Equalisation on a custom DSP core

by

A.T. Markettos (CAI)

Fourth-year undergraduate project in
Group E, 2001/2002
Cambridge University Engineering Department

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work. I also declare that the work I shall submit in the conference style paper is also my own.

Signed: _____ Date: _____

Summary

A simple digital signal processor (DSP) for digital data demodulation is described. Development tools are produced, allowing testing of complex assembler programs. The DSP is used as a computation engine within a greater signal processing development system.

The RLS algorithm tested was stable in 8.8 fixed point format. RLS and LMS are assessed for assembler implementation, and an LMS-based filter written. It was stable, and give up to a 18dB improvement in BER over the unfiltered signal, independent of precision. DSP implementation suffers small errors. System meets its 4ms training deadline.

DSP architectural improvements speed up LMS algorithm by 38%.

Contents

Acknoweldgements

Thanks go to:

- Dr Ian Wassell for help, supervision, and suggestions.
- Malcolm Sellars and Neil Murphy at Cambridge Broadband Ltd for allowing me to do this project based on their work, and for their help.
- CUED and Gonville and Caius College for providing computing facilities
- The CUED computer operators, notably Paul Long, and John Fawcett for sorting unhappy computers out.
- Xilinx and TI for providing tools and hardware, even though I didn't use much of it.

Chapter 1

Background

Cambridge Broadband Ltd develop broadband (up to 60Mbit/s) wireless internet access products for point-to-point or point-to-multipoint (ISP to subscriber) systems, with a range of several miles.

This project is involved a part of a wireless link at 3.5 GHz. The project is based upon a simple digital signal processor (DSP) system designed to perform simple signal processing tasks such as convolution, within the modem section of the link. This receives baseband data symbols after downconversion. The aim of this project is to implement adaptive filtering on this DSP, for the purposes of echo cancellation.

Typically units are installed on buildings within a line of sight. However reflections from other objects cause multipath effects on the received signal, creating inter-symbol interference (ISI). It is desired to minimise such ISI with the use of an adaptive filter trained from a known sequence sent at regular intervals.

The baseband functions are implemented in a large Field Programmable Gate Array (FPGA). Part of this is a simple 16-bit DSP, which has been designed to perform signal processing tasks that can be varied in the field. The use of an FPGA means the design of the DSP can be modified, with the result that it is possible to alter its architecture to suit the tasks it is to perform.

Chapter 2

Previous work

The following foundations were supplied at the beginning of the project:

2.1 RLS algorithm

An implementation of a Recursive Least-Squares (RLS) algorithm was available, implemented in floating point arithmetic, running on a Linux machine and a fast off-the-shelf floating point DSP. It was desired to convert this to fixed point, to run on a cheaper off-the-shelf device and subsequently the custom DSP.

2.2 DSP implementation

The DSP itself is written in Verilog, a Hardware Description Language, which allows construction of logic systems in a textual manner, concentrating on the functionality of the design rather than its construction (contrasted with schematic capture, where functionality is hidden by the interconnects).

As supplied, the DSP has an instruction memory which is hard-coded to contain the instruction words it is to execute — the memory contents are supplied by the assembler from the program being run. In the modem design, a controlling microprocessor inserts data into its memory and activates the DSP when it has correlated the incoming signal — this means that in programming it we can assume that data is ready in its memory when it starts up.

2.3 Assembler

The DSP executes a simple assembly language (see section 2.2 for more details). To aid development, an assembler in Python converts textual mnemonics to numeric values which can be read by the DSP. The assembler can be called directly by the Verilog DSP generator to assemble a file and place it into the instruction memory of a newly-generated

DSP. In addition the assembler outputs the binary program in the form of a C structure, which can for insertion into a C program.

2.4 Emulator

A simple emulator, written in Python, was also supplied. This takes each instruction word, decodes and executes it, and hence allows the debugging of programs on a PC rather than using the DSP design directly. To accurately simulate bugs in a program, it is required that all of the pipeline and scheduling issues which are visible to the DSP programmer are also visible in the emulator. This makes emulation more complex to get right (having to make sure glitches in the implementation are also visible in the emulator), and means it must follow the DSP design, and design's progress, accurately.

2.5 DSP architecture

As supplied, the DSP is a simple 16 bit architecture, which has been developed piecemeal for the tasks it is intended from. It is designed so that one clock cycle of the DSP can fit with one clock cycle of the target FPGA, currently at 80MHz.

A block diagram of the DSP is shown in figure 2.1. The DSP contains 16 general purpose 16 bit registers, a program counter, and a 32 bit accumulator. It includes a 4 Kword data memory and a 512 word instruction memory, both instruction and data words being 16 bits wide. There is no cache since both memories are implemented locally on the same FPGA. Theses memories may be enlarged, subject to FPGA area being available.

A listing of the instruction set is shown in figure 2.2. The general opcode format is shown in table 2.1.

The DSP contains 6 pipelined stages, which are each capable of executing one phase of an instruction at a time, the timing of operation of the phases being shown in figure 2.2. There is no pipeline stalling or data forwarding implemented, with the result that after every ALU instruction the result is available in the register bank *after* the next instruction (a latency of 2 instructions). Hence the next instruction will use the previous value of the register. For example the code sequence:

```
LDIEXT          0x12,r0         ; r0 = 0x12
LDIEXT          0x23,r1         ; r1 = 0x23
```

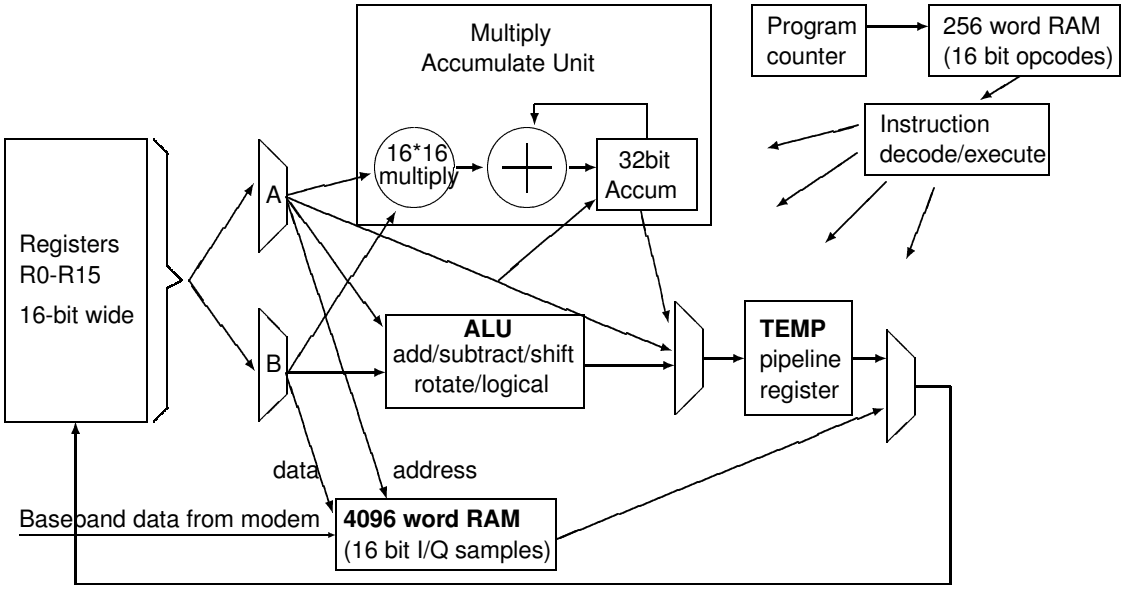


Figure 2.1: DSP block diagram

Type	bits			
	15-12	11-8	7-4	3-0
General	Opcode	Dest reg	Source B reg	Source A reg
			Immediate constant	
Escaped	0xF	Dest reg	Escaped opcode	Source A reg
Special	0x0	Special opcode	Source B reg	Source A reg
			Immediate offset	

Table 2.1: DSP general opcode format

```
NOP                             ; do nothing
ADD             r0,r1,r0        ; r0 = r0+r1
MV              r0,r2           ; r2 = r0
ADD             r0,r0,r3        ; r3 = r0+r0
NOP
```

results in `r0` containing `0x35` but `r2` still holding `0x12` (since by the `MV`, the new value of `r0` has not propagated through yet), while `r3` contains `0x6A` ($2 \times$ `0x35`) because by the time it has executed the new `r0` *has* been written back. Similarly memory loads will not be written back until after the third succeeding instruction (latency of 4), and multiplies after the fifth (latency 6). It is possible for a clash to occur, where an ALU instruction (latency 2) and memory load (latency 4) both try to write back to the register file in the

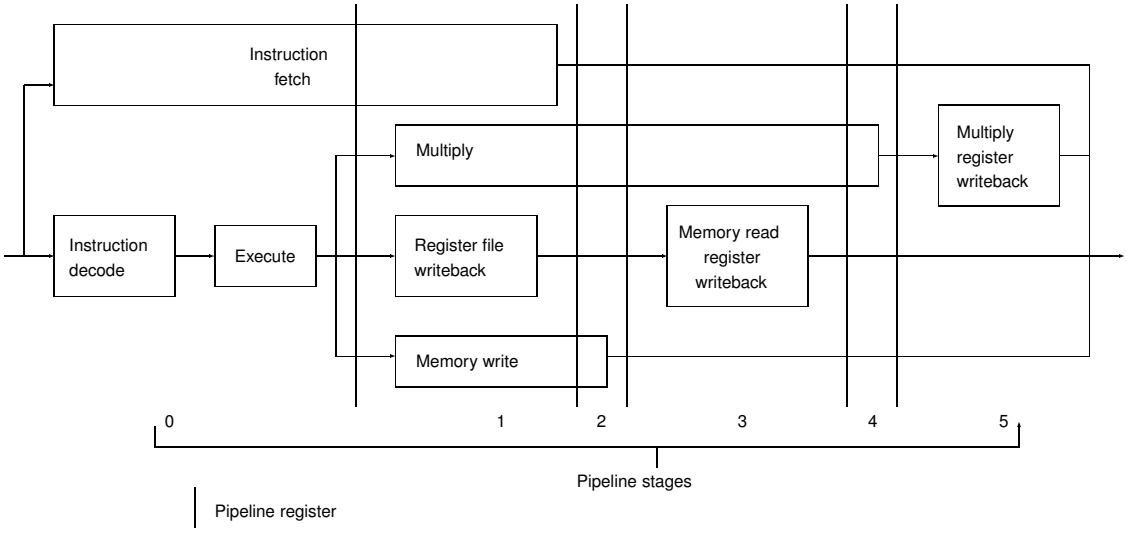


Figure 2.2: Time-based schematic of DSP pipeline

same cycle. Since it only has one write port (a function of the FPGA technology) this is impossible. The DSP makes no attempt to prevent this error - it is up to the programmer to avoid it.

The initial instruction set is shown in table 2.2.

Instruction	Operands	Opcode	Latency/cycles	Flags	Operation
LDIEXT	imm, Rd	0x1—	2		Rd := sign_extend(imm)
LDISHIFT	imm, Rd	0x2—	2		Rd := (temp≪8) — imm
LD	[Rb,offset],Rd	0x3—	4		Rd := Memory(Rb+offset)
ST	Ra,[Rb,offset]	0x7—	1		Memory(Rb+offset) := Ra
ADD	Ra,Rb,Rd	0x9—	2	ZC	Rd := Ra + Rb
ADDC	Ra,Rb,Rd	0x4—	2	ZC	Rd := Ra + Rb + carry
SUB	Ra,Rb,Rd	0xA—	2	ZC	Rd := Ra - Rb
SUBC	Ra,Rb,Rd	0x5—	2	ZC	Rd := Ra - Rb - carry
AND	Ra,Rb,Rd	0xB—	2	Z	Rd := Ra AND Rb
OR	Ra,Rb,Rd	0xC—	2	Z	Rd := Ra OR Rb
XOR	Ra,Rb,Rd	0xD—	2	Z	Rd := Ra XOR Rb
SLA	Ra,Rd	0xF-1-	2	ZC	Rd := [Ra[14:0], 0]; carry := Ra[15]
SRA	Ra,Rd	0xF-2-	2	ZC	Rd := [Ra[15], Ra[15:1]]; carry := Ra[0]
ROL	Ra,Rd	0xF-3-	2	ZC	Rd := [Ra[14:0], carry]; carry := Ra[15]
ROR	Ra,Rd	0xF-4-	2	ZC	Rd := [carry, Ra[15:1]]; carry := Ra[0]
LDACCH	Ra	0xF15-	2		Accum[31:16] := Ra
LDACCL	Ra	0xF05-	2		Accum[15:0] := Ra
STACCH	Rd	0xF-61	2		Rd := Accum[31:16]
STACCL	Rd	0xF-60	2		Rd := Accum[15:0]
MV	Ra,Rd	0xF-7-	2		Rd := Ra
NOP		0x00–	0		Do nothing for one cycle
HALT		0x01–	2		Execute 2 more instructions then halt
MAC	Ra,Rb	0x02–	6		Accum := Accum + (Ra * Rb) (signed multiply)
BZS	imm	0x07–	2		If Z set, PC := PC + signed(imm)
BZC	imm	0x08–	2		If Z clear, PC := PC + signed(imm)
BCS	imm	0x09–	2		If C set, PC := PC + signed(imm)
BCC	imm	0x0A–	2		If C clear, PC := PC + signed(imm)
BRA	imm	0x0B–	2		PC := PC + signed(imm)

Table 2.2: Initial DSP instruction set. Notation: [a,b] is a word formed by concatenating the bits from a with the bits from b. word[x:y] is bits x to y of word (inclusive). Z and/or C imply instruction sets Zero and/or Carry flags.

