Increasing the Spectral Efficiency of Continuous Phase Modulation Applied to Digital Microwave Radio: A Resource Efficient FPGA Receiver Implementation

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Engineering
in
Electronics and Computer Systems Engineering

at Massey University, Palmerston North, New Zealand

Andrew B. Bridger
B.E. (Hons 1)

November 2, 2009

Abstract

In modern point to point microwave radio systems used to backhaul cellular voice and data traffic, quadrature amplitude modulation (QAM) is the norm. These systems require a highly linear power amplifier which is expensive and has relatively low power efficiency. Recently, continuous phase modulation (CPM) has been deployed in this market. The CPM transmitted waveform has a constant envelope and so a non-linear RF power amplifier can be used. This significantly reduces cost and improves power efficiency.

Two important disadvantages of CPM are receiver complexity and inferior spectral efficiency compared to QAM. This thesis demonstrates a 50% spectral efficiency improvement over an existing CPM configuration without loss of detection efficiency. This is achieved by moving to coherent demodulation and extending the duration of the CPM phase pulse to 3 symbol periods.

This new CPM configuration of h=1/4, M=4, L=3, is evaluated against ETSI requirements for a 28 MHz channel carrying 24 E1 circuits. Simulation of the receiver floating point model demonstrates all requirements are met. The detection efficiency requirement is exceeded by 4.7 dB. Carrier recovery, phase and timing synchronisation are assumed to be ideal.

The 50% increased symbol rate, coherent reception and a longer smoother phase pulse, conspire to increase receiver complexity substantially. The Viterbi algorithm is used to perform maximum-likelihood detection resulting in a 128 state trellis. This application has a stringent cost requirement that limits the implementation target to a Field Programmable Gate Array (FPGA) costing less than US$30. To demonstrate this demanding cost target is met, the two most computationally expensive receiver functions, the branch metric unit and path metric processing unit, are implemented in VHDL and targeted to a Xilinx Spartan 3A-DSP 1800 FPGA. The implementation uses 67% of the available logic resources, thus meeting the cost requirement.

The branch metric unit is implemented using a distributed arithmetic technique that performs the equivalent of 27.6 giga-multiplies/s, consuming only 23% of the available FPGA logic cells. This is very efficient compared to a conventional approach using all the FPGA's embedded multipliers which combined can only achieve 21 giga-multiplies/s.

The Viterbi path metric processing unit is implemented using a more conventional state-parallel architecture. To reduce state metric routing complexity, states are grouped into radix-4 units comprising dual add-compare-select (ACS) units. By utilising a spare cycle in the deep ACS pipeline, each ACS unit processes two output state metrics, thus halving the number of ACS units required. This implementation uses 44% of the available FPGA resources and meets timing at 204.5 MHz, exceeding the throughput requirement of 54 Mbit/s.

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

Acknowledgements

I would like to acknowledge and thank those who have helped me during my research and made it all possible.

This thesis was completed under a Technology Industry Fellowship (TIF) Education scholarship in conjunction with Harris Stratex Networks (NZ) LTD and Massey University. This support from FRST is much appreciated.

A big thanks to Dr Philip Secker, my project mentor from Harris Stratex Networks, for his role in kick starting the project and securing funding from FRST and Harris Stratex Networks. Also for taking the time out from a busy schedule to offer guidance, read my long emails and answer my many questions throughout the year, and provide feedback on this thesis.

I would like to thank Harris Stratex Networks for their commitment to the project in the form of Philip's time and a stipend contribution.

Many thanks to Dr Xiang Gui, my academic supervisor in Palmerston North, for his time spent listening and offering guidance during our weekly progress meetings, for ensuring I had access to the university's resources I needed, and for taking the time to review my thesis.

I would also like to thank Dr Edmund Lai, my co-supervisor based in Wellington, for his wise words, excellent thesis writing resources on his website, and for taking the time to provide feedback on my thesis.

Colin Plaw and Patrick Rynhart helped me with VPN access to the university labs and ensured I had access to a source code repository. Managing all the material created throughout the year would have been much much harder without it, thank you.

To my parents, Mum, Dad, Ma and Baba, thank you for all your encouragement and support, and for all the dinners that have been cooked for me over the last 6 months while I have been writing up this thesis. Also, thank you Mum for proof reading the whole thesis.

Finally, a very special thank you to my lovely wife Mandira, for convincing me to pursue my FPGA and digital signal processing passion in the form of a Masters in Engineering. Thank you for your constant encouragement and understanding during the thesis writeup phase.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xii**

List of Tables **xiii**

Glossary **xvi**

1 Introduction **1**
1.1 Introduction 1
1.2 Scope 2
1.3 Summary of Thesis Contributions 4
1.4 Overview 4

2 Background and Related Work **7**
2.1 Introduction 7
2.2 Notation 7
2.3 Continuous Phase Modulation Signal Model 7
2.4 Maximum Likelihood Receiver 10
2.4.1 Rational h 11
2.4.2 Viterbi Trellis Decode 12
2.5 Viterbi Decoder Architecture 12
2.6 Literature Review 13
2.6.1 CPM Receiver Implementations 13
2.6.1.1 Viterbi Path Metric Processing 13
2.6.1.2 Viterbi Path Metric Normalisation 14
2.6.2 CPM Configurations and their Energy/Bandwidth Consumption 15
2.6.3 Complexity Reduction 15
2.6.4 Literature Review Summary 16

3 CPM Parameter Selection **17**
3.1 Introduction 17
3.2 CPM Candidates 18
3.2.1 Choice of Symbol Alphabet Size (M) and Phase Pulse Shape 19
3.2.2 Modulation Index (h) and Phase Pulse Duration (L) Candidates 21

3.3 ETSI Requirements . . . 23
3.3.1 Specific Application . . . 23
3.3.2 Bandwidth: Transmit Power Spectral Density (PSD) . . . 24
3.3.3 Detection Efficiency: Bit Error Rate as a function of Receive Signal Level . . . 24
3.3.4 Interference Rejection . . . 24
3.4 Simulation Results . . . 25
3.4.1 Simulation System Model . . . 25
3.4.2 Bandwidth: Transmit Power Spectral Density (PSD) . . . 25
3.4.3 Detection Efficiency and Interference Rejection Performance: h=1/4, L=3 . . . 27
3.4.3.1 Detection Efficiency: Bit Error Rate as a function of SNR . . . 27
3.4.3.2 1st Adjacent Channel Interference . . . 29
3.4.3.3 Co-channel Interference . . . 30
3.4.4 Detection Efficiency and Interference Rejection Performance: h=1/5, L=2 . . . 30
3.5 Conclusion . . . 32

4 Fixed Point Modelling **35**
4.1 Introduction . . . 35
4.2 Implementation Target . . . 35
4.3 Sampling Rate . . . 37
4.4 Fixed Point Modelling . . . 38
4.4.1 Floating Point vs Fixed Point Numeric Representation . 39
4.4.2 Quadrature and In-phase Received Signal Word-length . 39
4.4.3 Branch Metric Filter Bank Coefficient Word-length 39
4.4.4 Branch Metric Word-length . . . 41
4.4.5 State Metric Normalisation . . . 42
4.5 Survivor Path History . . . 43
4.6 Conclusion . . . 45

5 Branch Metric Implementation **47**
5.1 Introduction . . . 47
5.2 Computation Complexity . . . 48
5.3 CPM and Distributed Arithmetic . . . 50
5.3.1 Phase State Symmetry . . . 51
5.4 FPGA Implementation . . . 52
5.4.1 Throughput Requirements . . . 52
5.4.2 Efficient Mapping of a DA Filter into FPGA Hardware . 53
5.4.2.1 DALUT . . . 53
5.4.2.2 Scaling Accumulator . . . 54
5.4.2.3 FPGA Resource Use Summary . . . 54
5.4.2.4 Additional Resource Use Required to Meet Timing . . . 54
5.4.3 Implementation Results . . . 55
5.4.4 Functional Verification . . . 56
5.5 DA vs Embedded Multipliers . . . 56
5.6 Conclusion . . . 57

6 Path Metric Implementation 59
6.1 Introduction . . . 59
6.2 Background . . . 60
6.3 Proposed Solution . . . 61
6.3.1 State-Parallel Radix-4 Decomposition . . . 61
6.3.2 Add-Compare-Select Unit . . . 64
6.3.2.1 Resource Use Estimate . . . 65
6.4 Implementation Results . . . 66
6.4.1 Functional Verification . . . 67
6.5 Conclusion . . . 67

7 Conclusions and Future Work 69

Appendix 72

A VHDL Implementation Functional Verification 75
A.1 VHDL Functional Verification Architecture . . . 75

B Receive Signal Level to SNR Conversion 77

C Baseband I/Q Modulator Derivation 79

D VHDL Source Code 81
D.1 Branch Metric Filter Bank . . . 81
D.1.1 Synthesis Top Level . . . 81
D.1.2 Top Level . . . 81
D.1.3 4-Tap Distributed Arithmetic Filter . . . 87
D.1.4 Filter Coefficients . . . 92
D.1.5 Testbench . . . 92
D.1.6 Test Vectors Package . . . 96
D.2 Viterbi Trellis Path Metrics . . . 96
D.2.1 Synthesis Top Level . . . 96
D.2.2 Top Level . . . 96
D.2.3 Radix-4 Add-Compare-Select Unit . . . 105
D.2.4 Viterbi Trellis Package . . . 121
D.2.5 Testbench . . . 121
D.2.6 Test Vectors Package . . . 122
D.3 Branch Metrics Filter Bank and Viterbi Trellis Path Metrics . . . 122
D.3.1 Synthesis Top Level . . . 122
D.3.2 Top Level . . . 122
D.3.3 Testbench . . . 122
D.4 Placed Primitives Modules . . . 122
D.4.1 Adder with Subtract and Clear Controls . . . 122
D.4.2 Adder with Subtract and 2 Input Operand Mux . . . 122
D.4.3 16 Deep ROM using Distributed Ram . . . 122
D.4.4 2 Input Mux . . . 122
D.4.5 Shift Register . . . 122
D.4.6 Register . . . 122
D.4.7 Relative Location Constraint(RLOC) Helper Package . . . 122
D.5 Sundry Packages . . . 123
D.5.1 Key Project Constants . . . 123

E Matlab Source Code **125**
E.1 Analytical CPM Code Performance 125
E.1.1 CPM Code Minimum Euclidean Distance Upper Bound 125
E.1.2 CPM Code Baseband Double Sided Bandwidth 125
E.2 Floating and Fixed Point M-file Models 125
E.3 VHDL Trellis Representation and Test Vector Generation 125
E.3.1 Export Matlab Data to VHDL Writer 126
E.3.2 VHDL Writer 126
E.4 Miscellaneous 126
E.4.1 Hekstras Method Bound Calculation 126

F Implementation Results **127**
F.1 Branch Metric Filter Bank 127
F.2 Viterbi Trellis Path Metrics 129

G Software Tool Versions **133**
G.1 High Level Modelling: Matlab and Simulink 133
G.2 FPGA Implementation: Xilinx ISE 10.1 133
G.3 VHDL Simulation: Modelsim 133

Bibliography **135**

List of Figures

1.1 CPM Receiver Functionality Studied in this Thesis 3
1.2 Modelling and Implementation at Complex Baseband 4

2.1 Raised Cosine (RC) and Rectangular (REC) Frequency Pulse and Phase Pulse 9
2.2 Standard Viterbi Decoder Implementation Architecture 12

3.1 Effect of Symbol Alphabet Size (M) on Detection Efficiency, L=1, Raised Cosine Phase Pulse 19
3.2 Effect of Phase Pulse Duration (L) on Detection Efficiency, M=4 20
3.3 Relative Detection Efficiency and Spectral Efficiency for several CPM Configurations 22
3.4 Simulation System Model 26
3.5 Simulated Transmit Power Spectral Density of Candidate CPM Configurations, Various (h, L), M=4, 27 Msymbols/s) 26
3.6 Zoomed in version of Figure 3.5 27
3.7 Simulated Bit Error Probability with and without Reed Solomon FEC, AWGN Channel, No ACI Reject Filter, h=1/4, L=3RC, 27 Msymbols/s 28
3.8 Simulated Bit Error Rate with Adjacent Channel Interference, h=1/4, L=3, M=4, 27 Msymbols/s 29
3.9 Simulated Bit Error Rate Demonstrating Effect of ACI Reject Filter and Adjacent Channel Interference, h=1/4, L=3, M=4, 27 Msymbols/s 30
3.10 Simulated Bit Error Rate with Co-Channel Interference, h=1/4, L=3, M=4, 27 Msymbols/s 31
3.11 Simulated Bit Error Rate with and without Adjacent Channel Interference, h=1/5, L=2, M=4, 27 Msymbols/s 32

4.1 Approximations to the maximum-likelihood receiver 36
4.2 Effect of Sampling Rate on Detection Efficiency 37
4.3 Effect of Quantised Received Signal Word-length on Receiver Detection Efficiency 40
4.4 Effect of Quantised Branch Metric Filter Bank Coefficient Word-length on Receiver Detection Efficiency 41
4.5 Effect of Branch Metric Word-length on Receiver Detection Efficiency 42
4.6 Effect of Survivor Path History Depth on Detection Efficiency . 44

5.1 4-Tap Distributed Arithmetic FIR Filter Block Diagram 52

6.1 CPM Viterbi Detection 59
6.2 Add-Compare-Select Processing Required per State 61
6.3 128 State CPM Viterbi Detector Comprising 32 Radix-4 Add-Compare-Select Units 62
6.4 CPM Radix-4 Trellis (h=1/4, L=3, M=4) 63
6.5 Add-Compare-Select (ACS) Unit Detail and Pipeline 64
6.6 Radix-4 Unit Comprising Dual ACS Units 65

A.1 VHDL Implementation Test Architecture 76

List of Tables

2.1 Notation 7

3.1 Candidate Modulation Indices (h) 21
3.2 Candidate CPM Configurations 23
3.3 ETSI Co-Channel and 1st Adjacent Channel Interference Performance [1, Table D.7] 24

4.1 Comparison of DSP and FPGA Multiplication Capability 36
4.2 Detection Efficiency Degradation due to Branch Metric Quantisation 42

5.1 CPM Branch Metric Filter Bank Complexity 49
5.2 Distributed Arithmetic Filter Fmax Requirement 53
5.3 Estimated Branch Metric Filter Bank FPGA Resource Use 54
5.4 Branch Metric Filter Bank Implementation Results 55

6.1 Single Add-Compare-Select Unit FPGA Resource Use Estimate 66
6.2 Path Metric Processing Unit Implementation Results 67

B.1 ETSI Received Signal Level Converted to SNR and $\frac{E_b}{N_o}$ 77

G.1 Matlab and Simulink Software Versions 133
G.2 Xilinx Synthesis and Implementation Software Versions 134

Glossary

ACI	Adjacent Channel Interference
ACS	Add-Compare-Select
ACSU	Add-Compare-Select Unit
ADC	Analog to Digital Converter
AGC	Automatic Gain Control
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BMU	Branch Metric Unit
BRAM	Block Random Access Memory
CCI	Co-Channel Interference
CFO	Carrier Frequency Offset
CPFSK	Continuous Phase Frequency Shift Keying
CPM	Continuous Phase Modulation
DA	Distributed Arithmetic
DALUT	Distributed Arithmetic Look-Up Table
DSP	Digital Signal Processor
DUT	Device Under Test
ETSI	European Telecommunications Standards Institute
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
GMSK	Gaussian Minimum Shift Keying
IF	Intermediate Frequency
LC	Logic Cell
LUT	Look-Up Table
ML	Maximum-Likelihood
MLSD	Maximum-Likelihood Sequence Detector

MSK	Minimum Shift Keying
PAM	Pulse Amplitude Modulation
PSD	Power Spectral Density
QAM	Quadrature Amplitude Modulation
RC	Raised Cosine
REC	Rectangular
RF	Radio Frequency
RLOC	Relative Location
RSL	Received Signal Level
SER	Symbol Error Rate
SMU	Survivor Management Unit
SNR	Signal to Noise Ratio
SRAM	Static Random Access Memory
SRC	Spectrally Raised Cosine
STA	Static Timing Analysis
TFM	Tamed Frequency Modulation
UHF	Ultra-High Frequency
VHDL	VHSIC hardware description language
VHSIC	Very-High-Speed Integrated Circuit
VME	VERSA-module Europe

Chapter 1

Introduction

1.1 Introduction

In a data market, bit-rate is everything. Vendors in the digital microwave radio cellular backhaul market are under pressure to reduce costs and cope with increasing data rate requirements due to the rapidly increasing availability and consumer uptake of high speed mobile data services. Traditionally, microwave radio backhaul systems have used quadrature amplitude modulation (QAM). Radio spectrum is a limited resource and so the trend has been toward larger QAM constellation sizes to increase spectral efficiency. However, since QAM modulates carrier phase and amplitude, the transmitter requires a linear power amplifier. This component is a significant portion of the cost and power consumption of the microwave radio outdoor unit.

A recently released microwave radio product has significantly reduced cost and improved power efficiency by using continuous phase Modulation (CPM). By modulating carrier phase only and ensuring a smooth phase transition between symbols, the transmitted CPM waveform has a constant envelope. This allows the use of a low-cost non-linear power amplifier in the transmitter.

Nevertheless, two important disadvantages of CPM are receiver complexity and inferior spectral efficiency compared with QAM. Spectrally efficient CPM configurations require at least a quaternary symbol alphabet and a smoothed symbol phase pulse lasting several symbol periods. This leads to a maximum-likelihood receiver with a large matched filter bank and a Viterbi decoder with a large number of states; the implementation cost can be prohibitive.

This thesis proposes a CPM configuration that achieves a 50% improvement in spectral efficiency over an existing microwave radio CPM product, while maintaining receiver detection efficiency. This is achieved by moving to coherent demodulation and lengthening the phase pulse duration to 3 symbol periods. The ETSI channel bandwidth of 28 MHz is constant, so this new scheme increases data throughput by 50%.

The increased symbol rate (27 MSymbols/s), coherent reception, and a longer smoother phase pulse, conspire to increase receiver complexity substantially. The maximum-likelihood receiver contains a matched filter bank of 128 filters, followed by Viterbi path metric processing of 128 Viterbi states. The matched filtering operation alone is shown to consume 27.6 giga-multiplies/s. A cost

effective implementation is a challenge.

The application has a stringent cost requirement that limits the implementation target to a Field Programmable Gate Array (FPGA) costing less than US$30 at a volume price. To demonstrate the proposed CPM configuration is able to meet the cost target, the two most computationally expensive receiver functions are implemented in VHDL and targeted to a Xilinx Spartan 3A-DSP 1800 FPGA. The designs are synthesised to confirm FPGA resource use and cost. Static timing analysis on the placed and routed netlist confirms data throughput. A VHDL functional simulation verifies operation of the implemented design.

The literature tackles the complexity problem by using a variety of algorithm level complexity reduction techniques that have been shown to give significant reductions in complexity with a range of performance degradations relative to the maximum-likelihood receiver. Many of these techniques have not been tested in the presence of adjacent channel interference (ACI), and some complexity reduction techniques have shown increased sensitivity to ACI. This thesis avoids this issue and focuses on a maximum-likelihood FPGA implementation. This allows a dollar cost to be put on any quaternary CPM configuration with a symbol rate in the region of 10-30 MSymbols/s. This fills a gap in the literature where there are very few published details of CPM receiver FPGA implementations.

A low-cost CPM filter bank FPGA implementation is proposed. It uses a distributed arithmetic technique to implement 27.6 giga-multiplies per second of filter bank multiplications consuming 23% of the available FPGA logic cells. A conventional approach would use the FPGA's embedded multipliers but these resources provide a maximum of only 21 giga-multiplies/s. This design meets timing at 215.6 MHz, exceeding the minimum throughput requirements by 14%. The main drawback of this technique is that it adds one symbol period of processing latency. Since the branch metric filter bank is likely to be inside the phase recovery loop, this added latency degrades phase recovery performance. This is the price to be paid for a low-cost implementation.

The Viterbi path metrics processing implementation follows a more standard architecture. The 128 state trellis is decomposed into 32 radix-4 units. Each radix-4 unit comprises 2 add-compare-select (ACS) units that calculate 4 path metrics every symbol period. This implementation uses 44% of the FPGA's available logic cell resources. This design meets timing at 204.5 MHz achieving a throughput of 58.4 Mbit/s which is 8% more than the minimum requirement of 54 Mbit/s for the target application.

Symbol and phase synchronisation for CPM is an ongoing area of research and is beyond the scope of this thesis; we assume ideal symbol and phase synchronisation.

1.2 Scope

The purpose of this work is two-fold. Firstly, a new CPM configuration must be found that achieves a 50% spectral efficiency improvement over the existing product. Secondly, a practical, low-cost implementation must be demonstrated with an FPGA targeted VHDL implementation.

In the search for an efficient CPM configuration we assume single-h CPM

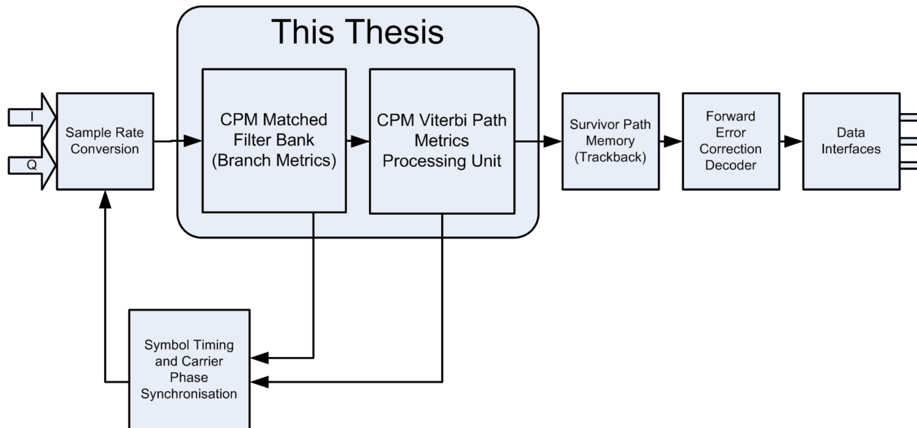

Figure 1.1: CPM Receiver Functionality Studied in this Thesis

and coherent reception. We assume perfect timing and carrier phase recovery and assume a zero carrier frequency offset. Although this is an active area of research it is beyond the scope of this thesis.

The CPM configuration's bandwidth consumption, SNR performance, Adjacent Channel and Co-Channel Interference (ACI, CCI) performance must meet an application specific set of ETSI standard requirements [1]. We assume 239/255 Reed Solomon forward error correction. The data rate requirement is 54 Mbit/s or 27 MSymbols/s for a quaternary CPM symbol alphabet. This provides for the transport of 24 E1 circuits and an additional approximately 5 Mbit/s for framing and auxiliary channel overhead.

The VHDL implementation in this thesis focuses purely on the two most computationally expensive, and thus costly, functions of the CPM receiver. See Figure 1.1. This is the branch metrics filter bank and the Viterbi path metric add-compare-select functionality. Not included within the scope of this thesis are E1 data circuit interfaces, forward error correction, framing, sample rate conversion and front end band-limiting.

Survivor path management is not implemented. A traceback architecture is cost effective because it uses FPGA block memory to store the Viterbi path history. Section 4.5 shows that the memory required to implement this function is small; one or two block rams for this application. Meeting throughput requirements is straight forward since this function is outside the Viterbi iteration loop and so the logic can be pipelined extensively. However, traceback memory bandwidth requirements are high when tracing back once every symbol. The bandwidth requirements can be reduced significantly by tracing back only once every several symbols using the technique described in [2]. The cost is added latency but this is negligible in the context of the latency through a complete receiver. Phase and timing recovery require early, tentative symbol decisions and these would not be generated by tracing back [3] [4].

The find-the-best metric operation has also not been implemented. If tracing back once every few symbols, bit-serial techniques are appropriate and the FPGA resources consumed can be kept to a minimum. Phase and timing recovery determine the constraints on this latency which is beyond the scope of

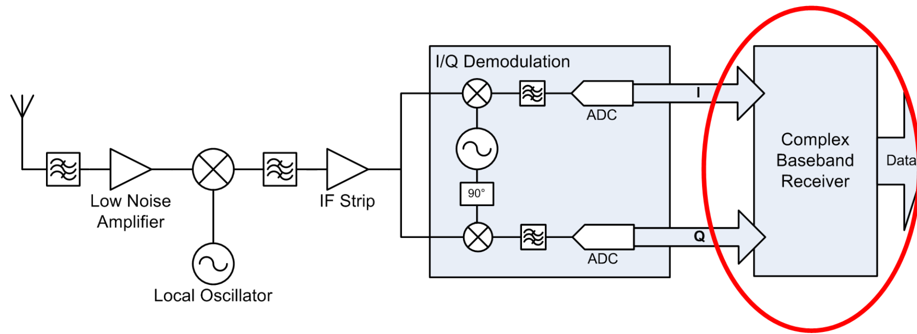

Figure 1.2: Modelling and Implementation at Complex Baseband

this thesis. For reasonable latencies, the cost of this operation is small relative to the rest of the receiver.

The VHDL implementation is proved with synthesis results, static timing analysis and a VHDL functional simulation. This approach is justified in appendix A.

All modelling and implementation is done at complex baseband as shown in Figure 1.2.

1.3 Summary of Thesis Contributions

The main contribution of this thesis is a resource efficient and low-cost FPGA implementation of the two most computationally expensive components of a CPM receiver of moderate throughput (54 Mbit/s). The CPM configuration achieves a 50% increase in spectral efficiency over an existing product.

Other contributions include:

- Demonstration of a 50% spectral efficiency improved CPM configuration meeting a specific ETSI microwave radio standard which specifies requirements for tranmsitted power spectral density and receiver performance in the presence of channel noise and adjacent and co-channel interference.
- CPM Viterbi demodulator fixed point simulation results.
- The application of Hekstra's path metric normalisation method to a CPM detector.
- The application of a bit-serial distributed arithmetic filter algorithm to the FPGA implementation of a CPM branch metric unit.
- Application of a Viterbi radix-4 decomposition to a CPM trellis.

1.4 Overview

- Chapter 2 presents background material relevant to the work in this thesis. The CPM signal model is developed and the key equations for the

maximum likelihood receiver are stated. This chapter also contains a review of the CPM literature relevant to this thesis.

- Chapter 3 investigates the choice of CPM parameters to improve spectral efficiency by 50%. Floating point models developed in Matlab and Simulink simulate the CPM transmitter and receiver at baseband. We choose the CPM configuration with lowest complexity that still meets the ETSI standard transmit power spectrum mask and receiver detection efficiency specification.

- Chapter 4 begins the transition to a fixed point hardware implementation. Matlab simulation results demonstrate the effect on receiver detection efficiency of sampling rate, word-length and Viterbi path history depth. There is a direct tradeoff between hardware complexity and detection efficiency. This chapter concludes by selecting the most appropriate sample rate, word-length and path history depth to be used in the FPGA implementation to follow.

- Chapter 5 presents the branch metrics filter bank VHDL implementation. A distributed arithmetic algorithm implements each filter in the filter bank. Synthesis results show that the FPGA resource use meets the cost requirement, and static timing analysis results confirm throughput. Functional simulation results demonstrate that the VHDL implementation matches the Matlab fixed point model precisely.

- Chapter 6 describes the Viterbi path metric VHDL implementation. This trellis decode add-compare-select (ACS) processing is implemented using a state-parallel structure using radix-4 units comprising dual ACS units. Results from synthesis, static timing analysis and functional simulation demonstrate that this design meets requirements.

- Chapter 7 concludes the work presented in this thesis and suggests possibilities of investigation for the future.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter begins by introducing the CPM signal model used throughout this thesis. A maximum-likelihood receiver is presented that comprises a filter bank followed by a Viterbi trellis search. Practical implementations use the Viterbi algorithm with an implementation comprising a branch metric unit, Viterbi path metric processing unit and a survivor management unit.

The second half of this chapter presents a review of the CPM literature relevant to this thesis.

2.2 Notation

This thesis uses the notation in Table 2.1.

$\Re(x)$	real component of x.
$\Im(x)$	imaginary component of x.
$\tilde{x}(t)$	baseband complex envelope of passband signal $x(t)$ where $x(t) = \Re\{\tilde{x}(t)e^{j\omega_c t}\}$ and ω_c is the carrier angular frequency.

Table 2.1: Notation

2.3 Continuous Phase Modulation Signal Model

A radio frequency (RF) carrier modulated by a baseband message carrying signal can be described by Equation (2.1). The amplitude $a(t)$ and phase $\phi(t)$ of the carrier f_c are available to be modulated by the message signal [5].

$$s(t) = a(t)cos(2\pi f_c t + \phi(t)) \tag{2.1}$$

An equivalent exponential notation is given in Equation (2.2) where $\tilde{s}(t)$ is the baseband complex envelope and ω_c is the RF carrier angular frequency.

Equation (2.3) represents $\tilde{s}(t)$ in terms of its in-phase and quadrature components and equivalently, Equation (2.4) shows the amplitude $a(t)$ and phase $\phi(t)$ components explicitly. For the purposes of this thesis, all the interesting properties of the modulation are described by its complex envelope $\tilde{s}(t)$ and so the passband formulation is not considered any further.

$$s(t) = \Re\{\tilde{s}(t)e^{j\omega_c t}\} \tag{2.2}$$

$$\tilde{s}(t) = s_I(t) + js_Q(t) \tag{2.3}$$

$$\tilde{s}(t) = a(t)e^{j\phi(t)} \tag{2.4}$$

Keeping $a(t)$ constant and using the message signal to modulate phase $\phi(t)$ only, the transmitted RF signal has a constant envelope and consequently is robust to non-linearities in the signal path. By smoothly transitioning the phase from symbol to symbol the spectral occupancy is reduced. This is continuous phase modulation [6].

The standard definition for how the phase is modulated by the message signal is defined in Equation (2.5). The message signal is represented as digital data coded into M-ary symbols $\mathbf{a}$, coming from a symbol set of size M, as defined by equation (2.6). In this thesis M is considered a power of 2 only.

$$\phi(t, \mathbf{a}) = 2\pi h \sum_{i=-\infty}^{\infty} \alpha_i q(t - iT) \tag{2.5}$$

$$\alpha_i \in \{\pm 1, \pm 3, ..., \pm(M-1)\} \tag{2.6}$$

The modulation index, h, is a parameter that trades off bandwidth and energy performance. Although it is possible to vary h from symbol to symbol, we only consider CPM schemes with a single, fixed h [7] [1].

$q(t)$ is the all important phase smoothing function or phase pulse. Two examples are shown in Figure 2.1. For example, the raised cosine frequency pulse is described by Equation (2.7). The phase pulse is defined by Equation (2.8). This function starts at 0 at the beginning of a symbol duration so that the phase is continuous from one symbol period to the next. By convention this function is 1/2 at the end of the symbol duration. An infinite number of phase smoothing functions are possible but there are several standard pulses defined in the literature.

$$g(t) = \frac{1}{2LT}(1 - \cos(\frac{2\pi t}{LT})), 0 \leq t \leq LT \tag{2.7}$$

$$q(t) = \int_{-\infty}^{t} g(\tau)d\tau \tag{2.8}$$

The frequency pulse $g(t)$ has finite duration of length LT where T is a nominal symbol period. The symbol period relates to the data bit rate and alphabet

[1] If h is allowed to vary from one symbol to the next, it is called multi-h CPM. Further gains in spectral efficiency and energy consumption are possible, at the expense of further complication in the receiver[8].

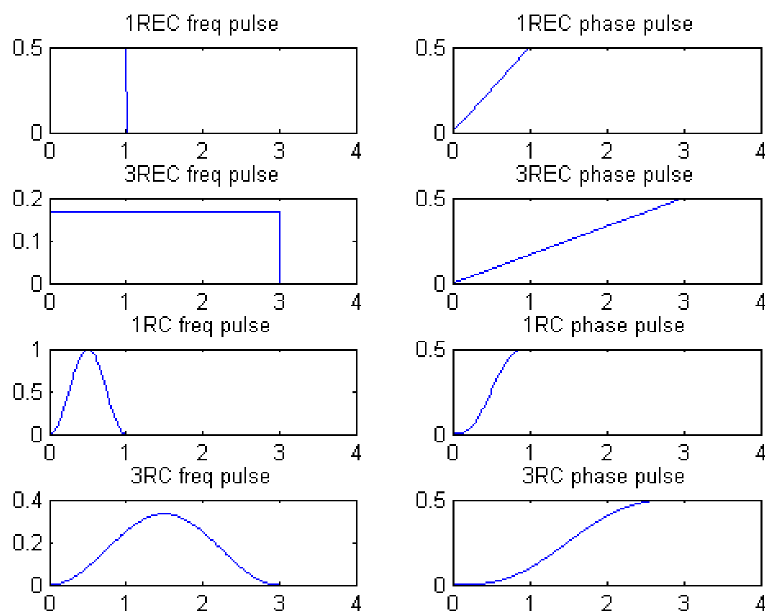

Figure 2.1: Raised Cosine (RC) and Rectangular (REC) Frequency Pulse and Phase Pulse

size as in Equation (2.9). L controls the degree of overlap between consecutive symbols in the modulator. CPM schemes with $L = 1$ are called full response and those with $L > 1$ are called partial response. CPM schemes with $L > 1$ spread the phase pulse in time and reduce the bandwidth of the transmitted signal [9].

> The use of partial response CPM systems yields a more attractive tradeoff between error probability and spectrum than does the full response systems [9].

$$T = \frac{\log_2 M}{bitrate} \tag{2.9}$$

A CPM configuration is uniquely defined by the three parameters h, M and $q(t)$. These parameters must be chosen to meet the requirements of the application at hand. Chapter 3 investigates the choice of these parameters with regard to meeting spectral occupancy, SNR performance and cost requirements for a specific point to point microwave radio application.

