added to the processor and then the processor must be tested on the FPGA. To be able to implement programs such as convolution and FFT multiplication will be required. To accomplish this the RISC-V multiplication extension may need to be implemented and any other potential extensions which are needed, such as floating-point arithmetic. The registers may have to be modified to 64-bits wide to handle the multiplication.

Next the processor must be modified to allow communication with the CODEC, which is onboard the De2, this will allow the ADC and DAC on board the FPGA to be utilised. External circuitry must be designed and added to control the equaliser and synthesiser. The equaliser and synthesiser designs must be decided upon, a parametric equaliser and speech synthesiser are the preferred implementation, however this decision will be made later. C programs of these will then be created and these will need to be turned in to RISC-V assembly. This will be done by hand or by an interpreter and then the assembly will be converted to RISC-V machine code using an assembler.

If time permitting an audio compressor will be designed and implemented as part of the equaliser which functions to improve the dynamic range of the output signal. A Gantt chart showing past progress and expected future progress is shown in Figure 5.

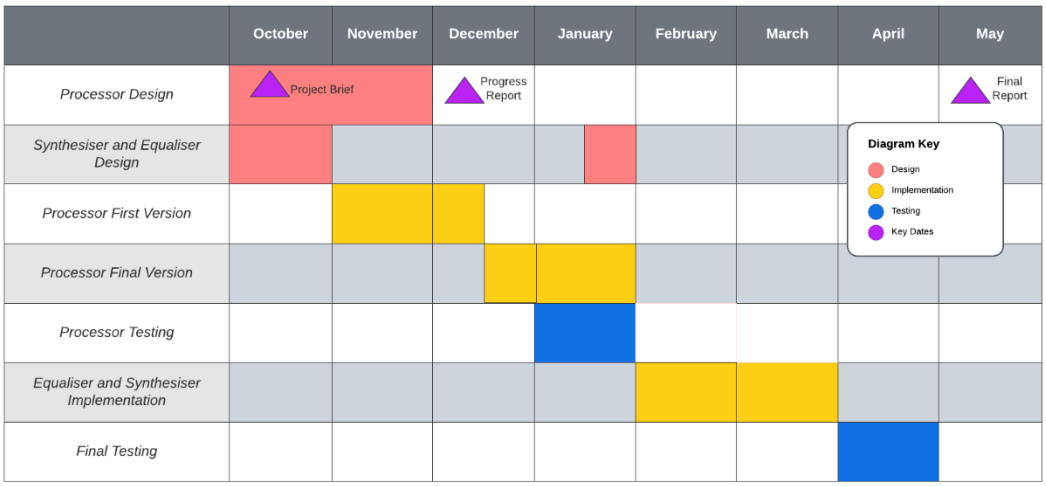

Figure 5: Project Gantt Chart