Chapter 3

Software tools

Once provided with the basic tools, it was necessary to produce a development environment for writing and testing programs to run on the DSP. The shortcomings of the supplied environment include:

- No tools for simulating Verilog were supplied (the ones in use at CBL were too expensive to provide).
- Verilog tools are slow (since they need to simulate on a signal level) and difficult to extract debugging information from.
- Assembler only handles native instruction set – doesn't allow pseudoinstructions for functions that can be represented in one or more instructions.

  For example to load a 16-bit value x into a register, one might use:

```
LDIEXT    Ra,x[7:0]
LDISHIFT  Ra,x[15:8]
```

  This could be represented in a single pseudoinstruction LDIL Ra,x which saves typing, makes code clearer and prevents errors.
- No means to use the DSP as a signal processing device. We would like to be able to pipe data into the DSP and run programs on this data.

3.1 Choice of Verilog tools

Before any work could begin, it was necessary to find a Verilog toolset which could cope with the supplied processor core. A number of toolsets were evaluated:

Icarus Verilog A free compiler and simulator. The DSP stressed them, showing up a number of bugs. While these were fixed quickly after reporting, it was not a good use of the limited time spent debugging others' code.

Ver Structural Compiler/Behavioural Simulator Another free compiler and simulator. Only a structual compiler, so doesn't support behavioural statements such as `initial` (which initialises register values). These are required, for example, to define the DSP's program. Therefore this wasn't any use.

Xilinx Foundation v1.5 An early version of the Xilinx toolset available in the Department. However no Verilog licences were available.

Xilinx Foundation v3.1 A later version of *Foundation*. Tries to synthesise Verilog, hence has same `initial` problems as *Ver*. After removal, it spent 30 minutes disc thrashing with no end result. FPGA tools are memory intensive, so without a more powerful machine to try, this did not seem to be an option worth pursuing.

ModelSim XE (Xilinx Edition) This is a downloadable Verilog compiler/simulator for Windows. This slows down exponentially above a fixed number of entities (as an encouragement to buy the full version). After working around the entity limit the DSP worked, running a 6 instruction program in 2 seconds.

Mentor Graphics tools These are installed on the Departmental Teaching System and include a version of ModelSim without arbitrary limitations, with remote access available. CPU time is shared with other users, so it would not be fair to use this extensively.

As a result of this evaluation, ModelSim XE was used for most development work, with a little being done on the Mentor Graphics tools. Filesharing between multiple platforms provided an easy-to-use development environment.

3.2 Emulator

The low-level of Verilog simulation made it unsuitable to developing large programs, in terms of access and efficiency. It is useful to be able to use the DSP as a computation engine in a language such as Matlab, and to get data and information out (such as put in print statements) which is difficult in Verilog.

To aid the above, and allow ease of debugging, the emulator was investigated. The existing emulator was written in Python which, while easy to program, is a very high level language not recommended for CPU intensive tasks. Typical DSP algorithms take

millions of instructions to execute, for which it is helpful to have a fast emulator. It was decided to redevelop the emulator in another language more suited to running programs at speed.

3.2.1 Choice of language

A number of possibilities were considered:

Python Existing emulator is already written - but Python is too slow.

C Fast, good for shifting bits around which is useful for emulating hardware. Difficult memory management, but the DSP has static memory requirements. It is possible to use Object Oriented (OO) techniques to achieve functions such as data hiding. This software engineering technique aids maintainability and reuse, however only a subset of the full OO techniques are available in C through OO provided 'by hand'.

C++ Comments as for C, plus true OO features. Some of C++'s OO is overly complex and has pitfalls, but not too much of a problem if the OO is kept simple. C++ compilers are often worse than corresponding C compilers.

Java C++ without many of the pitfalls. Memory management is handled nicely. Portable, with easy access to graphics. However, the need for a virtual machine or JIT compiler means it is slower than C.

To use the DSP as a computational element in a package such as Matlab requires an interface to be written between the emulator and Matlab. Matlab supports calling C, Java and Fortran, but not C++. It is possible however to interface C++ through a C veneer.

OO isn't strictly necessary for developing an emulator – since it emulates a sequential procedural processor, the emulator can use this model. For this reason, it was decided to start in C, and move to C++ if many OO features were required.

3.2.2 Emulator architecture

The emulator must faithfully emulate the implementation of the DSP as represented by the Verilog rather than the higher level specification (such as that detailed in table 2.2). This is because we wish to develop programs to run on the hardware DSP, and so

must expose our emulated programs to the same timing glitches and pipeline hazards present in the hardware. Therefore the emulator architecture must mirror the hardware architecture.

A very close mirror would involve handling hardware signals such as read/write strobes to registers. While accurate, this is difficult to maintain, slow, and unnecessarily complex. Alternatively the pipeline structure shown in figure 2.2 can be retained, but represent each pipeline stage as a software function. Also required is the concept of a register which simulates a hardware latch — so a value written is only available on the output after a clock cycle. This clock drives shifting of the whole pipeline. mirroring the hardware pipeline.

Important constructs in the emulator include:

Register Simulates a hardware register, containing an output value and a next value. On a clock cycle, the next value is transferred to the output value. Writes update the next value, reads are from the output.

DSPState An object containing all the state for the emulator, including register bank, data and code memory contents, pipelined signals and flags. This is the only memory used by the emulator.

Pipeline A file containing code to perform the DSP's pipeline stages. In figure 2.2 these are performed right-to-left to shuffle data along the pipeline. In order of operations performed to an instruction (left-to-right order), they are:

Memory access Read/write data memory, fetch next instruction.

Stage 0 Read the instruction most recently fetched, decode it, execute it.

Stage 1 Write arithmetic results to registers, access accumulator for LD/STACC, compute multiply part of MAC.

Stage 2 Do nothing - on real DSP multiply or LD happen here.

Stage 3 Do write-back of memory value read by LD to registers.

Stage 4 Do nothing - on real DSP multiply is happening here.

Stage 5 Do accumulate part of MAC, and write to accumulator.

Clock Clock all registers, so that new values propagate through.

Opcode execution Instead of a large `switch` statement to determine which opcode to call, function pointers are used. An opcode dictionary structure contains a function pointer for each major opcode type (opcodes of the form Txxx in table 2.2). This

function pointer is looked up and called to execute the instruction. The major opcodes which represent groups of minor opcodes (0Txx and FxTx) call similar tables to decode them. This reduces the time to decode the instruction to be one or two array lookups and function calls, rather than searching a long `switch` statement.

Opcode functions are listed in the opcode dictionary, and perform the operations specific to each opcode, including ALU operations. Undefined instructions simply contain a pointer to a function to halt the DSP.

The emulator performs a few basic checks, such as ensuring that two operations do not attempt to commit their results to the register file or accumulator at the same time (possible because of variable instruction latencies) — this halts emulation. It outputs a trace of each instruction it is executing, allowing program flow to be followed and pinpointing commit clashes.

The MEX interface is used to connect with Matlab, so it can called as a Matlab function. This involves linking it with Matlab libraries, and also writing an interface function to take Matlab matrices, convert to DSP fixed point representations, and insert them into the DSP's memory.

The assembler outputs a header file containing the DSP program, so the emulator must be recompiled every time the program is changed. It is a simple extension to be able to load in a program at runtime. However, for the MEX interface to insert data into memory locations, it must know the addresses of memory locations referred to in the assembly. To do this the assembler produces a header file containing all labels defined in the assembly code. When the emulator is compiled, these #defines are used to write the Matlab data in the right places in memory ready for the assembler. This could be done at runtime, but a recompile is not too difficult as the emulator is fairly small — a series of shell scripts perform preprocessing (see section 3.3), assembly, rebuild the emulator and run the emulator automatically.

The emulator was written to be portable, and as such has been run on x86, PA-RISC and ARM CPUs under Windows, Linux, HP-UX and RISC OS with no changes required.

3.3 Assembler

The existing assembler is written in Python, is slow, and does not support pseudoinstructions, for which it would be convenient for users to define their own. There existed a number of possibilities for developing the assembler:

Extend the existing assembler. However it is slow, and developing it would probably slow it down even further.

Bolt a frontend onto the existing assembler. This doesn't make it any faster, but has the advantage that the intermediate code can be inspected - it is important when a pseudoinstruction maps onto more than one real instruction. As instruction timing is critical, sometimes pseudoinstructions looking like a single instruction have an unexpected effect due to being composed of multiple real instructions.

Port an existing crossassembler. A crossassembler assembles code for multiple target CPUs, based on a definition file outlining the assembly language for each target. However, there are very few free crossassemblers available, and the notable exception (the GNU assembler `gas`[4]) is far too complex to consider porting.

Rewrite the assembler in a faster language. This might include using a compiler compiler such as `yacc`/`lex`. This is the most elegant solution, although the most time consuming. It was not felt that there was enough time available to consider this.

As a compromise, it was decided to attach a preprocessor to the existing assembler to convert it into a macroassembler. This was done with GNU `m4`, a generic macro processor. This does textual substitution and macro expansion on text files of any type, and is widely used as a text processing tool. A shell script runs an assembly file through `m4`, then passes the result to the assembler to assemble in the normal way.

`m4` is used to define a library of functions, which can then be invoked in the assembler. Notable functions include:

Register naming allows meaningful names to be defined for registers (eg `sp` for stack pointer instead of `r14`).

Pseudoinstructions permit higher level functions to be defined as a single operation, even though they might take more than one real instruction to implement. These are documented in section 3.4.

Stack : the DSP does not provide any support for stacking, but macros allow a stack pointer to defined, and the macros handle automatic adjustment of the stack pointer after storing/loading values from memory. Since `m4` is a functional language, it is necessary to code this using tail recursion. The following pseudocode illustrates the way it works:

```
\\ user calls, for example PUSH(r1,r6,r8)
function PUSH(regs)
  \\ regs is a list of registers to store on the stack
  stack_offset = 0;
  PUSH_one_register(stack_offset, regs);
  assemble_instruction(add stack_offset to stack_pointer);
end

function PUSH_one_register(stack_offset, r::rest)
  \\ r::rest means set r to the first element of the list supplied,
  \\ and rest to the remainder of it
  if (r==[]) end;  \\ if at end of the list, finish
  assemble_instruction(store r at stack_offset+stack_pointer);
  PUSH_one_register(rest, stack_offset+1);
end
```

so the example `PUSH(r1,r6,r8)` with `stack_pointer = r14` produces:

```
ST        r1,[r14,0]
ST        r6,[r14,1]
ST        r8,[r14,2]
ADDI        3,r14
```

(where `ADDI` is an extra instruction to add an immediate constant to a register – see section 3.4). POP works in a similar way except, since LD cannot accept a negative offset, it performs the decrement first.

Structures in memory can be defined using the structure macros. They define a macro which is a counter to act as a current pointer in memory. Each item contains its length, which updates the structure pointer. These again use tail recursion to compute the address count. A structure can be defined as:

```
structure_start(100)
structure(a,20)
structure(b,1)
structure(c,4)
structure_end
```

which produces output:

```
a         EQU         100
b         EQU         120
```

```
c        EQU        121
next item here would go at 125, and so on
```

`EQU` is an assembler directive which sets the label on the left hand side to the right hand value. This can then be used instead of a numeric constant in expressions.

Conditional assembly. Assembler output may be controlled using `m4_ifdef` constructs around sections of code. These can either be defined at the top of the file or at the command line (for example,`m4_define('DO_FILTER')` and `m4 -P -DDO_FILTER < input > output` respectively). This means a section of code may be included/excluded more simply than having to edit the code each time.

Assembler macros. Chunks of code can be written as `m4` macros. They can be invoked multiple times, and parameterised if very similar block of code need to be assembled more than once. For example, filtering for i and q components performs the same calculation, but with different data blocks for input and output. By passing the data pointers as parameters, an instance for each of i and q components can be created. Whilst taking more instruction memory, this enables the programmer to be sure each version is identical which aids debugging.

3.4 Instruction set extensions

When developing code for the DSP, it was found that adding extra instructions made common programming tasks much easier (reducing their instruction count), and in some cases permitted previously impossible features. Since the CPU executes one instruction per cycle, reducing the instruction count implies both increasing speed and reducing code size. Working against this is the restriction that opcodes must fit in a single 16 bit word, which severely limits the number of instructions that can be added. In particular, each instruction taking 3 register arguments occupies $1/16$ of the opcode space. Hence the merits of an instruction must be carefully considered before including it. Opcodes solely used for debugging are not under such stringent requirements, since they can be removed without loss of functionality.

The instructions added were as follows. The fraction of opcode space used is given in brackets:

LDR [Ra,Rb],Rd adds `Ra` and `Rb` together and loads the value pointed to by the sum into `Rd`. This is very useful, especially in signal processing where most computation is done as array accesses. It allows `Ra` to be the array base, and `Rb` the element to access, performing `Rd = Ra[Rb]`. Previously a temporary register was required to hold the sum of `Ra` and `Rb`, and an extra two cycles to complete this sum. In the final LMS code, the code is 515 instructions long, of which 36 are `LDR`s. Without them, the code would have been 587 instructions long (ignoring the need to stack registers or use of the delay slot when doing the temporary add) - an increase in size of 14%, and correspondingly slowed down by this amount. $(1/16)$

The counterpart STR is indeed possible, but requires three register bank reads. The FPGA technology in use can handle this, but it would have required substantial modification to the DSP core to permit it. Since only 18 ST instructions are used, the corresponding performance hit is approximately 7%. It would also occupy a large chunk $(1/16)$ of opcode space.