2.4 Maximum Likelihood Receiver

In this thesis we are interested in an implementation of the maximum-likelihood receiver. This is presented below.

The received signal $\tilde{r}(t)$ is a distorted version of the transmitted signal $\tilde{s}$ in the presence of additive white gaussian noise (AWGN) $\tilde{n}(t)$, as shown by equation (2.10). We assume perfect channel equalisation, and we set the phase and time offset terms to zero, implying perfect timing recovery and perfect carrier phase synchronisation with no carrier frequency offset. Appendix C shows how the transmitted baseband signal, $\tilde{s}$, is generated in practice.

$$\tilde{r}(t) = \tilde{s}(t, \alpha) + \tilde{n}(t) \tag{2.10}$$

In [6], a maximum likelihood sequence estimating (MLSE) receiver is derived and it requires finding the symbol sequence α that maximises the correlation in Equation (2.11). This is a correlation of the received signal with all possible transmitted signals.

$$J(\alpha) = \Re\{\int_{-\infty}^{\infty} \tilde{r}(t)\tilde{s}^*(t, \alpha)dt\} \tag{2.11}$$

The number of symbol sequences grows exponentially with the sequence length making a receiver using a direct implementation of this equation impractical. By taking an iterative approach as in Equation (2.12), and performing the correlation over one symbol period at a time as in Equation (2.13), the receiver becomes practical. The index n identifies a single symbol period.

$$J_n(\alpha) = J_{n-1}(\alpha) + Z_n(\alpha) \tag{2.12}$$

$$Z_n(\alpha) = \Re\{\int_{nT}^{(n+1)T} \tilde{r}(t)\tilde{s}^*(t, \alpha)dt\} \tag{2.13}$$

2.4.1 Rational h

A further requirement is for h to be rational so that $\tilde{s}^*(t, \alpha)$ lies within a finite set of waveforms over a single symbol period, making Equation (2.13) realisable. By restricting h to be rational according to Equation (2.14) then the phase takes on a finite set of phases modulo 2π at symbol time boundaries and a trellis with a finite number of states can be used to represent the phase transitions.

$$h = \frac{2k}{p}, k, p \in integers \tag{2.14}$$

For CPM we have

$$\tilde{s}(t) = e^{j\phi(t)} \tag{2.15}$$

where

$$\phi(t, \mathbf{a}) = 2\pi h \sum_{i=-\infty}^{\infty} \alpha_i q(t - iT) \tag{2.16}$$

And when h is rational the phase signal can be divided into two terms as shown in Equation (2.17).

$$\phi(t, \mathbf{a}) = 2\pi h \sum_{i=n-L+1}^{n} \alpha_i q(t - iT) + 2\pi h \sum_{i=-\infty}^{n-L} \alpha_i q(LT) \tag{2.17}$$

$\theta(t, \mathbf{a})$ in Equation (2.18) describes how the phase changes during the nth symbol interval due to the current α_n symbol and the previous $L - 1$ symbols.

$$\theta(t, \mathbf{a}) = 2\pi h \sum_{i=n-L+1}^{n} \alpha_i q(t - iT) \tag{2.18}$$

θ_n in Equation (2.19) is the accumulated phase change due to all symbols prior to and including the a_{n-L} symbol. This is called the phase state. [2]

$$\theta_n = \pi h \sum_{i=-\infty}^{n-L} \alpha_i \pmod{2\pi} \tag{2.19}$$

$$\phi(t, \mathbf{a}) = \theta(t, \mathbf{a}) + \theta_n \tag{2.20}$$

Placing Equation (2.20) into Equation (2.13) gives Equation (2.21).

$$Z_n(\alpha_n, \theta_n) = \Re\{\int_{nT}^{(n+1)T} \tilde{r}(t) e^{-j[\theta(t,\alpha_n)+\theta_n]} dt\} \tag{2.21}$$

$$Z_n(\alpha_n, \theta_n) = \Re\{e^{-j\theta_n} \int_{nT}^{(n+1)T} \tilde{r}(t) e^{-j\theta(t,\alpha_n)} dt\} \tag{2.22}$$

Since α_n can take on M^L different sequences and θ_n takes on p different values, Equation (2.22) represents M^L complex matched filters followed by p phase rotations [9].

[2] In general, θ_n does not represent the actual signal phase at the start of a symbol period because $\theta(t, \mathbf{a})$ is non-zero at the beginning of a symbol period. The phase state is the cumulative contribution of past symbols to the signal phase up to L-1 symbols prior to the current time.

2.4.2 Viterbi Trellis Decode

The Viterbi algorithm is an efficient way to evaluate Equation (2.12). $J_n(\alpha)$ represents the accumulated path metrics at time nT and $Z_n(\alpha)$ is the set of branch metrics for the interval from $t = nT$ to $t = (n+1)T$. The trellis state is defined by Equation (2.23) where the phase state θ_n is defined by Equation (2.24) and the correlative state is defined by Equation (2.25). This gives a Viterbi trellis with pM^{L-1} states [9].

$$\sigma_n = (\theta_n, \alpha_{n-1}, \alpha_{n-2}, \ldots, \alpha_{n-L+1}) \tag{2.23}$$

$$\theta_n = \frac{2\pi i}{p}, i \in \{0, 1, 2, \ldots, p-1\} \tag{2.24}$$

$$CorrelativeState = (\alpha_{n-1}, \alpha_{n-2}, \ldots, \alpha_{n-L+1}) \tag{2.25}$$

2.5 Viterbi Decoder Architecture

In the literature, Viterbi decoder implementations are typically treated by partitioning the functionality into three parts as shown in Figure 2.2.

- Branch Metric Unit (BMU) - Computes hamming or Euclidean distance between the received symbol and the various possible transmitted symbols. For a CPM receiver this implements Equation (2.21).

- Path Metric Processing Unit - Accumulates the path metric and selects a survivor path from each of the trellis connections inbound to each state. The path metrics are accumulated as defined by Equation (2.12).

- SMU or TBU - Survivor management unit or Traceback unit. These units extract the decoded data from the ultimate survivor path.

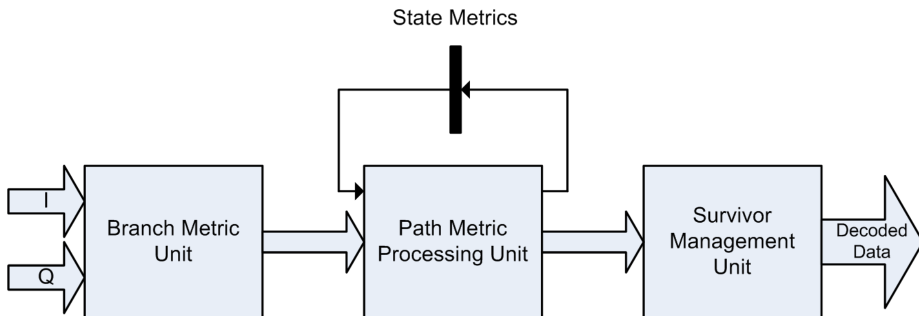

Figure 2.2: Standard Viterbi Decoder Implementation Architecture

2.6 Literature Review

2.6.1 CPM Receiver Implementations

Nova Engineering use a multi-h CPM waveform to increase spectral efficiency by 3 times compared to legacy PCM/FM telemetry waveforms at the same detection efficiency. They use a modulation index of h=1/4,5/16, a raised cosine phase pulse of 3 symbol periods (L=3RC) and a quaternary alphabet (M=4). Viterbi trellis complexity is 512 states [10].

Nova Engineering have also designed a product called Hypermod MMD22 which uses multi-h CPM with M=4 and 128 state Viterbi trellis complexity. The complete transceiver is implemented on a board with 5 Xilinx Virtex E XCV2000E [3] FPGAs. One FPGA is allocated to the Viterbi trellis update calculations which consume 80% of the logic resources and 40% of the block ram. The data rate is 22 Mbit/s [11].

In [12], turbo-detected coded CPM is used in a military UHF satellite communications application. A data rate of 80 kbit/s is transmitted in a 25 kHz channel. At $\frac{E_b}{N_o}$ of 11 dB, the bit error rate is 10^{-5}. They use M=8, h=1/8 and a rectangular phase pulse of 1 symbol duration (L=1REC). The modem implementation consists of 2 VERSA-module Europe (VME) cards in a VME chassis. Most of the signal processing functions are performed by a TMS320C6701 DSP but the iterative decoder is implemented in VHDL and uses 70% of an XCV2000E FPGAs resources.

Because these are commercial products, very few details of these FPGA implemenations have been published.

2.6.1.1 Viterbi Path Metric Processing

There is a large body of literature describing algorithms and implementation details for Viterbi trellis decoding, mostly targeting the decoding of convolutional codes and for ASIC implementation. Although few papers were found that directly address the implementation of a CPM Viterbi trellis, many of the general Viterbi results are applicable.

For low data rates or small Viterbi trellises, a fully state-serial approach uses the least hardware resources. A 64 state Viterbi decoder was implemented using a single add-compare-select (ACS) processing unit targeting a Xilinx Spartan 3 FPGA. The implementation used only 128 slices (approximately 256 logic cells) and 2 block rams to support a data rate of 2.4 Mbit/s [13].

In contrast, the CPM application in this thesis calls for a Viterbi decoder with a large number of states (128) and for a throughput of 54 Mbit/s which is moderately high for a low-cost FPGA implementation target. The traditional approach to high throughput, large trellis size Viterbi decoders is a fully state-parallel approach in which each state is processed with individual add-compare-select units. This consumes a lot of hardware resources and the ACS

[3] An XCV2000E has 38400 logic cells and supports a 130 MHz clock rate with 4 LUT levels. Although this FPGA is almost 10 years old, this part is roughly equivalent in density and performance to a Spartan 3-ADSP XC3S1800ADSP, the FPGA implementation target in this thesis. Virtex FPGA's are Xilinx's premium brand so offer higher performance than the spartan family, but at significantly higher cost.

path metric routing is complicated. A bit-serial approach to the ACS processing reduces the hardware requirements enormously, whilst also reducing the amount of ACS to ACS connectivity required since only a 1 bit wide path metric bus is required [2] [14]. However, this bit-serial approach does put an upper limit on throughput, dependent on the path metric wordlength.

By using multiple ACS units where each ACS unit processes multiple states, a hybrid of the fully state-serial and state-parallel approaches is possible. Shung proposes a systematic approach to allocating states to ACS units. By pipelining the ACS operation a single ACS unit processes multiple states at once. It is claimed this provides a favourable area-time tradeoff [15].

In general, a Viterbi trellis can be decomposed into radix-k sub-units, where k is a power of 2. For example, a radix-2 trellis has 2 inbound and 2 outbound paths per state and the radix-4 form of this trellis has 4 inbound and 4 outbound paths per state. Each Viterbi iteration of a radix-4 trellis is equivalent to 2 iterations of the radix-2 trellis. In this way a radix-4 trellis doubles the available time to perform the ACS operations. [16] is an ASIC implementation that nearly doubles throughput by decomposing a 32 state convolutional code radix-2 trellis into a radix-4 trellis. They achieve a decoding throughput of 140 Mbit/s in 1.2um CMOS. All ACS units within a radix-4 unit share input path metrics keeping all routing within a radix-4 unit local. This thesis shows that the natural radix-4 decomposition of the CPM trellis used in our microwave radio application, brings the same advantages to a CPM trellis Viterbi decoder.

Survivor path traceback has been regarded as another bottleneck to throughput. However, by increasing the survivor path memory size and tracing back less often than once per symbol, the traceback memory bandwidth requirements may be reduced significantly [2].

Sub-optimum detection based on the T-algorithm has been applied to the VLSI implementation of a coherent CPM detector [17]. Another non maximum-likelihood technique is the adaptive Viterbi algorithm which reduces the average amount of computation required by searching a subset of the full trellis based on channel conditions [18] [19] [20]. A systolic array approach to the branch metric and path metric processing is proposed in [21].

Both of the two main SRAM based FPGA vendors, Altera and Xilinx, have developed Viterbi Decoder IP cores. The Altera core can implement a 256 state trellis with a throughput of 16 Mbit/s using 3800 logic cells and 18 9-kbit block rams in a Cylone III family FPGA (EP3C10F256C6). They use 3 bit branch metrics [22]. Xilinx's serial IP core implements a 64 state trellis using 983 slices (equivalent to approximately 1966 logic cells) at a throughput of 15 Mbit/s in a Spartan 3A-DSP family FPGA (XC3SD3400A-4) [23].

2.6.1.2 Viterbi Path Metric Normalisation

The Viterbi path metrics grow without bound over time. Several techniques have been developed for scaling or normalising the metrics so they can be represented in fixed point arithmetic [24]. Since only the difference between path metrics is required for path selection in the Viterbi add-compare-select unit, and because this difference is bounded, Hekstra proposes the use of 2's complement arithmetic as an alternative to scaling or normalisation. This method eliminates the need for additional normalisation or rescaling hardware [25]. Path metric difference bounds are required to size the path metric wordlength

[26]. It has not been shown that these techniques can be applied to a CPM Viterbi trellis.

2.6.2 CPM Configurations and their Energy/Bandwidth Consumption

Aulin and Sundberg [27] [9] show a method for calculating a CPM code's minimum distance which predicts detection efficiency. They also plot various CPM codes on the energy bandwidth plane. Anderson, Aulin and Sundberg [6] [7] show results for a wider variety of CPM configurations on the energy bandwidth plane, mostly using the raised cosine (RC) phase pulse. However, they do not measure performance against ETSI microwave radio standards.

Svensson [28] considers the choice of CPM configuration with regard to meeting a spectral mask requirement, seemingly also from the same ETSI specification [1] as required in our application. They target 37.5 Mbit/s in a 14 MHz channel (2.68 bits/s/Hz) whereas the application in this thesis requires 54 Mbit/s in a 28 MHz channel (1.93 bits/s/Hz). They also provide results for adjacent channel interference with a carrier to interference ratio the same (-5dB) as required for the application considered in this thesis. Strangely, they locate the interferer at the 2nd adjacent channel whereas the ETSI specification calls for a 1st adjacent channel interferer.

Svensson [29] develops an empirical model for CPM and shows that for a constant effective bandwidth, M=8 is the optimum power of 2 symbol alphabet size in the range from 4 to 32, in order to maximise minimum distance squared (detection efficiency). However, compared to M=4, the advantage in terms of d^2min of M=8 is only 0.55 dB. They also describe a saturation L, beyond which brings little improvement to d^2min. For M=8 it is 3, and for M=4 it is 7. However, for M=4 the advantage of L=7 over L=4 is only .57 dB.

Optimising the shape of the phase pulse has been investigated. In [30] an optimised phase pulse for M=8, L=3 and h=1/8 gave only a 0.2dB gain over a GMSK phase pulse. For other M, gains up to 0.9 dB were found. [31] also investigates optimised phase pulses but concluded "the commonly known signal shapes are not too far from optimal performance".

Multi-h CPM is summarised in [8] and shows for the same bandwidth consumption, multi-h CPM has about a 2 dB d^2min improvement for M=4, 3RC, across a range of h. However, the increase in receiver complexity is considered beyond the scope of this thesis.

2.6.3 Complexity Reduction

Spectrally efficient CPM configurations have a non-binary symbol alphabet and smooth phase pulses lasting multiple symbol periods which leads to maximum-likelihood receivers with high complexity [7]. There is a significant body of literature proposing reduced complexity detectors, summarised by Perrins in [32].

The size of the receiver matched filter bank has been reduced by truncating the phase pulse and also by using a modified and reduced set of basis functions [33] [34]. The number of Viterbi trellis states has been reduced by

searching only a subset of the full trellis or by using decision feedback [35] [36] [6]. Combining these techniques have been studied in [32] [37] [28].

Laurent proposed a pulse amplitude representation [38] and by ignoring the smaller amplitude pulse receiver complexity is reduced. This was extended to M-ary CPM by Mengali [39]. Kaleh [40] presents a near optimum reduced complexity Viterbi receiver based on the PAM decomposition.

Other than in [28], adjacent channel interference performance of these reduced complexity schemes has not been tested. For a carrier to interference ratio of -5 dB, their 64 state detector has approximately only 0.5 dB loss; the maximum-likelihood detector is 1280 states. Unfortunately, they place the interferer in the 2nd adjacent channel but in our application there is the far more stringent requirement of the interferer being in the 1st adjacent channel. Also, Simmons claims the reduced trellis size decoders have significantly increased susceptibility to adjacent channel interference (ACI) [41], although the carrier to interference levels used were large (-10 and -20 dB).

2.6.4 Literature Review Summary

Bandwidth and energy consumption of CPM configurations are well studied, and a few papers evaluate the CPM codes bandwidth properties against ETSI standard spectral masks. As one would expect, CPM code performance has not been evaluated against the specific 28 MHz ETSI channel that is the focus of the microwave radio application considered by this thesis.

Although there is a vast range of Viterbi decoder literature, there are few published details for FPGA targeted CPM receiver implementations. There were no published results found for fixed point CPM receiver models.

Several techniques for complexity reduction give large reductions in complexity with minimal performance degradation, however adjacent channel interference performance of these algorithms is largely unproven. This thesis avoids this issue by implementing a maximum-likelihood receiver and focusing on a low-cost, resource efficient FPGA implementation.

The CPM literature typically measures complexity in terms of the number of matched filters and Viterbi states. This is a limitation for the purpose of this thesis since here we must meet a specific cost requirement. FPGA cost and resource use is measured in logic cells and block rams rather than the number of Viterbi states or branch metric filters.

Chapter 3

Improving Spectral Efficiency: CPM Parameter Selection

3.1 Introduction

In microwave radio cellular backhaul applications, improving spectral efficiency is desirable, as long as receiver detection efficiency and interference sensitivity are not compromised. An increase in spectral efficiency means the same data rate can be transported using less bandwidth allowing the operator to lower costs by using less costly radio spectrum licenses. Alternatively, operators keep constant their use of the already limited radio spectrum, and provide higher data rates to support, for example, the growing demand for mobile data services.

An existing CPM microwave radio product transports 16 E1 data circuits (36 Mbit/s including overhead) in a ETSI 28 MHz channel. This is a nominal spectral efficiency of 1.3 bits/s/Hz. This modem achieves a 10^{-6} bit error rate (BER) at an SNR of 14 dB [42].

In this chapter, we show a 50% higher data rate, 24 E1 data circuits (54 Mbit/s including overhead), can be transported in the same 28 MHz channel without any degradation in receiver detetection efficiency. This provides a margin of 4.7 dB to the ETSI receive signal level threshold specification assuming a radio noise figure of 6 dB.

The spectral efficiency is improved to 1.9 bits/s/Hz by moving to a new CPM configuration with a longer duration phase pulse and a smaller modulation index. Both these factors reduce detection efficiency; this degradation is mitigated by moving to coherent CPM demodulation. We assume perfect timing and carrier phase recovery and assume a zero carrier frequency offset.

Others have attempted more ambitious schemes such as 37.5 Mbit/s in a 14 MHz channel; this is a spectral efficiency of 2.7 bits/s/Hz. However, the optimum detector for this scheme is complicated requiring 2048 matched filters and a Viterbi decoder of 1280 states. A reduced complexity approach was taken resulting in a detector that is not maximum-likelihood. Also, only 2nd adjacent channel interference (ACI) sensitivity was investigated; the microwave radio application in this thesis requires compliance with a more stringent 1st adjacent channel interference sensitivity specification [28].

There are a large number of CPM configurations that potentially meet our requirements. Symbol alphabets of 2, 4 or 8 have been used in real world implementations, phase pulse durations from 1 to 5 symbol periods, phase pulse shapes of raised cosine, rectangular, GMSK and others, and a wide range of modulation indexes are possible [10] [11] [12] [28] [32]. This chapter chooses a CPM configuration that meets the ETSI channel transmit spectral mask and interference rejection requirements, whilst providing an acceptable tradeoff between detection efficiency and receiver implementation complexity measured in terms of the Viterbi trellis and branch metric matched filter bank size.

This chapter starts by identifying 4 candidate CPM configurations by examining their theoretical detection and bandwidth efficiency. These candidate configurations are then simulated and their performance evaluated against an ETSI standard specifying limits for transmit power spectrum, interference sensitivity and receive signal threshold performance. The CPM configuration of h=1/4, L=3RC, M=4 is chosen because it meets the ETSI requirements, and has the lowest complexity of any scheme that exceeds the ETSI receive signal threshold requirements by more than 4 dB.

3.2 Analytical CPM Performance: Identifying CPM Configuration Candidates

There is a large number of possible CPM configurations, each with its own specific detection efficiency and bandwidth consumption characteristics. There are 4 parameters that specify a CPM configuration:

- Symbol Alphabet Size (M)
- Modulation Index (h)
- Phase Pulse Duration (L)
- Phase Pulse Shape

These 4 parameters are reflected in the standard equation for how the phase of a CPM modulated signal changes with time and data symbols; see Equation (3.1). $q(t)$ determines the shape of the phase pulse and α_i is a data symbol from an alphabet of size M. "n" refers to the nth data symbol transmitted.

$$\theta(t, \mathbf{a}) = 2\pi h \sum_{i=n-L+1}^{n} \alpha_i q(t - iT) \tag{3.1}$$

Continuous phase modulation is a coded modulation and by calculating a CPM configuration's minimum Euclidean distance, relative detection efficiency performance comparisons can be made. Symbol error rate is derived from the minimum distance squared, d^2min, as shown in Equation (3.2). The procedure for calculating a CPM configuration's minimum distance is well documented[6]. For the purposes of this thesis, this was implemented in Matlab ; Appendix E.1.1 contains the source code.

$$P_e \approx Q(\sqrt{d^2min\frac{E_b}{N_0}}) \tag{3.2}$$

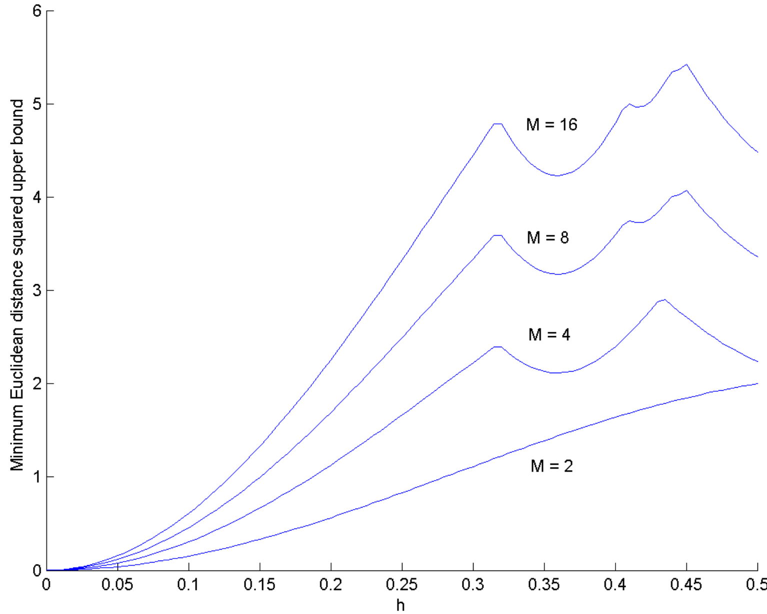

Figure 3.1: Effect of Symbol Alphabet Size (M) on Detection Efficiency, L=1, Raised Cosine Phase Pulse

Figure 3.1 illustrates how M and h affect detection efficiency for L=1 and a raised cosine (RC) phase pulse. Increasing M improves detection efficiency and the general trend for increasing h is an increase in detection efficiency. Increasing these parameters also causes an increase in bandwidth consumption. It is worth noting that this is an upper bound. At certain so-called "weak" modulation indices d^2min no longer reflects actual symbol error rate performance. None of the CPM configurations considered in this thesis fall into this category [6].

Figure 3.2 shows the effect of phase pulse durations from 1 to 5 nominal symbol periods for a quaternary alphabet and raised cosine phase pulse. The effect of a longer, smother phase pulse is to reduce detection efficiency; the flip side is a reduction in bandwidth consumption [6].

It is clear that in order to choose a CPM configuration it is necessary to evaluate it in a combined detection and bandwidth efficiency sense.

3.2.1 Choice of Symbol Alphabet Size (M) and Phase Pulse Shape

In this thesis, only a quaternary symbol alphabet (M=4) is investigated. This is almost certainly superior to a binary symbol alphabet (M=2). Across a range of modulation indices, M=4, L=3RC has almost 3dB better detection performance compared to M=2, L=3RC when comparing CPM configurations with the same

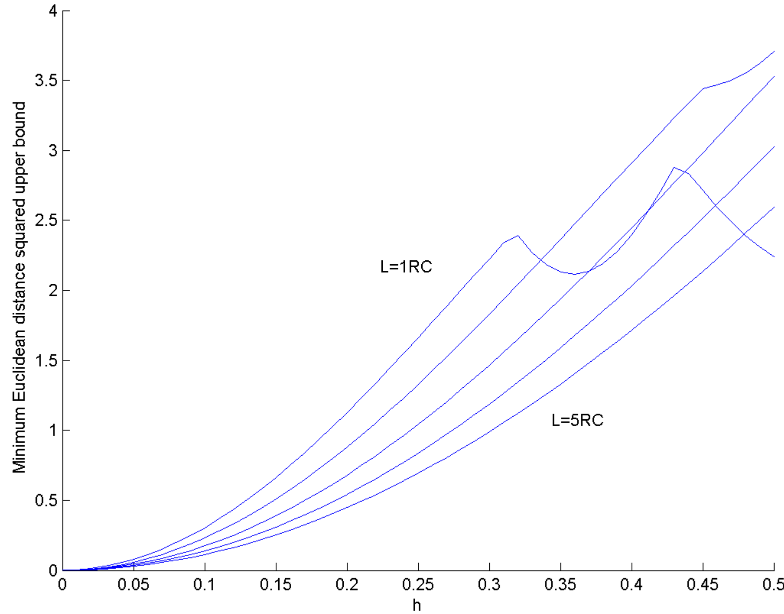

Figure 3.2: Effect of Phase Pulse Duration (L) on Detection Efficiency, M=4

bandwidth consumption [6]. An empirical model was developed that shows M=8 may be the best even-integer alphabet size [29]. However, in the example given it is only 0.55 db better than M=4 yet has substantially higher complexity so is not considered further. [1]

There are an infinite number of possible phase pulses although there are a standard few described in the literature: raised cosine (RC), spectrally raised cosine (SRC), rectangular (REC), Gaussian minimum shift keying (GMSK), tamed frequency modulation (TFM) and continuous phase frequency shift keying (CPFSK). Raised cosine is arguably the most popular and is often used as the baseline for comparison [6]. A phase pulse shape optimised to the bandwidth and detection efficiency requirements is possible but Asano concludes "the commonly known signal shapes are not too far from optimal performance" [31].

In any event, the implementation proposed in Chapters 5 and 6 supports alternative phase pulses. The phase pulse shape is reflected in the matched filter bank coefficients which are stored in volatile memory. The design can be easily modified to support external reloading of this memory. For these reasons, the choice of phase pulse is not investigated; the simulations and FPGA

[1] M=8 has higher complexity since the number of matched filters and Viterbi states increases exponentially with M. However, an advantage of moving to M=8 is that the symbol rate drops by 50% since the number of bits per symbol increases by 50%. This eases the throughput requirement on the implementation since the amount of processing time available for each symbol is now %50 higher. This is particularly significant for the Viterbi iteration loop since it contains feedback that causes a bottleneck in pipelined FPGA implementations.

h (fraction)	h	k	p
1/3	0.33...	1	6
2/7	0.29...	1	7
1/4	0.25	1	8
1/5	0.2	1	10

Table 3.1: Candidate Modulation Indices (h)

implementations in this thesis use a raised cosine phase pulse. [2]

We have chosen a quaternary alphabet (M=4) and a raised cosine phase pulse. Candidates for the modulation index (h) and phase pulse duration (L) are selected next.

3.2.2 Modulation Index (h) and Phase Pulse Duration (L) Candidates

In order to identify modulation index and phase pulse duration candidates, the CPM configuration must be evaluated in a combined detection efficiency and bandwidth efficiency sense. Bandwidth of the transmitted baseband CPM signal is calculated using a numerical method described in [7, pg 231]. We define the double-sided bandwidth as the bandwidth containing 99% of the transmitted power. This numerical method was implemented by the author using Matlab; Appendix E.1.2 contains the source code listing.

For coherent reception, the modulation index must be of the form $h = \frac{2k}{p}$ where k and p are integers. The modulation indices considered are shown in Table 3.1. Modulation indices above this range consume too much bandwidth and values below this range have too low a detection efficiency. There are other k, p combinations that sit within this range, but all require large values of p and so are considered too costly in terms of implementation. For example, $h = \frac{3}{11} = 0.27$ is interesting, but when put into the form $h = \frac{2k}{p}$, k=3 and p=22. 22 phase states is considered too costly for implementation.

Phase pulse durations from the range 1 to 5 symbols are considered. Combined with the 4 modulation index values of Table 3.1 gives 20 CPM configurations to be evaluated.

All configurations are evaluated in a combined detection efficiency and spectral efficiency sense with the results shown in Figure 3.3. Detection efficiency is calculated in terms of d^2min relative to minimum shift keying (MSK). MSK has a d^2min of 2. Spectral efficiency is calculated from the bandwidth calculation described above.

The existing CPM product's performance is also plotted in Figure 3.3. The spectral efficiency of the CPM product is calculated as the symbol rate data rate (36 Mbit/s) divided by the ETSI channel bandwidth (28 MHz). When comparing the existing CPM product's spectral efficiency with these new CPM configurations, there is an assumption that the transmit power spectrum determines the channel spacing. This is not necessarily true as adjacent channel

[2]The current FPGA design uses a LUT4 primitive to store the matched filter coefficients. By replacing the LUT4 with an SRL16 primitive the filter coefficients can be loaded serially, yet still read and addressed as a standard memory.

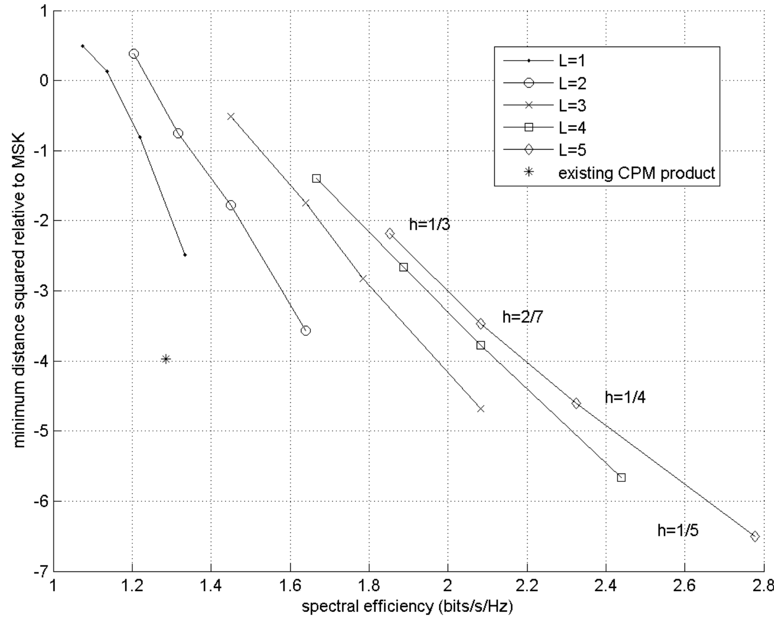

Figure 3.3: Relative Detection Efficiency and Spectral Efficiency for several CPM Configurations

interference tolerance may require the transmitted bandwidth to lie some extra distance inside the transmit channel mask.

The existing CPM product's detection efficiency relative to MSK is approximated by calculating its equivalent d^2min using the existing products threshold performance of 14 dB SNR at 10^{-6} bit error rate and Equation (3.2) solved for d^2min. This is an approximation because the threshold performance is specified at a bit error rate, whereas Equation (3.2) calculates a symbol error rate.

The first point to note is that h acts to tradeoff detection efficiency and spectral efficiency. Only by increasing L can both detection efficiency and spectral efficiency be improved. However, the gains get less and less at each higher L, while the complexity increases exponentially with L [6].

CPM configurations with L=4 or L=5 are ruled out as their complexity is considered too high. Indeed, Chapters 5 and 6 implement a CPM configuration with L=3 that meet the cost requirement for the application in this thesis. Moving to L=4 increases the complexity by 4 times which would cause the cost requirement to be exceeded.

There are only 4 remaining CPM configurations that are in the vicinity of meeting the 50% increase in spectral efficiency goal of 1.9 bits/s/Hz. These are listed in Table 3.2. Three of the 4 schemes promise to improve detection efficiency compared with the existing CPM product. This table also shows the maximum-likelihood receiver complexity in terms of matched filters and Viterbi trellis states.

h	L	M	no. matched filters	no. Viterbi trellis states
1/5	2	4	32	40
2/7	3	4	128	112
1/4	3	4	128	128
1/5	3	4	128	160

Table 3.2: Candidate CPM Configurations

3.3 CPM Configuration Evaluation Metrics: ETSI Compliance

A floating point Simulink model is used to carry out simulations confirming the analytical detection efficiency and bandwidth consumption results presented in the previous section and evaluate the candidate CPM configurations against an ETSI microwave radio standard. The cellular backhaul microwave radio application considered in this thesis requires a product to meet this standard.

The ETSI specification [1, Annex D] constrains three aspects of the modulation:

1. Bandwidth - The transmitted power spectrum must fit within a low-pass spectral mask.

2. Detection Efficiency - At a specified received signal power level, the receiver bit error rate (BER) must be less than a specified value.