Such figures are overestimates, since in some cases there may be nothing gained from using an `LDR`/`STR` over `LD`/`ST`

LDACCX Ra writes the 16-bit value to both halves of the 32-bit accumulator.Thus if `Ra=0`, the accumulator can be cleared in one instruction. There are only 4 `LDACCX`s in the final code, so it only provides 0.7% code size saving. But it uses little hardware or opcode space, so this is a cheap performance gain. $(1/4096)$

LDPC Ra loads the PC with the value of a register. This permits a jump to an arbitrary address rather than one specified during assembly, allowing branch tables, function pointers and other previously impossible structures. $(1/256)$

BL label (branch with link), stores the program counter (PC) into a register (called the link register, nominally `r15`) and branches to `label`. In conjunction with `LDPC` this allows subroutines, since `BL` can be used to set up `r15` with a return address, which is jumped to with `LDPC`. This combination allows structured programming with subroutines, rather than the GOTO-like constructs using branches. Note that the stored program counter points after the delay slot following `BL`, so if a `BL`/`LDPC` combination is used, this delay slot only gets executed once.$(1/256)$

LDACCM/STACCM Ra load and save the middle of the accumulator from/to `Ra`. In 8.8 fixed point format (8 bits integer, 8 fractional), the result of a multiply lies in bits 8 to 15 of the accumulator. `STACCM` stores this 16-bit word into register `Ra`. `LDACCM` performs the reverse operation, except it sign extends the value from `Ra` to fill the top 8 bits of the accumulator with its sign bit, and clears the lower 8 bits of the accumulator. These permit 8.8 fixed point arithmetic, which would otherwise be difficult – `STACCM` would require `STACCH` then `STACCL` plus masking off and ORing the relevant sections together, at least 10 instructions. `LDACCM` is even more complex due to the sign extend. The LMS code uses 8 `STACCM`s and 2 `LDACCM`s, so a very rough size saving might be $(8 + 2) \times 10 = 100$ instructions, or 19%. ($1/4096$ each)

ADDI/SUBI v,Rd add/subtract immediate constant `v` to `Rd`, then write it back to `Rd`. Originally these were implemented as separate `ADDI`/`SUBI` instructions with a 4 bit constant field (each occupying $1/256$). They were combined to make a single `ADDI` instruction taking a single 8 bit signed constant, which gives a much larger (and more useful) range, at the expense of taking up $1/16$ of the opcode space. There are 11 `ADDI`s in the LMS code, which would otherwise take 3 instructions. Thus there is a speedup of 22 instructions, or 4.3%

STPC Rd stores the PC into register Ra. A major problem occurs with `BL` once the program grows over 128 instructions, because `BL` can only handle an offset in the range ± 128. We could have a series of branches each within 128 instruction range of each other to jump to the faraway function, but this means the programmer has to worry about how long each section is. Using `STPC` allows a solution to be devised:

```
; ra contains the address of the subroutine we wish to call
STPC r15
LDPC ra
NOP
```

This stores the PC pointing to the `NOP` into r15, then calls the subroutine pointed to by `ra` (executing the delay slot `NOP` as usual). On return, the `NOP` is executed again. Alternative strategies include transposing `LDPC` and `STPC` above, but this means the PC that `STPC` stores is already that of the subroutine – this could be remedied, but with difficulty. `STPC` could store PC+1, but this would be messy.

Defining the behaviour of `STPC` to it executing the delay slot twice seems the least painful solution.

The following instructions were devised for debugging, and were only implemented in the emulator:

HALT v inserts a constant into the redundant immediate field in the existing HALT. This reveals the exact place the DSP halted, to the programmer in the emulator or to extra (unimplemented) hardware in the DSP. $(1/256)$

ASSERT Rx,Ry compares whether the contents of Rx and Ry are equal, and halts the emulator if they are not. Compliance tests can use this, since it is possible to compare the result of a calculation with the expected value, and stop the program if they are not the same. `ASSERT` always prints a register dump, so `ASSERT Rx,Rx` will print a register dump and not stop the program. $(1/256)$

PRINT Rn,v prints the tag `v` followed by the contents of `Rn` in both hex and as an 8.8 fixed point value. This aids debugging, where the programmer wishes to print out the contents of a particular register. The tag is used to distinguish between printouts from different parts of the program. $(1/256)$

PRINTC char prints the character `char`. This can be used to trace program flow, or format output from `PRINT`. $(1/256)$

PRINTM v : PRINT Miscellaneous. The only opcode defined is `v=1`, which prints the accumulator. ($1/256$ although this could be reduced).

The following instructions were found to be possible using other instructions, and hence removed as being redundant:

SLA Ra,Rb can be represented by `ADD Ra,Ra,Rb` (ie `Ra`$\ll$`1` $\equiv$ `Ra+Ra`)

ROL Ra,Rb can be represented by `ADDC Ra,Ra,Rb`
(ie `Ra`$\ll$`1 | carry` $\equiv$ `Ra+Ra+carry`)

Together these save $2/256$ worth of opcode space.

As well as opcode space, extra instructions are constrained by FPGA timing requirements at the clock frequency (80MHz). In this design, the TEMP register is heavily used

and this becomes the critical path for timing analysis. The new design was tested by fitting it with the FPGA tools to the device used in the modem. It fitted into a higher speed grade part easily, but the 80MHz part required some rearrangement which suggests it is approaching the limit of timing. However timing analysis reported a speed of 90.5MHz, which leaves a 10% margin on the required performance.

Combining the theoretical results based on the analysis of real code, suggests that roughly 38% of code size/execution time is saved by these improvements, at the expense of 12.5% of opcode space. The debugging instructions make debugging possible, although not straightforward.

3.4.1 Pseudoinstructions

Using the macro assembly capabilities of `m4`, common operations involving more than one instruction were encapsulated into a single pseudoinstruction. This makes code easier to read and debug. Due to the way `m4` invokes its macros, they must have brackets around their arguments:

LDIL(value,ra) loads a 16 bit `value` into register `ra` (`LDIEXT` then `LDISHIFT`)

ADR(label,ra) loads register `ra` with the address of `label` (functionally this is the same as `LDIL`).

STR(rv,ra,rb,rt) stores `rv` at the address (`ra+rb`), `rt` is a scratch register. This performs the `STR` function above in 3 instructions.

BLL label is a long branch - load the address of `label` temporarily into `15`, then branch with link to it using `STPC` and `LDPC` as outlined above. At the expense of 6 instructions, this can branch anywhere in the code memory.

DELAYL performs 3 `NOP`s to ensure a load operation has completed before using the result.

DELAYM similarly performs 5 `NOP`s for a `MAC`.

PRINTACC performs `PRINTM 1` to print the accumulator.

NEWLINE performs `PRINTC 10`, printing a newline character.

To ease developing functions, and provide a structure for register use within larger programs, a procedure calling standard was developed based on the ARM Procedure Call Standard with `m4` to do register naming. This involves:

- Registers 0-3 (called `a1-a4`) are used to supply parameters to functions. The result of a function is returned in `a1`, and all of `a1-a4` may be corrupted by a function call.
- Registers 4-9 (called `v1-v6`) may be used within a function, but must be preserved across function calls.
- Register 12 (`ws`) is the workspace pointer, which points to the base of memory used by this program (this allows multiple programs to run on the DSP, by giving each a different base workspace pointer). In this case it is useful because in the hardware DSP, the base of memory is written to by the controlling CPU.
- Register 14 (`sp`) is the stack pointer, used to push and pop registers from the stack.
- Register 15 (`lr`) is the link register, which stores the function return address.

3.5 Conclusion

These instruction set modifications, as well as being more efficient, made the DSP much easier to program by hand, which is a key factor in developing code for it. The design including these changes was shipped off for production in an ASIC (Application Specific Integrated Circuit), which is currently in the process of fabrication.

Chapter 4

Finite precision effects

Recursive least squares (RLS) algorithms suffer finite precision effects[7], which cause them to become unstable or fail to converge. Much effort has been put into analysing the instability and devising variations on the RLS algorithm for particular problems (see [2, 3, 8] for a selection).

Before an equalisation algorithm may be implemented on the DSP, it is useful to consider its behaviour when using finite precision arithmetic. In particular, it is helpful to decide what size of fixed point words and the position of the binary point in simulation, before committing to a DSP implementation where these values are difficult to change.

The starting point was some existing code that performs the filter system shown in figure 4.1[9]. This was designed to run in C using floating point arithmetic on a Linux machine and also on an off-the-shelf floating point DSP chip. As well as analysing the fixed point behaviour, it is useful to implement the RLS algorithm on a cheaper off-the-shelf fixed point DSP chip.

The aim was to convert the existing floating point code into fixed point with as few

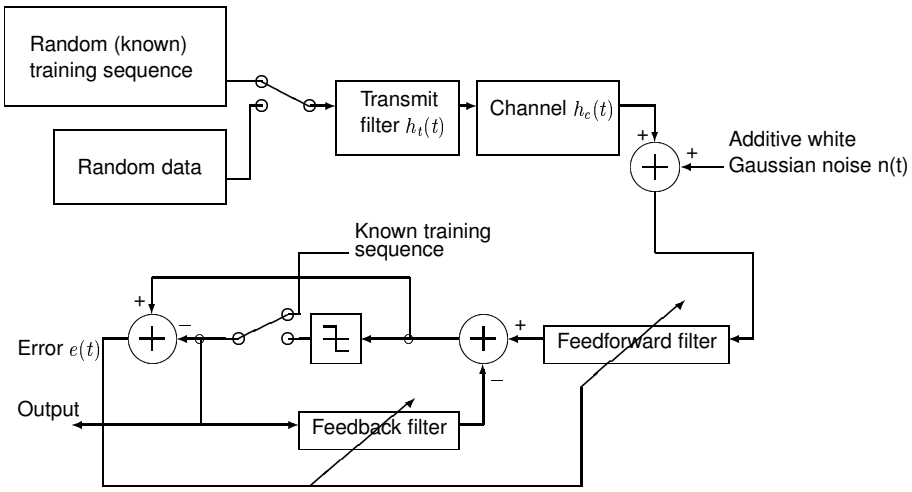


Figure 4.1: Decision feedback equaliser (DFE) filter system block diagram, trained with RLS

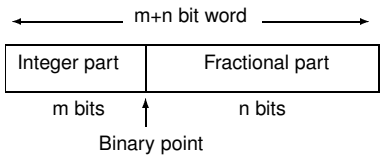


Figure 4.2: Fixed point representation

structural changes as possible, and to allow the investigation of different fixed point formats to study the effects of finite precision. A restriction is that the implementation must be efficient since it runs on an embedded device to real time constraints. Hence, for example, a C++ class with operator overloading for fixed point values would be too heavyweight, since it is less efficient and requires a C++ compiler for the DSP (which brings in extra unwanted overheads and typically for embedded hardware compiles inferior code to a C compiler).

We consider the method of representing fixed point numbers using a single integer word, dividing this into an integer part and a fractional part using a binary point.

For the most general case, consider two numbers a with binary point bit position p_a and b with point p_b, where $p_i = n$ in figure 4.2. We can thus define an arithmetic in this representation ($\ll n$ and $\gg n$ represent left and right shifts by n places respectively):

Operation	Pre-operation	Arithmetical operator	Point of result
Assignment		$a = b \ll (p_a - p_b)$	p_a
Addition/ subtraction	$a' = a \ll (p_b - p_a)$ $b' = b \ll (p_a - p_b)$	$q = a' \pm b$ $q = a \pm b'$	p_b p_a
Multiplication		$q = a \times b$	$p_a + p_b$
Division	$a' = a \ll p_b$	$q = a' \div b$	p_a

Care is required in multiplication to ensure that the result does not overflow the result representation. Even more care is required for division, since the point position is a tradeoff between binary places in representation of numerator and denominator, and binary places of result.

As can be seen, much of the extra work requires shifting values around before performing arithmetic on them. Also note that if $p_a = p_b$, $p_a - p_b = 0$, and hence addition, subtraction and assignment require no shifting. If we choose values for p_a and p_b that are equal, we substantially reduce the work required for these operations. It would be useful to take advantage of this optimisation in such common cases.

Also remember that typically the point positions will be fixed in the program. This is

the purpose of fixed point arithmetic - allowing them to move requires the complexities of floating point which is what we are aiming to avoid in both hardware and software.

4.1 Implementation

Ideally, we wish the compiler to perform the work of keeping track of the binary points, and insert shifts only when necessary. Modern optimising compilers are designed to remove extraneous operations, a feature we can use to achieve this.

To provide a friendly interface to the programmer, a number of C preprocessor macros were developed to hide the complexity of shifting, and look like normal arithmetical operations. These build on C's shift and predicate
(`condition ? resultIfTrue : resultIfFalse`)
operations. Some notable examples are documented here, but a full listing of the macros can be found in the Appendix.

4.1.1 Fixed point macros

All macros rely on the datatype `fixed`, which is a fixed point dataword (of length which can be assigned), and each variable `x` has an associated `#define`d point position `POINT_X`.

```
#define SHIFT_RIGHT(value,shift)
  (((shift)>0) ? ((value)>>(shift)) : ((value)<<(-(shift))))
```

This macro is invoked to perform a right shift of `value` by `shift` places. C cannot handle shifts of negative amounts, so this macro performs a left shift if `shift` is negative. `SHIFT_LEFT` is defined as `SHIFT_RIGHT` with a negative `shift`.

```
#define FIXED_ASSIGN(srcValue,srcPoint,destPoint) \
  (SHIFT_RIGHT(srcValue,srcPoint-destPoint))
```

Usage `destValue = FIXED_ASSIGN(srcValue, srcPoint, destPoint)`. Returns `srcValue` shifted so it has its binary point at `destPoint`, ready for assignment to `destValue`.

```
#define FIXED_ADD(aValue,aPoint,bValue,bPoint,resultPoint) \
  (aPoint>bPoint) ? \
    SHIFT_LEFT(((bValue) + \
      SHIFT_RIGHT((aValue),((aPoint)-(bPoint)))),((resultPoint)-(bPoint))) : \
    SHIFT_LEFT(((aValue) + \
      SHIFT_RIGHT((bValue),((bPoint)-(aPoint)))),((resultPoint)-(aPoint)))
```

Returns `aValue+bValue` with binary point `resultPoint`, by shifting one of the operands to the binary point of the other, doing the addition, then shifting to the point of the result. The binary point used for the addition is chosen to retain as much precision as possible. `FIXED_SUB` (subtract) is just `FIXED_ADD` with `bValue` negated.