3. Interference Rejection - In the presence of adjacent channel interference (ACI) or co-channel interference (CCI), detection efficiency can degrade by no more than specified limits.

It is worth noting that the ETSI detection efficiency specification is the minimum performance required to achieve ETSI compliance. System gain is an important product marketing specification, and since improvements to detection efficiency (receiver sensitivity) directly improve system gain, it is desirable to maximise the margin to this specification. For example, improving the detection efficiency by 3 dB allows the use of an antenna approximately half the size and therefore significantly reduced cost. The chosen CPM configuration must meet the ETSI requirements whilst providing an acceptable tradeoff between detection efficiency and receiver implementation complexity and cost.

3.3.1 Specific Application

The specific application of interest to this thesis is covered by the ETSI standard: Fixed radio systems, Characteristics and requirements for point-to-point equipment and antennas. The frequency bands of interest are 13 GHz and 15 GHz which are covered by Annex D of this standard. Our 50% improved spectral efficiency CPM configuration transports 24 E1 circuits within the 28 MHz channel. The ETSI standard classifies such a system as spectrum efficiency class 2, system D.1. MHz channel [1, Annex D].

interference type	carrier to interference ratio (dB)	allowed SNR degradation (dB)	new SNR (dB)	new $\frac{E_b}{N_o}$ (dB) (M=4)
co-channel	23	1	19.7	16.7
co-channel	19	3	21.7	18.7
1st adjacent channel	0	1	19.7	16.7
1st adjacent channel	-4	3	21.7	18.7

Table 3.3: ETSI Co-Channel and 1st Adjacent Channel Interference Performance [1, Table D.7]

3.3.2 Bandwidth: Transmit Power Spectral Density (PSD)

The transmitted signal must lie within the radio frequency spectrum mask shown in Figure 3.5. The channel spacing is 28 MHz. This mask is specified relative to the carrier frequency f_o. The transmitted spectrum is assumed to be symmetrical and so only single sided limits are specified. The 0dB point on the mask corresponds to the power spectral density (PSD) at the carrier frequency [1, Table D.4].

In this thesis the transmitter is modelled at baseband and so the simulated baseband transmit power spectrum is directly evaluated against the spectrum mask shown in Figure 3.5. It is assumed the up-conversion process does not alter the shape of the transmitted power spectrum.

3.3.3 Detection Efficiency: Bit Error Rate as a function of Receive Signal Level

It is widely known that in general, detection efficiency performance can be traded off against bandwidth efficiency performance. Section 3.3.2 constrains the bandwidth and here we constrain detection efficiency; at a receive signal level of -75 dBm, the bit error rate (BER) must be less than 10^{-6} [1, Table D.6].

For the purposes of simulation, -75 dBm receive signal level is equivalent to a signal to noise ratio (SNR) of 18.7 dB and energy per bit to noise ratio ($\frac{E_b}{N_o}$) of 15.7 dB. Appendix B details this calculation.

3.3.4 Interference Rejection

The radio must achieve a minimum level of detection efficiency in the presence of co-channel and adjacent channel interference. Table 3.3 specifies the strength of the interferer and the amount by which SNR may be degraded while still achieving a 10^{-6} bit error rate.

In practice, the candidate CPM configurations are evaluated against a more stringent specification to provide engineering margin. We increase the carrier to interference ratios in Table 3.3 by 1 dB and bit error rate is measured without forward error correction (FEC) and is relaxed to 10^{-5}. This leaves approximately two orders of magnitude of BER margin before exceeding the error correcting capabilities of the Reed Solomon FEC. The code used has a threshold at about 10^{-3}; a BER of 10^{-3} at the FEC input results in approximately 10^{-6} at the output.

3.4 Simulated CPM Performance: Selecting a CPM Configuration For Implementation

The simulation system model is presented first, followed by simulation results evaluating the candidate CPM configurations against the ETSI requirements. The h=2/7, L=3 configuration does not meet the spectral mask and so is rejected. The h=1/4, L=3 configuration meets all ETSI requirements and so meets the 50% increase in spectral efficiency target; the h=1/5, L=3 configuration has a smaller minimum distance so is rejected. The h=1/5, L=2 configuration does not meet the adjacent channel interference rejection requirement.

3.4.1 Simulation System Model

The simulation system model is shown in Figure 3.4. The 6 key parts of this model are:

- Transmitter - The transmitter comprises a linear feedback shift register data source of polynomial $x^{15} + x^{14} + 1$, a Reed Solomon forward error correction encoder of codeword length (n) 255 and message length (k) 239, and a CPM modulator. The symbol rate is 27 MSymbols/s and 8 samples per symbol.

- Receiver - The receiver comprises a low pass filter, CPM demodulator with traceback depth of 32 symbols and a Reed Solomon decoder. The low pass filter provides close-in adjacent channel rejection.

- Channel - The channel is modelled with additive white Gaussian noise only. There is no channel delay. Phase and symbol synchronisation are ideal.

- Interference Generator - A second transmitter model generates co-channel and adjacent channel interference. Gain, phase and frequency are adjustable.

- Bit Error Rate (BER) Checker - Transmitted bits and symbols are compared with the received symbols and bits, both before and after FEC decoding to determine the symbol and bit error rate.

- Transmit Power Spectral Density Measurement - Transmitter power spectral density is measured using a periodogram. The FFT length is 2048 samples, it uses a Hanning window and the periodogram averages over 256 spectra.

3.4.2 Bandwidth: Transmit Power Spectral Density (PSD)

The simulated baseband transmit power spectrum of the 4 candidate CPM configurations is shown in 3.5 together with the relevant ETSI spectral mask. Figure 3.6 is a zoomed in view of the same data and shows that h=2/7, L=3 CPM configuration is the only one that fails to meet the spectral mask.

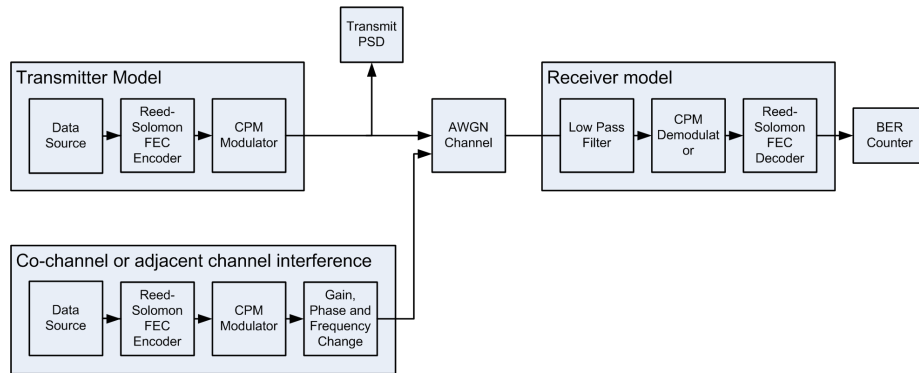

Figure 3.4: Simulation System Model

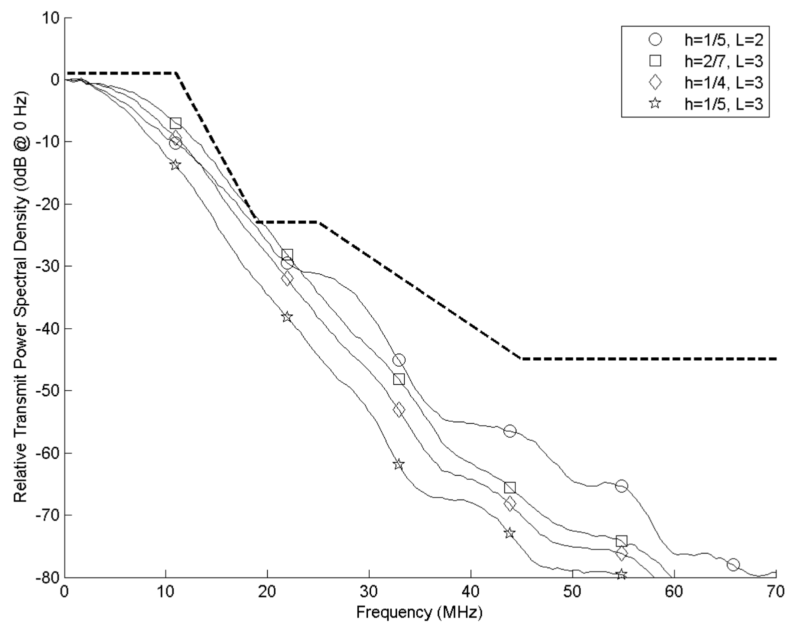

Figure 3.5: Simulated Transmit Power Spectral Density of Candidate CPM Configurations, Various (h, L), M=4, 27 Msymbols/s)

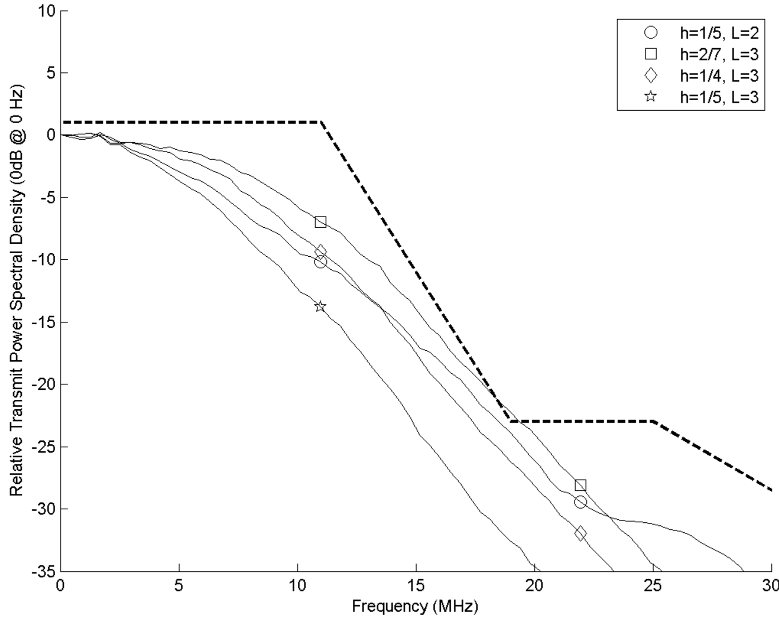

Figure 3.6: Zoomed in version of Figure 3.5

3.4.3 Detection Efficiency and Interference Rejection Performance: h=1/4, L=3

3.4.3.1 Detection Efficiency: Bit Error Rate as a function of SNR

With an AWGN channel only, simulated error rate performance is shown in Figure 3.7. The symbol and bit error rate data was collected by accumulating a minimum of 100 symbol errors, or for the FEC BER graph a minimum of 100 bit errors after FEC. The receiver low-pass filter is removed for this simulation.

A BER of 10^{-6} is achieved with an $\frac{E_b}{N_o}$ of 14 dB. Furthermore, when the Reed Solomon FEC is included then $\frac{E_b}{N_o}$ is approximately 10.6 dB at a BER of 10^{-6}. This is 5.1 dB better than the ETSI requirement of 15.7 dB.

The existing CPM modem product achieves a 10^{-6} BER at an SNR of 14 dB or 11 dB $\frac{E_b}{N_o}$ [42] assuming ideal timing synchronisation and no degradation due to fixed precision arithmetic. [3] These results show that the new CPM configuration has the potential to be 0.4 dB better in terms of detection efficiency. However, a low-pass adjacent channel rejection filter is required to meet the ACI rejection requirements. This filter also degrades clear channel performance as described in the next section.

[3] SNR is 3 dB higher than $\frac{E_b}{N_o}$ for a quaternary alphabet.

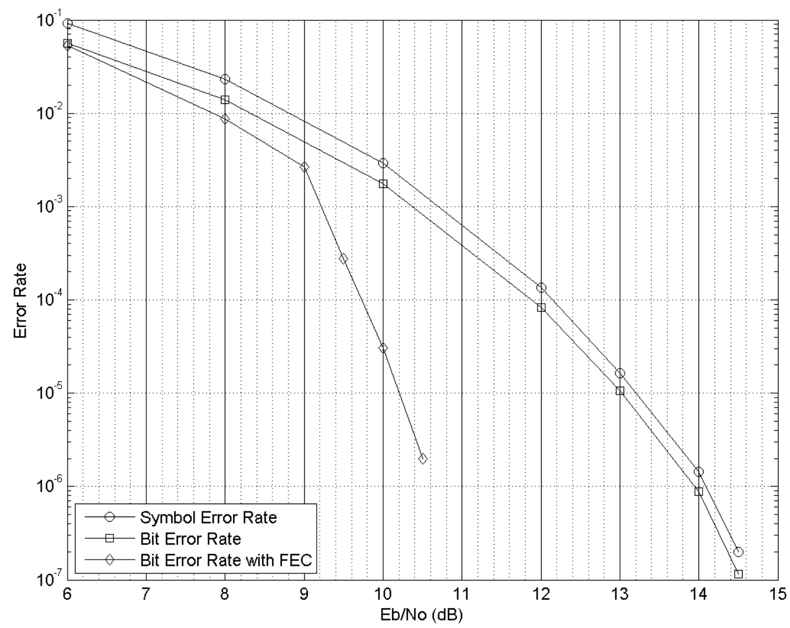

Figure 3.7: Simulated Bit Error Probability with and without Reed Solomon FEC, AWGN Channel, No ACI Reject Filter, h=1/4, L=3RC, 27 Msymbols/s

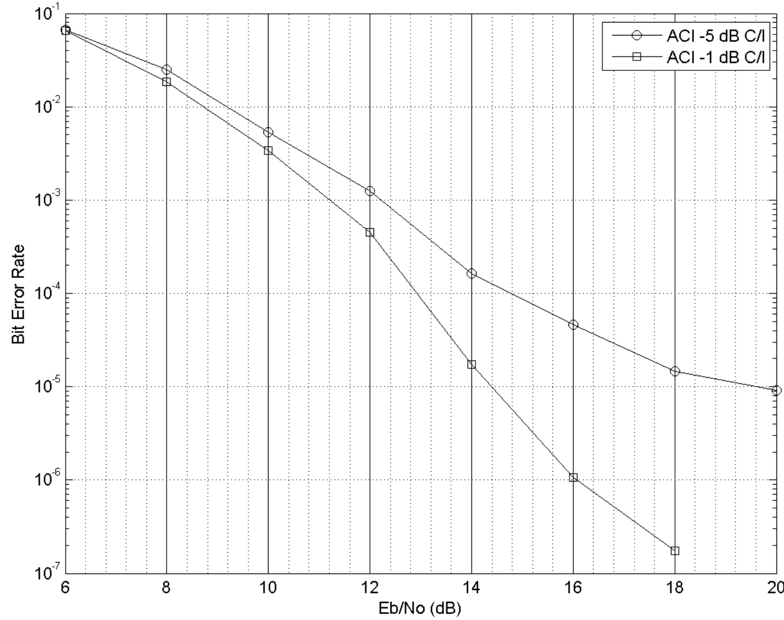

Figure 3.8: Simulated Bit Error Rate with Adjacent Channel Interference, h=1/4, L=3, M=4, 27 Msymbols/s

3.4.3.2 1st Adjacent Channel Interference

Figure 3.8 shows adjacent channel performance with carrier to interference (C/I) ratios of -5 dB and -1 dB. This adjacent channel interference is 1 dB stronger than specified in the ETSI specification. For the 5 dB adjacent channel interferer test, the bit error rate is very close to meeting the target of 10^{-5} at an $\frac{E_b}{N_o}$ of 18.7 dB. This is well below the 10^{-3} Reed Solomon FEC threshold and so with FEC present, the BER falls well below the ETSI specification limit of 10^{-6}.

The less stressful 1 dB ACI test shows a bit error rate of less than 10^{-6} at 16.7 dB $\frac{E_b}{N_o}$, clearly meeting the ETSI requirement.

The radio's intermediate frequency (IF) amplifier stages contribute significantly to the receiver's selectivity. Nevertheless, the final IF stage in this radio is 36 MHz wide which passes a significant amount of 1st adjacent channel signal power. The front end of the digital portion of this receiver contains decimation stages and other low pass filters which provide adjacent channel rejection. These are modelled with a single low pass filter of 96 taps.

The low pass filter cutoff frequency has a significant impact on receiver BER performance. If the cutoff is set too low then the in-band signal is distorted too much and BER performance is degraded. On the other hand, if the filter cutoff is set too high, then too much adjacent channel power passes into the demodulator and BER performance is degraded. A cutoff frequency of 14 MHz is chosen to provide a compromise between clear channel performance and adjacent channel rejection.

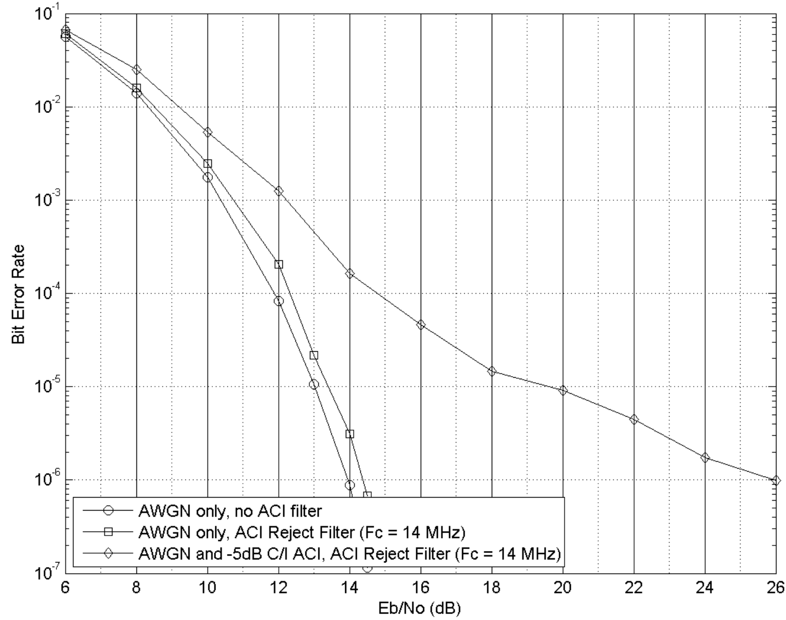

Figure 3.9: Simulated Bit Error Rate Demonstrating Effect of ACI Reject Filter and Adjacent Channel Interference, h=1/4, L=3, M=4, 27 Msymbols/s

Figure 3.9 shows the effect of the low pass filter on clear channel performance. At a BER of 10^{-6}, $\frac{E_b}{N_o}$ is 14.4 dB, a degradation of 0.4 dB due to the low pass filter. Without the filter present the new CPM configuration was 0.4 dB better than the existing product. This means that with the filter present the new CPM configuration has the same level of detection efficiency as the existing CPM product. This is 4.7 dB better than the minimum required in the ETSI standard.

3.4.3.3 Co-channel Interference

Co-channel interference is simulated at carrier to interference (C/I) ratios of 19 dB and 23 dB and the results are shown in Figure 3.10. In both cases the bit error rate is less than 10^{-5} at an $\frac{E_b}{N_o}$ of 16 dB. The requirement is for a bit error rate of less than 10^{-5} at $\frac{E_b}{N_o}$ of 18.7 dB and 15.7 dB. This is a clear pass to the ETSI requirements.

3.4.4 Detection Efficiency and Interference Rejection Performance: h=1/5, L=2

The h=1/5, L=2 CPM configuration is interesting because compared to h=1/4, L=3 it has a matched filter bank 1/4 the size and Viterbi trellis with 1/3 of the

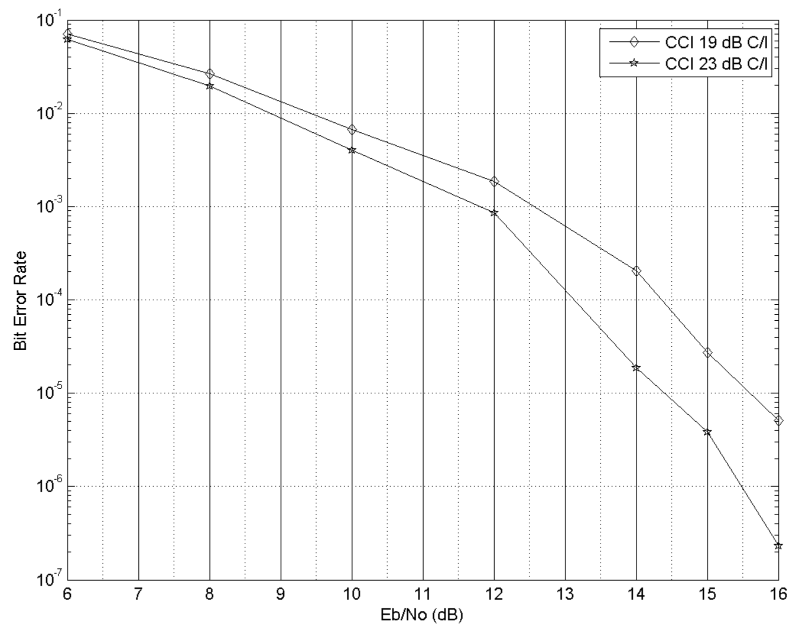

Figure 3.10: Simulated Bit Error Rate with Co-Channel Interference, h=1/4, L=3, M=4, 27 Msymbols/s

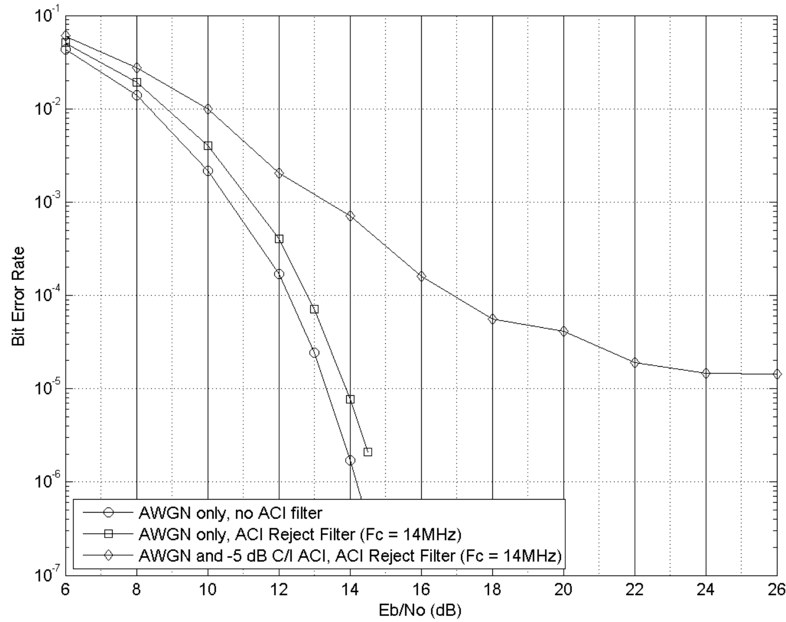

Figure 3.11: Simulated Bit Error Rate with and without Adjacent Channel Interference, h=1/5, L=2, M=4, 27 Msymbols/s

states. The simulated detection efficiency performance of this CPM configuration is shown Figure 3.11.

At a BER of 10^{-6}, $\frac{E_b}{N_o}$ is approximately 14.8 dB with the low pass ACI reject filter present. This is 0.4 dB worse than the h=1/4, L=3 CPM configuration.

However, the ACI rejection performance is considerably worse. At an $\frac{E_b}{N_o}$ of 18.7 dB the BER is $5 * 10^{-5}$. This does not meet the stated requirement of a BER less than 10^{-5} at 18.7 dB $\frac{E_b}{N_o}$.

The ACI performance could be improved by reducing the low-pass ACI reject filter cutoff frequency. However, this also increases the amount of in-band signal removed at the band edge. The clear channel performance is already reduced by 0.6 dB due to the low-pass ACI reject filter, so further lowering the cutoff frequency will degrade the in-band performance even further.

3.5 Conclusion

The 50% spectral efficiency improvement requirement has been met by a move to a CPM configuration with a longer duration phase pulse (L=3) and coherent reception. This new CPM configuration (h=1/4, L=3, M=4) is simulated and shown to exceed the ETSI clear channel detection efficiency requirement by 4.7 dB including Reed Solomon forward error correction. This level of detection efficiency is the same as the existing CPM microwave radio product, and because the spectral efficiency is 50% higher, this is a significant improvement.

However, the longer phase pulse and move to coherent reception result in a more computationally expensive receiver. For this new CPM configuration to have practical significance its implementation cost must be low enough. The following chapters in this thesis now focus on a low cost implementation. The next chapter demonstrates the transition to a fixed point simulation model that is used to optimize word-lengths in the branch metrics and path metrics processing units of a coherent CPM receiver.

Chapter 4

Toward an FPGA Implementation: Fixed Point Modelling

4.1 Introduction

The previous chapter demonstrated that a 50% increase in spectral efficiency without loss of detection efficiency is obtained by moving to coherent reception and increasing the phase pulse length to 3 symbol periods. This new CPM configuration of h=1/4, L=3, M=4, requires a CPM Viterbi decoder of 128 matched filters and 128 trellis states, all computed at a 27 MSymbols/s rate.

Before moving to an FPGA implementation, several approximations and optimizations to the floating point maximum-likelihood receiver must be made to make the implementation realizable and cost effective. These are:

- Reduced sampling rate
- Fixed point numeric representation
- Reduced survivor path memory size

These approximations are relevant to the receiver functional blocks as shown in Figure 4.1. This chapter demonstrates the effect of these approximations on receiver detection efficiency. The choice of sampling rate, fixed point word-lengths and survivor path memory depth are optimised for implementation size (product cost) and performance.

4.2 Implementation Target: ASSP, ASIC, FPGA, DSP

An FPGA is the targeted implementation technology because it is the most cost effective implementation technology currently available. An existing CPM microwave radio [42] uses an FPGA modem. Although a standard cell based application specific integrated circuit (ASIC) offers higher performance, lower power, and consumes less silicon area [43], the non recurring engineering cost

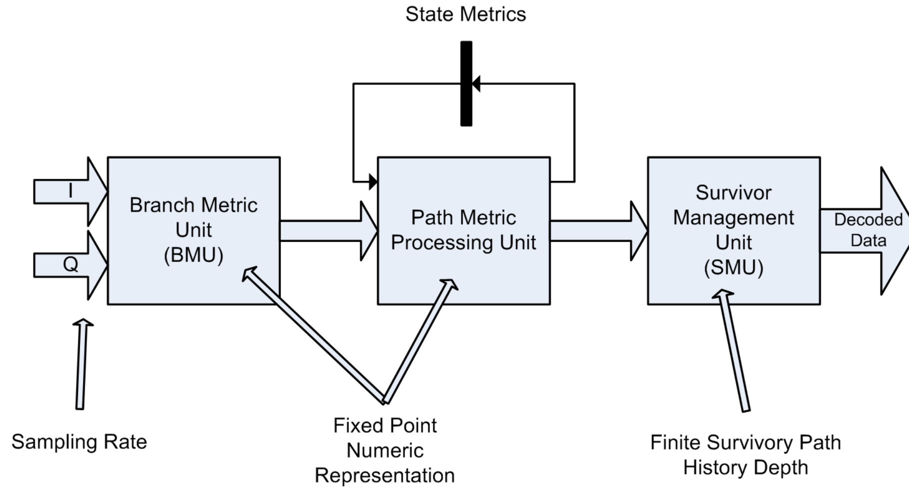

Figure 4.1: Approximations to the maximum-likelihood receiver

	DSP	FPGA
Part	Texas Instruments TMS320C6421	Xilinx Spartan 3 XC3SD1800A
Technology Process	90nm	90nm
Peak 16 bit x 16 bit multiplies per second	1.6 GMULT/S	21 GMULT/s [44]
Cost (US$)	$8.95 @10k	$29.85 @25k
$ per GMULT/S	$5.60	$1.40

Table 4.1: Comparison of DSP and FPGA Multiplication Capability

of an ASIC cannot be justified by the expected product volumes. Besides which, the design flexibility and time-to-market advantage of an FPGA cannot be ignored.

A digital signal processor based platform simply does not meet the high computation requirements of this CPM receiver. For example, this CPM receiver's branch metric filter bank alone requires 27.6 giga-multiplies/s. In this thesis the targeted low-cost FPGA is a Xilinx Spartan 3 1800-ADSP. Table 4.1 provides a comparison between this FPGA and a Texas Instruments DSP. This DSP was picked for comparison purpose because it is the lowest cost per multiply TI DSP at the same technology process node. In this comparison, the FPGA performance is 10 times higher than the DSP and on a cost per multiply basis the FPGA is 4 times cheaper.

Provigent provide application specific standard product (ASSP) solutions for the cellular backhaul microwave radio market. However, these chipsets do not support CPM and so are ruled out for the purposes of this thesis [45].

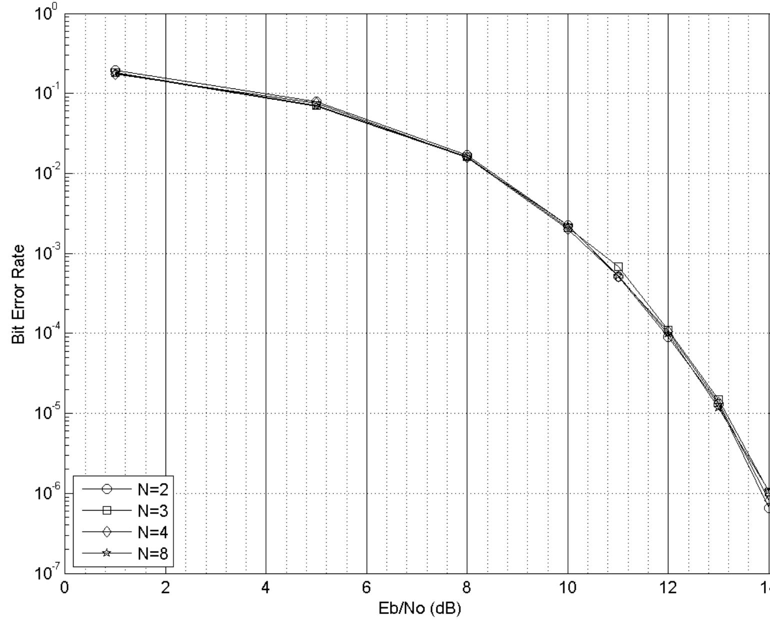

Figure 4.2: Effect of Sampling Rate on Detection Efficiency

4.3 Sampling Rate

The floating point model in Chapter 3 approximated continuous time by representing each symbol with 8 samples. At a symbol rate of 27 MSymbols/s this is a sample rate of 216 MSamples/s. Due to Nyquist sampling theory, this sample rate allows frequencies up to 108 MHz to be represented without aliasing. This is obviously too high a sample rate for an implementation because the received signal does not contain significant energy at these high frequencies; it is band-limited by intermediate frequency (IF) filtering stages within the receiver. For example, the existing CPM receiver IF 3dB bandwidth is 36 MHz.

For the purposes of this thesis, we assume the analog I/Q signal is sampled appropriately to meet the anti-aliasing and dynamic range requirements for the receiver. This is followed by a multi-rate filter to reduce (or increase) the sample rate to whatever the requirement is for this coherent CPM receiver processing. The multi-rate filter, analog anti-aliasing low pass filtering and analog to digital converter sample rate choice is beyond the scope of this thesis.

Since there is a direct correlation with sampling rate and computational complexity, finding the minimum sampling rate that does not significantly degrade receiver detection efficiency is important. Figure 4.2 shows bit error rate vs $\frac{E_b}{N_o}$ simulation results for sampling rates from 2 to 8 samples per symbol. The Viterbi path memory size is large (32 symbols deep) and the receiver model uses 64-bit floating point arithmetic.

The results show minimal, if any, performance degradation due to the re-

duced sample rate. Therefore, the implementation will use the minimum sample rate required for timing recovery which is 2 samples per symbol (54 MSamples/s). This result is not surprising when considering the transmitted power spectral density and receiver ACI reject filter cutoff frequency. At 27 MHz (54/2), the transmitted power spectral density is more than 40 dB down, besides which, the low pass ACI reject filter cutoff is 14 MHz. i.e. very little useful received signal energy is present at 27 MHz so higher sampling rates do not improve performance.

Furthermore, the FPGA branch metrics implementation described in Chapter 5 uses a distributed arithmetic approach which becomes particularly resource efficient for 4-input LUT (look-up table) FPGAs when the sample rate is twice the symbol rate.

4.4 Fixed Point Modelling

Fixed point modelling of the receiver is a significant step in determining the FPGA implementation size and cost. The models of Chapter 3 used large word-lengths (64 bit) and floating point arithmetic so that receiver performance is not degraded by arithmetic overflow and quantisation noise. By moving to fixed point arithmetic and using word-lengths much less than 64 bits, the FPGA resources used, and hence cost of the implementation is minimised. This section presents fixed point modelling results that give insight into the relationship between word-length (hardware cost) and receiver detection efficiency.

Although the literature contains fixed point modelling results for Viterbi decoding in error correction applications, there are no known fixed point simulation results for CPM Viterbi detection. This thesis performs fixed point simulations using the same system model presented in Chapter 3 but replacing key floating point data types with Matlab's fixed point number data type (fi()). Also, specific parts of the receiver model were rewritten to match the FPGA VHDL implementation presented in Chapters 5 and 6.

Developing a fixed point model is also important because it creates a baseline to which the FPGA implementation can be tested against. The VHDL verification strategy described in Appendix A uses results from the fixed point model stored in VHDL arrays to verify the VHDL implementation.