Multiplies need particular care because, depending on the word length of the machine the code is running on, they may lose their sign bits when shifting. For portability, a function strips off the sign bits, performs the multiply then adds the sign back again. While the arithmetic may be simple, it requires a function call which adds overhead. With a loss of generality this can be made into a macro call designed for a particular word length machine. It is also the case that a multiplication requires the machine to have a word twice the length of the word being used in the calculations, because multiplying two k bit words produces a $2k$ bit result. If such a word is not available, it can be emulated using two k bit words, but requires more work.

The issues for multiplication also apply to division. Additionally, large denominator terms typically give rise to small results (and vice versa). Often trying to represent both with the same binary point will result in one or other losing precision. Care must be taken in choosing the representation for these values. We can code special cases, notably the reciprocal operator, with binary points chosen for situations where the parameters are large or small. However, this is still a significant source of lost precision.

4.1.2 Macros in use

We can investigate the quality of code output by a compiler from the macros, to see how efficiently they are being compiled. Consider the following function, using 32 bit fixed point values:

```
#include "fixed.h"
#define PD 16 // default binary point

fixed sumofproducts(fixed a, fixed b, fixed c, fixed d)
{
  fixed e;
  e = FIXED_SUM_OF_PRODUCTS(a,PD,b,PD,c,PD,d,PD,PD,PD);
  return e;
}
```

`FIXED_SUM_OF_PRODUCTS` is a macro that performs $a \times b + c \times d$, using the above features. The preprocessor output, showing all the macros expanded, is (only for illustration):

```
fixed sumofproducts(fixed a, fixed b, fixed c, fixed d) {
  fixed e;
  e =  ( ( 16 > 16 ) ?   ((( (-( (( 16 )-( 16 )) )) )>0) ? (( (( ( fixed_mul ( c , 16 , d , 16 , 16 )) ) +
((( (( 16 )-( 16 )) )>0) ?   (( ( ( fixed_mul ( a , 16 , b , 16 , 16 )) )
)>>( (( 16 )-( 16 )) )) : ((
 ( ( fixed_mul ( a , 16 , b , 16 , 16 )) ) )<< (-( (( 16 )-( 16 )) )))) ) )>>( (-( (( 16 )-( 16 )) )) )) :
 (( (( ( fixed_mul ( c , 16 , d , 16 , 16 )) ) + ((( (( 16 )-( 16 )) )>0) ?   (( ( (
fixed_mul ( a , 16 , b , 16 , 16 )) ) ) >>( (( 16 )-( 16 )) )) : (( ( ( fixed_mul ( a , 16 , b , 16 , 16 )) ) )<<
  (-( (( 16 )-( 16 )) )))) ) )<<(-( (-( (( 16 )-( 16 )) )) ))))   :   ((( (-( (( 16 )-( 16 )) )) )>0) ?
 (( (( ( fixed_mul ( a , 16 , b , 16 , 16 )) ) +   ((( (( 16 )-( 16 )) )>0) ? (( ( ( fixed_mul ( c , 16 , d , 16 , 16 )) ) )
>>( (( 16 )-( 16 )) )) : (( ( ( fixed_mul ( c , 16 , d , 16 , 16 )) ) )<<  (-( (( 16 )-( 16 )) )))) ) )>>
( (-( (( 16 )-( 16 )) )) )) : (( (( ( fixed_mul ( a , 16 , b , 16 , 16 )) )  ((( (( 16 )-( 16 )) )>0) ?
  (( ( ( fixed_mul ( c , 16 , d , 16 , 16 )) ) )>>( (( 16 )-( 16 )) )) : (( ( ( fixed_mul ( c , 16 , d , 16 , 16 )) ) )
<<(-( (( 16 )-( 16 )) )))) ) )  <<(-( (-( (( 16 )-( 16 )) )) ))))   ) ;
  return e;
}
```

A cursory look at this code suggests that there are many extraneous tests (is $16 < 16$?) which can be optimised out. The assembler output of the compiler (generating for a simple-to-follow ARM target), is:

```
; parameters: a1=a, a2=b, a3=c, a4=d
        MOV     v2,a3     ; keep c and d safe for later
        MOV     v1,a4
        MOV     a4,#&10   ; 5th parameter of fixed_mul
        STMDB   sp!,{a4}  ; is passed on the stack
        MOV     a3,a2
        MOV     a2,#&10   ; point position of multiply parameters
        BL      fixed_mul ; a1 = a*b
        ADD     sp,sp,#4
        MOV     v3,a1     ; v3=a1 for safe keeping
        MOV     a4,#&10
        STMDB   sp!,{a4}
        MOV     a3,v1     ; pass c and d as parameters
        MOV     a1,v2
        MOV     a2,#&10
        BL      fixed_mul ; a1 = c*d
        ADD     a1,v3,a1  ; a1 = a1+v3 = a*b+c*d
; result is in a1
```

Ignoring shuffling of values between registers for parameter passing, this is two calls to the multiply function and an add - hence all the complexity has been optimised away. Other than the overhead of the multiply function (as discussed above), there has been no performance hit from using the fixed point macros.

4.2 Fixed point RLS

Using these macros, it is possible to give each variable a separate static point position. Modifying the existing floating point code to use them is simple, because it involves replacing arithmetical operations with macro calls. An additional feature of the macros is that the size of the representation can be changed – here both 16 and 32 bit words

Variables:
$\hat{\boldsymbol{h}}(n)$ = estimate of coefficient at step n.
$\boldsymbol{k}(n)$ = Kalman gain vector.
$d(n)$ = desired output response.
$\boldsymbol{\Phi}(n) = \boldsymbol{P}^{-1}(n)$ = correlation matrix.
$\boldsymbol{u}(n)$ = input sample vector.
$\boldsymbol{\eta}(n)$ = estimation error.

For step n of the RLS algorithm:

$$\text{Compute gain vector } \boldsymbol{k}(n) = \frac{\boldsymbol{P}(n-1)\boldsymbol{u}(n)}{1 + \boldsymbol{u}^T(n)\boldsymbol{P}(n-1)\boldsymbol{u}(n)} \tag{4.1}$$

$$\text{Compute error } \boldsymbol{\eta}(n) = d(n) - \boldsymbol{u}^T(n)\hat{\boldsymbol{h}}(n-1)$$

$$\text{Update coefficients } \hat{\boldsymbol{h}}(n) = \hat{\boldsymbol{h}}(n-1) + \boldsymbol{k}(n)\boldsymbol{\eta}(n)$$

$$\text{Update correlation } \boldsymbol{P}(n) = \boldsymbol{P}(n-1) - \boldsymbol{k}(n)\boldsymbol{u}^T(n)\boldsymbol{P}(n-1)$$

Table 4.1: Vanilla recursive least-squares (RLS) algorithm

were tested. Defining a version of the macros that call floating point functions, the same code can be run in 16 or 32 bit integer or floating point mode. The points of particular variables can be moved independently, or by setting all to a default value they can all be changed together.

The vanilla RLS algorithm being used is detailed in table 4.2.

The stability of the RLS algorithm was investigated by running it with different point positions for key variables, and with different word sizes. Performance was measured by examining the training of the algorithm, and computing an average Mean Squared Error (MSE) for all of the training steps. To avoid large (poorly trained) masking small (correctly training) MSE figures, the MSE at each training step was capped to 2 – in general an average MSE around 2 suggests the equaliser has failed to train usefully.

4.2.1 Results

The RLS trained correctly in less than 32 samples, which was tested for a wide variety of channels and noise. Figures 4.3–4.7 show the effect on the MSE of varying different parameters for a channel of $[0, 0.2 + 0.1\text{i}, 1, 0.3 + 0.4\text{i}, 0.2 - 0.3\text{i}]$. On all these plots, the

line on the graph gives the MSE when using the floating point version as a comparison.

From these it can be seen that the algorithm is stable for the middle range of point positions - in other words retaining both integer and fractional bits. This test was repeated for a number of different simulated channels, and the behaviour was similar to the results shown here.

4.3 Conclusions

The particular RLS under test is not susceptible to finite precision effects, at least in the scenario it is being tested in. It is stable down to a 16 bit dataword, the optimal point

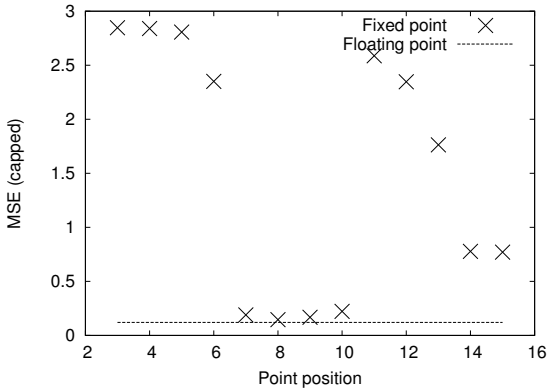


Figure 4.3: MSE of RLS, 16 bit datawords, varying default point position.

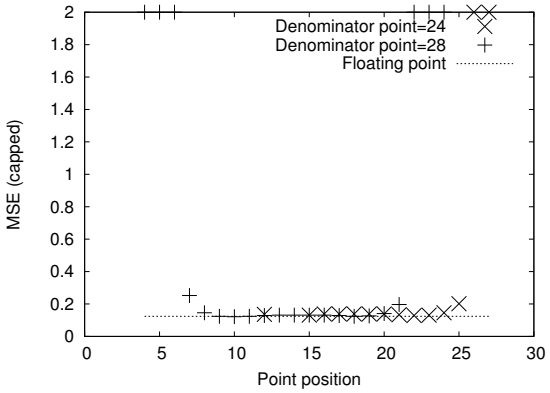


Figure 4.4: MSE of RLS, 32 bit datawords, varying default point position.

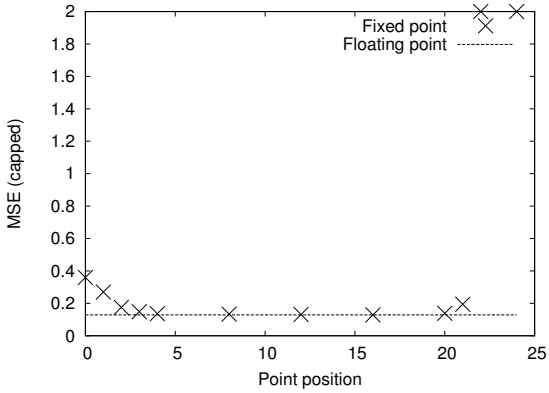


Figure 4.5: MSE of RLS, 32 bit datawords, point of denominator in equation 4.1.

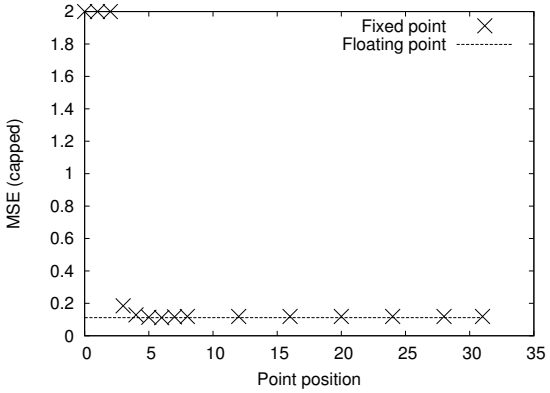


Figure 4.6: MSE of RLS, 32 bit datawords, point of inverse of denominator in equation 4.1.

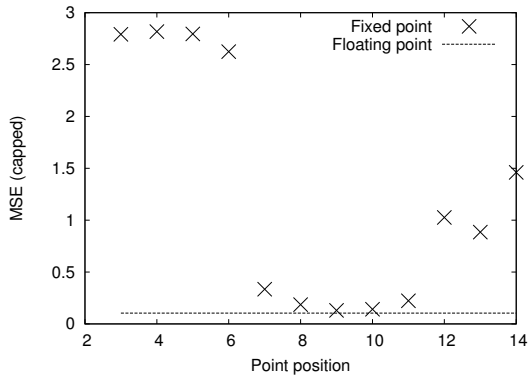


Figure 4.7: MSE of RLS, 16 bit datawords, point of $\boldsymbol{P}$ matrix in equation equation 4.1.

position being at bit 8. Hence an 8.8 (8 bits integer, 8 bits fractional) fixed point format is recommended for DSP implementation. [7] suggests that numerical instability is caused by the $\boldsymbol{P}$ matrix becoming non-positive definite, but these results suggest that control over the point for $\boldsymbol{P}$ can prevent this.

This C code was ported to an off-the-cheap fixed point DSP, substantially cheaper than the floating point device it was designed for, the only problems being caused by the DSP development system.