The number of samples per symbol is 2 and the Viterbi path history depth is 16. At least 100 bit errors are accumulated at each $\frac{E_b}{N_o}$ simulation data point. The low-pass adjacent channel interference reject filter is not present. Adjacent channel interference rejection performance of the fixed point model has not been investigated. Forward error correction is not included.

There are 4 word-lengths that must be determined:

1. In-phase and quadrature received signal word-length.
2. Branch metric filter bank coefficient word-length.
3. Output branch metrics word-length.
4. Path metric word-length.

Simulation is used to investigate the first three. Path metric word-length is chosen analytically.

4.4.1 Floating Point vs Fixed Point Numeric Representation

Fixed point addition and multiplication generally require less FPGA resources than the equivalent operations in floating point. For example, floating point addition requires an extra operation to normalise the smallest operand's mantissa before adding. Also, for the same word-length, fixed point numbers provide higher precision since bits are not consumed by the exponent representation. However, the dynamic range of fixed point numbers is severely restricted when compared to a floating point representation.

The limited dynamic range of fixed point numbers is particularly relevant in a communications receiver, since received signal levels vary over a wide range. It is assumed that automatic gain control (AGC) circuitry keeps the sampled received signal within the dynamic range of the ADC. And once inside the FPGA, front end low pass filtering would be followed by further AGC. AGC is outside the scope of this thesis but it has been assumed that the received signal is scaled such that there is 1 bit of headroom for the in-phase and quadrature inputs to the branch metric unit.

4.4.2 Quadrature and In-phase Received Signal Word-length

For this experiment, the received signal is quantised in the range of word-lengths between 5 and 8 bits. In contrast, the simulation results from the previous chapter represented the received signal using 64-bit floating point numbers. Figure 4.3 presents the results.

At word-lengths of 5 and 6 bits the receiver detection efficiency is degraded by approximately 0.4 dB and 0.2 dB at a BER of 10^{-4}. There is a small degradation for a word-length of 7 bits and for 8 bits the results closely match the floating point model.

The received signal word-length is a critical factor in determining the amount of FPGA resources used to implement the filter bank. Chapter 5 proposes a bit-serial arithmetic technique which is very low cost if the received signal word-length is 7 bits or less. For larger word-lengths the amount of FPGA resources must double in order to keep the FPGA operating frequency low enough such that static timing is met. Hence the received signal word-length is chosen to be 7 bits.

The filter bank coefficient word-length is 16 bits and the matched filter arithmetic is carried out such that there is no rounding nor loss of precision. The resulting branch metrics are kept at full precision and converted to 64 bit floating point for the remainder of the model. Path metric computations use double precision (64 bit) floating point arithmetic.

4.4.3 Branch Metric Filter Bank Coefficient Word-length

The filter bank coefficients are also quantised to a word-length from 5 to 8 bits. The coefficients are quantised such that they extend across the complete dynamic range of the fixed point representation.

Figure 4.4 presents the results. At a word-length of 5 bits the detection efficiency degrades by 0.5 dB at a BER of 10^{-4} and at 6 bits there is a slight degradation. At word-lengths of 7 and 8 bits there is very little degradation

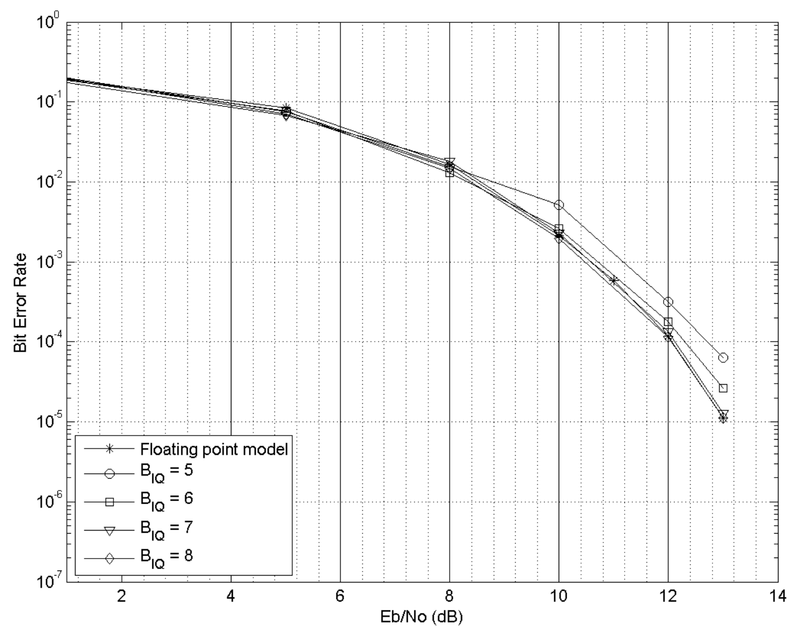

Figure 4.3: Effect of Quantised Received Signal Word-length on Receiver Detection Efficiency

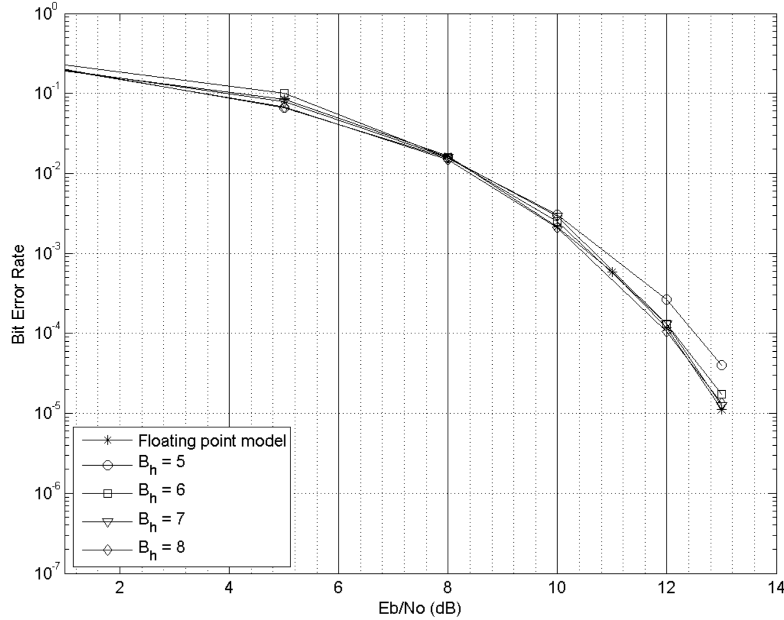

Figure 4.4: Effect of Quantised Branch Metric Filter Bank Coefficient Word-length on Receiver Detection Efficiency

at a BER of 10^{-5}. 7 bits is chosen as the minimum word-length that does not degrade performance.

The in-phase and quadrature received signal are quantised to a word-length of 16 bits and the branch metric arithmetic is performed at full precision. The resulting branch metrics are kept at full precision and converted to 64 bit floating point for the remainder of the model.

4.4.4 Branch Metric Word-length

This experiment investigates the effect of branch metric word-length and demonstrates the overall receiver detection efficiency degradation due to the transition to a fixed point model. The received signal word-length and matched filter bank coefficient word-lengths are set to the minimum that did not cause receiver performance degradation. i.e. 7 bits. The branch metric outputs are calculated in full precision, the top 2 bits truncated and the result then rounded down to a word-length of 7, 8, 9 or 10 bits.

The path metric word-length is always 2 bits more than the branch metric word-length; this is justified by the use of Hekstra's method as discussed in section 4.4.5

Figure 4.5 presents the results in graphical form and Table 4.2 shows the receiver detection efficiency degradation for each branch metric word-length.

Branch metric word-lengths of 7 and 8 bits cause significant degradation.

Branch Metric Word-length	SNR Degradation at BER of 10^{-4} (dB)
7	1.3
8	0.4
9	0.2
10	0.2

Table 4.2: Detection Efficiency Degradation due to Branch Metric Quantisation

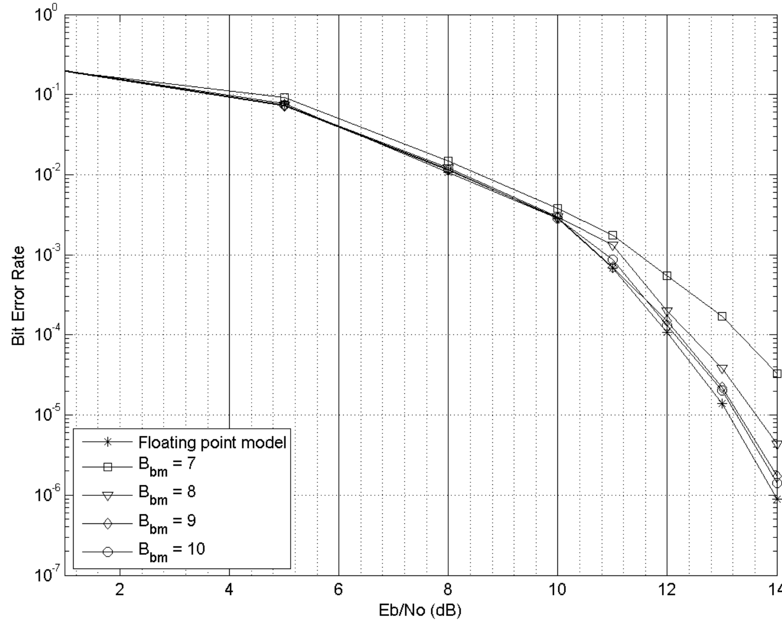

Figure 4.5: Effect of Branch Metric Word-length on Receiver Detection Efficiency

At 9 and 10 bits the degradation is approximately only 0.2 dB at a BER of $2 * 10^{-6}$. Hence, the implementation will use branch metrics rounded to 9 bits and consequently a path metric word-length of 11 bits.

4.4.5 State Metric Normalisation

Viterbi state metrics grow unbounded with time and since an implementation must use a numeric representation with finite range, a strategy for coping with this unbounded growth is required. Several techniques have been developed for scaling or normalising the metrics so that overflow is avoided [24]. Most of these techniques cost FPGA resources and add latency inside the critical Viterbi iteration loop.

Hekstra proposes an alternative technique which relies on the overflow behaviour of 2's complement arithmetic and the fact that it is the difference be-

tween path metrics rather than their absolute value that is important for survivor path selection. Since 2's complement arithmetic is the norm for implementation, Hekstra's method avoids the need for any extra hardware resources [25].

Although Hekstra's method has been applied to Viterbi decoders for use in forward error correction applications [2], it has not yet been applied to CPM Viterbi detectors. Hekstra's method requires the difference between state metrics to be bounded. It can be proven the metrics are bounded by considering the 2 dimensional state transition matrix T. Source and destination states are represented by each dimension, and a 1 entry indicates the state transition is possible, and a 0 entry indicates the transition is not possible. If T can be raised to a power n, such that all entries are strictly positive, then the metrics are bounded[25].

For the h=1/4, L=3, M=4 state transition matrix, n could not be found. Indeed, simulation shows the state metric range growing without bound over time. However, a direct consequence of the modulation index having an even valued p=8, (h=2k/p) is that every state transition occurs from a state with even valued phase state to odd valued phase state and vice versa. Odd to odd or even to even state transitions are not possible in the state transition matrix T. This means the trellis comprises two sub-trellises that do not share any connections and hence the difference in metrics between sub-trellises is unbounded.

If T is defined for the odd or even sub-trellis only, then n is found to be 3, and metrics within a sub-trellis are bounded. (See Appendix E.4.1 for the Matlab source code used to find n). Once a starting phase state is defined, all further transitions are within a single sub-trellis only and so Hekstra's method is applicable. The implication for the phase recovery algorithm is that phase slips between the two sub-trellises must not be allowed.

Since only one sub-trellis is relevant, it is suggested that the other sub-trellis Viterbi calculations are redundant and the overall Viterbi processing requirements can be halved. This has not been investigated in the FPGA implementation of Chapter 6.

To avoid overflow, Hekstra claims the path metric word-length B_{pm} must satisfy equation (4.1). B is an upper bound on the absolute values of branch metrics and is given by equation (4.2). Conservatively $m = 2n$ [25]. B_{bm} is the branch metric word-length.

$$B_{pm} \geq \log_2((m+2)B+1)+1 \tag{4.1}$$

$$B = 2^{B_{bm}-1} \tag{4.2}$$

For a branch metric word-length of 9 bits and n=3, this equation suggests the path metric word-length must be at least 11 bits. The simulation of section 4.4.4 uses 11 bit path metric word-lengths. That simulation was repeated with a path metric word-length of 10 bits with the same result. However, moving to 9 bits causes a very high error rate.

4.5 Survivor Path History Depth

The Viterbi algorithm provides maximum-likelihood estimates of the transmitted symbol when the Viterbi path history has infinite depth. In practice, near

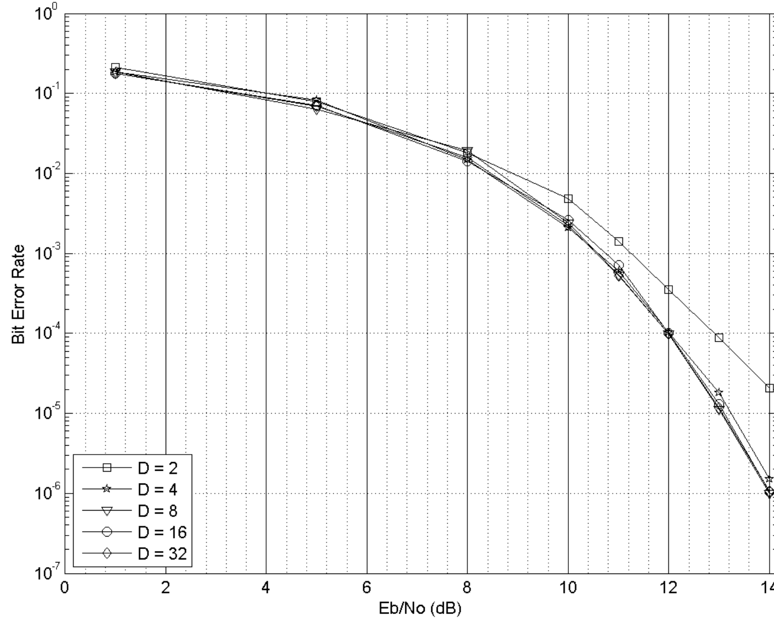

Figure 4.6: Effect of Survivor Path History Depth on Detection Efficiency

optimal performance is achievable with practical path memory sizes. For convolutional decoder applications, a well known rule of thumb is to size the path history to be 5 times the convolutional code constraint length.

For CPM Viterbi receivers, Anderson et al [6] suggest that the path history depth should be at least as large as the observation interval required to achieve the upper bound on the CPM code's minimum squared Euclidean distance (d^2min). [6, Figure 3.31, pg 103] shows that for a quaternary, 3RC, $h < 0.5$ CPM configuration, the observation interval must be at least 12 symbols long to achieve the performance predicted by d^2min. This implies a minimum path history depth of 12 symbols for our h=1/4, L=3RC, M=4 CPM receiver.

Simulation shows that a path history depth of only 8 symbols does not degrade detection efficiency. Figure 4.6 shows simulated receiver bit error rate vs $\frac{E_b}{N_o}$ for a variety of Viterbi path history depths. The low-pass ACI reject filter is not present, all arithmetic is 64-bit floating point and there are 8 samples per symbol. A depth of only 2 symbols degrades the performance significantly and a depth of 4 symbols only slightly.

Although the Viterbi path history implementation is beyond the scope of this thesis, a traceback architecture using the memory management technique described in [46] is likely to be cost effective. For quaternary CPM symbols, this technique stores 2 bits per state per Viterbi iteration. An h=1/4, L=3, M=4 CPM configuration has a 128 state trellis, and for a path history depth of 8 iterations the path history memory required is 2048 bits. This easily fits within a single 18 kbit block RAM within the FPGA. The FPGA targeted in this thesis has 84

such block RAMs so the memory cost is low.

Viterbi path memory bandwidth requirements are high when tracing back once every symbol. The bandwidth requirements can be reduced significantly by tracing back only once every several symbols using the technique described by Chang [2].

4.6 Conclusion

Sampling rate, fixed point word-length and survivor path history depth have a critical effect on the CPM receiver implementation cost and detection efficiency performance. By using Matlab fixed point data types and rewriting specific portions of the CPM receiver floating point model used in the previous chapter, simulations have been performed to find an acceptable tradeoff between cost and performance for these parameters.

Simulation results show that compared to the floating point model, a fixed point model with 2 samples per symbol and 16 symbol deep path history, degrades detection efficiency by only 0.2 dB at a BER of $2 * 10^{-6}$. To achieve this small degradation, the in-phase and quadrature received signal inputs must be quantised to at least 7 bits, the branch metric filter bank coefficients at least 7 bits and the branch metrics are rounded down to at least 9 bits. The path metric word-length is 11 bits.

Path metrics grow without bound over time. This is commonly overcome by periodically rescaling or normalising the metrics. A clever technique published by Hekstra avoids rescaling entirely by exploiting the fact that path metric differences are bounded and it is the difference between metrics, not their absolute value that are important in a Viterbi decoder. This technique is applied to a CPM receiver; as far as the author knows, this has not been published before. Hekstra's calculations show that a path metric word-length of 11 bits is sufficient to avoid incorrect survivor path selections and this has been confirmed in simulation.

Chapter 5

CPM Branch Metric Filter Bank FPGA Implementation

5.1 Introduction

The previous chapter introduced fixed point arithmetic, finite Viterbi path memory size and a practical sampling rate. It was shown that these approximations cause a detection efficiency degradation of only 0.2 dB. This still meets the radio performance requirements and provides for a significantly lower complexity and lower cost receiver than an implementation based on a high sample rate and large word-length floating point arithmetic.

The application considered in this thesis has a stringent cost requirement, and to prove this requirement can be met an implementation is required. The two most computationally expensive parts of the receiver are the Viterbi branch metric filter bank and Viterbi path metrics processing unit. This section of the thesis considers the branch metrics filter bank implementation.

A key contribution of this thesis is the proposed application of a well known distributed arithmetic technique to the implementation of a coherent CPM receiver branch metrics filter bank. The implementation is generalised to support any reasonable h, L, M, quantised received signal word-length and filter coefficient word-length.

The technique is proven with a VHDL implementation for the CPM configuration found in Chapter 3 and further refined in Chapter 4. That is h=1/4, L=3, M=4, raised cosine phase pulse. Synthesis results are presented which show FPGA resource usage meets the cost requirements specified for this application. It would seem this technique is more than 6 times cheaper than a conventional technique using the FPGA's dedicated multipliers.

Data throughput is verified by performing static timing analysis on the placed and routed design. Timing closure is achieved for a 215.6 MHz clock allowing a symbol rate of up to 30.9 MSymbols/s, meeting the application requirement of 27 MSymbols/s. Correct functional performance is demonstrated with a VHDL simulation producing results that match the Matlab fixed point model precisely (bit for bit).

Distributed arithmetic uses the FPGA logic fabric very efficiently by performing the filter calculations in a bit-serial fashion. For the application con-

sidered in this thesis, it is most FPGA hardware cost efficient for symbol rates around 30 MSymbols/s. For 2 samples per symbol this technique applies most efficiently to the 4 input LUT architecture which is used in the current generation of low cost FPGAs from the dominant FPGA market players Xilinx and Altera.

A significant advantage of this technique is that it allows symbol rate to be traded off against implementation size (and hence cost), although that has not been demonstrated here. The proposed implementation supports any h, L, M, input or coefficient word-length.

The main disadvantage of this bit-serial processing is that it introduces a symbol period of delay. It is likely this branch metric processing is inside the carrier phase recovery loop and so this added latency will degrade phase recovery performance. This is the performance cost incurred in order to meet the demanding cost requirements.

Another challenge with this technique is in meeting timing due to the high clock rates caused by fully bit-serial processing. This was overcome by designing with low-level FPGA resource primitives directly and algorithmically floorplanning these elements; this "manual" approach beats the automatic tools algorithms by 10's of MHz for this design.

5.2 Computation Complexity

The CPM receiver literature typically expresses branch metric complexity as the number of matched filters. The phase rotation computations are ignored [28] [32] [34]. In this thesis we focus on implementation and so a more appropriate measure of complexity is real valued multiplications (and adds) per decoded symbol. By taking account of the symbol rate an even more useful metric is multiplications per second and additions per second. These metrics do not account for filter coefficient word-length nor received signal sample word-length. We account for that in a later section by expressing complexity directly in terms of FPGA resources.

The maximum-likelihood receiver calculates branch metrics according to Equation (5.1), previously presented in section 2.4.

$$Z_n(\alpha_n, \theta_n) = \Re\{e^{-j\theta_n} \int_{nT}^{(n+1)T} \tilde{r}(t) e^{-j\theta(t,\alpha_n)} dt\} \tag{5.1}$$

Express in terms of real operations only by substituting $\tilde{r}(t) = r_I(t) + jr_Q(t)$ into Equation (5.1) we get Equation (5.2) [6].

$$\begin{aligned} Z_n(\alpha_n, \theta_n) = \cos(\theta_n)[\int_{nT}^{(n+1)T} r_I(t)\cos(\theta(t,\alpha_n))dt + \int_{nT}^{(n+1)T} r_Q(t)\sin(\theta(t,\alpha_n))dt] \\ + \sin(\theta_n)[\int_{nT}^{(n+1)T} r_Q(t)\cos(\theta(t,\alpha_n))dt - \int_{nT}^{(n+1)T} r_I(t)\sin(\theta(t,\alpha_n))dt] \end{aligned} \tag{5.2}$$

Each integral represents a matched filter. $\cos(\theta(t,\alpha_n))$ and $\sin(\theta(t,\alpha_n))$ are the matched filter coefficients where α_n is an L length symbol sequence. There are M^L such possible symbol sequences so Equation 5.2 represents $4M^L$

h	p	L	M	N	R (1/T)	Branch Metric multiplies/s	Branch Metric adds/s
1/4	8	3	4	2	27 MSamples/s	6.9 GMults/s	3.5 GAdds/s

Table 5.1: CPM Branch Metric Filter Bank Complexity

matched filters. However, every α_n sequence has a corresponding negative sequence and using the trigonometric identities in Equation (5.3), the number of matched filters is halved. The maximum-likelihood branch metric filter bank is $2M^L$ matched filters [6].

$$\begin{aligned} \cos(A) &= \cos(-A) \\ \sin(A) &= -\sin(-A) \end{aligned} \tag{5.3}$$

$\cos(\theta(t, \alpha_n))$ and $\sin(\theta(t, \alpha_n))$ are precomputed filter coefficients. Each integral (filter) requires N real multiplications and N-1 additions, where N is the number of samples per symbol. For a symbol rate of $R = 1/T$ symbols/s where T is the symbol period, Equation (5.4) describes the matched filter multiplication complexity and Equation (5.5) is the matched filter add complexity. Table 5.1 shows this complexity for the CPM configuration selected in Chapter 3.

$$MatchedFilters_{multiplies/s} = R2M^L N \tag{5.4}$$

$$MatchedFilters_{adds/s} = R2M^L(N-1) \tag{5.5}$$

To achieve these high computational rates, a traditional solution might use the embedded multipliers now present in the fabric of modern low-cost FPGAs. The XC3S1800A-DSP Spartan 3A-DSP FPGA is the largest Xilinx FPGA that meets the cost requirements for this application. This part has 84 multipliers in addition to the general purpose LUT and flip-flop fabric. These multipliers can be clocked at a maximum of 250 MHz which provides for a processing capability 21 giga-multiplies/s [44].

At a first analysis it appears that the embedded multipliers have the multiplication capacity to support the matched filter processing. However, Equation (5.4) and Table 5.1 ignore the phase rotation multiplications. Ultimately there are $pM^L/2$ (256) unique branch metrics to be calculated yet there are only 84 multipliers available. With 128 matched filters, the processing might be arranged such that each embedded multiplier performs the processing for 2 matched filters. This requires the two results per multiplier to be stored adding further complexity. A solution using the majority (76%) of the available multipliers is undesirable since other parts of the CPM receiver also require the use of the multipliers. For example, front end adjacent channel filtering and the adaptive equaliser.

In some respects the Xilinx datasheet claim of 21 giga-multiplies/s overstates the devices capability. Firstly, every clock cycle must be used to generate a result. This requires extra logic to sequence operands and store results. And secondly, the multipliers must be clocked at their maximum rate of 250 MHz and the design must meet timing. Getting data into and out of the multipliers from the general logic fabric at this rate is a design challenge. Furthermore, it is unlikely other parts of the FPGA design would operate at this frequency and

so extra logic and complexity is required to transfer the data between this high speed 250 MHz clock domain and the rest of the design.

The phase state rotation complexity can be reduced and almost eliminated for some specific values of h. For example, with h=1/4 there are 8 phase states, yet $\cos(\theta(t, \alpha_n))$ and $\sin(\theta(t, \alpha_n))$ take on values of 0,1 or +- 0.707 only. Multiplication by 0 and 1 is trivial so the multiplication by 0.707 is the only one required per matched filter output. This thesis proposes an implementation that ignores this optimisation and so it is generalised for any h, L, and M.

5.3 Applying Distributed Arithmetic to a CPM Filter Bank

The microwave radio application considered in this thesis requires a symbol rate of 27 MSymbols/s. This is almost an order of magnitude less than clock rates achievable in current generation low cost FPGAs. For example, the Spartan 3A-DSP FPGA multipliers support operation at up to 250 MHz[44] and as shown in section 5.4.3, clock rates above 200 MHz are achievable in the FPGA logic fabric. The implementation proposed by this thesis exploits this by carrying out the processing in a bit-serial fashion using a well known technique called distributed arithmetic.

Distributed arithmetic (DA) filters are a logic resource efficient way of implementing a finite impulse response (FIR) filter [47] [48]. The signal to be filtered is passed to the filter in a bit-serial fashion, least significant (LSB) first. This requires a system clock at a rate several times that of the incoming sample rate. In this way, a higher system clock is traded off against lower hardware costs. In our application we have a symbol rate of 27 MSymbols/s and Spartan 3ADSP clock rates of more than 200 MHz are achievable, making DA filters attractive for this application.

A distributed arithmetic filter implements the standard FIR filter equation precisely. Equation (5.6) describes a M tap FIR filter with filter coefficients $h(k)$ and input signal $x(m)$ where m is the discrete time variable [47].

$$y(m) = \sum_{k=0}^{M-1} x(m-k)h(k) \tag{5.6}$$

The key to applying this to a CPM branch metric filter bank is to bring the phase rotations inside the integral and combine the in-phase and quadrature matched filters into a single filter. This is shown starting with Equation (5.7) and ending in Equation (5.8) where the branch metrics computations are expressed in a sum of products form that is equivalent to Equation (5.6).

$$\begin{aligned} Z_n(\alpha_n, \theta_n) &= \Re\{e^{-j\theta_n} \int_{nT}^{(n+1)T} \tilde{r}(t) e^{-j\theta(t,\alpha_n)} dt\} \\ &= \Re\{\int_{nT}^{(n+1)T} \tilde{r}(t) e^{-j(\theta(t,\alpha_n)+\theta_n)} dt\} \\ &= \int_{nT}^{(n+1)T} r_I(t)\cos[\theta(t,\alpha_n)+\theta_n] + r_Q(t)\sin[\theta(t,\alpha_n)+\theta_n] dt \end{aligned} \tag{5.7}$$

We now move to discrete time notation where m is the discrete time sample index, $r_I(m)$ and $r_Q(m)$ are discrete time versions of $r_I(t)$ and $r_Q(t)$ sampled evenly throughout a symbol period. N is the number of samples per symbol. Perfect timing recovery is assumed which means that the I and Q samples always occur at the same place relative to symbol boundaries.

$hc(m)$ and $hs(m)$ represent discrete time versions of the matched filter coefficients $\cos[\theta(t,\alpha_n)+\theta_n]$ and $\sin[\theta(t,\alpha_n)+\theta_n]$.

$$Z_n(\alpha_n,\theta_n)=\sum_{m=0}^{N} r_I(m)hc(m)+\sum_{m=0}^{N} r_Q(m)hs(m) \tag{5.8}$$

And by letting $x(m)=\{r_I(0),r_I(1),\ldots,r_I(N-1),r_Q(0),r_Q(1),\ldots,r_Q(N-1)\}$, $h(m)=\{hc(0),hc(1),\ldots,hc(N-1),hs(0),hs(1),\ldots,hs(N-1)\}$ and $M=2N$, we see that Equation (5.8) is equivalent to Equation (5.6) and therefore a single distributed arithmetic filter calculates a single branch metric.

A distributed arithmetic filter requires the filter coefficients to be constant [47] [1]. $\theta(t,\mathbf{a})$ is given by Equation (5.9) and for a fixed h and a specified fixed phase pulse $q(t)$ then the coefficients are only a function of $\mathbf{a}$ and $\mathbf{a}$ is a constant for each matched filter. A practical CPM microwave radio link has a constant CPM configuration (h, L, M, phase pulse shape).

$$\theta(t,\mathbf{a})=2\pi h\sum_{i=n-L+1}^{n}\alpha_i q(t-iT) \tag{5.9}$$

5.3.1 Phase State Symmetry

A CPM receiver requires pM^L branch metrics to be calculated and fed into the Viterbi trellis decoder. Since each branch metric is implemented with a separate DA filter the receiver has a branch metric filter bank containing pM^L filters operating in parallel. For h=1/4, M=4, L=3 this is 512 filters. The number of filters can be halved by exploiting symmetry in the phase state.

As previously defined in Chapter 2, Equation (5.10) defines the phase state and the number of phase states p is related to h as in Equation (5.11). When p is even then $\theta_n+\pi \mod 2\pi$ is also a phase state. And because $\cos(A+\pi)=-\cos(A)$ and $\sin(A+\pi)=-\sin(A)$, Equation (5.7) only needs to be calculated for $\theta_n<\pi$. The remaining branch metrics for $\theta_n>=\pi$ are simply the negative of the branch metrics calculated for $\theta_n<\pi$.

$$\theta_n=\frac{2\pi i}{p}, i\in\{0,1,2,\ldots,p-1\} \tag{5.10}$$

$$h=\frac{2k}{p}, k,p\in integers \tag{5.11}$$

This negation operation can be rolled into the Viterbi add-compare-select unit without any additional LUT or flip-flop cost. However, there is still an FPGA routing cost because each branch metric filter output must now route to 2 add-compare-select units. The Viterbi trellis add-compare-select implementation is discussed in detail in Chapter 6.

[1] This is not quite true. By replacing the distributed arithmetic LUT with a Xilinx SRL shift register primitive, new filter coefficients can be loaded serially. This can be used to build adaptive distributed arithmetic filters. This is not relevant to the CPM receiver considered in this thesis.

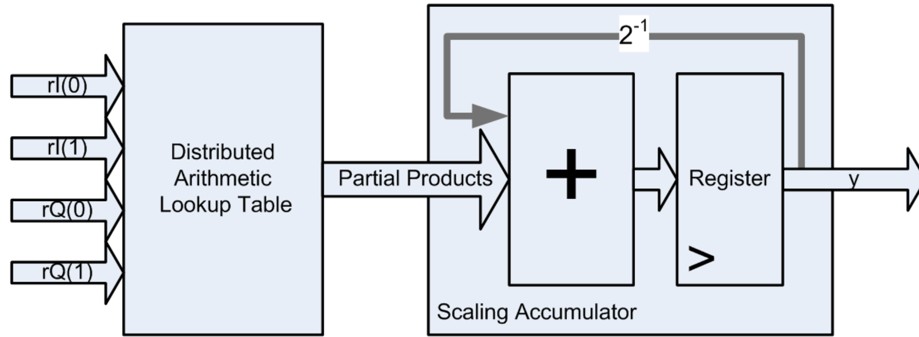

Figure 5.1: 4-Tap Distributed Arithmetic FIR Filter Block Diagram

5.4 4-Tap Distributed Arithmetic FIR Filter Implementation

Chapter 4 showed that a sample rate of 2 samples per symbol (N=2) does not degrade receiver detection efficiency. This means a 4-tap DA filter is required to calculate each branch metric. The Xilinx Spartan 3A-DSP FPGA provides 4 input LUTs [44] which means the 4-tap DA filter maps very efficiently to this FPGA's logic fabric [47]. [2]

The basic structure of a 4-tap DA filter is shown in Figure 5.1. There are two parts to it. Firstly, a DA lookup table (DALUT) which takes in 1 bit from each of 4 input samples every clock cycle. This lookup table stores all 2^4 combinations of partial products given by the 4 filter coefficients. The 4 input sample bits are concatenated to form the lookup table address. These partial products are pre-computed and loaded into the lookup table during FPGA configuration. Secondly, as each new bit from the input samples is clocked through, the DALUT partial products are added with a scaling accumulator. After the input sample word-length (B_{IQ}) number of clock cycles, a new filter output is valid.

5.4.1 Throughput Requirements

The maximum throughput (symbol rate) is determined by the maximum frequency of operation of the design. Define this frequency as f_{max} . This parameter is determined by running the placed and routed design through Xilinx's static timing analysis tool called Trace. Trace uses a detailed timing model of the FPGA to find the slowest static path between registers in the design and hence the maximum operating frequency. Since every FPGA manufactured by Xilinx is factory tested against this timing model, the design is guaranteed to function in the real world across all corners of process, voltage and temperature (PVT) [49].

For a fully bit-serial DA filter the symbol rate throughput R is given by

[2] 6 input LUTs are used in Xilinx's premium FPGA brand called Virtex-5. These FPGA's offer higher performance and density but at significantly higher cost. Spartan-6 and Virtex-6 are the next generation of Xilinx FPGA's and they use a 6 input LUT. They are not currently released for production.