Chapter 5

DSP equaliser

Once decided on a data format for fixed point implementation of RLS, we can implement this on the DSP. First it is useful to develop a test harness around the DSP, to generate data suitable for filtering. It was decided to reimplement the data generation system in figure 4.1 in Matlab for ease of programming (to enable it to run on faster hardware, it was later converted to run under GNU Octave, a free Matlab lookalike – however this had little effect on its functionality, and can be disregarded). The generation system performs the following functions:

- Specify the channel including reflectors
- Generate a training sequence of random symbols
- Generate a data sequence of random symbols
- Concatenate both sequences, and map to QPSK symbols
- Convolve with the channel response
- Add white Gaussian noise to inphase and quadrature components
- Pass this data and clean training sequence to filter
- Calculate MSE of training and BER of data sequence
- When the known sequence is exhausted, continue to train on decisions made on the data

This is straightforward to implement in Matlab, and provides a means of testing equalisers. This then allows equalisers to be developed.

5.1 Zero forcing equaliser

Initially a simple Zero Forcing Equaliser was developed in Matlab to provide a standard to compare other equalisers.

As an illustration, consider a channel $[h_1\ h_2\ h_3]$ and tap vector $[x_1\ x_2\ x_3]$. Here we wish to solve:

$$\begin{pmatrix} h_1 & 0 & 0 \\ h_2 & h_1 & 0 \\ h_3 & h_2 & h_1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} desired \\ output \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$
$$\mathsf{A} \qquad \boldsymbol{x} \qquad = \qquad \boldsymbol{b}$$

where A is formed from the channel time shifted in various positions. Solution by Gaussian elimination gives the trivial solution $\boldsymbol{x}_1 = 1/a_{11},\ \boldsymbol{x}_i = 0$ for $i > 1$. We wish to find the least-squares solution of $\mathsf{A}\boldsymbol{x} = \boldsymbol{b}$.

Hence consider the QR-factorisation of A:

$$\begin{aligned} \mathsf{A} &= \mathsf{QR} \\ \mathsf{QR}\boldsymbol{x} &= \boldsymbol{b} \\ \mathsf{R}^T\mathsf{Q}^T\mathsf{QR}\boldsymbol{x} &= \mathsf{R}^T\mathsf{Q}^T\boldsymbol{b} \\ \mathsf{R}\bar{\boldsymbol{x}} &= \mathsf{Q}^T\boldsymbol{b} \end{aligned}$$

where $\bar{\boldsymbol{x}}$ is the least-squares solution. Thus:

$$\boldsymbol{c} = \mathsf{Q}^T\boldsymbol{b} \quad \text{and} \quad \mathsf{R}\bar{\boldsymbol{x}} = \boldsymbol{c}$$

which we can solve by Gaussian elimination. We can calculate the residual output $\boldsymbol{b'}$ from the filter by augmenting A with the residues of the channel:

$$\begin{pmatrix} h_1 & 0 & 0 \\ h_2 & h_1 & 0 \\ h_3 & h_2 & h_1 \\ 0 & h_3 & h_2 \\ 0 & 0 & h_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \end{pmatrix}$$
$$\mathsf{A'} \qquad \boldsymbol{x} \qquad = \qquad \boldsymbol{b'}$$

Therefore we can calculate the eye opening for the filter from $\boldsymbol{b'}$. However this assumes ideal arithmetic when filtering, which we do not have, and given the typically small nature of the residues means they may not be present in the output.

When implemented in Matlab, the above equation meant it was possible to compare the taps produced from other methods to see whether they were training correctly.

5.2 Recursive least-squares (RLS)

Once the data generation system was tested satisfactorily, it was decided to convert the existing RLS algorithm in C to DSP assembler. C is a useful intermediary between an algorithm in Matlab and assembler, because it expresses the underlying processing that needs to be performed more clearly, especially in terms of matrix and vector operations.

A start was made converting some of the key sections to DSP assembler. However, it ran against the following problems:

- RLS requires many vector/matrix operations - meaning lots of loops. Loops are quite tedious to code in assembler, and require care to get right (avoid off-by-one errors and so on).
- Keeping track of register usage in assembler is difficult because there aren't enough registers to hold all values being used, yet moving registers back and forth to memory is slow.
- The operations in RLS rely on results from each other. This makes dividing up into function calls difficult, without the use of memory as temporary storage (which is complex and slow).
- The DSP delay slot means that code is difficult to read and write. For optimal performance non-dependent instructions should be interleaved together to make use of the delay slot - this means effectively two algorithms could be running at one time. Thus the programmer needs to either interleave two programs, or reorder instructions in non-intuitive ways to gain maximum performance, neither of which aid readability.
- The delay slot also makes debugging difficult, because the programmer must remember that an operation does not complete until at least 2 instructions later. If this is forgotten, the code may fail in subtle ways which are often hard to detect.
- Getting debugging output from a program is hard, despite the availability of printouts. Due to the restrictive instruction length and DSP architecture, it can only print register values, with perhaps a few single character identifiers. Thus substantial effort if necessary to decipher the debug output. Since `PRINT` instructions take up an instruction word in the program, they affect the delay slot so hence adding/removing `PRINT` actions may change the behaviour of the program.

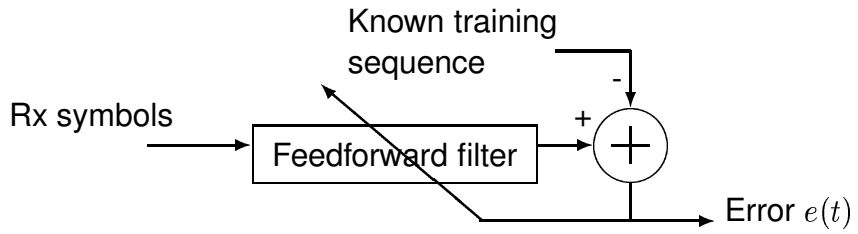


Figure 5.1: FIR-based linear equaliser

- There is no easy way to detect pipeline commits clashing (see section 3.2.2) other than running the program until a clash occurs. The assembler should be able to warn of this situation. However, either careful thought or trial and error is required to eliminate the clash, because placing `NOP` instructions to prevent one clash may trigger another.

Many of these problems could be solved by additional software tools, some ideas for which are presented in chapter 6. Since there was not enough time to look into these in depth, it was decided to implement the simpler LMS algorithm instead.

5.3 Least mean squares (LMS)

To start with the simplest algorithm, then work up if there was more time available, it was decided to implement a linear equaliser trained with the LMS. Instead of the complex feedforward/back and decision mechanism in figure 4.1, a simple FIR filter is used (see figure 5.1).

Therefore the DSP needs to do training and filtering of the data. This algorithm is given in table 5.3.

5.3.1 Implementation

While this appears straightforward, it must be remembered that the DSP is only capable of real arithmetic. Hence this needs to be split into real and imaginary parts for implementation. We also wish to optimise it for the multiply-and-accumulate architecture of the DSP.

Considering the error computation, we can divide this into:

Variables:
M = number of taps.
$\hat{\boldsymbol{w}}(n)$ = tap weight vector estimate at step n.
$\boldsymbol{u}(n)$ = input sample vector.
$d(n)$ = desired output response.
$e(n)$ = error from desirec response.
μ = step size.

For step n of the LMS algorithm:

$$\text{Compute error } e(n) = d(n) - \hat{\boldsymbol{w}}^H(n)\boldsymbol{u}(n)$$

(where A^H is the Hermitian conjugate of $\mathsf{A} = \mathsf{A}^{T*}$)

$$\text{Compute tap updates } \hat{\boldsymbol{w}}(n+1) = \hat{\boldsymbol{w}}(n) + \mu\boldsymbol{u}(n)e^*(n)$$
$$\text{Filter output } q(n) = \hat{\boldsymbol{w}}(n)\boldsymbol{u}(n)$$

Table 5.1: Least mean squares algorithm with linear filtering

$$\hat{\boldsymbol{w}}^H(n)\boldsymbol{u}(n) = \sum_{p=1}^{M} \left(w_{ip} - \mathrm{i}w_{qp}\right)\left(u_{ip} + \mathrm{i}u_{qp}\right) \tag{5.1}$$

(where x_{ip} is the in-phase component of $x(n)_p$)

$$e(n) = \left(d_i(n) - \sum_{p=1}^{M}\left(w_{ip}u_{ip} + w_{qp}u_{qp}\right)\right) + \mathrm{i}\left(d_q(n) - \sum_{p=1}^{M}\left(w_{ip}u_{qp} - w_{qp}u_{ip}\right)\right)$$

This we can do by evaluating each component separately. The sums can each be done on the DSP accumulator, which means the adds are performed with 32 bit precision, then only converted down to 16 bit after all have been done.

Considering the tap updates, we can take this in sections:

$$\boldsymbol{u}(n)e^*(n) = \left(\boldsymbol{u}_i(n) + \mathrm{i}\boldsymbol{u}_q(n)\right)\left(e_i^*(n) + \mathrm{i}e_q^*(n)\right)$$

$$= \left(\boldsymbol{u}_i(n)e_i^*(n) - \boldsymbol{u}_q(n)e_q^*(n)\right) + \mathrm{i}\left(\boldsymbol{u}_i(n)e_q^*(n) + \boldsymbol{u}_q(n)e_i^*(n)\right)$$

Thus to optimise the arithmetic, we can precalculate:

$$\alpha_i = \mu e_i^*(n), \alpha_q = \mu e_q^*(n)$$

then:

$$\hat{\boldsymbol{w}}_{\boldsymbol{i}}(n+1) = \hat{\boldsymbol{w}}_{\boldsymbol{i}}(n) + \alpha_i \boldsymbol{u}_i(n) + \alpha_q \boldsymbol{u}_q(n)$$

for which each row can all be done on the accumulator (the first term can be added with `MAC 1,`$\hat{w}_{ip}(n)$ or alternatively done in a register, in a tradeoff between speed and accuracy). A similar process applies to the quadrature component.

Similarly, we can express the filtering as:

$$q(n) = \left(\sum_{p=1}^{M} (w_{ip} u_{ip} - w_{qp} u_{qp}) \right) + \mathrm{i} \left(\sum_{p=1}^{M} (w_{ip} u_{qp} + w_{qp} u_{ip}) \right) \tag{5.2}$$

each component of which is easily calculated on the accumulator.

This algorithm was implemented on the DSP in assembler. The values used for training and filtering are stored on a small shift register containing the last M received symbols, rather than the received symbol array itself. This is cleared to zero at startup, to ensure that symbols received before correlation are ignored (these are not stored in this implementation, so attempting to read them would read undefined memory). The system runs over the data twice, once in training mode, and once in filter mode. Training mode performs tap updates for the length of the training data. Filter mode does filtering for the whole dataset. Noticing that the expansion of equation 5.1 and equation 5.2 are the same, this code is reused for both training and filtering. The error is also stored to examine the progress of training.

While this algorithm is fairly straightforward to implement, it is very difficult to debug. This is because of the limited debugging facilities available on the DSP. Notably, the debug output is cryptic, and it is difficult to control so it either produces too much, or not enough to be helpful. Also the extra precision in the accumulator means it can't easily be compared with an algorithm running in another language since their multiplies will be subtly different. If the algorithm begins to fail many cycles into training it is difficult to identify exactly where, and which operation has gone wrong – it may be a combination building up over a number of cycles. While LMS is not sensitive to finite precision errors, many programming errors cause it to diverge yet are not easy to identify. In particular, too large a value of μ or getting a sign wrong are simple errors that are difficult to detect.

Such bugs may cause the filter to fail to train at all (see figure 5.2), diverge slowly (see figure 5.3) or train erratically (figure 5.4).

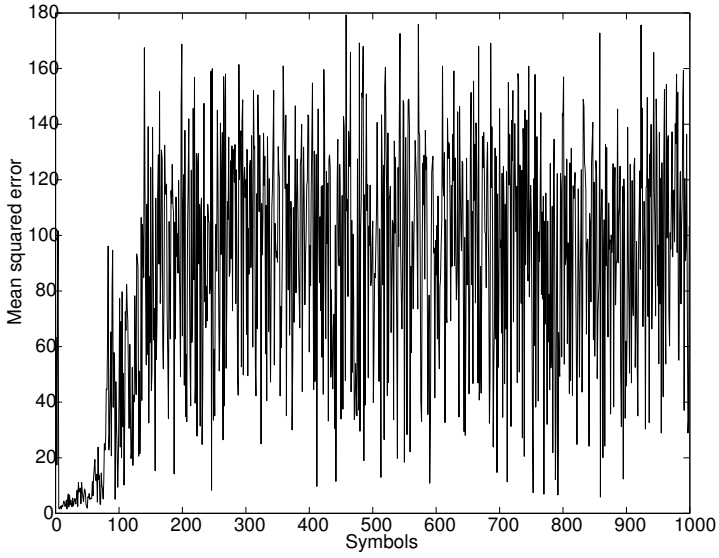


Figure 5.2: Blowing up LMS, with $\mu = 0.1$ (too high)

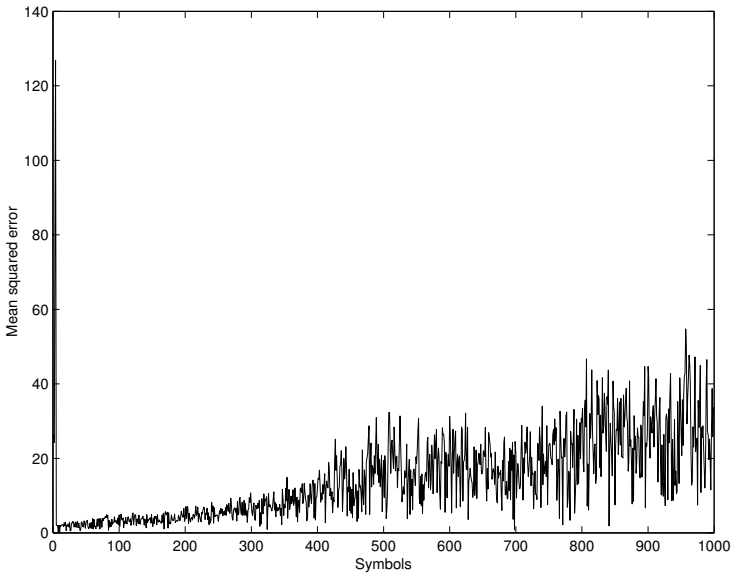


Figure 5.3: Diverging LMS, using initial coefficients chosen by ZFE, $\mu = 0.01$