R (Symbol Rate MSymbols/s)	f_{max}	B_{IQ}
27 MSymbols/s	189 MHz	7

Table 5.2: Distributed Arithmetic Filter Fmax Requirement

Equation (5.12). Table 5.2 shows that for our application throughput requirement of 27 MSymbols/s and a filter input sample word-length of 7 bits, the FPGA design must achieve an f_{max} of 189 MHz. This is a significant challenge when targeting a low-cost FPGA such as a Xilinx Spartan3A-DSP FPGA.

$$R = \frac{f_{max}}{B_{IQ}} \tag{5.12}$$

5.4.2 Efficient Mapping of a DA Filter into FPGA Hardware

Define B_C as the filter coefficient word-length in bits and B_{IQ} as the in-phase and quadrature sample word-lengths.

5.4.2.1 DALUT

The DALUT has 4 address bits so has 16 locations. Each location stores the partial product from accumulating 4 coefficients. Each time two B bit numbers are added, the resulting sum must be B+1 bits to guarantee overflow is avoided. When accumulating 4 coefficients then the resulting sum word-length must be $B_C + 2$ to avoid overflow. Therefore, the DALUT size is 16x($B_C + 2$).

In Xilinx Spartan 3-ADSP FPGAs there are 3 types of memory [44]:

- Block Ram (BRAM) - Large 18kbit blocks of dual ported ram which can be used in configurations from 16k x 1 [3] through to 512x36.
- Flip Flops - Each FPGA logic cell contains a single flip-flop which can store 1 bit of data.
- LUT RAM - Each FPGA logic cell contains a 4 input LUT. This LUT can be configured as a 16x1 ROM and half the available LUTs can be used as 16x1 RAMs (aka "distributed ram").

For our application, the LUT RAM is by far the most efficient way of implementing the DALUT. The LUT RAM stores 16 times more bits per logic cell than compared to a flip-flop. The block ram capacity of 18 kbits is many times larger than required, and with only 84 BRAMs available on the FPGA, there are simply not enough BRAMs to implement 256 filters. The most logic efficient solution is to use the LUT RAM.

Each LUT is used as 16x1 ROM so a single filter DALUT requires $B_C + 2$ LUTs. It is also worth noting that the actual implementation also registers the DALUT output to improve the maximum frequency of operation. This register essentially comes for "free" since every logic cell contains 1 LUT and 1 flip-flop. However the register does add one extra clock cycle of latency.

[3] Block ram parity bits are not available in the 16kx1 configuration hence only 16k addressable locations are available.

B_{IQ}	B_C	Logic Cells per filter	h	L	M	# Filters	Total Logic Cells	Total Logic Cells (% of available) [5]
7	7	25	$\frac{1}{4}$	3	4	256	6400	19.2%

Table 5.3: Estimated Branch Metric Filter Bank FPGA Resource Use

5.4.2.2 Scaling Accumulator

The scaling accumulator adds the current partial product to a right shifted version of the previous clock cycles accumulation. The partial product is always aligned to the most significant bit of the scaling accumulator result. In order to avoid overflow, the scaling accumulator word-length is the DALUT word-length plus 1 bit growth every clock cycle. i.e. $B_C + 2 + B_{IQ}$ since there are B_{IQ} clock cycles required per input sample.

The scaling accumulator comprises an adder followed by a register. The right shift does not consume any logic resources; a bit right shift is wired into the accumulator feedback path. It is well known that an adder consumes 1 logic cell per bit, and the register uses the flip-flop in the same logic cell. This means the scaling accumulator consumes a maximum of $B_C + 2 + B_{IQ}$ bits.

5.4.2.3 FPGA Resource Use Summary

The total resources used by a single 4-tap CPM DA filter are given in Equation (5.13). The complete CPM receiver branch metric filter bank requires many of these filters operating in parallel. Table 5.3 shows the calculated resource usage for the specific CPM configuration chosen in Chapter 3 and further refined in Chapter 4. For a single 4-tap DA filter with 7 bit coefficients and 7 bit samples, the FPGA implementation requires 25 logic cells per filter and is guaranteed to avoid overflow and without any loss of precision.

For h=1/4, L=3, M=4 then the complete filter bank requires 256 filters which is estimated to consume 6400 logic cells. This is 19% of the available FPGA logic cell resources which meets the low-cost requirements for this application. Confirmation of throughput, static timing analysis and actual resource use is demonstrated in section 5.4.3. [4]

$$DAFIR_{LCs} = (B_C + 2) + (B_C + 2 + B_{IQ}) \tag{5.13}$$

5.4.2.4 Additional Resource Use Required to Meet Timing

Given the high f_{max} requirement of 189 MHz (see section 5.4.1), signal routing delays must be kept to a minimum. Signals with high fanout have higher routing delays because the average distance the signal travels is further; the signal passes through more switch boxes within the FPGA's routing fabric; and the

[4] Xilinx datasheets specify "equivalent logic cells" which Xilinx calculate by applying an arbitraty factor to the actual number of logic cells in the device. One assumes this is done for marketing purposes. This thesis always uses actual logic cells. For SC3S1800A-DSP the Xilinx datasheet reports 37440 "equivalent logic cells" and reports 4160 actual complex logic blocks(CLB). Each CLB contains 8 logic cells (LCs) giving the part a total of 33280 actual logic cells or in terms of slices this is 16640 slices.

flip-flops	LUTs	slices	BRAMs	DSP48s	best case achievable period	f_{max}
7460	4896	3863	0	0	4.639 ns	215.6 MHz

Table 5.4: Branch Metric Filter Bank Implementation Results

capacitive loading on the high fanout net is higher. The filter bank signal input $x(m)$ routes to all 256 filters, and each filter DALUT is 9 LCs so the fanout is 256x9=2304. This is a very high fanout and experiments show that f_{max} can fall well below 100 MHz, reducing the design throughput well below the required 27 MSymbol/s. The solution is straight forward. The register storing $x(m)$ is duplicated many times which reduces the fanout by the same ratio but increases the design cost. It was found that a good balance between cost and performance is achieved when $x(m)$ register is duplicated 128 times. Pairs of filters are driven by their own dedicated $x(m)$ register thus reducing the fanout to 18 LCs. The increase in LC usage is 4x (256/2) = 512 LCs which is an 8% increase in the filter bank size.

The filter bank contains two more high fanout signals; both are scaling accumulator controls [6]. The first control signal indicates the input signal LSB which zeros the scaling accumulator. The second control signal indicates the input signal MSB which configures the scaling accumulator to perform a subtraction. These two signals were register duplicated on a per filter basis, so the increase in logic cell usage is 2*256=512 LCs.

The total increase in design size is thus 1024 LCs bringing the total design estimate to 7424 LCs.

5.4.3 Implementation Results

The DA filter bank is written in VHDL and implemented using the Xilinx ISE tool suite. Appendices D.1 and D.4 contain the VHDL source and Appendix G lists the implementation tool versions.

A period timing constraint of 216 MHz was applied to the place and route (PAR) tools. The default synthesis and PAR options were modified slightly: "keep hierarchy" is set to yes to allow relative location (RLOC) constraints to propagate correctly through the hierarchy, and "add I/O buffers" is set to false since this design does not connect to FPGA device input or output pins [7].

The FPGA resource use and static timing analysis results are summarised in Table 5.4. Appendix F.1 contains more detailed output from the mapper and PAR tools.

[6]Actually there is a third extremely high fanout signal - the clock. FPGAs have dedicated global clock routing networks that have very low skew. Timing analysis results show that clock skew can consume more than 300 ps from the timing budget. But there is relatively little that can be done about it. The filter bank could be split in half, thus halving the clock fanout. But this introduces a 2nd clock into the design which raises the specter of crossing clock domains at some point further downstream in the data path.

[7]The ISE tools mapper strips all logic not connected to input or output pins. This is avoided by manually instantiating IBUFs on the branch metric filter bank inputs and applying the mapper save attribute to the branch metric filter bank outputs

Since every logic cell (LC) contains a LUT and flip-flop, the actual flip-flop resource usage of 7460 flip-flops closely follows the theoretical estimate of 7424 LCs. The small difference can be explained by two functionalities not included in the resource use estimate: a simple state machine and registers that serialise incoming parallel data ($x(m)$) for the bit-serial DA filter bank.

The slice usage result is more difficult to interpret. Each spartan 3A-DSP slice contains two LCs and each LC contains a LUT and a flip-flop. If a single LUT or flip-flop is used then the tools still report slice usage of 1. This explains why 3863 slices x 2 = 7726 LCs is larger than the actual number of flip-flops or LUTs used in the design.

The FPGA resource usage of 3863 slices is 23% of the available FPGA slices thus comfortably meeting the applications cost requirement. The remaining 77% of slices and 100% of block ram and embedded multipliers are available for the Viterbi path metric calculations described in Chapter 6. This design meets timing at 215.6 MHz thus supporting a data throughput of 215.6 / 7 = 30.9 MSymbols/s. This is 14.1% higher throughput than the application's requirement of 27 MSymbols/s, giving a considerable margin. In practice an extra timing allowance for clock jitter is required depending on the quality of the clock source. Also, the branch metric results are not routed and attached to the Viterbi trellis decoder. Altough this routing will be kept short and local by placing each branch metric filter close to its associated Viterbi add-compare-select unit, the extra routing resources consumed will degrade timing.

5.4.4 Functional Verification

The VHDL implementation is verified by executing the design in a VHDL simulator. Test vectors are generated by using the Matlab fixed point model to write a VHDL package containing input and output from the Matlab model. The VHDL testbench applies I and Q samples to the filter bank and then checks the filter bank output branch metrics against the expected outputs also stored in the VHDL package. Any errors are counted and reported. This test strategy is justified and described in more detail in Appendix A.

The testbench VHDL source code is in Appendix D.1.5 and the test vectors package in Appendix D.1.6. 100 random symbols worth of I and Q samples were applied to the filter bank. The testbench reports 0 errors. This simulation demonstrates that the VHDL branch metrics precisely match the Matlab fixed point model branch metrics.

5.5 Comparison: Distributed Arithmetic vs Embedded FPGA Multipliers

For input sample word-lengths significantly less than 18 bits, a distributed arithmetic approach achieves a much higher throughput than is achievable with the FPGAs embedded multipliers. The DA filter bank presented in this chapter performs the equivalent of 31.6 giga-multiplies/s while using slightly less than a quarter of the available FPGA slices. The absolute maximum throughput available using all the FPGA's embedded multipliers is 21 giga-multiplies/s

[44]. Using a quarter of the embedded multipliers (21) provides for only 21 x 250 MHz = 5.3 giga-multiplies/s, almost 6 times less than the DA approach.

The DA filter multiplication complexity is calculated using the filter bank implemented in section 5.4.3. That is 256 filters running at a symbol rate of 30.9 MSymbols/s. Each filter performs 4 multiplications per symbol. I.e. 4 x 256 x 30.9 = 31.6 giga-multiplies/s.

On the other hand, the embedded multipliers provide for significantly higher operand word-lengths. Up to 18 bits compared with only 7 bit for the DA filter bank presented in this thesis. As the DA filter input sample and coefficient word-lengths increase, so does the FPGA logic resource use. A longer word-length also implies a longer carry chain for the scaling accumulator which may degrade f_{max} and throughput as well. Nevertheless, it was shown in Chapter 4 that input sample word-lengths greater than 7 bits provide minimal receiver detection efficiency improvements. And this is the elegance of distributed arithmetic designs; FPGA resource use is precisely matched to the applications requirements.

5.6 Conclusion

This chapter showed that distributed arithmetic can be applied to an FPGA implementation of the branch metrics component of a CPM receiver. Although DA is a well known technique, its application to a CPM receiver has not previously been published. The key to making this technique work for CPM is to combine the branch metric matched filter and phase rotation into a single filter. Although this increases the total number of filters required, each filter is implemented bit-serially using distributed arithmetic. Each filter consumes very few FPGA logic resources; in this application each filter consumes 25 logic cells.

The complete filter bank of 256 filters occupies 23% of the low cost target FPGA. The design meets timing at 215.6 MHz, meeting the minimum requirement of 189 MHz and thus supporting a symbol rate in excess of 27 MSymbols/s. This provides a margin of 14%. The main drawback of this bit-serial processing is the added 1 symbol latency. Since the branch metric filter bank is likely to be inside a phase recovery loop, this extra processing latency degrades phase recovery performance. This is the price to be paid for a low-cost implementation meeting the strict cost requirements for this microwave radio application.

Chapter 6

CPM Path Metric FPGA Implementation

6.1 Introduction

The previous chapter presented an FPGA implementation for the branch metric unit which used 23% of the available logic resources and requires a 189 MHz system clock to meet the 27 MSymbol/s (54 Mbit/s) throughput requirement. As shown in Figure 6.1, the next stage of processing in this CPM receiver is the path metric processing unit.

Each symbol interval this unit calculates 4 path metrics inbound to each of the 128 trellis states and selects one surviving path per state to be the output state metric. Two decision bits represent the selected path and these are forwarded onto a survivor management unit which tracks the trellis path history for each state. The survivor management unit traces back the path with the largest metric to a finite depth and outputs the symbol at the tail of this path.

For such a large trellis, and moderate throughput requirement, a low-cost FPGA implementation is a challenge. The path metric computations alone require 13.8 giga-adds per second. Furthermore, this add-compare-select (ACS) processing is inside the Viterbi iteration loop so each added pipeline stage re-

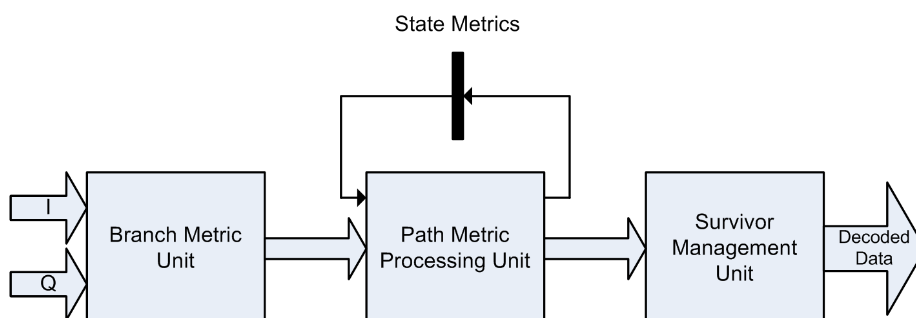

Figure 6.1: CPM Viterbi Detection

duces the amount of time available to complete the required processing. With a system clock frequency of at least 189 MHz, deep pipelining is required to pass static timing analysis, leaving limited opportunities to time-share FPGA resources.

Although the open literature contains many examples of Viterbi decoders targeting ASIC implementations for convolutional decoder applications [16] [2] [14], there are none that present FPGA implementations of CPM Viterbi detection to any significant level of detail.

The standard large trellis, high-throughput architecture is a state parallel solution in which every trellis state has dedicated hardware resources performing the ACS operations. For large trellises the state metric routing is complex [16]. The solution presented here is to partition the ACS processing into radix-4 units, each of which processes 4 states. The advantage is that the state metric read routing is short since it is local to a single radix-4 unit. The state metric write routing is still lengthy; this is mitigated by devoting 1 complete pipeline cycle to the state metric write routing.

The proposed VHDL implementation is deeply pipelined to meet the 189 MHz system clock requirement. The ACS unit uses 6 out of the 7 available clock cycles per Viterbi iteration to produce 1 survivor state metric. This unused cycle is exploited by processing two output state metrics per ACS unit, thus requiring only 2 ACS units per radix-4 unit. The FPGA resource cost is halved.

6.2 Viterbi Algorithm: Updating State Metrics

The background material presented in Chapter 2 explained that the Viterbi algorithm is an iterative approach to find the most likely transmitted symbol sequence. Equation (6.1) shows how the nth Viterbi iteration has state metrics $J_n(\alpha)$ which are calculated from the previous iteration state metrics J_{n-1} and branch metrics $Z_n(\alpha)$ [6]. This Viterbi iteration is repeated once per received symbol duration.

$$J_n(\alpha) = J_{n-1}(\alpha) + Z_n(\alpha) \tag{6.1}$$

For an h=1/4, L=3, M=4 CPM configuration there are 128 states and hence 128 state metrics. Each state is uniquely identified by a combination of a phase state and correlative state as defined in Equation (6.2). The phase state is defined in Equation (6.3) where $p = 8$ for a modulation index of $h = 1/4$.

$$\sigma_n = (\theta_n, \alpha_{n-1}, \alpha_{n-2}) \tag{6.2}$$

$$\theta_n = \frac{2\pi i}{p}, i \in \{0, 1, 2, \ldots, p-1\} \tag{6.3}$$

For quaternary symbols (M=4) each state has 4 possible inbound paths. The path metrics are calculated for all 4 paths and the path with the highest metric is selected to be the survivor path, also called the output state metric. In the next Viterbi iteration, this output state metric becomes an input state metric somewhere else in the Viterbi trellis. Figure 6.2 shows this per state processing in diagram form.

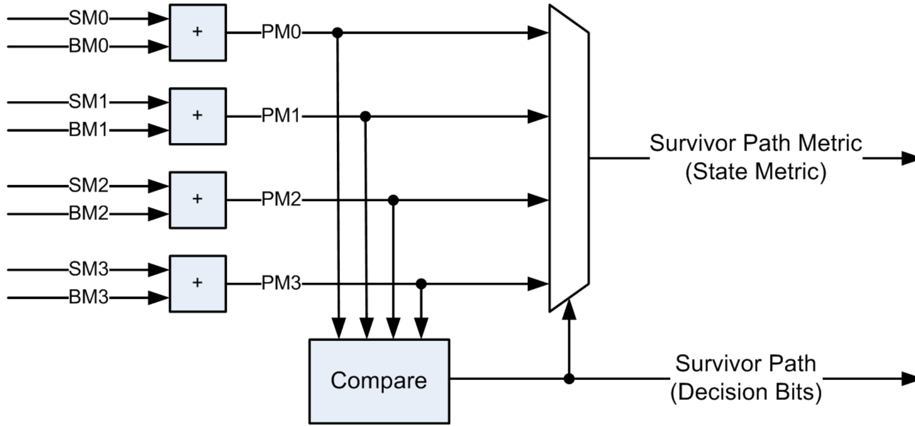

Figure 6.2: Add-Compare-Select Processing Required per State

6.3 Proposed Solution

6.3.1 State-Parallel Radix-4 Decomposition

The standard large trellis, high-throughput architecture is a state parallel solution in which every trellis state has dedicated hardware resources performing the ACS operations. For large trellises the state metric routing is complex [16]. The solution presented here is to partition the ACS processing into radix-4 units, each of which processes 4 states. The advantage being that the state metric read routing is short since it is local to a single radix-4 unit. The state metric write routing is still lengthy; this is mitigated by devoting 1 complete pipeline cycle to the state metric write routing.

Figure 6.3 shows how path metric processing is carried out by 32 radix-4 units. The radix-4 units have branch metrics and source state metrics as inputs. The radix-4 unit outputs a survivor state metric which becomes a source state metric on the next Viterbi iteration. Decision bits are output to indicate the selected path.

By appropriate partitioning of the states into radix-4 units comprising 4 states, the 4 source state metrics are shared by all ACS units within the radix-4 unit. By placing the state metric registers with radix-4 unit, the state metric output (read) routing is kept completely local to the radix-4 unit, is short, and consequently fast. It is no longer a bottleneck to meeting timing.

A single radix-4 unit implements the fully-connected trellis shown in Figure 6.4 which is for h=1/4, L=3, M=4. A (θ_D, α_{n-1}) tuple uniquely identifies the source and destination state metrics for a radix-4 unit. There are 8 possible phase states for θ_D and 4 possible symbols for α_{n-1} giving 32 unique radix-4 units to carry out the processing for all 128 states in the trellis. A Matlab generated VHDL constant array contains the mapping of radix 4 output state metrics onto the state metrics bus, and from the state metric bus to the radix-4 unit state metrics inputs.

A standard implementation would use 4 add-compare-select units within the radix-4 unit. In this design we use two time shared ACS units to perform

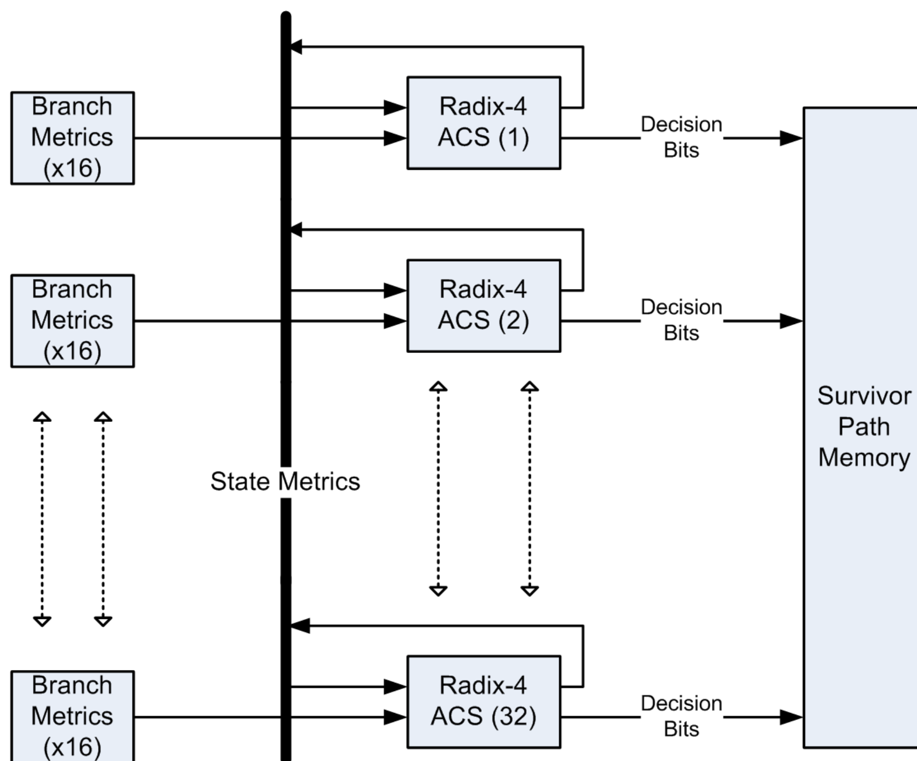

Figure 6.3: 128 State CPM Viterbi Detector Comprising 32 Radix-4 Add-Compare-Select Units

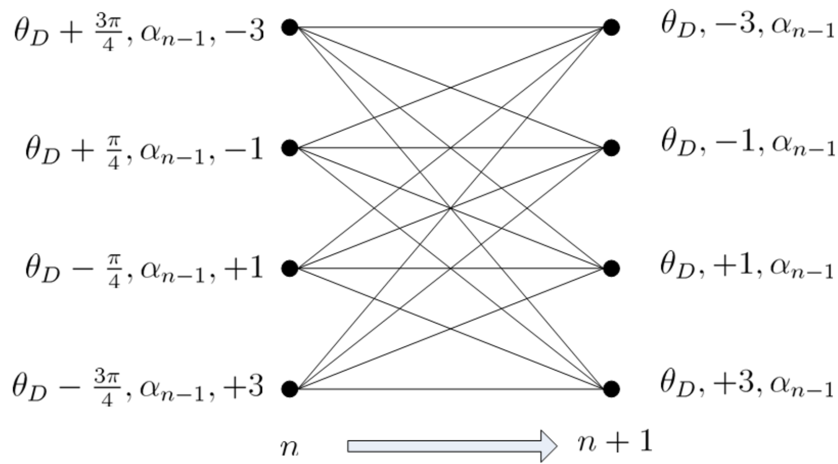

Figure 6.4: CPM Radix-4 Trellis (h=1/4, L=3, M=4)

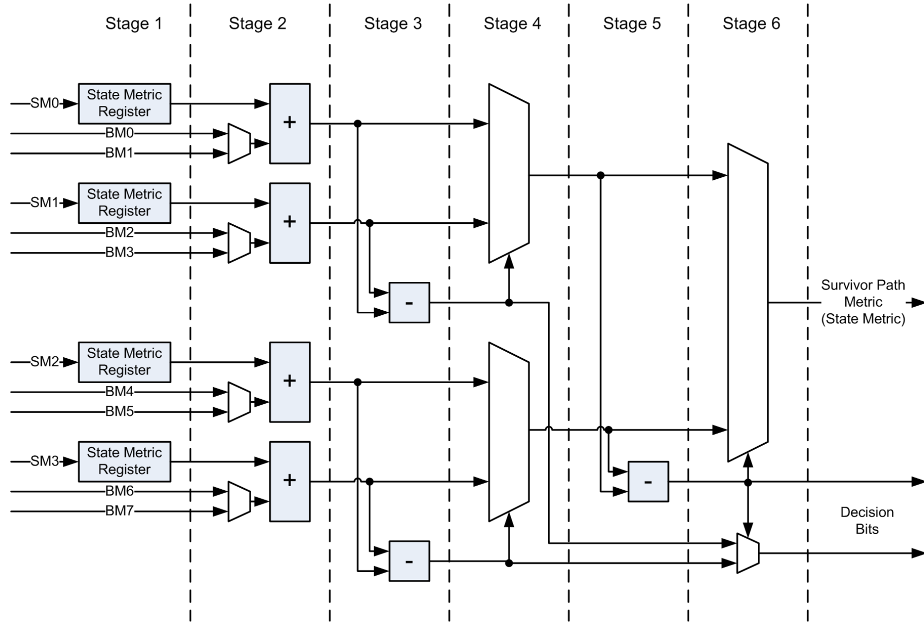

Figure 6.5: Add-Compare-Select (ACS) Unit Detail and Pipeline

the equivalent processing of 4 ACS units. This is shown in Figure 6.6.

6.3.2 Add-Compare-Select Unit

The proposed ACS unit design is shown in Figure 6.5. The implementation has a deep pipeline that is 6 stages long. This keeps the logic within each stage to be only 1 LUT logic level deep. i.e. there is only ever a maximum of 1 LUT level between registers. This results in only 1 route per stage when the pipeline flip-flop is co-located with the LUT driving the flip-flop. This level of pipelining is required to meet timing for the 189 MHz system clock [1].

Each pipeline stage has a specific function:

1. State Metric Register - Store the previous Viterbi iterations surviving state metrics, ready for input into the current iterations path metric calculations. During this stage the state metrics route from an ACS state metric output to the register input. This allows one complete cycle for state metric routing.

2. Path Metric Accumulation - State metric is added to the branch metric.

[1]Given the large number of branch metric connections that connect the branch metric unit and path metric unit (PMU), it is not practical to run the PMU at a clock frequency different to the BMU, because of the added cost of the clock domain crossing logic that would be required. Nevertheless, it may be possible to clock enable the ACS unit twice every Viterbi iteration and apply a 3 cycle multicycle timing constraint to relax the 189 MHz period constraint. This would allow 15.9 ns for the complete ACS processing and state metric routing. It seems unlikely this would meet timing, although this has not been investigated. Besides which, multicycle constraints are an added complexity and potentially disastrous if applied incorrectly since they relax timing on specific paths on the 189 MHz clock.

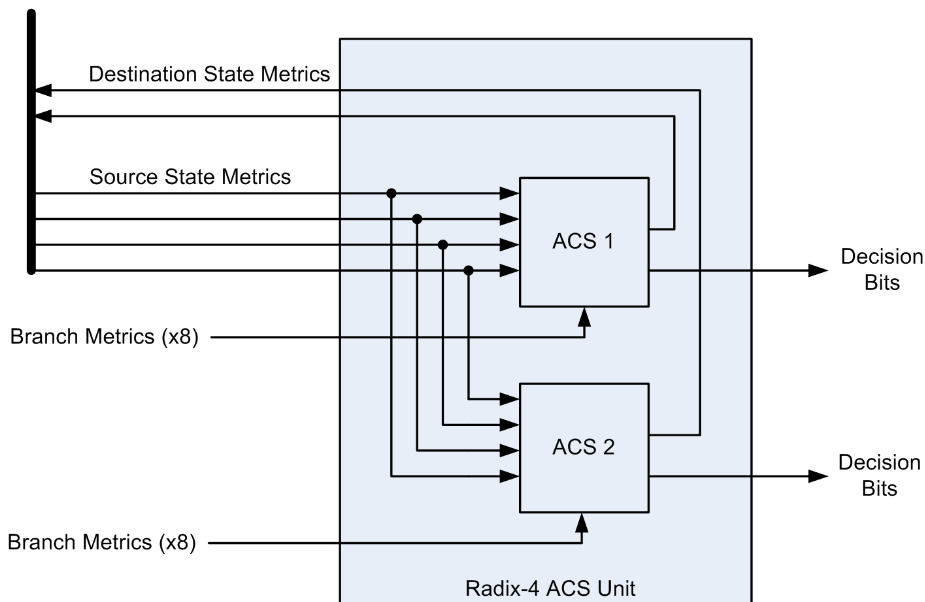

Figure 6.6: Radix-4 Unit Comprising Dual ACS Units

3. 1st Comparison - Subtract 2 metrics in pairs in the first stage of comparison.
4. 1st Selection - 2:1 multiplexer to select the largest path metric for each pair.
5. 2nd Comparison - Subtract the 2 remaining metrics to find the largest.
6. 2nd Selection - Final 2:1 multiplexer to select the surviving path metric.

For a clock frequency of 189 MHz, there are 7 clock cycles available per Viterbi iteration. Since the pipeline is 6 deep, the spare cycle is used to send a second set of branch metrics through the pipeline and compute the path metrics and select the survivor path for a second output state. This allows a single ACS unit to perform the state metric processing for 2 states, thus halving the number of ACS units required; a radix-4 unit comprises 2 ACS units as shown in Figure 6.6.

Since there are now 2 branch metrics to pass through the pipeline, a 2:1 multiplexer is incorporated into the stage 2 adder. This is possible because the FPGA fabric uses a 4 input LUT; 3 inputs are taken by the state metric operand and 2 branch metric operands, the 4th input is the select input for the multiplexer.

6.3.2.1 Resource Use Estimate

FPGA resource use is estimated in terms of logic cells in Table 6.1. Each logic cell contains a flip-flop and a 4 input LUT for the Xilinx Spartan 3A-DSP family of FPGAs. The FPGA logic fabric has been designed such that adders and 2 to

Pipeline Stage	Function	Logic Cell Estimate (LCs)	Notes
1	State Metric Register	$4 * B_{pm}$	
2	Path Metric Add	$4 * B_{pm}$	
3	1st Comparison	$3 * B_{pm}$	Each path metric must be registered to match the pipeline delay of the subtraction
4	1st Selection	$2 * B_{pm} + 2$	2 decision bits pipelined
5	2nd Comparison	$3 * B_{pm} + 2$	2 decision bits pipelined
6	2nd Selection	$1 * B_{pm} + 2$	1xMux and 2 decision bits
All Stages		$17 * B_{pm} + 6$	

Table 6.1: Single Add-Compare-Select Unit FPGA Resource Use Estimate

1 multiplexers consume 1 logic cell per bit. The logic cell usage is expressed as a function of the path metric (state metric) word-length B_{pm}.

A radix-4 unit comprises dual ACS units, but since there is a single set of state metric registers shared between both ACS units, the radix-4 unit estimate is double the ACS unit estimate minus a set of state metric registers. The radix-4 unit estimate is $2 * (17 * B_{pm} + 6) - 4 * B_{pm} = 30 * B_{pm} + 12$ logic cells.

The h=1/4, L=3, M=4 CPM trellis decomposes into 32 radix-4 units, and with a path metric wordlength of 11 bits, the total path metric processing FPGA resource use estimate is $32 * (30 * 11 + 12) = 10944$ logic cells.

6.4 Implementation Results

The path metric processing unit is written in VHDL and implemented using the Xilinx ISE tool suite. Appendices D.2 and D.4 contain the VHDL source and Appendix G lists the implementation tool versions used.

A period timing constraint of 215 MHz was applied to the place and route tools. The default synthesis and PAR options were modified slightly: "keep hierarchy" is set to yes to allow RLOC (relative location) constraints to propagate correctly through the hierarchy, and "add I/O buffers" is set to false since this design does not connect to FPGA device input or output pins [2].

The FPGA resource use and static timing analysis results are summarised in Table 6.2. Appendix F.2 contains more detailed output from the mapper and PAR tools.

The estimated resource use of 10944 logic cells comes close to matching the actual flip-flop usage results. The 7303 slice usage represents 43.9% of the FPGAs available 16640 slices. Static timing analysis shows the design meets the 189 MHz requirement, thus meeting the application's 54 Mbit/s throughput requirement.

[2] The ISE tools mapper strips all logic not connected to input or output pins. This is avoided by manually instantiating IBUFs on the path metric processing unit inputs and applying the mapper save attribute to the decision bit outputs

flip-flops	LUTs	slices	BRAMs	DSP48s	best case achievable period	Fmax
11338	7174	7303	0	0	4.891 ns	204.5 MHz

Table 6.2: Path Metric Processing Unit Implementation Results

6.4.1 Functional Verification

The same verification strategy as used for the branch metrics implementation is used to verify the path metrics processing implementation. This test strategy is justified and described in more detail in Appendix A.

The path metrics implementation testbench VHDL source code is in Appendix D.2.5 and the test vectors package in Appendix D.2.6. One hundred random symbols worth of I and Q samples were applied to the filter bank to generate the branch metrics for input to the VHDL design. The Matlab fixed point model stores the resulting state metrics values in a VHDL constant array. The path metrics test bench checks these state metrics, bit for bit, with the ones generated by the VHDL model, once every Viterbi iteration. The testbench reports no errors. This simulation demonstrates that the VHDL generated state metrics precisely match the Matlab fixed point model state metrics, and hence verifies operation of the design.

6.5 Conclusion

A state-parallel architecture comprising 32 radix-4 units, each containing two deeply pipelined ACS units performs the required path metric processing for the 128 state trellis. The proposed architecture has been implemented in VHDL and targeted toward a low-cost Xilinx Spartan 3ADSP FPGA. The implementation consume 43.9% of the available FPGA resources and passes static timing analysis at 204.5 MHz providing a healthy margin to the 189 MHz minimum requirement. A VHDL functional simulation verifies that the VHDL model precisely matches the fixed point model.

Chapter 7

Conclusions and Future Work

Continuous phase modulation is a promising entrant to the microwave radio cellular backhaul market. Its constant envelope property enables the use of non-linear radio frequency power amplifiers in the outdoor unit. This significantly reduces cost and improves power efficiency. Nevertheless, the spectral efficiency of CPM is inferior to traditional radios using large quadrature amplitude modulation constellations. Furthermore, receiver implementation complexity and cost are an issue for spectrally efficient CPM configurations which require multiple symbol duration phase pulses to reduce spectral occupancy.

This thesis proposes a CPM configuration that achieves a 50% improvement in spectral efficiency without degrading detection efficiency or system gain compared to a CPM microwave radio recently released to the market. This is achieved by moving to coherent demodulation in the receiver and using a longer, smoother, phase pulse. The new CPM configuration has a modulation index (h) of 1/4, phase pulse symbol duration (L) of 3, raised cosine phase pulse shape and has a quaternary (M=4) symbol alphabet size. By simulating a floating point model, this CPM configuration is shown to meet specific ETSI standard requirements [1, Annex D] that constrain the transmitted power spectral density and require minimum standards of performance for receiver detection efficiency. The application is for a 28 MHz wide channel and data rate of 54 Mbit/s, sufficient to transport 24 E1 circuits plus framing, forward error correction and auxiliary channel overhead. This is a significant improvement in data rate compared to the existing product which transports only 16 E1 circuits within the same 28 MHz ETSI channel.

Nevertheless, this new CPM configuration has high complexity and for this result to have practical significance, a low cost implementation is required. The literature contains many examples of non-optimal, complexity reduced CPM receivers, some of which show increased sensitivity to adjacent channel interference and have degraded detection efficiency compared to an optimal receiver. In this thesis, these issues are avoided by implementing the maximum-likelihood receiver using the Viterbi algorithm to demodulate the baseband CPM signal. The CPM configuration described above leads to a Viterbi trellis with 128 states, 4 inbound paths per state and 512 branch metrics that must be calculated each symbol period. The application requires a throughput of 27 MSymbols/s (54 Mbit/s), making a low cost FPGA implementation a challenge.

In order to explore the tradeoff between implementation word-length (cost) and detection efficiency degradation, a fixed point model of the CPM receiver has been developed. The degradation is only 0.2 dB when the received signal is quantised to 7 bits, branch metric filter bank coefficients quantised to 7 bits, branch metrics word-length of 9 bits and path metric word-length of 11 bits. There was no observed degradation due to the use of 2 samples per symbol processing and a Viterbi path history depth of 16 symbols.

The two most computationally expensive parts of the receiver are the branch metrics unit and path metrics processing unit. This thesis presents a novel approach to the branch metrics filter bank by using the well known distributed arithmetic algorithm and applying it to a CPM receiver branch metric unit for the first time. This technique performs 27.6 giga-multiplies per second, in a bit-serial fashion, making very efficient use of the FPGA logic fabric. One undesirable consequence of the bit serial processing is an added delay of 1 symbol duration which is likely to degrade the carrier phase recovery control loop performance. This is the price to be paid for a low-cost implementation.

The branch metric unit is implemented in VHDL and targeted to a low-cost Xilinx Spartan-3ADSP FPGA. The implementation consumes 23% of the available logic cells and the placed and routed design passes static timing analysis at 215 MHz, meeting the 189 MHz requirement to achieve a throughput of 54 Mbit/s. Functionality is verified using a VHDL testbench simulated by Modelsim. VHDL test vectors are generated by the fixed point model and are passed through the design, and results automatically checked to confirm that the VHDL implementation precisely matches the fixed point model.

Although all work in this thesis has been carried out in simulation, the same VHDL models are used in simulation and for FPGA implementation. By passing the VHDL implementation models through the FPGA vendors synthesis and place and route software and verifying the design meets static timing, and because the design uses a single clock domain, there is a high degree of confidence that the VHDL simulation matches the real world FPGA behaviour.

Viterbi path metric processing is implemented using a traditional state-parallel design. Static metric routing complexity is a recognised problem for large state Viterbi decoders; this thesis takes the approach of grouping states into radix-4 units so that all add-compare-select processing units within a radix-4 unit share the same set of state metrics. This keeps much of the state metric routing local and short. In order to operate at the high clock frequency of 189 MHz, the ACS unit is deeply pipelined. This results in 2 spare cycles within the Viterbi iteration loop, one of which is devoted to state metric output routing, to make it easier to meet timing on the state metric routing. The other spare cycle is used to send a second set of states through the ACS pipeline, thus halving the number of required ACS units to 2 per radix-4 unit bringing about a halving of FPGA resource use.

The Viterbi path metric processing VHDL implementation consumes 44% of the available FPGA resources and passes static timing analysis at 204.5 MHz, thus achieving the throughput requirement of 54 Mbit/s. The VHDL implementation was verified in simulation by precisely matching the VHDL model output with that of the Matlab fixed point model.

A Viterbi CPM demodulator also requires a survivor path management unit and a search for the best state metric. Future work is required to implement these.

For the first time, Hekstra's method of normalising path metrics has been applied to a CPM receiver. For this technique to work, the difference between state metrics must be bounded. The CPM configuration presented in this thesis comprise two unconnected sub-trellises and within each sub-trellis the state metric differences are bounded. Once synchronised, the transmitted symbol sequence only exists in one sub-trellis so there is an opportunity to halve the amount of processing required. This is for future study.

This thesis fills a gap in the CPM literature where there are few details of FPGA targeted CPM receivers. Simulation results presented here show a new CPM configuration achieving a 50% improvement in spectral efficiency compared to a recently released CPM microwave radio product. A VHDL implementation of this CPM configuration is presented and results show that a low cost FPGA implemenation is practical. This work has assumed ideal carrier phase recovery and timing recovery. Future work is required to implement carrier phase and timing recovery and integrate these functions with the receiver implementation proposed here.

Appendix

Appendix A

VHDL Implementation Functional Verification

A VHDL functional simulation and static timing analysis (STA) is used to verify the FPGA implementation. The only purpose of the VHDL functional simulation is to prove the VHDL implementation matches the fixed point model precisely. The Matlab floating point models demonstrate that the application's spectral efficiency and BER performance requirements are met. The Matlab fixed point simulation demonstrates receiver performance after moving to reduced precision fixed point arithmetic. The fixed point model is also used to generate test vectors for the VHDL functional simulation.

Testing the VHDL design in a real world FPGA is not required in order to demonstrate that the application's cost requirements are met. For this design, static timing analysgis is straightforward. The design is fully synchronous: it uses a single clock and no asynchronous inputs or outputs. By using the automated Xilinx STA tool, Trace, timing through all paths is checked and guarantees the design will function as simulated over process, voltage and temperature. This is because Trace checks the placed and routed designs timing against a set of speedfiles specific to the FPGA device. Each FPGA device manufactured by Xilinx is tested against this same set of FPGA speedfiles, thus guaranteeing the customers design will function correctly in every FPGA shipped by Xilinx [49].

A.1 VHDL Functional Verification Architecture

Figure A.1 shows how the Matlab fixed point model is used to generate test vectors which are then applied to the VHDL CPM receiver implementation in a Modelsim VHDL simulation testbench. The Matlab fixed point model is the same as used in chapter 4 but with function hooks (see Appendix E.3.1) added to export results from specific points in the model. The exported results are read by a VHDL writer function written in Matlab. The VHDL writer function generates a VHDL package file containing the exported results as VHDL constant arrays. The VHDL writer source code is in Appendix E.3.2.

A VHDL testbench then imports this VHDL package, instantiates the VHDL device under test(DUT), applies the input test vectors to the DUT and checks

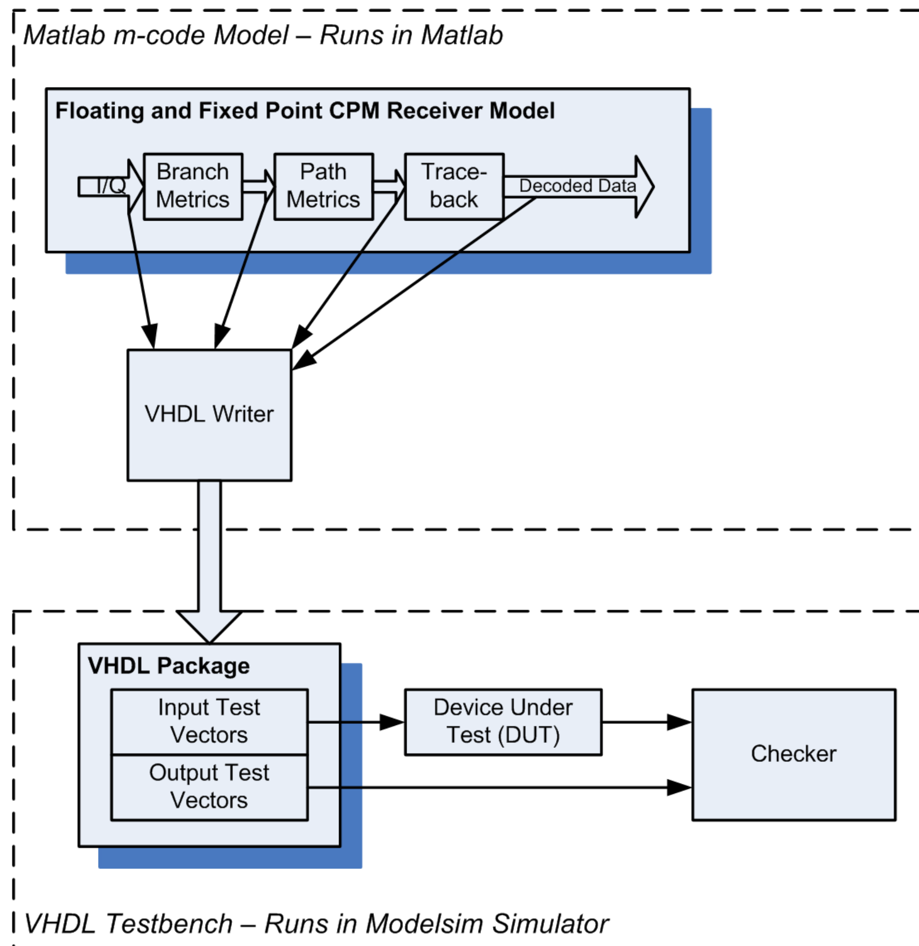

Figure A.1: VHDL Implementation Test Architecture

the DUT output against the correct output in the test vectors. Every error is counted, and a VHDL assert prints an error message in the simulator if any errors are found. For example, Appendix D.1.5 contains the VHDL testbench for the branch metrics VHDL implementation.

Appendix B

Receive Signal Level to SNR Conversion

The ETSI radio frequency performance specification used in this thesis specifies detection efficiency by requiring the bit error rate to be less than 10^{-6} at a receive signal level (RSL) of -75 dBm [1, Table D.6].

Received signal strength is expressed as a signal to noise ratio (SNR) for the purposes of simulation and analysis. This is more useful metric because it is independent of bandwidth. We assume two noises sources for the purposes of this calculation. Firstly, equation B.1 defines the thermal noise in the symbol rate bandwidth (B), modelled as additive white gaussian noise. Secondly, noise added by the receiver itself. We conservatively assume a receiver noise figure of 6 dB. SNR is calculated from RSL using equation B.2 [50].

Energy per bit relative to noise, $\frac{E_b}{N_o}$, is also commonly used as a receiver figure of merit because it is independent of bandwidth and number of bits per symbol (M). Equation B.3 converts between SNR and $\frac{E_b}{N_o}$.

$$P_{thermalnoise} = -174dBm + 10\log_{10}(B) \tag{B.1}$$

$$SNR_{dB} = RSL - P_{thermal_noise} - NF_{receiver} \tag{B.2}$$

$$SNR_{dB} = 10log_{10}(\log_2(M)) + \frac{E_b}{N_o} \tag{B.3}$$

Table B.1 summarises the conversion from -75 dBm receive signal level to SNR and $\frac{E_b}{N_o}$. This is equivalent to a SNR of 18.7 dB and $\frac{E_b}{N_o}$ of 15.7 dB.

ETSI RSL [1, Table D.6]	Symbol Rate Bandwidth(B)	$P_{thermalnoise}$	$NF_{receiver}$	SNR	$\frac{E_b}{N_o}$ (M=4)
-75 dBm	27 MHz	-99.7 dBm	6 dB	18.7 dB	15.7 dB

Table B.1: ETSI Received Signal Level Converted to SNR and $\frac{E_b}{N_o}$

Appendix C

Baseband I/Q Modulator Derivation

Communication systems are often described in terms of their in-phase and quadrature baseband signal components. At some point these I and Q channels must modulate the actual carrier. This section derives the standard I/Q modulator.

Consider the passband signal (C.1) that might be amplitude modulated with $a(t)$ and phase modulated with $\theta(t)$.

$$s(t) = a(t)\cos(\omega_c t + \theta(t)) \tag{C.1}$$

$$s(t) = \Re\{a(t)e^{j\theta t}e^{j\omega t}\} \tag{C.2}$$

Now define $\tilde{s}(t) = a(t)e^{j\theta t}$ as the baseband complex envelope of $s(t)$.

$$s(t) = \Re\{\tilde{s}(t)e^{j\omega t}\} \tag{C.3}$$

Expressing $\tilde{s}(t)$ in terms of its real $s_I(t)$ and imaginary $s_Q(t)$ parts gives (C.4).

$$s(t) = \Re\{(s_I(t) + js_Q(t))e^{j\omega t}\} \tag{C.4}$$

$$s(t) = s_I(t)\cos(\omega t) - s_Q(t)\sin(\omega t) \tag{C.5}$$

Appendix D

VHDL Source Code

This appendix lists the VHDL source code files developed. In a few key cases, the source code is printed here. Otherwise, please see the CD associated with this thesis for the source code in electronic format.

D.1 Branch Metric Filter Bank

D.1.1 Synthesis Top Level

/fpga/src/branch_metrics_dafir_synth.vhd

D.1.2 Top Level

```
--
   ****************************************************************************************************
-- Title:   Branch metric filter bank using distributed arithmetic fir
   filters
-- Author:  Andrew Bridger
-- Date:    1 July 2008
-- High Level Module Description:  This module implements a bank of
   filters with coefficients provided by
--  a matlab generated VHDL package. The input signal I and Q
   components are serialised and fed to all
--  filters in parallel lsb first. The filter output appears B_I = B_Q
    cycles + pipelining stages later.
--
-- Notes/Limitations: TODO round and saturate the filter outputs.
--  1) XST produces this silly warning :WARNING:Xst:2677 - Node <
   i_shift_reg_1_0> of sequential type is unconnected in block <
   branch_metrics_dafir>.
--     Using the RTL viewer you can see XST uses the name x(0) instead
   of i_shift_reg_1_0.
--
-- Synthesizable: Yes
--
-- Testbench: branch_metrics_dafir_tb.vhd
--
-- Note: The version control system in use is the repository for
   information regarding bug fixes,
-- versions, feature additions etc.
```

```
--
    ********************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.PkgStdType.all;
use work.pkg_project.all;
use work.pkg_mf_coeffs.all;
use work.pkg_rloc.all;

library unisim;
use unisim.vcomponents.all;

entity branch_metrics_dafir_primitives is
  generic(
    PLACE              : natural;              --1 = spartan 3
        placed, 0 = unplaced
    B_COEFF            : positive;             --Filter coefficient
        wordlength
    B_IQ               : positive;             --I/Q input
        wordlength
    FILTER_BANK_COEFFS : filter_bank_coeffs_ta ); --filter
        coefficients
  port (
    clk                :in   std_logic;        --system clock
    i,q                :in   iq_ta;            --array of samples for
        1 symbol period
    new_samples        :in   std_logic;        --active during first
        clk period in which new IQ samples are presented
    branch_metrics     :out  branch_metrics_ta;--valid for only 1
        clock period
    branch_metrics_rdy :out  std_logic         --active when output
        valid
    );
end branch_metrics_dafir_primitives;

architecture xprim of branch_metrics_dafir_primitives is


  signal iq_lsb, iq_msb, iq_lsb_buf, iq_msb_buf: std_logic;

  subtype x_t is std_logic_vector(3 downto 0);
  signal x               : x_t;
  signal bm_rdy          : std_logic_vector(NUM_FILTERS downto 1);
  signal dummy_dout0, dummy_dout1, dummy_dout2, dummy_dout3 :
      std_logic_vector(i(1)'range);

begin
  assert false report "branch_metrics_dafir_primitives: check
      create_mf_dalut_coeffs.m has INCLUDE_NEG_MF_COEFFS = 0 " severity
       warning;
  --I'm confused. Shouldn't INCLUDE_NEG... be set to 1 because
      pkgproject says numfilters = size of coeff
  --array /2.!

  --Serialise I and Q samples and generate controls for the dafir
      filters.
  control: process( clk )
```

```
  variable bit_count                    : natural range 0 to B_IQ-1;
begin
  if rising_edge(clk) then
    iq_lsb <= '0'; --default assignments
    iq_msb <= '0';
    if new_samples = '1' then --load shift reg
      bit_count    := 0;
      iq_lsb       <= '1';
    else --shift out, lsb first
      if bit_count = (B_IQ-1) then
        bit_count := 0;
      else
        bit_count := bit_count + 1;
      end if;
    end if;
    --Keep track of iq bit position and signal when msbit reached.
        iq_msb goes high for 1 cycle only.
    if bit_count = (B_IQ-1) then
      iq_msb <= '1';
    end if;
  end if;
end process;

--Form the dalut address input by picking off the lsb of the shift
    reg
--i(n) serialise
i_n_serialise: entity work.shift_reg(xprim)
generic map(
  SHIFT_DIRECTION => RIGHT, --0 right (towards lsb), 1 shift left (
      towards msb)
  PLACE           => PLACE)
      port map(
  clk        => clk,
  ce         => '1',
  load       => new_samples,
  din        => std_logic_vector(i(1)),
  dout       => dummy_dout0,
  serial_in  => '0',
  serial_out => x(0));

--i(n-1) serialise
i_n_1_serialise: entity work.shift_reg(xprim)
generic map(
  SHIFT_DIRECTION => RIGHT, --0 right (towards lsb), 1 shift left (
      towards msb)
  PLACE           => PLACE)
      port map(
  clk        => clk,
  ce         => '1',
  load       => new_samples,
  din        => std_logic_vector(i(2)),
  dout       => dummy_dout1,
  serial_in  => '0',
  serial_out => x(1));

--q(n) serialise
q_n_serialise: entity work.shift_reg(xprim)
generic map(
  SHIFT_DIRECTION => RIGHT, --0 right (towards lsb), 1 shift left (
      towards msb)
  PLACE           => PLACE)
      port map(
```

```
    clk         => clk,
    ce          => '1',
    load        => new_samples,
    din         => std_logic_vector(q(1)),
    dout        => dummy_dout2,
    serial_in   => '0',
    serial_out  => x(2));


  --q(n-1) serialise
  q_n_1_serialise: entity work.shift_reg(xprim)
  generic map(
    SHIFT_DIRECTION => RIGHT, --0 right (towards lsb), 1 shift left (
        towards msb)
    PLACE               => PLACE)
        port map(
    clk         => clk,
    ce          => '1',
    load        => new_samples,
    din         => std_logic_vector(q(2)),
    dout        => dummy_dout3,
    serial_in   => '0',
    serial_out  => x(3));

  --Buffer the dafir control signals to match the 1 clk latency
      introduced by the x buffering
  iq_msb_flop: FDRSE
  generic map ( INIT => '0') -- Initial value of register ('0' or '1')
  port map( C    => clk,
            D    => iq_msb,
            Q    => iq_msb_buf,
            CE   => '1',
            R    => '0',
            S    => '0');
  iq_lsb_flop: FDRSE
  generic map ( INIT => '0') -- Initial value of register ('0' or '1')
  port map( C    => clk,
            D    => iq_lsb,
            Q    => iq_lsb_buf,
            CE   => '1',
            R    => '0',
            S    => '0');

  --bank of matched filters that generate branch metrics at their
      output. Inputs are serialised versions of
  --I and Q. The filter coefficients are contained in pkg_mf_coeffs.vhd
      .
  --Each filter calculates I(n)*C(0) + I(n-1)*C(1) + Q(n)*C(2) + Q(n-1)
      *C(3).
  --2 Samples per symbol period are assumed.
  generate_mf_bank: for bmid in 1 to NUM_FILTERS/2 generate
    --Placement. Arrange in a 2D array. Place 2 filters side by side at
        a time, then order is
    --column by column.
    --Todo generalise to a function and put into pkg_rloc.
    --define size of object being placed
    attribute RLOC          : string;
    --constant PLACE          : natural := 1;
    constant YSIZE_SLICES : natural := 6; --hardcoded - todo calc
        properly
    constant XSIZE_SLICES : natural := 4;
    constant objs_per_col  : natural := 16;
```

```
    --constant objs_per_row : natural := 16;
    constant idx          : natural := bmid -1;  --zero referenced.
    constant x_rloc       : natural := (idx/objs_per_col) *
        XSIZE_SLICES;
    constant y_rloc       : natural := (idx mod objs_per_col) *
        YSIZE_SLICES;

--    constant xy_str        : string  := "x" & itoa(x_rloc) & "y" &
    itoa(y_rloc);
--    constant rloc_str      : string  := pick_string(PLACE, xy_str);
    --Strange - pkg_rloc was missing, yet synthesized fine??
        pick_string should have caused error
    attribute RLOC of matched_filter_left :
      label is pick_string(PLACE, "x" & itoa(x_rloc) & "y" & itoa(
          y_rloc));
    attribute RLOC of matched_filter_right :
      label is pick_string(PLACE, "x" & itoa(x_rloc+2) & "y" & itoa(
          y_rloc));

    signal x_buf: std_logic_vector(x'range);
    constant bmid_mf1 : natural := idx*2 + 1;
    constant bmid_mf2 : natural := idx*2 + 2;
  begin
    --matched_filter: entity work.DAFir4Tap(rtl)
    matched_filter_left: entity work.DA_fir_4tap_all_primitives(rtl)
    generic map(
      PLACE           => PLACE,
      B_COEFF         => B_COEFF,              --Coefficient wordlength
      COEFF           => FILTER_BANK_COEFFS(bmid_mf1), --Coefficients
      B_X             => B_IQ)                 --x(n) word length
    port map (
      clk      => clk,
      x        => x_buf,
      x_lsb    => iq_lsb_buf,
      x_msb    => iq_msb_buf,
      y        => branch_metrics(bmid_mf1),
      y_rdy    => bm_rdy(bmid_mf1));

    matched_filter_right: entity work.DA_fir_4tap_all_primitives(rtl)
    generic map(
      PLACE           => PLACE,
      B_COEFF         => B_COEFF,              --Coefficient wordlength
      COEFF           => FILTER_BANK_COEFFS(bmid_mf2), --Coefficients
      B_X             => B_IQ)                 --x(n) word length
    port map (
      clk      => clk,
      x        => x_buf,
      x_lsb    => iq_lsb_buf,
      x_msb    => iq_msb_buf,
      y        => branch_metrics(bmid_mf2),
      y_rdy    => bm_rdy(bmid_mf2));

    --If the filter bank is large, these x inputs are very high fanout.
        Add flop buffers every so often
    --to reduce fanout, and reduce the routing length.
    gen_x_flop_buf: for j in 0 to 3 generate
      --placement. mf RPM has F row empty, so fit these flops in there.
      attribute BEL : string;
      attribute BEL of x_flop_buf  : label is pick_string(PLACE, "FFX")
          ; --??seems to ignore this constraint?
      constant flop_buf_xy_str     : string  := "x" & itoa(x_rloc + j)
          & "y" & itoa(y_rloc);
```

```
      attribute RLOC of x_flop_buf : label is pick_string(PLACE,
          flop_buf_xy_str);
    begin
      x_flop_buf: FDRSE
      generic map ( INIT => '0') -- Initial value of register ('0' or
          '1')
      port map( C    => clk,
                D    => x(j),
                Q    => x_buf(j),
                CE   => '1',
                R    => '0',
                S    => '0');
    end generate;
  end generate;

  --All bms ready at the same time
  branch_metrics_rdy <= bm_rdy(1);
end xprim;


  --Serialise I and Q samples and generate controls for the dafir
      filters.
--    control: process( clk )
--      variable i_shift_reg, q_shift_reg : iq_ta;
--      variable bit_count                : natural range 0 to B_IQ-1;
--    begin
--      if rising_edge(clk) then
--        iq_lsb <= '0'; --default assignments
--        iq_msb <= '0';
--        if new_samples = '1' then --load shift reg
--          bit_count   := 0;
--          iq_lsb      <= '1';
--          i_shift_reg := i;
--          q_shift_reg := q;
--        else --shift out, lsb first
--          if bit_count = (B_IQ-1) then
--            bit_count := 0;
--          else
--            bit_count := bit_count + 1;
--          end if;
--          for n in 1 to SAMPLES_PER_SYMBOL loop
--            i_shift_reg(n) := "0" & i_shift_reg(n)(i_shift_reg(n)'high
    downto i_shift_reg(n)'low+1);
--            q_shift_reg(n) := "0" & q_shift_reg(n)(q_shift_reg(n)'high
    downto q_shift_reg(n)'low+1);
--          end loop;
--        end if;
--        --Form the dalut address input by picking off the lsb of the
    shift reg
--        for n in 1 to SAMPLES_PER_SYMBOL loop
--          x(n-1)                      <= i_shift_reg(n)(i_shift_reg(n)'
    low);
--          x(n-1+SAMPLES_PER_SYMBOL) <= q_shift_reg(n)(q_shift_reg(n)'
    low);
--        end loop;
--        --Keep track of iq bit position and signal when msbit reached.
    iq_msb goes high for 1 cycle only.
--        if bit_count = (B_IQ-1) then
--          iq_msb <= '1';
--        end if;
--      end if;
--    end process;
```

—

D.1.3 4-Tap Distributed Arithmetic Filter

```
--
   ******************************************************************************************
-- Title:    Distributed Arithmetic FIR Filter (Only supports 4 taps),
   implemented with xilinx primitives
-- Author:  Andrew Bridger
-- Date:     23 July 2008
-- High Level Module Description: Fully bit serial DA Fir filter. See
   pg 62 "DSP with FPGA" Uwe Meyer Baese.
--  Designed as a component for CPM branch metric filter bank. Input
   signal serialisation not done here.
--  This design comprises a DALUT containing precomputed partial
   products followed by a scaling accumulator.
--  Input signal signed in 2's complement format.
--
--  This version completely implemented using xilinx primitives and
   with placement attributes.  This is
--  required to meet timing for high clock rates and when a large
   number of these filters are used. E.g. The
--  PAR tools did not place the DALUT rom and pipeline flops in the
   same slice, needlessly incurring a
--  a route that was more than 3ns in some cases.
--
--  The subtract and reset flop controls were also being optimised away
    for larger filter banks. Which
--  leads to very high fanout and long, slow routes.
--
--
-- Notes/Limitations:
--  1) Bizarre warning. WARNING:Xst:1610 - "C:/project/
   massey_stratex_cpm/demosvn/abr_masters_cpm/fpga/src/DAFir4Tap.vhd"
   line 78: Width mismatch. <partial_product> has a width of 9 bits
   but assigned expression is 10-bit wide.
--     In numeric_std "+" operator with signed inputs will produce a
   result the length of the operand
--     with the longest length. With B_COEFF = 7, both operands and the
    result all have 9 bits. Modelsim
--     gives no warnings.
--
-- Synthesizable: Yes
--
-- Testbench: DA_fir_4tap_tb.vhd
--
-- Note: The version control system in use is the repository for
   information regarding bug fixes,
-- versions, feature additions etc.
--
   ******************************************************************************************

--Abbreviations
--SA = scaling accumulator

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.PkgStdType.all;
```

```
use work.pkg_mf_coeffs.all;
use work.pkg_rloc.all;

library unisim;
use unisim.vcomponents.ALL;

entity DA_fir_4tap_all_primitives is
  generic(
    PLACE           : natural;                  --1 = spartan 3 placed ,
        0 = unplaced
    B_COEFF         : positive;                 --Coefficient
        wordlength
    COEFF           : single_filter_coeffs_ta;  --Coefficients
    B_X             : positive);                --x(n) word length
  port (
    clk             :in   std_logic;                      --system
        clock
    x               :in   std_logic_vector(3 downto 0);
    x_lsb           :in   std_logic;                      --active
        during first bit of x
    x_msb           :in   std_logic;                      --active when
        the sign bit of x is input
    y               :out  signed(B_X+B_COEFF+2-1 downto 0);-- valid for
        only 1 clk period
    y_rdy           :out  std_logic                       --active when
        y result is valid
    );
end DA_fir_4tap_all_primitives;


architecture rtl of DA_fir_4tap_all_primitives is
  --With 256 filters , the branch metric outputs far exceed real chip IO
      . So synth without IO pads. This
  --then requires the use of mapper save attribute to prevent the
      mapper from stripping the whole design
  --since there are no loads(IO pads)
  --Unfortunately , applying this attribute on the branch_metrics signal
       does not work. Possibly because
  --branch_metrics is an array of arrays. Hence do it here - this is
      only required when synthesizing without
  --IO pads.
  attribute s: string;
  attribute s of y: signal is "yes";

  constant LUT_SIZE           : positive := 4;  --only 4 input LUT
      currently supported.
  constant DALUT_BIT_GROWTH   : natural  := 2;  --For 6 input LUT, bit
      growth is 3
  --subtype partial_product_t is signed( B_COEFF+DALUT_BIT_GROWTH-1
      downto 0 );
  --type dalut_t is array (0 to (2**LUT_SIZE)-1) of partial_product_t;

 --Generate DALUT lookup table. All possible partial products are
     generated and returned in an array.
  function create_dalut(
    constant INTEGER_COEFFS : single_filter_coeffs_ta)    --filter
        coefficients
    return integer_ta is
      variable partial_product     : integer;
      variable dalut               : integer_ta(0 to (2**LUT_SIZE)-1);
      variable addr_bit, lut_addr  : integer;
  begin
```

```
    assert INTEGER_COEFFS'length = LUT_SIZE
      report "CreateDalut() - Exactly 4 coefficients must be provided."
          ;
    --Calculate all possible partial products for these coefficients.
        This is done one address at a time.
    for lut_addr in 0 to 2**LUT_SIZE -1 loop
      partial_product := 0;
      --Check each bit of the LUT address to determine which
          coefficients should be accumulated to
      --generate the final partial product for this address.
      for addr_bit in 0 to LUT_SIZE-1 loop
        if( to_unsigned(lut_addr, LUT_SIZE)( addr_bit ) = '1' ) then
          partial_product := partial_product + INTEGER_COEFFS(addr_bit)
              ;
        end if;
      end loop;
      dalut( lut_addr ) := partial_product;
    end loop;
    return dalut;
  end function;

  constant SIGN_EXTEND_BIT : natural := 1;
  signal sa_add, op_a, op_b : std_logic_vector(SIGN_EXTEND_BIT +
      DALUT_BIT_GROWTH + B_COEFF-1 downto 0);
  signal sa_shift           : std_logic_vector(B_X-2 downto 0);
  signal reset_sa_n, subtract_pp : std_logic;
  signal reset_sa_n_noreg   : std_logic;
  signal y_slv              : std_logic_vector(y'range);

  constant dalut_init : integer_ta := create_dalut(COEFF);
  signal pp           : std_logic_vector(DALUT_BIT_GROWTH + B_COEFF-1
      downto 0);
--   constant dummy_zero : std_logic_vector(pp'range) := (others => '0')
    ;

  --Placement. Arrange DALUT and scaling accumulator in adjacent
      columns, lsb aligned.
  attribute RLOC : string;
  --constant xy_str            : string := "x0y0" & itoa((i/2)) ;
  --constant rloc_str          : string := pick_string(PLACE, xy_str);
  attribute RLOC of dalut                : label is pick_string(PLACE,
      "x0y1");
  attribute RLOC of scaling_accumulator  : label is pick_string(PLACE,
      "x1y1");
  --place below the lsb of dalut/sa since lsb has longest carry chain
      path.
  attribute RLOC of reset_sa_n_flop      : label is pick_string(PLACE,
      "x1y0");
  attribute RLOC of subtract_flop        : label is pick_string(PLACE,
      "x0y0");
  --constant control_flop_y_pos : natural := (DALUT_BIT_GROWTH +
      B_COEFF+1) mod 2;
  --attribute RLOC of reset_sa_n_flop       : label is pick_string(
      PLACE, "x0y" & itoa(control_flop_y_pos));
  --attribute RLOC of subtract_flop         : label is pick_string(
      PLACE, "x1y" & itoa(control_flop_y_pos));
begin

  --x input 1 bit per clock, lookup the partial products sum based on
      the value of x
  --Register the partial product, and then accumulate in the scaling
      accumulator. The scaling accumulator
```

```
--is reset to 0 when x lsb presented. When the x msbit(sign bit) is
    presented the partial product
--is subtracted from the scaling accumulator result. (See Uwe Meyer-
    baese)

--scaling accumulator does => acc_pp = (2^-1)*previous acc + pp*2^(
    B_X-1).
--I.e. the pp is msbit aligned with (2^-1)*previous acc and then
    added. The (B_X-1) zeros
--post-pended to pp don't need to be added since A + 0 = 0!. This
    means we have two components to
--the scaling accumulator result. Called sa_add and sa_shift.
y_slv <= sa_add & sa_shift;
y     <= signed(y_slv);

--dalut lookup, registered.
--this lutram has a problem getting throught the mapper
--Pack:679 - Unable to obey design constraints (MACRONAME=filter_bank
    /generate_mf_bank[1].matched_filter/dalut/hset, RLOC=X0Y4) which
    require the combination of the following symbols into a single
    SLICEM component:
--      FLOP symbol "filter_bank/generate_mf_bank[1].matched_filter/
    dalut/bit_loop[8].yes_dout_flop.dout_flop" (Output Signal =
    filter_bank/generate_mf_bank[1].matched_filter/pp<8>)
--      RAM symbol "filter_bank/generate_mf_bank[1].matched_filter/
    dalut/bit_loop[8].ram_bit" (Output Signal = filter_bank/
    generate_mf_bank[1].matched_filter/dalut/dout_no_reg<8>)
--Function generator filter_bank/generate_mf_bank[1].matched_filter/
    dalut/bit_loop[8].ram_bit has a site constraint other than "G".
    Please correct the design constraints accordingly.
--    dalut: entity work.RAM16XnS(xprim)
--    generic map (
--      REG    => 1,            -- 0 no reg, 1 register RAM output
--      INIT   => dalut_init,   --initial ram contents
--      PLACE  => 1)            -- 0 unplaced, 1 spartan 3
--    port map (
--      dout   => pp,
--      din    => dummy_zero,
--      addr   => x,
--      clk    => clk,
--      we     => '0' );

--dalut lookup, registered.
dalut: entity work.lut_rom(xprim)
generic map (
  REG   => 1,           -- 0 no reg, 1 register RAM output
  INIT  => dalut_init,  --initial ram contents
  PLACE => PLACE)           -- 0 unplaced, 1 spartan 3
port map (
  dout  => pp,
  addr  => x,
  clk   => clk);

--Setup, and register the controls for the scaling accumulator adder
--x_msb, x_lsb are very high fanout signals when this filter is used
    in a large
--filter bank. These flops also serve to buffer these signals and
    reduce fanout.
reset_sa_n_noreg  <= not(x_lsb);

reset_sa_n_flop: FDRSE
generic map ( INIT => '0') -- Initial value of register ('0' or '1')
```

```
  port map( C    => clk,
            D    => reset_sa_n_noreg,
            Q    => reset_sa_n,
            CE   => '1',
            R    => '0',
            S    => '0');

  subtract_flop: FDRSE
  generic map ( INIT => '0') -- Initial value of register ('0' or '1')
  port map( C    => clk,
            D    => x_msb,
            Q    => subtract_pp,
            CE   => '1',
            R    => '0',
            S    => '0');

  --sign extend the operands into the scaling accumulator
  op_b  <= std_logic_vector(pp(pp'high) & pp);
  op_a  <= sa_add(sa_add'high) & sa_add(sa_add'high downto (sa_add'low
      +1)); -- 2^-1

  --Scaling accumulator. ans = a +/- b
  scaling_accumulator: entity work.adder_sub_clr(rtl)
    generic map( PLACE => PLACE )
    port map(
      clk          => clk,
      subtract_b   => subtract_pp,
      clear_a_n    => reset_sa_n,
      a            => op_a,
      b            => op_b,
      ans          => sa_add);

  --The shift register component of the scaling accumulator simply
      shifts right, taking the lsb of
  --the scaling accumulator adder portion. Except when the lsb of a new
       word is applied, in which
  --case the the shift reg must clear.
--     shift_component: entity work.shift_reg(xprim)
--     generic map(
--       SHIFT_DIRECTION => RIGHT, --0 right (towards lsb), 1 shift left
    (towards msb)
--       PLACE            => 1)
--         port map(
--       clk       => clk,
--       ce        => '1',
--       load      => not(reset_sa_n),
--       din       => init to zeros...,
--       shift_in  => sa_add(sa_add'low),
--       shift_out => sa_shift);


  shift_component: process(clk)
  begin
    if rising_edge(clk) then
      --The shift register component of the scaling accumulator simply
          shifts right, taking the lsb of
      --the scaling accumulator adder portion. Except when the lsb of a
           new word is applied, in which
      --case the the shift reg must clear.
      if reset_sa_n = '0' then
        sa_shift <= (others => '0');
      else
```

```
        sa_shift <= sa_add(sa_add'low) &  sa_shift(sa_shift'high downto
             (sa_shift'low+1));
      end if;
      --The subtract cycle is the last for the current word, i.e.
          result ready
      --at next clock edge.
      --If this module used in a large filter bank, this flop becomes
          repeated many times. Hopefully
      --tools will remove most of these redundant flops.
      y_rdy        <= subtract_pp;
    end if;
  end process;
end rtl;
```