To aid testing, a 32 bit emulated DSP was developed. Since the DSP does not inspect the data directly, but merely performs arithmetical operations of it, it is possible to change the size of bus widths within the DSP without affecting the computational function it performs. Registers can be extended to 32 bits, the accumulator to 64 bits,

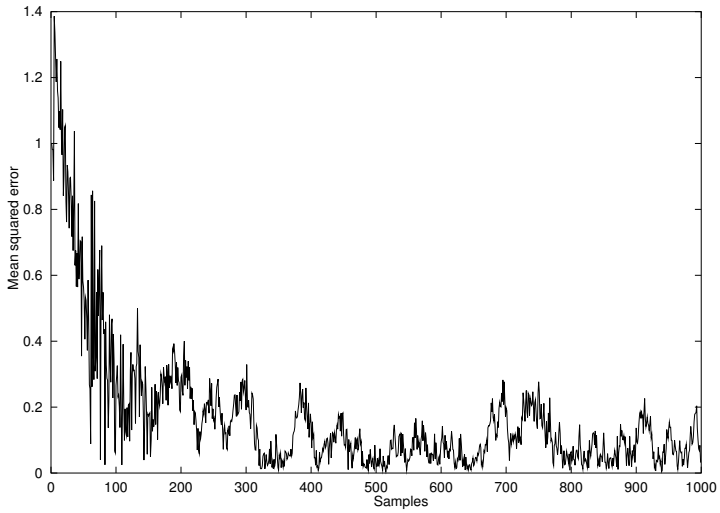


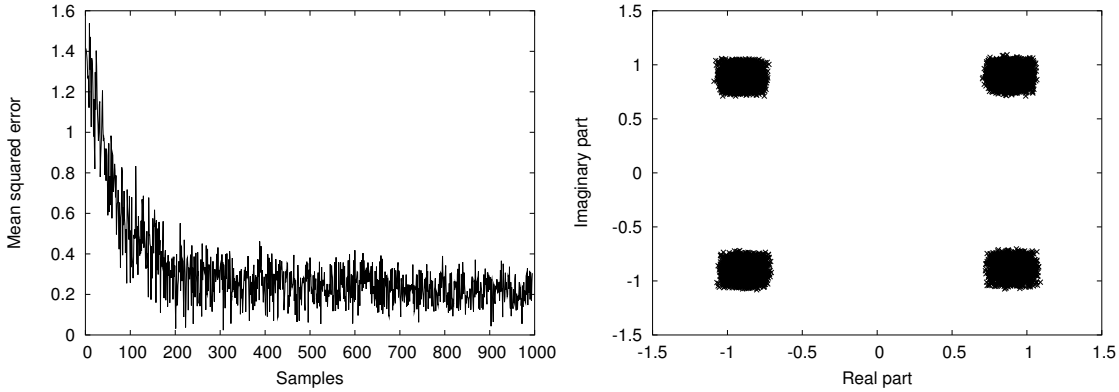
Figure 5.4: Erratic LMS (caused by numerical errors in algorithm)

and instructions modified for the correct behaviour on longer words. If a program does not, for example, test the sign bit of a value directly (for which it would need to know which bit is the sign bit), it can work equivalently on data of the relevant length without knowledge of what that length is. Since Matlab supplies the data into the DSP, only this requires modification for 32 bit support, not the assembly code itself.

The same technique with the fixed point macros was applied to the C version, to allow floating point, 32 bit fixed point and 16 bit fixed point versions. To aid debugging, the 16 bit C version was modified to perform arithmetic exactly as the DSP does (with a 32 bit wide accumulator). By making it produce debug output identical to that of the DSP, it is possible to use a comparison tool on the C and DSP output. This allowed tracking down of the final bugs, one of which was found to be in the emulator (incorrect rounding). After an extended period, this produced a system which trained and filtered correctly.

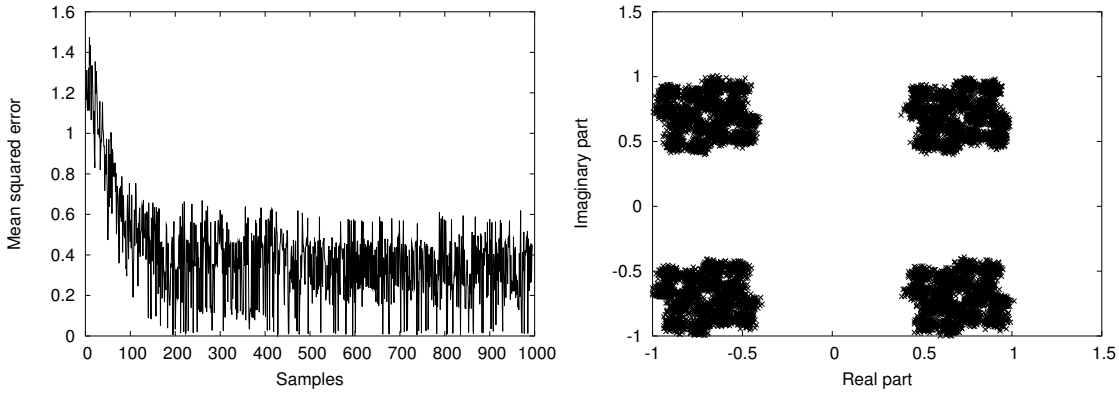
5.3.2 Results

Once the code had been debugged (which took about a month), the output for a noiseless channel looks something like figure 5.5 (C version) and figure 5.6 (DSP version). This shows the DSP is filtering almost correctly, but appears to have slight differences to the C version, causing inaccuracies. In particular, it seemed to be sensitive to the placement



(a) Training (b) Output constellation

Figure 5.5: LMS in C - noiseless channel, large reflector: SNR=100dB, channel = [1,0,0.4], 1000 training symbols, 10000 data symbols, 8 taps

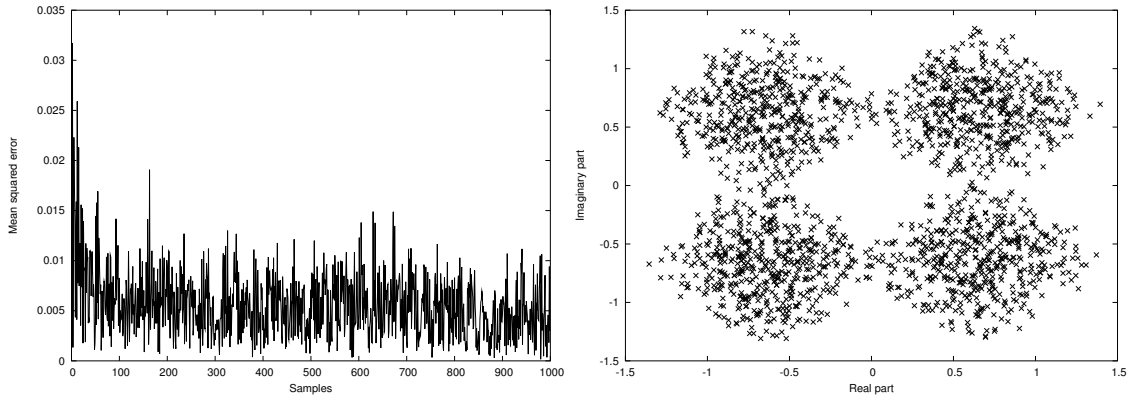


(a) Training (b) Constellation

Figure 5.6: LMS on DSP: SNR=100dB, channel = [1,0,0.4], 1000 training symbols, 10000 data symbols, 8 taps

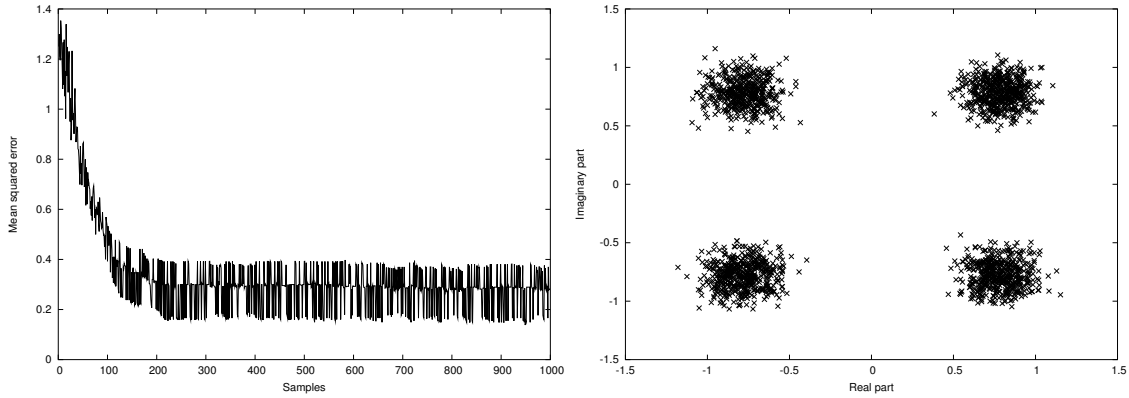
of printout instructions, suggesting there may be a subtle timing problem somewhere within it. When compared to the C version step-by-step, the outputs were identical, suggesting this bug may only trigger in certain rare circumstances.

These training sequences have been averaged over many runs, however this does not make the curves any smoother, which suggests the apparent noise on them is due to finite precision effects rather than noise. Figure 5.7 shows the results with very little noise, but a channel response which is longer than the filter, which shows errors because the filter cannot adequately echo cancel. This suggests that fixed point effects are the dominant



(a) Training (b) Constellation

Figure 5.7: LMS on DSP: too short filter: SNR=100dB, channel=[1,0,0.2+0.1i,0.2+0.5i], averaged over 1000 runs, 1000 training symbols, 10000 data symbols, 5 taps

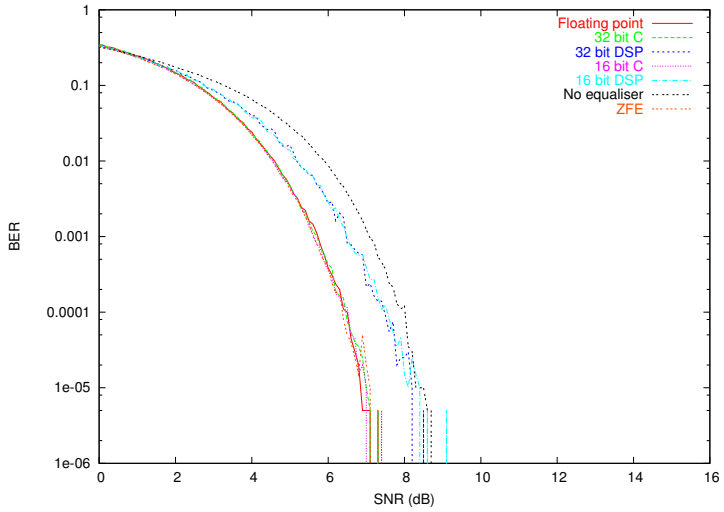


(a) Training (b) Constellation

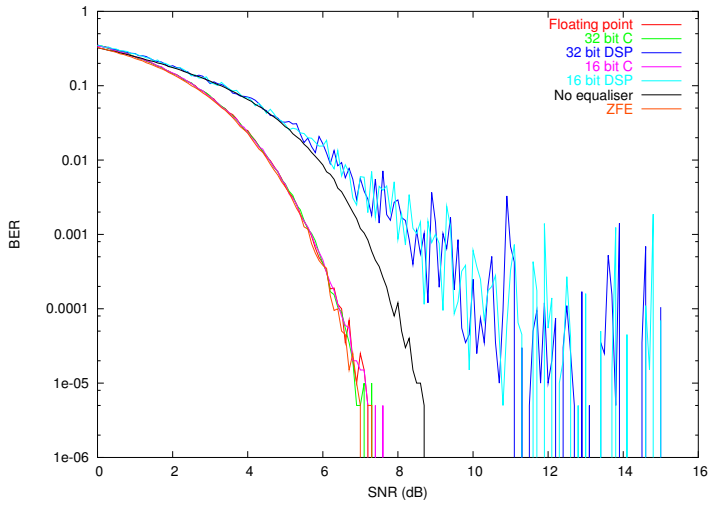
Figure 5.8: LMS on DSP, noisy channel, small reflector: SNR=10dB, channel=[1,0,0.1], 1000 training symbols, 10000 data symbols, 5 taps

restriction.

To evaluate the behaviour of the filter, the Matlab code also calculates the bit error rate (BER) of the filter output. The SNR-BER curves can be seen in figure 5.9(a) and figure 5.9(b) for 1000 and 32 training symbols respectively. It should be noted that because the DSP only supports signed 16 bit arithmetic, it can only address 32768 memory locations, which restricts the data length to about 10000. This means the smallest BER that can be measured is 10^{-4}. A better idea of the SNR is measured by averaging the results of multiple runs, but this involves training to potentially different values each



(a) 1000 training symbols



(b) 32 training symbols

Figure 5.9: BER v SNR: 10000 data symbols, 5 taps, channel [1,0.4], averaged over 20 runs.

time.