D.1.4 Filter Coefficients

/matlab/fpga/vhdl/pkg_mf_coeffs.vhd

D.1.5 Testbench

```
--
    ************************************************************************************
-- Title:    Testbench for dafir branch metric filter bank
-- Author:   Andrew Bridger
-- Date:     3 July 2008
-- High Level Module Description:  Instantiates the filter bank, inputs
     matlab generated stimulus and
--    checks result against matlab generated results.
-- Notes/Limitations:
--
-- Synthesizable: No
--
-- Testbench: n/a
--
-- Note: The version control system in use is the repository for
     information regarding bug fixes,
-- versions, feature additions etc.
--
    ************************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.PkgStdType.all;
use work.pkg_math.all;
use work.pkg_project.all;
use work.pkg_mf_coeffs.all;
--use work.pkg_mf_coeffs_h025_L3_M4.all;
use work.pkg_matlab_test_vectors.all;

library unisim;
use unisim.vcomponents.all;

entity branch_metrics_dafir_tb is
end branch_metrics_dafir_tb;

architecture branch_metrics_dafir_tb_arch of branch_metrics_dafir_tb is
```

```
  signal clk,reset            :  std_logic;         --system clock
  signal i,q                  :  iq_ta;             --array of samples
      for 1 symbol period
  signal new_samples          :  std_logic;         --active during first
       clk period in which new IQ samples are presented
  signal branch_metrics       :  branch_metrics_ta;--valid for only 1
      clock period
  signal branch_metrics_rdy   :  std_logic;          --active when output
       valid

  signal blahtest : std_logic;

begin

  --dut_filter_bank: entity work.branch_metrics_dafir(
      branch_metrics_dafir_arch )
  dut_filter_bank: entity work.branch_metrics_dafir_primitives(xprim)
    generic map(
      PLACE                 => 1,
      B_COEFF               => B_COEFF,
      B_IQ                  => B_IQ,
      FILTER_BANK_COEFFS    => FILTER_BANK_COEFFS   )
    port map(
      clk                   => clk,
      i                     => i,
      q                     => q,
      new_samples           => new_samples,
      branch_metrics        => branch_metrics,
      branch_metrics_rdy    => branch_metrics_rdy);

  main: process
    --100MHz clock
    procedure tick is
    begin
      clk <= '0';
      wait for 5 ns;
      clk <= '1';
      wait for 5 ns;
    end procedure;

    procedure GenReset( constant length : natural ) is
    begin
      reset <= '1';
      for i in 0 to length loop
        tick;
      end loop;
      reset <= '0';
      tick;
    end procedure;

    --Test two arrays for equality. Returns 0 if equal element wise,
        otherwise returns number of differences.
    function isequal(a, b : integer_ta) return natural is
      variable diff : natural := 0;
      constant same_length : boolean := (a'length = b'length);
      variable a_norm, b_norm : integer_ta(1 to a'length);
    begin
      assert same_length report "isequal: operands have different array
           lengths." severity failure;
      --normalise range definitions
      a_norm := a;
```

```
    b_norm := b;
    for i in 1 to a'length loop
      if (a_norm(i) /= b_norm(i)) then
        diff := diff + 1;
      end if;
    end loop;
    return diff;
  end function;

  type many_branch_metrics_ta is array (natural range <>) of
      branch_metrics_ta;
  type many_iq_samples_ta is array (natural range <>) of iq_ta;

  --Apply i/q stimulus and check result. However, i/q applied in a
      fairly simple manner; a new symbols
  --worth of samples are input after the previous result has been
      fully calculated. I.e the filter
  --is not running at maximum throughput. (the filter is pipelined so
       normally will have two symbols in the
  --pipeline at once).
  procedure test_filter_bank_simple( constant i_input, q_input :
      integer2D_ta;
                                     constant correct_result   :
                                         integer2D_ta ) is
    variable errors          : natural := 0;
    variable num_symbols     : natural := i_input'length(1);
    --normalise ranges
    constant i_input_norm : integer2D_ta(1 to num_symbols,1 to
        SAMPLES_PER_SYMBOL) := i_input;
    constant q_input_norm : integer2D_ta(1 to num_symbols,1 to
        SAMPLES_PER_SYMBOL) := q_input;
    --constant i_norm        : iq_ta(1 to SAMPLES_PER_SYMBOL);
    --constant q_norm        : iq_ta(1 to SAMPLES_PER_SYMBOL);
  begin
  --loop through all input test vectors
    for symbol in 1 to num_symbols loop
      --apply new input
      new_samples <= '1';
      for n in 1 to SAMPLES_PER_SYMBOL loop
        i(n) <= to_signed(i_input_norm(symbol,n),B_IQ);
        q(n) <= to_signed(q_input_norm(symbol,n),B_IQ);
        --i_norm(n) := to_signed(i_input_norm(symbol,n),B_IQ);
        --q_norm(n) := to_signed(q_input_norm(symbol,n),B_IQ);
        --i          <= i_norm(n);
        --q          <= q_norm(n);
      end loop;
      tick;
      new_samples <= '0';
      --clock until result arrives
      while(branch_metrics_rdy = '0') loop
        tick;
      end loop;
      --check for error in result
      for bmid in branch_metrics'low to branch_metrics'high loop
        if (to_integer(branch_metrics(bmid)) /= correct_result(symbol
            ,bmid)) then
          errors := errors + 1;
        end if;
      end loop;
      --errors := errors + isequal(to_integer(branch_metrics),
          correct_result(symbol));
    end loop;
```

```
    assert errors = 0 report "test_filter_bank: branch metric errors
        found." severity failure;
  end procedure;

  --This test more closely represents how the filter will be used in
      practice. To keep clock rate to a
  --minimum, the input samples will be applied at the max rate - e.g.
      for 7 bit inputs then a new sample is
  --input once every 7 clocks. The result comes out with a latency of
      2 cycles.
  procedure test_filter_bank( constant i_input, q_input :
      integer2D_ta;
                              constant correct_result   :
                                  integer2D_ta;
                              constant clks_per_input   : natural) is
    variable errors           : natural := 0;
    variable num_symbols      : natural := i_input'length(1);
    variable last_result_read : boolean := false;
    variable clk_count, symbol_input_idx, bm_check_idx : natural :=
        0;
    --normalise ranges
    constant i_input_norm : integer2D_ta(0 to num_symbols-1,1 to
        SAMPLES_PER_SYMBOL) := i_input;
    constant q_input_norm : integer2D_ta(0 to num_symbols-1,1 to
        SAMPLES_PER_SYMBOL) := q_input;
  begin
    --loop until all symbols have been input and the last result has
        been read
    last_result_read  := false;
    clk_count         := clks_per_input;
    symbol_input_idx  := 0;
    bm_check_idx      := 1;
    while(not last_result_read) loop
      --input symbol at max rate for this filter
      if (clk_count = clks_per_input) then
        clk_count := 0;
        --input a symbol to the dut
        new_samples <= '1';
        for n in 1 to SAMPLES_PER_SYMBOL loop
          i(n) <= to_signed(i_input_norm(symbol_input_idx,n),B_IQ);
          q(n) <= to_signed(q_input_norm(symbol_input_idx,n),B_IQ);
        end loop;
        symbol_input_idx := (symbol_input_idx + 1) mod num_symbols;
            --inc ptr to next symbol to input
      else
        new_samples <= '0';
      end if;

      --read design output when available, and check if its correct
      if (branch_metrics_rdy = '1') then
        --check for error in result
        for bmid in branch_metrics'low to branch_metrics'high loop
          if (to_integer(branch_metrics(bmid)) /= correct_result(
              bm_check_idx,bmid)) then
            errors := errors + 1;
          end if;
        end loop;
        --check if the last result has been read
        last_result_read := (bm_check_idx >= num_symbols);
        bm_check_idx := bm_check_idx + 1;
      end if;
```

```
        --clock the design
        tick;
        clk_count := clk_count + 1;
      end loop;
      assert errors = 0 report "test_filter_bank: branch metric errors
          found." severity failure;
    end procedure;

    constant ANS_SIMPLE   : integer2D_ta(1 to 3,1 to 4) :=
        ((-63,3,63,36),(-126,6,126,72),(-189,9,189,108));
    constant I_SIMPLE     : integer2D_ta(1 to 3,1 to 2) := ((1,0),
        (2,0), (3,0));
    constant Q_SIMPLE     : integer2D_ta(1 to 3,1 to 2) := ((0,0),
        (0,0), (0,0));

  begin
    for n in i'low to (i'low+SAMPLES_PER_SYMBOL-1) loop
      i(n) <= (others => '0');
      q(n) <= (others => '0');
    end loop;
    new_samples  <= '1'; --hold filter bank in reset with this control
    GenReset(5);

    --test_filter_bank_simple( I_SIMPLE, Q_SIMPLE, ANS_SIMPLE );    --
        don't forget to rename pkg_mf_coeffs_simpletest
    test_filter_bank(i_into_bm, q_into_bm, bm_result_unrounded, B_IQ);
    tick;
    tick;
    tick;

    assert false
      report "sim end"
      severity failure;
  end process;
end branch_metrics_dafir_tb_arch;
```

D.1.6 Test Vectors Package

`/fpga/src/pkg_matlab_test_vectors.vhd`

D.2 Viterbi Trellis Path Metrics

D.2.1 Synthesis Top Level

`/fpga/src/cpm_viterbi_decoder_synth.vhd`

D.2.2 Top Level

```
--
    ****************************************************************************************
-- Title:                          CPM viterbi decoder
-- Author:                         Andrew Bridger
-- Date:                           24 September 2008
-- High Level Module Description: This module peforms the viterbi
    decoder processing for the complete
-- CPM trellis. This module instantiates several radix-4 units that
    carry out the state metric
```

```
-- add-compare-select processing. The state metrics between radix-4
    units are interconnected based
-- on a matlab generated vhdl description of the CPM trellis.

-- This module currently expects branch metrics as inputs, and outputs
    decision bits every viterbi iteration.
-- These decision bits identify the survivor path for each state. A
    traceback unit(external to this module)
-- would store and process these bits further, before outputting
    decoded data.
-- (todo - likely in future the bms will be brought in here to tie
    placement to the radix4 unit placement)
--
-- The branchmetrics must be supplied at the correct time. The current
    implementation requires the 1st
-- bms to be supplied on the 3/4 cycles following new_viterbi_iteration
     . Since a DAFIR BM implementation
-- only has the bms valid for 1 clock cycle, they must be supplied on
    precisely the right clock cycle,
-- indeed the cycle they are read.
--
-- Various implemenations are supported using if .. generate statements
     and a generic parameter to select
-- the desired implementation.

-- Notes/Limitations:

-- Synthesizable: Yes
--
-- Testbench: cpm_viterbi_decoder_tb.vhd and also some test code
    embedded in this module itself.
--
-- Note: The version control system in use is the repository for
    information regarding bug fixes,
-- versions, feature additions etc.
--
    ****************************************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.PkgStdType.all;
use work.pkg_math.all;
use work.pkg_rloc.all;
use work.pkg_project.all;
use work.pkg_trellis.all;
-- synthesis translate_off
use work.pkg_matlab_test_vectors_viterbi.all;
use work.pkg_to_string.all;
-- synthesis translate_on

library unisim;
use unisim.vcomponents.all;

entity cpm_viterbi_decoder is
  generic(
    PLACE                    : natural := 0;     --default unplaced
    DESIGN_TYPE              : natural := 0;      --currently not
        supported - perhaps should be enumerated type
```

```
    CHECK_STATE_METRICS        : boolean := false
    );
  port (
    clk                     :in   std_logic;
    reset                   :in   std_logic;            --synchronously
        resets source state metrics to 0
    new_viterbi_iteration   :in   std_logic;
    branch_metrics          :in   branch_metrics_rounded_ta(NUM_FILTERS*2
         downto 1);
    decision_bits           :out  std_logic_vector(
        DECISION_BITS_BUS_WIDTH-1 downto 0);  --Used during traceback
        to determine trellis path taken
    decision_bits_rdy       :out  std_logic             --High when
        decision_bits are first valid
    );
end cpm_viterbi_decoder;

architecture rtl of cpm_viterbi_decoder is
  --General purpose search function. Future move to a library?
  --Finds the first occurance of the item in search_me, returns
      location row in 1 and column in 2
  --If item not found returns (0,0) - which may not be part of
      search_me of course.
  --This means the user must assume the item is in the array, which
      really limits the usefulness of this
  --function. Future fix up.
  function find_first_occurance( constant item      : integer;
                                 constant search_me : integer2D_ta )
                                 return integer_ta is
    variable found_location : integer_ta(1 to 2) := (0,0);
  begin
    for row in search_me'range(1) loop
      for col in search_me'range(2) loop
        if search_me(row,col) = item then
          found_location := (row, col);
          return found_location;
        end if;
      end loop;
    end loop;
    assert false report "find_first_occurance: did not find item."
        severity failure;
    return found_location;
  end function;

  --given a state metric, find which bus cycle its valid in.
  --This must match the radix4_acs implementation obviously.
  function stateid_to_bus_cycle( constant state_id : positive;
                                 constant output_state_mapping :
                                     integer2D_ta
                                 ) return natural is
    variable output_sm_port    : integer            := 0;
    variable locationyx        : integer_ta(1 to 2)  := (0,0);
  begin
    --find which cycle the sm is valid on by finding which cycle it was
         generated on by:
    --1)find out which radix 4 unit sm port generated it, by searching
        in the output_state_mapping
    --table. (there are 4 ports per radix4 unit mapped onto 2 buses).
    locationyx       := find_first_occurance( state_id,
        output_state_mapping );
    output_sm_port   := locationyx(2);    --each column is a new port
```

```
  --2)ports 1,3 are on the first bus cycle, ports 2,4 are on the
      second cycle
  if (output_sm_port = 1) or (output_sm_port = 3) then
    return 0;
  else
    return 1;
  end if;
end function;

--Each radix4 unit latches statemetrics on different bus cycles. (1
    of 2 possible currently).
--Generate the config array that specifies which sms are valid on
    which bus cycles for a
--specific radix4 unit.
function generate_smvalid_buscycle_table( constant radix4_id
              : positive;
                                          constant
                                              output_state_mapping :
                                              integer2D_ta;
                                          constant
                                              input_state_mapping  :
                                              integer2D_ta
                                          ) return natural_ta is--
                                              sm_valid_bus_cycle_t is
  variable valid_bus_cycle  : natural_ta(3 downto 0) := (0,0,0,0);
  variable source_state     : integer := 0;
begin
  --for each source state in the radix 4 unit
  for j in 1 to 4 loop
    source_state          := input_state_mapping(radix4_id,j);
    valid_bus_cycle(j-1)  := stateid_to_bus_cycle(source_state,
        output_state_mapping);
  end loop;
  return valid_bus_cycle;
end function;

--Generate a table showing which bus cycle each state_metric is valid
     on. Used by the test code
--at the end of this module.
function stateid_to_buscycle_table( constant output_state_mapping :
    integer2D_ta
                                   ) return natural_ta is
  constant NUM_STATES : positive := output_state_mapping'length(1) *
      output_state_mapping'length(2) ;
  variable table      : natural_ta(NUM_STATES downto 1) := (others =>
       99);
begin
  --for each state_metric
  for i in 1 to NUM_STATES loop
    --find which bus cycle its output in
    table(i) := stateid_to_bus_cycle(i, output_state_mapping );
  end loop;
  return table;
end function;

--constants/signals
constant  ACS_UNITS_PER_RADIX4_UNIT       : positive := 2;
constant  STATE_METRICS_PER_RADIX4_UNIT   : positive := 4;
constant  BRANCH_METRICS_PER_RADIX4_UNIT  : positive := 16;

signal    state_metrics                   : state_metrics_ta(
    NUM_VITERBI_STATES downto 1);
```

```
  --signal    state_metric_rdy               : std_logic_vector(
      NUM_RADIX4_UNITS downto 1);

begin
  --Implement the viterbi trellis processing by instantiating radix-4
      units, and wiring up the output
  --state metrics to input state metrics as defined by the CPM trellis.
  place_radix4acs_units: for radix4_id in 1 to NUM_RADIX4_UNITS
      generate
    --placement.
    attribute RLOC : string;

    --place R4 units in a square shape, bounded top and bottom by the
        dcm edges, and from the left
    --bram/mult column.
    constant Y_SLICES_PER_R4     : natural := ((B_SM+1)/2) * 4 + 1;  --
        vhdl rounds down on /
    constant X_SLICES_PER_R4     : natural := 10; --actually 9, but
        round up to even number to get L/M slice alignment right
    constant AVAILABLE_Y_SLICES  : natural := 160; --XC3SD1800A from
        bottom dcms edge to top dcms edge.

    constant R4_PER_COLUMN              : natural := AVAILABLE_Y_SLICES /
         Y_SLICES_PER_R4;
    constant IDX                        : natural := radix4_id -1;  --zero
         referenced.
    constant X_RLOC                     : natural := (IDX/R4_PER_COLUMN)
            * X_SLICES_PER_R4;
    constant Y_RLOC                     : natural := (IDX mod
        R4_PER_COLUMN)  * Y_SLICES_PER_R4;
    attribute RLOC of radix4_acs_unit : label is pick_string(PLACE,
        xy_str(X_RLOC,Y_RLOC));

    signal source_state_metrics         : state_metrics_ta(
        STATE_METRICS_PER_RADIX4_UNIT downto 1);
    signal dest_state_metric_bus        : state_metrics_ta(
        ACS_UNITS_PER_RADIX4_UNIT downto 1);
    signal radix4_branch_metrics        : branch_metrics_rounded_ta(
        BRANCH_METRICS_PER_RADIX4_UNIT downto 1);
    --Branch metrics are shared by two radix-4 units. One of the radix
        4 units is configured to subtract
    --the branch metrics. Radix4 unit Ids map to a (thetaD,a(n-1))
        tuple. The radix4 unit that is
    --(thetaD + pi, a(n-1)) is the r4 unit that shares the branch
        metrics and is configured to subtract them.
    --(thetaD + pi, a(n-1)) maps to an r4 unit id that is
        NUM_RADIX4_UNITS/2 greater than the first r4 id.
    constant SUBTRACT_BRANCH_METRICS  : boolean   := radix4_id > (
        NUM_RADIX4_UNITS/2);
    function bms_radix4_id_func( radix4_id : natural range 1 to
        NUM_RADIX4_UNITS ) return positive is
    begin
      if radix4_id <= (NUM_RADIX4_UNITS/2) then
        return radix4_id;
      else
        return radix4_id - (NUM_RADIX4_UNITS/2);
      end if;
    end function;
    constant bms_radix4_id: natural := bms_radix4_id_func( radix4_id );
  begin
    --assert false report "radix4_id =" & Image(radix4_id) severity
        warning;
```

```
--radix-4 units that do the add-compare-select viterbi processing
radix4_acs_unit: entity work.radix4_acs(rtl)
  generic map(
    PLACE                    => 0,
    DESIGN_TYPE              => 1, -- 0 is vhdl, 1 is primitives
    SUBTRACT_BRANCH_METRICS => SUBTRACT_BRANCH_METRICS,
    SM_VALID_BUS_CYCLE       => generate_smvalid_buscycle_table(
        radix4_id,
                                                              work
                                                                  .
                                                                  pkg_trellis
                                                                  .
                                                                  radix4_output_state_mapping
                                                                  ,

                                                              work
                                                                  .
                                                                  pkg_trellis
                                                                  .
                                                                  radix4_input_state_mapping
                                                                  )
                                                                  )

  port map(
    clk                  => clk,
    reset                => reset,                  --synchronously
         resets source state metrics to 0
    new_viterbi_iteration => new_viterbi_iteration, --high for 1
        clock cycle, new source_sms are ready for input
    source_state_metrics  => source_state_metrics,  --source state
        metrics - dest sms from previous viterbi iteration
    branch_metrics        => radix4_branch_metrics,
    dest_state_metric     => dest_state_metric_bus, --state metric
        bus output from this viterbi iteration
    --Used during traceback to determine trellis path taken
    decision_bits         => decision_bits(4*radix4_id -1 downto
        4*(radix4_id -1))
    );
--branch metric wiring. Hookup the branch metric hardware to the
    right radix-4 unit bm inputs.
--The upper half of radix4 ids reuse the bms of the lower half
input_bm_wiring: for radix4_bmid in 1 to
    BRANCH_METRICS_PER_RADIX4_UNIT generate
begin
  radix4_branch_metrics(radix4_bmid) <=
    branch_metrics( work.pkg_trellis.radix4_bm_mapping(
        bms_radix4_id, radix4_bmid) );
    --branch_metrics( work.pkg_trellis.radix4_bm_mapping(radix4_id,
         radix4_bmid) );
end generate;

--state metric wiring. Connect radix-4 unit state metric bus
    outputs back to the state metric
--inputs.
--map state metrics ids to the radix4 acs inputs
input_sm_wiring: for radix4_input_id in 1 to
    STATE_METRICS_PER_RADIX4_UNIT generate
begin
  source_state_metrics(radix4_input_id)
    <= state_metrics( work.pkg_trellis.radix4_input_state_mapping(
        radix4_id, radix4_input_id) );
end generate;
```

```
--map radix4 acs outputs to state metric ids
output_sm_wiring: for radix4_output_id in 1 to
    STATE_METRICS_PER_RADIX4_UNIT generate
begin
  state_metrics( work.pkg_trellis.radix4_output_state_mapping(
      radix4_id, radix4_output_id) )
    <= dest_state_metric_bus((radix4_output_id+1)/
        ACS_UNITS_PER_RADIX4_UNIT);
end generate;

end generate;

assert false report "warning: bms hooked up temporaily - set
    correctly! also the +ve/-ve acs config too" severity warning;

--Control the radix-4 units and organise the viterbi iteration timing
    . This has been "hard coded" and
--assumes branch metrics are input every 7 clock cycles, and the
    radix-4 acs unit takes 7 clock cycles
--to complete a viterbi iteration. Also assumed the clock rate is
    matched to 7x the symbol rate.
--In general these assumptions may not be true, e.g. clock faster
    than required, mismatch between
--branch metric and viterbi cycles required. And so in the future it
    may be worthwhile re-writing this
--control logic to work with these assumptions relaxed.
--new_viterbi_iteration <= branch_metrics_rdy; --only correct since
    both viterbi and bm take 7 clocks

--upper layer responsibility to activate reset at the right time.
    Future improvement may be to
--synchronise this with a start of viterbi iteration so results of
    this reset are predictable. E.g.
--resetting part way through a cycle will have no effect, since the
    state metrics get re-written
--at the end of the viterbi iteration.
--reset_source_sm       <= reset_source_sm;

--The first state metric and decision bits are ready 6 cycles after
    the start of the viterbi iteration.
--The 2nd stat metic ready on the 7th cycle.
--BETTER IDEA> just use a shift reg, insert new_vit_iter at the front
     - extended an extra clock,
--then bits rdy is just the output of the shift reg, shift reg is the
     length of the pipeline long.
--Conceptually very easy to understand!!

--Generate a ready signal that indicates when decision bits are valid
    . Every time new-viterbi_iteration
--goes high, a new iteration starts with bus cycles to input the
    state metrics to the radix-4 acs
--units. Then the radix-4 pipeline proceses the sms and produces the
    decision bits after some latency.
--Use a simple shift reg to represent the pipeline: its depth is the
    same as the real pipeline, and
--new_viterbi_iteration is fed in the front, and the decision bits
    ready signal is therefore just
--the output of the shift reg.
generate_decision_bits_rdy: process( clk )
  variable shift_reg         : std_logic_vector(R4_SM_INPUT_BUS_CYCLES
      + R4_PIPELINE_DEPTH -2 downto 0);
```

```
    variable shift_in          : std_logic;
    variable bus_cycle_count   : positive range 1 to
        R4_SM_INPUT_BUS_CYCLES;
  begin
    if rising_edge( clk ) then
      --shift reg serial input
      if reset = '1' then
        shift_in := '0';
        bus_cycle_count := R4_SM_INPUT_BUS_CYCLES;  --set default value
            , and to come out of reset cleanly
      elsif new_viterbi_iteration = '1' then
        shift_in := '1';
        bus_cycle_count := 1;
      elsif bus_cycle_count < R4_SM_INPUT_BUS_CYCLES then --extend 1's
          input for duration of bus cycles
        shift_in := '1';
        bus_cycle_count := bus_cycle_count + 1;
      else
        shift_in := '0';
      end if;
      --shift
      if reset = '1' then --precludes srl16
        shift_reg := (others => '0');
      else
        shift_reg := shift_reg(shift_reg'high-1 downto shift_reg'low) &
            shift_in;
      end if;
      --output
      decision_bits_rdy <= shift_reg( shift_reg'high );
    end if;
  end process;

  --Verify correct viterbi decoder operation by checking the state
      metrics calculated at the end of
  --every viterbi iteration. They are checked against statemetrics
      generated in the matlab fixed point
  --model. This code is ignored for synthesis, and can also be "
      switched off" using the CHECK_STATE_METRICS
  --generic parameter.
  -- synthesis translate_off
  chk_state_metrics: process( clk )
    variable viterbi_iteration                   : natural := 1;
    variable error, good_iteration_count         : natural := 0;
    variable implementation_answer, matlab_answer, difference :
        state_metric_t;
    variable valid_state_metrics                 : state_metrics_ta(
        NUM_VITERBI_STATES downto 1);
    constant stateid_to_buscycle_tbl             : natural_ta
             := stateid_to_buscycle_table( work.pkg_trellis.
                 radix4_output_state_mapping );
    variable bus_cycle         : natural              := 0;
    variable got_all_sms       : boolean              := false;
    variable pipeline_state    : string(13 downto 1)  := "xxxxxxxxxxxxx";
    variable cycle             : natural              := 0;
    variable test_bus_cycle    : natural              := 55;
    constant PRECISE_MATCH_REQUIRED : boolean := false; --set to false
        if a difference of 1 is acceptable
                                                     --input test
                                                         vectors
                                                         occasiaonally
                                                          have the -
                                                         ve BMS out
```

```
                                                      by 1 - think
                                                       its a
                                                      rounding
                                                      issue in the
                                                       matlab
                                                      model?
begin
  if rising_edge( clk ) then
    if CHECK_STATE_METRICS then
      if reset = '1' then
        viterbi_iteration      := 0;
        good_iteration_count   := 0;
        bus_cycle              := 100;
        for stateid in valid_state_metrics'range loop
          valid_state_metrics( stateid ) := (others => '1');
        end loop;
        assert PRECISE_MATCH_REQUIRED
          report "cpm_viterbi_decoder: warning, statemetrics are
              checked to within +/- 2. Precise match not required"
          severity warning;
      elsif is_x(std_logic_vector(state_metrics(1))) then
      --don't run test if inputs not defined
        assert false report "cpm_viterbi_decoder: state_metrics xxx
            so results not checked" severity warning;
      else
        --At the beginning of every viterbi iteration, gather a
            complete set of state metrics. These
        --are produced over 1 or more cycles, depending on the
            radix4_acs implementation. I.e. the
        --state_metrics array is only valid for specific states on
            specific cycles.
        --First determine which cycle we are on.
        --got_all_sms := false;
        if new_viterbi_iteration = '1' then
          bus_cycle := 0;
        else
          bus_cycle := bus_cycle + 1;
        end if;
        --Loop through all metrics, see if we are on the cycle for
            that sm, if yes then read it.
        if (bus_cycle = 0) or (bus_cycle = 1) then
          for stateid in 1 to NUM_VITERBI_STATES loop
            if (bus_cycle) = stateid_to_buscycle_tbl(stateid) then
              valid_state_metrics(stateid) := state_metrics(stateid);
            end if;
          end loop;
        end if;
        --Once got all sms from all the bus cycles, check them
            against the matlab data.
        got_all_sms := (bus_cycle = 2);
        if got_all_sms then
          if viterbi_iteration /= 0 then --ignore 1st iteration
            error := 0;
            for state_id in 1 to NUM_VITERBI_STATES loop
              implementation_answer := valid_state_metrics(state_id);
              matlab_answer         := to_signed(work.
                  pkg_matlab_test_vectors_viterbi.
                                       viterbi_state_metrics(
                                           viterbi_iteration,
                                           state_id),B_SM);
              difference := implementation_answer - matlab_answer;
              if PRECISE_MATCH_REQUIRED then
```

```
              if difference /= 0 then
                error := error + 1;
              end if;
            else
              if (difference < -2) or (difference > 2) then --diff
                  of 1 still lots of errors?
                error := error + 1;
              end if;
            end if;
          end loop;
          --assert every viterbi iteration with errors.
          if error = 0 then
            good_iteration_count := good_iteration_count + 1;
          end if;
          assert error = 0
            report "cpm_viterbi_decoder: implementation results do
                 not match matlab results. " &
                    "Iteration=" & work.pkg_to_string.Image(
                        viterbi_iteration ) & ", " &
                    "Good iterations=" & work.pkg_to_string.Image(
                        good_iteration_count ) & ", " &
                    "Statemetrics in error this iter=" & work.
                        pkg_to_string.Image(error)
            severity failure;
        end if;
        viterbi_iteration := viterbi_iteration + 1;
      end if;
    end if;
    --Aid debug by providing a string labeling each cycle
    if (reset = '1') or (new_viterbi_iteration = '1') then
      cycle := 0;
    else
      cycle := cycle + 1;
    end if;
    case cycle is
      when 0 => pipeline_state     := "sm_bus_cycle0";
      when 1 => pipeline_state     := "sm_bus_cycle1";
      when 2 => pipeline_state     := "sm_plus_bm    ";
      when 3 => pipeline_state     := "compare1      ";
      when 4 => pipeline_state     := "mux1          ";
      when 5 => pipeline_state     := "compare2      ";
      when 6 => pipeline_state     := "mux2          ";
      when others => pipeline_state := "undefined     ";
    end case;
  end if;
  --test
  --test_bus_cycle := stateid_to_bus_cycle(37, work.pkg_trellis.
      radix4_output_state_mapping );
 end if;
end process;
-- synthesis translate_on
end rtl;
```