These plots show that the C versions perform slightly better than the DSP versions, and that the DSP version does not train as quickly and so has not trained properly after 32 symbols. This can be seen in figure 5.5(a) and figure 5.6(a), where the steady-state MSE for the DSP is greater than that of the C version. From figure 5.9(a), we can measure that the improvement in the SNR due to the C filter is about 18dB, and the 6dB due to the DSP filter. Clearly this bug is affecting the DSP's output.

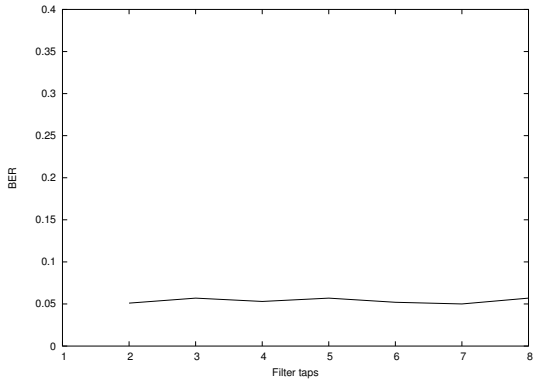


Figure 5.10: BER when varying filter taps: 3dB SNR, 1000 training samples, channel=[1,0,0.4]

The BER when varying the number of taps was also investigated. For an SNR of 3dB, with 1000 training symbols, figure 5.10 shows the effect of 2 to 8 taps. Such low SNR is required to get any bit errors at all, but this means the filters train incorrectly, so this is perhaps not an ideal test. A lot more CPU time would provide a better picture at lower BER.

Also the frequency response of the channel and filter are shown in figure 5.11. This shows that while the filters have trained to the response correctly, it cannot compensate for the frequency nulls caused by the channel. The ZFE is expected to provide an inverse of the channel, subject to limits on the number of taps and representation used (here floating point). LMS trades off noise amplification with received signal power. Here, in the presence of no noise, they should be the same. Figure 5.11 shows that they have roughly the same envelope (which is the inverse of the channel), but at different amplitude levels. This is likely to be due to the size of the representations used in training. This could be solved by either sacrificing precision from the fractional part by moving the binary point, or using a longer word length as discussed earlier. The DSP version has a lower amplitude response than the C version, agreeing with the flaw in training noted above.

The system must meet the deadline of training within 4ms. If we disable the filtering function (not included in this target), 1000 training symbols requires 993,546 cycles which take 12.4ms. This is too slow. If we reduce the training symbols to 250 (still more than enough as the above shows), the system trains in 251,795 cycles or 3.15ms.

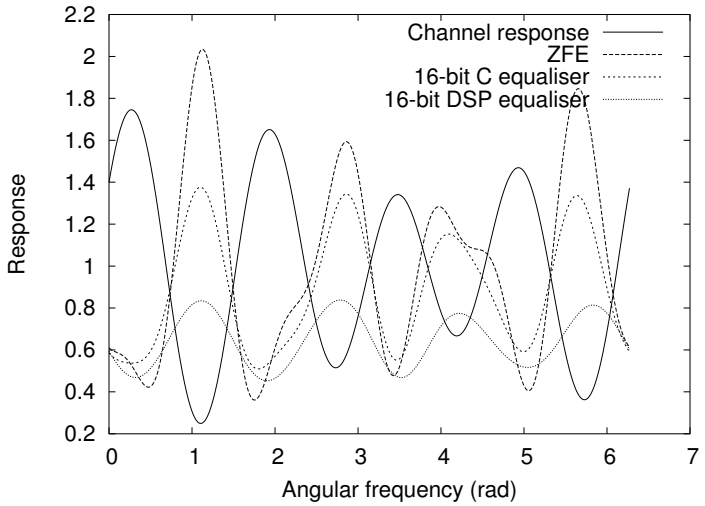


Figure 5.11: Frequency response of channel [1,0,0.2+0.1i,0.2+0.5i] and filters. SNR=100dB, 8 taps.

5.4 Conclusion

This section has shown how a filter system may be implemented for a realistic situation. It has examined several ways that it can be trained, and considered the software engineering implications of implementing them on a real platform. It has shown that a simple processor such as the one used here can be used to performing signal processing computations, with restrictions based on memory length and precision, and we have seen the effects of both of these. The filter as implemented in C trains correctly, and figure 5.11 shows its performance in the frequency domain and figure 5.9 the BER performance. The DSP version has a slight flaw in training, although the C version shows this is not related to the precision – further time spent debugging is required to iron out the last few problems. It can be seen that while precision affects the frequency response, it does not affect the BER. Hence LMS on a low precision processor such as this is a useful and valid tool. With 250 training samples it meets the real-time deadline imposed on it.

Chapter 6

Suggestions for further work

A multi-disciplined project such as this inevitably throws up many ideas of ways to pursue it further. Some of this are listed here – many were considered, but were not possible to be explored in the limited time available.

6.1 DSP tools

As the attempt at RLS implementation has shown, the tools available for programming the DSP are lacking in some respects. There are a number of ways they could be improved.

6.1.1 Scheduling assembler

The main problem is the delay slot(s), the need to code taking this into account and it catching the programmer unawares when this is forgotten. A scheduling assembler would be presented with an instruction stream written in sequential order (ie assuming no delay slot) – this instruction stream is easier to program, since there is no need to worry whether instructions have completed. The assembler takes this, and analyses the dependencies between each instruction in the stream. It reorders the instructions to make best use of the delay slot, avoid pipeline clashes, and inserts `NOP` instructions where required.

So, the input instruction stream:

```
1: LDIEXT  1,r0
2: ADD     r5,r0,r0
3: MAC     r1,r2
4: SUB     r2,r0,r1
```

(2 depends on 1, 4 depends on 2)

Might be rescheduled to:

```
LDIEXT  1,r0
MAC     r1,r2
ADD     r5,r0,r0
NOP
SUB     r2,r0,r1
```

This makes it impossible to be caught out by accessing registers before a result has been committed.

A simpler idea is just to insert a number of `NOP`s after each instruction to achieve the same result, but this has a significant impact on speed and code density.

The existing assembler could be rewritten in a faster language such as C, perhaps using ready-built tools such as `lex` and `yacc` as was suggested in section 3.3.

6.1.2 C compiler

If more complex algorithms are intended to be run on the DSP, the instruction set may become limiting. Many loops or array accesses may involve much fiddling with array indices or loop counters, which is hard to get right and distracts from the main purpose of the code. It may also be useful to port code from other platforms to the DSP.

Some kind of compiler is the obvious solution. Without a scheduling assembler, a compiler is also useful because it means programs can be assembled from tried and tested blocks, which each take care of the delay slot within them. An optimising compiler can remove unwanted instructions, and do the scheduling itself.

Rather than write a compiler from scratch, it may be possible to port an existing compiler. GCC[5] is the most widely used free retargetable C compiler. However it is designed primarily for heavyweight CPUs, and may have difficulty with our 16 bit DSP. It is also estimated that it takes about a month of work to produce a working backend without any optimisation, so this may be unsuitable. Another contender is LCC[6], which is designed for smaller CPUs and is quicker to port to a new architecture. However it has fewer optimisations and no support for rescheduling, which may cause problems in this case. Both of these were seriously considered for this project, but it was decided that there was not enough time to pursue them. Practically, the LCC documentation is mostly in book form, which is difficult to get hold of.

6.1.3 Compiling assembler

This is a half-way house between a compiler and an assembler. It enables the normal instruction set to be used, but this is compiled into a number of different instructions. So, for example, a virtual 32 bit DSP could be created by forming instructions referencing 32 bit registers (in memory, or a combination of two 16-bit registers). As far as the programmer is concerned it is running on a 32 bit wide DSP, but the underlying hardware implements it using 16 bit instructions. This is useful if the algorithm we wish to run

requires more precise arithmetic, but we cannot change the hardware to accommodate this.

Hence, a 32-bit instruction:	Might be implemented as:
`ADD r0,r1,r2`	`ADD r0,r2,r4` `ADDC r1,r3,r5`
(where `r0`–`r2` are 32-bit wide registers)	($\texttt{r0}_{32}$ stored in `r0`,`r1`, $\texttt{r1}_{32}$ in `r2`,`r3` etc)

Floating point arithmetic could be implemented in a similar way. This gives a trade-off between accuracy and speed/code size, but can still be done when the DSP hardware has been frozen. These measures enables code making use of different precisions to be written, without changing the underlying hardware.

6.1.4 Debug output

The previous chapters have highlighted the problems with the cryptic nature of debugging printouts. This is caused both by the DSP architecture (that it can't read instruction memory other than when fetching instructions) and the software tools (the assembler doesn't support more than one instruction generated per input statement). It would be helpful to support `printf` style debugging from the DSP.

This could be done by providing an external string table, perhaps in data RAM or in a separate memory addressable by special instructions on the DSP. The assembler could arrange to build this table from print statements in the source, and call a function from the assembler which would extract the statement and print the relevant text. At this level this can be implemented on both emulator and hardware. To speed up execution, most of this could be performed by the emulator, with perhaps one instruction (containing perhaps an 8-bit ID of the string to print out) invoking it. For maximum effectiveness, the string and registers required should be built by the assembler from directives in the assembly code, so that the string is next to the code it is called from.

6.1.5 Emulator

The current emulator on a 1GHz PC running Linux manages about 0.7 MIPS (Millions of Instructions Per Second). This is about 1/115 of full speed. It might be useful to have a faster emulator (say to test algorithms with real-time inputs and outputs). One way this could be achieved is the use of dynamic recompilation, which involves taking the

code for one CPU and compiling it, whilst being executed, for another[10]. This has the potential for an order of magnitude speed increase, although it must be targeted at a specific CPU architecture – reducing the portability of the emulator.

An alternative is to compile the assembly direct into target machine code, although the disadvantage is that some of the debugging features may be lost, and care is required to preserve all pipeline hazards visible to the programmer.

6.2 Equalisation

With LMS working, it is possible to consider more complex algorithms. A decision feedback equaliser trained with LMS would not be much more complex to write, although as with LMS debugging would take up much of the time. RLS is also possible, although it would benefit from better tool support (scheduling assembler or compiler).

6.2.1 Complex number support

To aid the implementation of an equalisation algorithm, it may be useful to consider adding hardware support to the DSP. Notably, complex number support would be helpful as it would reduce the algorithmic complexity required by splitting each complex number into real and imaginary parts.

It could be dealt with using a compiling assembler to generate code to drive the real DSP, but keeping track of real and imaginary parts without the programmer having to worry about them. This may generate sub-optimal code because tricks such as conjugation cannot be included in other operations, but this is a small price to pay for making the programmer's job easier, as it can be hand-optimised later.

An alternative method is to add extra hardware. In this case, it could be done by each register having real and imaginary parts, with a real and imaginary pipeline. These pipelines operate independently, but join when a multiply is concerned (see figure 6.1).

This requires about twice as much hardware for the ALU and registers, and four multipliers (to consider all parts of a complex multiply). The other problem is memory writes requires two writes to memory at once. This could be handled by having memory split into real and complex banks. Extra instructions might be required to do different operations on real and imaginary datapaths — such as finding the complex conjugate. However it would speed up both coding and execution dramatically — the latter at least

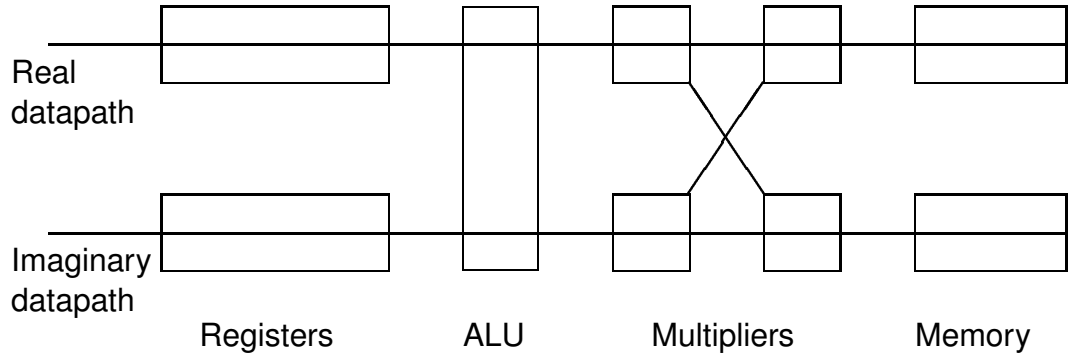


Figure 6.1: Suggested complex DSP datapath

a factor of two since the majority of operations involve complex data.

6.2.2 Vector support

Another use of hardware would be to perform multiple operations in parallel so, for example, all of the tap vector could be updated in one operation. This raises extra questions about how it would be achieved (are values loaded into registers 'by hand', or do we try to load a whole vector at once? If the latter, how do we get over hardware restrictions about how many words we can read at once?). It is also the case that, while parallelisable, some operations don't fit into this pattern – for example forming a dot product has some parallel multiplies, but then a serial add.

While n channel vector support would increase the hardware by n times, it would not be used for all operations, and hence the speedup would be much less than n times. It might still be worth it if it enabled meeting a deadline which would not otherwise be met.

6.2.3 Other structures

Analysis of the particular equalisation algorithm in use may lead to other parallel structures. For example, Asai and Matsumoto[1] detail an RLS processor based on systolic arrays. This involves a tree shaped array of computation cells (figure 6.2), through which data flows on a heartbeat (systole). This makes a very fast implementation, at the expense of much hardware. It may be possible to modify this for the DSP, to produce a unit performing the job of one cell. Data is passed through this computation unit as

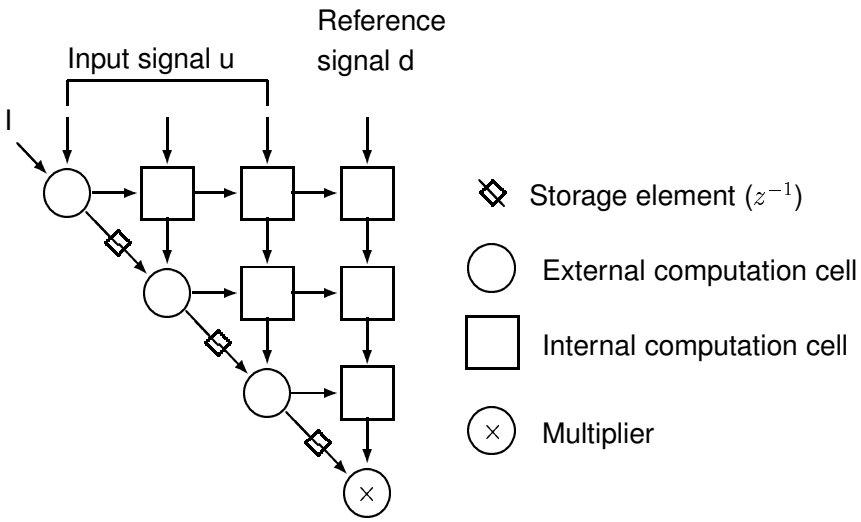


Figure 6.2: Systolic array structure

controlled by the instruction stream, with the result that the unit can be reused in one training cycle, vastly reducing the hardware impact. If pipelined, multiple data can be passing through this unit at once, allowing even more efficient use of the hardware. It is also the case that the array can be increased to arbitrary size by modifying the code, which is not possible with a conventional hardware array.

6.3 Conclusion

Many of these areas may be worth pursuing. On the software side, a scheduling assembler or compiler would make the programmer's job much easier. On the hardware side, complex support gives the best performance/extra hardware ratio. Systolic array support requires work to determine even if it is feasible to consider.

Chapter 7

Conclusions

This project has demonstrated the design process involved in producing a programmable signal processing system, considering aspects from many different angles. It has investigated the facets of hardware architecture, software development tools, signal processing infrastructure, software engineering and algorithmic implementation and performance.

This has a number of important implications:

- It has shown that hardware architecture has a strong bearing on the performance of the system. Approximately a 38% speed increase for the algorithms in use is shown by the addition of extra instructions by the use of 12.5% of opcode space. Other instructions allow software structures such as subroutines that would be impossible otherwise. Restrictions on this such as hardware usage and timing must always be considered when making these tradeoffs.
- The software tools provided contribute greatly to the productivity or otherwise of the programmer. These may or may not require support in the hardware architecture. Being able to extend the tools when required adds greatly to their flexibility.
- Under 16-bit 8.8 fixed point arithmetic, the vanilla RLS algorithm is stable, at least under the conditions seen here.
- Large-scale signal processing tasks such as RLS are not suited to programming in assembler, particularly due to their repetetive nature and need to use memory as temporary storage. In particular, if the tool support is lacking, debugging such programs can be a slow and painstaking task.
- Modern CPUs are sometimes difficult to program in assembler, due to exposure to their inner workings. Tool support can help, whilst a slightly higher level language may be significantly easier whilst causing little performance hit.
- LMS is insensitive to the precision of computation, as are finite precision filters trained by it. It is well suited to a low precision processor, and has a useful benefit over an unfiltered system. The implementation here works with some caveats, filters correctly in limited 8.8 precision, and meets its real-time deadlines.

Bibliography

[1] T. Asai and T. Matsumoto. A systolic array RLS processor. *IEICE Trans. on Comm.*, E84B(5):1356–1361, May 2001.

[2] G.E. Bottomley and S.T. Alexander. A novel approach for stabilizing recursive least-squares filters. *IEEE Trans. Signal Process.*, 39(8):1770–1779, August 1991.

[3] P.S.R. Diniz and M.G. Siqueira. Fixed-point error analysis of the QR-recursive least-square algorithm. *IEEE Trans. Circuits Syst. II–Analog Digit. Signal Process.*, 42(5):334–348, May 1995.

[4] Free Software Foundation. GNU binutils. http://www.gnu.org/software/binutils/binutils.html.

[5] Free Software Foundation. The GNU compiler collection. http://www.gnu.org/software/gcc/gcc.html.

[6] C.R. Fraser and D.R. Hanson. *A retargetable C compiler: design and implementation*. Addison-Wesley, 1995.

[7] Simon Haykin. *Adaptive Filter Theory*. Prentice Hall, fourth edition, 2002.

[8] A.P. Liavas and P.A. Regalia. On the numerical stability and accuracy of the conventional recursive least squares algorithm. *IEEE Trans. Signal Process.*, 47(1):88–96, January 1999.

[9] Malcolm Sellars. *Low complexity equalization techniques for broadband indoor radio*. PhD thesis, Cambridge University Engineering Department, March 2000.

[10] David Sharp. *tARMac: A dynamically recompiling ARM Emulator*. BSc dissertation, University of Warwick, 2001.

Appendix

Fixed point macros: source code

```
/* fixed.h:  Fixed point macros with movable points
 * by Theo Markettos 2001
 */

#ifndef FIXED_H
#define FIXED_H

#include <assert.h>

/* define this to using floating point values throughout instead of fixed point */
//#define FLOAT

/* default fixed point position, used for variables where the point is not explicitly
 * specified */
#define POINT_DEFAULT 16

/* Used in computing inverse of large number - this is the position of the point in
 * the 1.0 used as numerator (this is a tradeoff between integer and fraction parts
 * in the result) */
#define ONEOVERLARGE_POINT_ONE 13

/* Likewise, when computing inverse of a small number, this sets the point of the result
 * (again a tradeoff between integer and fractional parts */
#define ONEOVERSMALL_POINT_RESULT 4

/* _int64 is an MSVC type - change this to whatever the local 64 bit integer type is
 * (eg long long for gcc) */
#define int64 _int64

/* number of bits in our default representation */
#define FIXED_BITS 32

/* our representation of a fixed point number and a long fixed point number -
 * long being used for divides and multiplies
 */
typedef long fixed;
typedef int64 fixedLong;

/* definitions of constants - these depend heavily on the
 * representation and default point */
#define FIXED_ONE (1u<<POINT_DEFAULT)
#define FIXED_MINUS_ONE ((-1)<<POINT_DEFAULT)

//#define USE_OWN_LONG_MULTIPLY

#ifndef FLOAT

/* the sign is held in the MSB, irrespective of where the point is */
#define FIXED_SIGN(value,point) ((value & ((fixed) (1)<<(FIXED_BITS-1))) ? -1 : 1)

/* define bitmasks for different parts of a number
 * if point is at place 16, 1<<16 is 0x00010000 (32 bit representation),
 * so 1<<16 - 1 = 0x0000ffff
 * similarly for the integer part, except shifted */
#define FRACTIONAL_MASK(point) (((fixed) (1)<<(point))-(fixed) (1))
#define INTEGER_MASK(point) ((((fixed) (1)<<(FIXED_BITS-point))-(fixed) (1))<<point)

/* find the absolute value of a number: */
#define FIXED_ABS(value,point) FIXED_SET_SIGN(value,point,1)

/* Set the sign of a number:
 * if the sign is wrong, it is negated */
#define FIXED_SET_SIGN(value,point,sign) ((FIXED_SIGN(value,point)==sign) ? value : ((~value)+(fixed) (1)))

/* C can't cope if we try to do shifts with negative shift values, so here we
 * have some macros to convert from a<<-b to a>>b and vice versa */
#define SHIFT_RIGHT(value,shift) (((shift)>0) ? ((value)>>(shift)) : ((value)<<(-(shift))))
#define SHIFT_LEFT(value,shift) SHIFT_RIGHT(value,(-(shift)))

/* When we're just working with raw figures, not our result type, we may just want
 * to assign (eg long aFixedDataItem = double x;) another type to it
 * These do this, but don't call the assert because sometimes we can't
 * consider printf("%d\n",MAKE_FIXED_....);
 */
```

```
#define MAKE_FIXED_INT(val,pnt) (fixed) ((val)<<(pnt))
#define MAKE_FIXED_FLOAT(val,pnt) (fixed) ((val)*((float) ((fixed) (1)<<pnt)))
#define MAKE_FIXED_DOUBLE(val,pnt) (fixed) ((val)*((double) ((fixed) (1)<<pnt)))

/* convert a fixed to a double:
 * 'declaration':
 * double FIXED_TO_DOUBLE(fixed value); */
#define FIXED_TO_DOUBLE(val,point) (((double) (val))/((double) ((fixed) (1)<<(point))))
#define FIXED_TO_FLOAT(val,point)  (((float ) (val))/((float ) ((fixed) (1)<<(point))))
#define FIXEDLONG_TO_FLOAT(val,point)  (((float ) (val))/((float ) ((fixedLong) (1)<<(point))))

/* A useful value to assign, so it's clear what is a fixed type */
#define FIXED_ZERO (fixed) 0

/* return destValue (ie result=FIXED_ASSIGN(srcPoint,srcValue,destPoint)
 * means result=srcValue) */
#define FIXED_ASSIGN(srcValue,srcPoint,destPoint) (SHIFT_RIGHT(srcValue,srcPoint-destPoint))

fixed fixed_div(fixed numValue, int numPoint, fixed denValue, int denPoint, int resultPoint);

fixed fixed_mul(fixed aValue, int aPoint, fixed bValue, int bPoint, int resultPoint);

fixedLong fixed_long_mul(fixed aValue, int aPoint, fixed bValue, int bPoint, int resultPoint);

/* return a+b */
#define FIXED_ADD(aValue,aPoint,bValue,bPoint,resultPoint) (aPoint>bPoint) ? \
        SHIFT_LEFT(((bValue) + SHIFT_RIGHT((aValue),((aPoint)-(bPoint)))),((resultPoint)-(bPoint))) : \
        SHIFT_LEFT(((aValue) + SHIFT_RIGHT((bValue),((bPoint)-(aPoint)))),((resultPoint)-(aPoint)))

#define FIXED_SUB(aV,aP,bV,bP,rP) FIXED_ADD(aV,aP,(-bV),bP,rP)

#define FIXED_MUL fixed_mul
#define FIXED_DIV fixed_div
#define FIXED_LONG_MUL fixed_long_mul

/* ##V are values, ##P are point positions corresponding to values,
 * iP is intermediate point (used for temporary values)
 * rP is result point */
#define FIXED_SUM_OF_PRODUCTS(a1V,a1P,a2V,a2P,b1V,b1P,b2V,b2P,iP,rP) \
  (FIXED_ADD((FIXED_MUL(a1V,a1P,a2V,a2P,iP)),iP, (FIXED_MUL(b1V,b1P,b2V,b2P,iP)),iP, rP))

#define FIXED_DIFFERENCE_OF_PRODUCTS(a1V,a1P,a2V,a2P,b1V,b1P,b2V,b2P,iP,rP) \
  FIXED_SUB((FIXED_MUL(a1V,a1P,a2V,a2P,iP)),iP, (FIXED_MUL(b1V,b1P,b2V,b2P,iP)),iP, rP)

fixed fixed_oneoversmall(fixed denValue, int denPoint, int resultPoint);
fixed fixed_oneoverlarge(fixed denValue, int denPoint, int resultPoint);

#define FIXED_ONEOVERLARGE fixed_oneoverlarge
#define FIXED_ONEOVERSMALL fixed_oneoversmall

#else /* not FLOAT */

#define int64 _int64
#define fixed float

#define SHIFT_RIGHT(value,shift) (value)
#define SHIFT_LEFT(value,shift) SHIFT_RIGHT(value,(-(shift)))

#define MAKE_FIXED_INT(val,pnt) (fixed) (val)
#define MAKE_FIXED_FLOAT(val,pnt) (fixed) (val)
#define MAKE_FIXED_DOUBLE(val,pnt) (fixed) (val)
#define FIXED_TO_DOUBLE(val,point) (((double) (val)))
#define FIXED_TO_FLOAT(val,point)  (((float ) (val)))

/* A useful value to assign, so it's clear what is a fixed type */
#define FIXED_ZERO (fixed) 0
#define FIXED_ONE 1

#define FIXED_ASSIGN(srcValue,srcPoint,destPoint) srcValue
#define FIXED_ADD(aValue,aPoint,bValue,bPoint,resultPoint) (aValue+bValue)
#define FIXED_SUB(aV,aP,bV,bP,rP) (aV-bV)
#define FIXED_MUL(aV,aP,bV,bP,rP) (aV*bV)
#define FIXED_DIV(aV,aP,bV,bP,rP) (aV/bV)

#define FIXED_SUM_OF_PRODUCTS(a1V,a1P,a2V,a2P,b1V,b1P,b2V,b2P,iP,rP) ((a1V*a2V) + (b1V*b2V))
#define FIXED_DIFFERENCE_OF_PRODUCTS(a1V,a1P,a2V,a2P,b1V,b1P,b2V,b2P,iP,rP) ((a1V*a2V) - (b1V*b2V))

#define FIXED_ONEOVERLARGE(dV,dP,rP) (1/dV)
#define FIXED_ONEOVERSMALL(dV,dP,rP) (1/dV)
#define FIXED_SIGN(v,p) v<0 ? 1 : -1
#define FIXED_ABS(v,p) ((float) fabs(v))
#define FIXED_SET_SIGN(v,p,s) (FIXED_SIGN(v,p)==s) ? v : (-v)

#endif /* not FLOAT */
#endif /* not FIXED_H */
```