D.2.3 Radix-4 Add-Compare-Select Unit

```
--
    ************************************************************************************************
-- Title:                          Radix-4 viterbi state metrics unit,
    contains 1 or more ACS units
-- Author:                         Andrew Bridger
-- Date:                           18 September 2008
```

```
-- High Level Module Description:   This modules groups together the
    viterbi add-compare-select processing
-- for 4 states of a quaternary(M=4) alphabet CPM trellis. This "radix
    -4" state composition is chosen
-- such that all four output state metrics share the same four source
    state metrics input to this module.
-- This simplifies the state metric routing complexity because now the
    routing from any source state metric
-- register is local to this radix-4 unit only.
--
-- The output state metric routing is still an issue however, since it
    is non-local. That issue is partly
-- addressed at a higher level in the design hierarchy, by appropriate
    grouping of radix-4 units, and their
-- relative layout.

-- This radix-4 unit contains 1 or more acs units that carry out the
    required processing.

-- Given source state metrisc ssm1-ssm4, destination state metrics dsm1
    -dsm4 and branch metrics bm1-bm16,
-- this unit calculates:
-- dsm1 = max{ssm1+bm1, ...,ssm4+bm4}
-- dsm2 = max{ssm1+bm5, ...,ssm4+bm8}
-- dsm3 = max{ssm1+bm9, ...,ssm4+bm12}
-- dsm4 = max{ssm1+bm13, ...,ssm4+bm16}

--For the 2x ACS implementation, dsm1 and then dsm2 are output on
    dest_state_metric bus 0
-- dsm3 and then 4 are output on dest_state_metric bus 1

-- The new_viterbi_iteration signal synchronises the operations that
    occur within this radix4 unit. This
-- signal goes active for 1 clock cycle, when the first cycle of
    statemetrics are valid at this
-- modules inputs. The second cycle of state metrics are input on the
    next cycle.
-- The branch metric inputs must be input in lock step with the
    statemetric input cycles.

-- Various implemenations are supported using if .. generate statements
     and a generic parameter to select
-- the desired implementation.

-- Notes/Limitations:  1) Consider putting decision bits in a known
    state at reset. Perhaps not essential
--                       but a nice thing to do for downstream modules?
--                     2) Consider moving scalarize and vectorize into
    a utilities package.
--silly functions so that std_logic can be applied to unconstrained
    vector ports. In future move
--these into a package. These will be required anytime you have a
    component defined with
--unconstrained slv, yet there may be cases where you want to pass in
    just 1 bit, e.g. a std_logic.
--
-- Synthesizable: Yes
--
-- Testbench: radix4_acs_tb
--
-- Note: The version control system in use is the repository for
    information regarding bug fixes,
```

```
-- versions, feature additions etc.
--
    ****************************************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.PkgStdType.all;
use work.pkg_project.all;
use work.pkg_rloc.all;
use work.pkg_math.all;


library unisim;
use unisim.vcomponents.all;

entity radix4_acs is
  generic(
    PLACE                     : natural := 0;     --default unplaced
    DESIGN_TYPE               : natural := 0;     --0 is vhdl, 1 is
        xilinx primitives
    SUBTRACT_BRANCH_METRICS : boolean := false; --if true subtract
        rather than add bms to state metrics
    --Configure the bus cycle to which each input state metric is
        registered on.
    SM_VALID_BUS_CYCLE        : natural_ta(3 downto 0) := (0,0,0,0)
    );
  port (
    clk                    :in   std_logic;
    reset                  :in   std_logic;             --synchronously
        resets source state metrics to 0
    new_viterbi_iteration  :in   std_logic;             --high for 1
        clock cycle, when first source sms are ready for input
    source_state_metrics   :in   state_metrics_ta(3 downto 0); --source
        state metrics - dest sms from previous viterbi iteration
    branch_metrics         :in   branch_metrics_rounded_ta(15 downto 0);
    dest_state_metric      :out  state_metrics_ta(1 downto 0); --state
        metric bus output from this viterbi iteration
    decision_bits          :out  std_logic_vector(3 downto 0)  --Used
        during traceback to determine trellis path taken
    );
end radix4_acs;

architecture rtl of radix4_acs is
  constant CHECK_ANSWER           : boolean := true;  --set to true for
      debug, false may speed up simulation
  constant PLACE_SUB_INSTANCE     : natural := PLACE;


  --synthesis translate off
  signal pipeline_state : string(13 downto 1) := "xxxxxxxxxxxxx";
  --synthesis translate on

  --Control to override PLACE control to force sub-instances to be
      placed.
  --constant FORCE_SUB_INSTANCE_PLACE_TRUE  : boolean := true;  -- <--
      normal value should be false.
  --constant PLACE_SUB_INSTANCE             : natural := PLACE when (
      not FORCE_SUB_INSTANCE_PLACE_TRUE) else 1;
```

```
begin

  --synthesis translate off
  --debug aid that prints the pipeline cycle into a string.
  pipeline_state_string: process( clk )
    variable cycle: integer :=0;
  begin
    if rising_edge(clk) then
      if (reset = '1') or (new_viterbi_iteration = '1') then
        cycle := 0;
      else
        cycle := cycle + 1;
      end if;
      case cycle is
        when 0 => pipeline_state      <= "sm_bus_cycle0";
        when 1 => pipeline_state      <= "sm_bus_cycle1";
        when 2 => pipeline_state      <= "sm_plus_bm    ";
        when 3 => pipeline_state      <= "compareA      ";
        when 4 => pipeline_state      <= "muxA          ";
        when 5 => pipeline_state      <= "compareB      ";
        when 6 => pipeline_state      <= "muxB          ";
        when others => pipeline_state <= "xxxxxxxxxxxxx";
      end case;
    end if;
  end process;
  --synthesis translate on

  --VHDL VERSION OF 2xACS RADIX4 UNIT. (Xilinx primitives version
      further below.)
  vhdl: if DESIGN_TYPE = 0 generate
    signal source_state_metrics_reg  : state_metrics_ta(3 downto 0);
    signal select_first_bm           : boolean;
  begin
    --Use two ACS units to generate the 4 state metrics outputs from
        this radix-4 unit. Each ACS unit
    --outputs two state metrics sequentially.
    radix4_acs: for i in 0 to 1 generate
    begin
      --peform add-compare-select processing for 2 states using 1 set
          of hardware. Pipeline is 6 deep, plus
      --1 clock for the 2nd state, so takes 7 clocks for complete
          processing.
      --Path metrics are stored as fixed point 2's complement numbers
          and have limited range. By using
      --Hekstra's method, dedicated normalisation logic is avoided by
          allowing the metrics to overflow and
      --performing the find-the-best-metric operation using 2's
          complement arithmetic. See the Matlab model for
      --more details.
      four_way_acs_vhdl_2states: process( clk )
        variable keep_pm23, keep_pm1, keep_pm3          : boolean;
        variable keep_pm1_pipe1, keep_pm1_pipe2, keep_pm3_pipe1,
            keep_pm3_pipe2 : boolean;
        variable pm0_reg, pm1_reg, pm2_reg, pm3_reg    : state_metric_t;
        variable pm01_reg, pm23_reg                    : state_metric_t;
        variable pm0, pm1, pm2, pm3, pm01, pm23        : state_metric_t;
        type branch_metric_muxed_ta is array(3 downto 0) of
            branch_metric_rounded_t;
        variable branch_metric                         :
            branch_metric_muxed_ta;
```

```
  --variable dest_state_metric_noreg                  :
      state_metric_t;
begin
  if rising_edge( clk ) then
    --READ BOTTOM UP
    --using variables so read from the bottom up to get a sense
        of the dataflow and pipelining.
    --The comments also make more sense if read from bottom up.
    --TODO DELETE6)Register the output (destination) best metric.
         Gives whole flop to flop timing path for routing.
    --dest_state_metric(i) <= dest_state_metric_noreg;

    --7)Finally, select (mux) the best metric
    if keep_pm23 then
      dest_state_metric(i) <= pm23_reg;
    else
      dest_state_metric(i) <= pm01_reg;
    end if;
    --Output the decision bits based on which path metric was
        kept. There are two decison bits for
    --each ACS unit. Encoded as follows.
    --pm0 - "00"
    --pm1 - "01"
    --pm2 - "10"
    --pm3 - "11"
    decision_bits(1+ 2*i downto 2*i) <= "00";
    if keep_pm23 then
      decision_bits(1 + 2*i) <= '1';     --msb
      if keep_pm3_pipe2 then
        decision_bits(0 + 2*i) <= '1';  --lsb
      end if;
    else
      if keep_pm1_pipe2 then
        decision_bits(0 + 2*i) <= '1';  --lsb
      end if;
    end if;
    --6)Peform the subtraction comparison between the best 2 path
         metrics
    keep_pm23   := (pm01 - pm23) < 0;
    pm01_reg    := pm01;
    pm23_reg    := pm23;
    keep_pm1_pipe2 := keep_pm1_pipe1;
    keep_pm3_pipe2 := keep_pm3_pipe1;
    --5)Select (mux) the best metric based on the sign bit of the
         subtraction
    if keep_pm1 then
      pm01 := pm1_reg;
    else
      pm01 := pm0_reg;
    end if;
    if keep_pm3 then
      pm23 := pm3_reg;
    else
      pm23 := pm2_reg;
    end if;
    keep_pm1_pipe1 := keep_pm1; --pipeline the decision bits -
        used later in the pipeline
    keep_pm3_pipe1 := keep_pm3;
    --4)Now start the process of finding the "best" path metric.
        This is done in two stages, firstly
    --finding the best of each of two pais. And then in the
        second stage finding the best of the
```

```
        --remaining two path metrics.
        --The key part of the matlab model for finding the "best"
            path metric is:
        --      subtractAns = val - x(i+1);
        --      if subtractAns < 0 %val is smaller
        --        val = x(i+1);
        --        idx = i+1;
        --This is just a subtraction followed by the <0 check which
            is just looking at the sign bit of the
        --subtraction result.
        --Perform the subtraction for each of two pairs of the state
            metrics and pass the sign bits
        --onto the next stage. Also store the path metrics in
            registers ready for the next pipeline stage.
        keep_pm1   := (pm0 - pm1) < 0;
        keep_pm3   := (pm2 - pm3) < 0;
        pm0_reg    := pm0;
        pm1_reg    := pm1;
        pm2_reg    := pm2;
        pm3_reg    := pm3;

        --3)accumulate the path metrics allowing 2's complement
            overflow. pm = state metric + branch metric
        --Since this acs unit process two states in serial, need to
            mux in the branch metrics for the
        --2nd state. (The mux and + map into a single LUT level - at
            least they should!)
        if select_first_bm then
          branch_metric := (branch_metrics(3+8*i), branch_metrics
              (2+8*i), branch_metrics(1+8*i), branch_metrics(0+8*i));
        else
          branch_metric := (branch_metrics(7+8*i), branch_metrics
              (6+8*i), branch_metrics(5+8*i), branch_metrics(4+8*i));
        end if;
        --branch metric computation halved by recognising that for
            every branch metric, there is an
        --equivalent -ve one. All 16 branch metrics input to a radix
            -4 unit are either added or
        --subtracted, so this add/subtract configuration is static ->
             compile time.
        if SUBTRACT_BRANCH_METRICS then
          pm0 := source_state_metrics_reg(0) - branch_metric(0); --(
              does synth tool automatically sign extend?)
          pm1 := source_state_metrics_reg(1) - branch_metric(1);
          pm2 := source_state_metrics_reg(2) - branch_metric(2);
          pm3 := source_state_metrics_reg(3) - branch_metric(3);
        else --add them
          pm0 := source_state_metrics_reg(0) + branch_metric(0);
          pm1 := source_state_metrics_reg(1) + branch_metric(1);
          pm2 := source_state_metrics_reg(2) + branch_metric(2);
          pm3 := source_state_metrics_reg(3) + branch_metric(3);
        end if;
        --2), 1) Two bus cycles to register the source state metrics.
             See below.
      end if;
    end process;
  end generate;

  --The source state metrics must be registered. (Stage 1 and 2 of
      the pipeline).
  --Share one set of these registers amongst both ACS units.
```

```
--Since this radix 4 unit outputs 2 sms on one bus, there are two
    bus cycles. Therefore, each sm must
--be registered on the correct cycle. That is specified by a
    generic paramater.
register_source_sms: process( clk )
  variable shift_reg              : std_logic_vector(1 downto 0);
  variable shift_in               : std_logic;
  constant SELECT_FIRST_BM_CYCLE  : natural := 1;
  variable load                   : std_logic_vector(3 downto 0);
     --debug
begin
  --READ TOP DOWN
  if rising_edge( clk ) then
    --apologies for following being a little "tricky". Intended to
        be easy to translate into
    --primitives. The acs processing pipeline is represented with a
         shift register. Each shift
    --register location corresponds to a different stage in the
        pipeline. New_viterbi_iteration is
    --the shift reg serial input, and by simpling looking at the
        shift reg contents we can figure
    --out what pipeline stage we are at, and what pipeline control
        signals need to be generated.
    --Note, currently only need to know about the first 2 stages,
        so the shift reg doesn't need to be
    --the complete pipeline depth.
    shift_in := new_viterbi_iteration;
    if reset = '1' then
      shift_reg := (others => '0');
    else
      shift_reg := shift_reg(shift_reg'high-1 downto 0) & shift_in;
    end if;
    --Decode the pipeline state and generate the required control
        signals.
    --1) sm bus cycle 1, 2) sm bus cycle 2, 3) add 1st set of bms,
        4) now mux in the 2nd set of bms
    --A) branch metric muxing.
    if shift_reg( SELECT_FIRST_BM_CYCLE ) = '1' then   --sm bus
        cycle 2
      select_first_bm <= true;  --setup to calc with 1st set of bms
    else
      select_first_bm <= false;
    end if;
    --B) input state metric registering.
    --Loop through each input state metric, and reset or register
        it on the correct cycle.
    for i in 0 to 3 loop
      load(i) := '0';     --debug
      if reset = '1' then
        source_state_metrics_reg(i) <= (others => '0');
      elsif shift_reg( SM_VALID_BUS_CYCLE(i) ) = '1' then
        load(i) := '1';   --debug
        source_state_metrics_reg(i) <= source_state_metrics(i);
      end if;
    end loop;
  end if;
end process;
end generate;

--XILINX INSTANTIATED PRIMITIVES VERSION OF 2xACS RADIX4 UNIT
--Read the following from here(top) down to get a feel for the data
    flow.
```

```
--The data flow pipeline is: register state metrics(2 cycles) -> path
     metrics = state metrics + bms ->
--first subtract/mux (2) -> second subtract/mux level (2).
primitives: if DESIGN_TYPE = 1 generate
  --Placement
  attribute RLOC_ORIGIN : string;
  attribute RLOC  : string;
  --DEBUG temp: RLOC origin to help with placement debug. Without it
      mapper seems to split the rpm.
  --Note: RLOC_ORIGIN will find the lowest left most element in the
      RPM to which pipeline_state belongs,
  --and map that to the rloc_origin specified.
  --attribute RLOC_ORIGIN of pipeline_state: label is pick_string(
      PLACE, xy_str(4,4));

  --standard y size(in slices) of the components we are placing.
  constant YSTEP          : natural := ((B_SM+1) / 2);  --+1 required
      to round up to nearest whole slice
  constant XSTEP          : natural := 1;               --todo support
      -ve 1 for right to left layout...
  constant ygrid                              : natural_ta(3 downto 0)
      := (3*YSTEP +1,2*YSTEP+1,YSTEP,0);
  --Layout is state metric regs through to final mux left to right or
       right to left. The four
  --state metric regs are layed out vertically, with an empty slice
      row in the middle to pipeline
  --control signals, misc bits and pieces of logic.
  attribute RLOC of select_first_bm_flop0A  : label is pick_string(
      PLACE, xy_str(XSTEP,2*YSTEP));
  attribute RLOC of select_first_bm_flop0B  : label is pick_string(
      PLACE, xy_str(XSTEP,2*YSTEP));
  attribute RLOC of select_first_bm_flop1A  : label is pick_string(
      PLACE, xy_str(3*XSTEP,2*YSTEP));
  attribute RLOC of select_first_bm_flop1B  : label is pick_string(
      PLACE, xy_str(3*XSTEP,2*YSTEP));
  --It is not obvious if 4 flops is an advantage over 2 flops based
      on timing results. 1->2 flops
  --certainly brings a benifit.
  constant bms_plus_sms_xpos                  : natural_ta(1 downto 0)
      := (3*XSTEP,XSTEP);
  attribute RLOC of pipeline_state            : label is pick_string(
      PLACE, xy_str(0,2*YSTEP));

  --The following type has been defined here so that it cannot be
      used throughout the general design.
  --(instead should use the one defined in pkg_project).
  --But we need an slv rather than signed definition for this low
      level primitve stuff.
  subtype state_metric_slv_t       is std_logic_vector(B_SM-1 downto
      0);
  type state_metric_slv_ta         is array (natural range <> ) of
      state_metric_slv_t;
  signal source_state_metrics_reg  : state_metric_slv_ta(3 downto 0);
  constant ZERO_SM                 : bit_vector(B_SM-1 downto 0) := (
      others => '0');

  signal shift_reg                 : std_logic_vector(1 downto 0);
  signal next_pipeline_cycle       : std_logic_vector(2 downto 0);
  signal select_first_bm           : std_logic;
  signal load_sm                   : std_logic_vector(3 downto 0);
  constant SELECT_FIRST_BM_CYCLE   : natural := 2;
  signal select_first_bm_pipe      : std_logic_vector(3 downto 0);
```

```
begin
  --PIPELINE CONTROL STATE MACHINE
  --apologies for following being a little "tricky". Intended to be
     easy to translate into
  --primitives. The acs processing pipeline is represented with a
     shift register. Each shift
  --register location corresponds to a different stage in the
     pipeline. New_viterbi_iteration is
  --the shift reg serial input, and by simply looking at the shift
     reg contents we can figure
  --out what pipeline stage we are at, and what pipeline control
     signals need to be generated.
  --Note, currently only need to output controls on the first 2
     stages, so the shift reg doesn't
  --need to be the complete pipeline depth.
  pipeline_state: entity work.shift_reg(xprim)
  generic map(
    SHIFT_DIRECTION => LEFT, --towards msb)
    PLACE           => PLACE_SUB_INSTANCE)
        port map(
    clk         => clk,
    ce          => '1',
    load        => reset,
    din         => "00",--ZERO_SHIFT_REG,
    dout        => shift_reg,
    serial_in   => new_viterbi_iteration,
    serial_out  => open);
  --pipeline_cycle must show the next pipeline cycle so that we can
     decode it and produce controls ready
  --for the next cycle.
  next_pipeline_cycle <= shift_reg & new_viterbi_iteration;

  --Decode the pipeline state and generate the required control
     signals.
  --1) sm bus cycle one, 2) sm bus cycle two, 3) add 1st set of bms,
     4) now mux in the 2nd set of bms
  --A)branch metric muxing.
  select_first_bm <= '1' when next_pipeline_cycle(
     SELECT_FIRST_BM_CYCLE-1 ) = '1' else '0';
  --This signal is high fanout. i.e. 8 sm+bm adders and at 11 bits
     each thats 88 loads. Duplicate
  --registers to reduce the fanout to bring this net off the timing
     critical path. (select_first_bm is
  --generated 1 cycle early, these buffer flops then delay by 1 cycle
      to align it to the pipeline
  --correctly).
  select_first_bm_flop0A: FDRSE
  generic map ( INIT => '0')
  port map( C   => clk,
            D   => select_first_bm,
            Q   => select_first_bm_pipe(0),
            CE  => '1',
            R   => '0',
            S   => '0');
  select_first_bm_flop0B: FDRSE
  generic map ( INIT => '0')
  port map( C   => clk,
            D   => select_first_bm,
            Q   => select_first_bm_pipe(1),
            CE  => '1',
            R   => '0',
            S   => '0');
```

```
select_first_bm_flop1A: FDRSE
generic map ( INIT => '0')
port map( C    => clk,
          D    => select_first_bm,
          Q    => select_first_bm_pipe(2),
          CE   => '1',
          R    => '0',
          S    => '0');
select_first_bm_flop1B: FDRSE
generic map ( INIT => '0')
port map( C    => clk,
          D    => select_first_bm,
          Q    => select_first_bm_pipe(3),
          CE   => '1',
          R    => '0',
          S    => '0');

--B)The source state metrics must be registered. (Stage 1 and 2 of
    the pipeline).
--Share one set of these registers amongst both ACS units.
--Since this radix 4 unit outputs 2 sms on one bus, there are two
    bus cycles. Therefore, each sm must
--be registered on the correct cycle. That is specified by a
    generic paramater.
--Loop through each input state metric, and reset or register it on
     the correct cycle.
load_sm(0) <= next_pipeline_cycle( SM_VALID_BUS_CYCLE(0) );
load_sm(1) <= next_pipeline_cycle( SM_VALID_BUS_CYCLE(1) );
load_sm(2) <= next_pipeline_cycle( SM_VALID_BUS_CYCLE(2) );
load_sm(3) <= next_pipeline_cycle( SM_VALID_BUS_CYCLE(3) );

--ONE SET OF STATE METRIC REGISTERS
sm_regs: for i in 0 to 3 generate
  attribute RLOC of sm_reg : label is pick_string(PLACE, xy_str(0,
      ygrid(i)));
begin
  sm_reg: entity work.reg(xprim)
  generic map(
    INIT    => ZERO_SM,
    PLACE   => PLACE_SUB_INSTANCE) --see pkg_rloc for definition
        port map(
    clk    => clk,
    ce     => load_sm(i),
    d      => std_logic_vector(source_state_metrics(i)),
    q      => source_state_metrics_reg(i),
    reset => reset,
    set    => '0');
end generate;

--assert PLACE = 1 report "PLACE not = to 1" severity warning;
--assert PLACE = 0 report "PLACE not = to 0" severity warning;
--assert PLACE_SUB_INSTANCE = 1 report "PLACE SUB INST not = to 1"
    severity warning;
--assert PLACE_SUB_INSTANCE = 0 report "PLACE SUB INST not = to 0"
    severity warning;
--TWO ADD-COMPARE-SELECT UNITS
--Use two ACS units to generate the 4 state metrics outputs for
    this radix-4 unit. Each ACS unit
--outputs two state metrics sequentially.
radix4_acs: for i in 0 to 1 generate
  --placement
```

```
--1st layer of subtract next to the sm+bm that generates on of
    its operands. The other operand
--is in a different ygrid row which is not ideal and is probably
    a cause of the subtract often
--being on the timing critical path.
attribute RLOC of subtract_A : label is pick_string(PLACE, xy_str
    (bms_plus_sms_xpos(i)+1,ygrid(1)));
attribute RLOC of subtract_B : label is pick_string(PLACE, xy_str
    (bms_plus_sms_xpos(i)+1,ygrid(2)));
--The first mux and all later stages are grouped such that the
    first ACS unit is on the bottom
--two ygrid rows, and the second ACS unit is on the top two ygrid
    rows in an attempt to minimise
--vertical routing distance and crossovers.
                                              --i          j
constant pm_regs_xpos : integer2D_ta(1 downto 0, 3 downto 0)
  := ((4*XSTEP,2*XSTEP,6*XSTEP,5*XSTEP),(6*XSTEP,5*XSTEP,4*XSTEP
     ,2*XSTEP));
constant pm_regs_ypos : integer2D_ta(1 downto 0, 3 downto 0) :=
  ((ygrid(3),ygrid(3),ygrid(2),ygrid(2)),(ygrid(1),ygrid(1),ygrid
     (0),ygrid(0)));

--Mux A/B and each ones associated input regs are put on the same
     row to minimise routing.
attribute RLOC of mux_A          : label is pick_string(PLACE,
    xy_str(5*XSTEP,ygrid(3*i)));
attribute RLOC of mux_B          : label is pick_string(PLACE,
    xy_str(7*XSTEP,ygrid(i+1)));
attribute RLOC of subtract_final : label is pick_string(PLACE,
    xy_str(8*XSTEP,ygrid(i+1)));
--Final mux put at far right hand side so the state metrics can
    route out here. The two regs
--that feed that final mux are place in same row to minimise
    routing.
attribute RLOC of reg_pm01       : label is pick_string(PLACE,
    xy_str(6*XSTEP,ygrid(3*i)));
attribute RLOC of reg_pm23       : label is pick_string(PLACE,
    xy_str(7*XSTEP,ygrid(3*i)));
attribute RLOC of mux_final      : label is pick_string(PLACE,
    xy_str(8*XSTEP,ygrid(3*i)));

--Place the miscelaneous control flops and decision bits flops in
     the empty slice row that
--runs through the middle of the rpm.
--Decision bits placed on the outside edge, opposite the state
    metrics. These will route out to
--traceback.
attribute RLOC of keep_pm1_pipe1_flop    : label is pick_string(
    PLACE, xy_str(2*XSTEP+2*i*XSTEP,2*YSTEP));
attribute RLOC of keep_pm3_pipe1_flop    : label is pick_string(
    PLACE, xy_str(2*XSTEP+2*i*XSTEP,2*YSTEP));
attribute RLOC of keep_pm1_pipe2_flop    : label is pick_string(
    PLACE, xy_str(5*XSTEP+i,2*YSTEP));
attribute RLOC of keep_pm3_pipe2_flop    : label is pick_string(
    PLACE, xy_str(5*XSTEP+i,2*YSTEP));
attribute RLOC of decision_bits_lsb      : label is pick_string(
    PLACE, xy_str(7*XSTEP+i,2*YSTEP));
attribute RLOC of decision_bits_msb      : label is pick_string(
    PLACE, xy_str(7*XSTEP+i,2*YSTEP));

--Silly functions so that std_logic can be applied to
    unconstrained vector ports. In future move
```

```
--these into a package. These will be required anytime you have a
     component defined with
--unconstrained slv, yet there may be cases where you want to
    pass in just 1 bit, e.g. a std_logic.
function vectorize(s: std_logic) return std_logic_vector is
  variable v: std_logic_vector(0 downto 0);
begin
  v(0) := s;
  return v;
end;
function scalarize(v: in std_logic_vector) return std_logic is
begin
  assert v'length = 1 report "scalarize: output port must be
      single bit!" severity FAILURE;
  return v(v'LEFT);
end;
signal decision_bits_lsb_dummy              : std_logic_vector(0
    downto 0);
--ACS
signal pm, pm_reg                           : state_metric_slv_ta
    (3 downto 0);
signal dest_state_metric_slv                : state_metric_slv_t;
signal keep_pm23, keep_pm1, keep_pm3        : std_logic;
signal keep_pm1_pipe1, keep_pm1_pipe2       : std_logic;
signal keep_pm3_pipe1, keep_pm3_pipe2       : std_logic;
signal pm01, pm23, pm01_reg, pm23_reg       : state_metric_slv_t;
signal pm0_minus_pm1, pm2_minus_pm3         : state_metric_slv_t;
signal pm01_minus_pm23                      : state_metric_slv_t;
begin
--peform add-compare-select processing for 2 states using 1 set
    of hardware. Pipeline is 6 deep, plus
--1 clock for the 2nd state, so takes 7 clocks for complete
    processing.
--Path metrics are stored as fixed point 2's complement numbers
    and have limited range. By using
--Hekstra's method, dedicated normalisation logic is avoided by
    allowing the metrics to overflow and
--performing the find-the-best-metric operation using 2's
    complement arithmetic. See the Matlab model for
--more details.

--Read top down to get the pipeline flow. Number on the left is
    the pipeline stage.
--1),2), Two bus cycles to register the source state metrics.
    See above.

--3)accumulate the path metrics allowing 2's complement overflow.
     pm = state metric + branch metric
--Since this acs unit process two states in serial, need to mux
    in the branch metrics for the
--2nd state. (The mux and + map into a single LUT level).
--branch metric computation halved by recognising that for every
    branch metric, there is an
--equivalent -ve one. All 16 branch metrics input to a radix-4
    unit are either added or
--subtracted, so this add/subtract configuration is static ->
    compile time.
bms_plus_sms: for j in 0 to 3 generate
  signal first_bm_sign_extended, second_bm_sign_extended :
      std_logic_vector(source_state_metrics_reg(0)'range);
  attribute RLOC of bm_plus_sm : label is pick_string(PLACE,
      xy_str(bms_plus_sms_xpos(i), ygrid(j)));
```

```
begin
  --both operands must be same length so need to sign extend the
      branch metrics. (assumption is
  --that branch metric wordlength always less than state metric
      wordlength).
  --numeric_std's function "resize" does this beautifully for you
      : its just wires, no logic
  first_bm_sign_extended  <= std_logic_vector( ieee.numeric_std.
      resize( branch_metrics(j    + 8*i), source_state_metrics_reg
      (0)'length ));
  second_bm_sign_extended <= std_logic_vector( ieee.numeric_std.
      resize( branch_metrics(j+4 + 8*i), source_state_metrics_reg
      (0)'length ));

  bm_plus_sm: entity work.adder_sub_mux(xprim)
  generic map( REG            => true,
               INIT           => ZERO_SM,
               SUBTRACT       => SUBTRACT_BRANCH_METRICS,
               CHECK_ANSWER   => CHECK_ANSWER,
               PLACE          => PLACE_SUB_INSTANCE)
        port map(
               clk    => clk,
               sel    => select_first_bm_pipe(2*i+j/2),
                           --0 selects b, 1 selects c.
               a      => source_state_metrics_reg(j),
               b      => second_bm_sign_extended,
               c      => first_bm_sign_extended,
               ans    => pm(j),                       --ans = a
                   +/- (b or c)  (sel select b or c)
               ce     => '1',                         --todo, only
                    enable for 2 cycles to help with debug and
                   may reduce power (less switching activity)
               reset => '0',
               set    => '0');
end generate;

--4)Now start the process of finding the "best" path metric. This
     is done in two stages, firstly
--finding the best of each of two pais. And then in the second
    stage finding the best of the
--remaining two path metrics.
--The key part of the matlab model for finding the "best" path
    metric is:
--     subtractAns = val - x(i+1);
--     if subtractAns < 0 %val is smaller
--       val = x(i+1);
--       idx = i+1;
--This is just a subtraction followed by the <0 check which is
    just looking at the sign bit of the
--subtraction result.
--Perform the subtraction for each of two pairs of the state
    metrics and pass the sign bits
--onto the next stage. Also store the path metrics in registers
    ready for the next pipeline stage.
subtract_A: entity work.adder_sub( xprim )
generic map( REG            => false,
             REG_CARRY_ONLY=> true,
             INIT           => ZERO_SM,
             CHECK_ANSWER   => CHECK_ANSWER,
             PLACE          => PLACE_SUB_INSTANCE)
      port map(
             clk            => clk,
```

```
                subtract        => '1',
                a               => pm(0),
                b               => pm(1),
                ans             => pm0_minus_pm1,  --ans = a - b
                ce              => '1',
                reset           => '0',
                set             => '0');
    --If sign bit of result is 1, then -ve result and so keep pm1.
        Register it.
    keep_pm1 <= pm0_minus_pm1(pm0_minus_pm1'high);
--      keep_pm1_flop: FDRSE
--      generic map ( INIT => '0')
--      port map( C   => clk,
--                D   => pm0_minus_pm1(pm0_minus_pm1'high),
--                Q   => keep_pm1,
--                CE  => '1',
--                R   => '0',
--                S   => '0');
    --2nd subtractor, now compare pm2 and pm3
    subtract_B: entity work.adder_sub( xprim )
    generic map(
      REG             => false,
      REG_CARRY_ONLY=> true,
      INIT            => ZERO_SM,
      CHECK_ANSWER    => CHECK_ANSWER,
      PLACE           => PLACE_SUB_INSTANCE)
          port map(
      clk             => clk,
      subtract        => '1',
      a               => pm(2),
      b               => pm(3),
      ans             => pm2_minus_pm3,  --ans = a - b
      ce              => '1',
      reset           => '0',
      set             => '0');
    --If sign bit of result is 1, then -ve result and so keep pm1.
        Register it.
    keep_pm3 <= pm2_minus_pm3(pm2_minus_pm3'high);
--      keep_pm3_flop: FDRSE
--      generic map ( INIT => '0')
--      port map( C   => clk,
--                D   => pm2_minus_pm3(pm2_minus_pm3'high),
--                Q   => keep_pm3,
--                CE  => '1',
--                R   => '0',
--                S   => '0');
    --now register all four pms, to balance this pipeline stage.
    pm_regs: for j in 0 to 3 generate
      attribute RLOC of pm_reg_comp : label is pick_string(PLACE,
          xy_str(pm_regs_xpos(i,j),pm_regs_ypos(i,j)));
    begin
      pm_reg_comp: entity work.reg(xprim)
      generic map(
        INIT    => ZERO_SM,
        PLACE   => PLACE_SUB_INSTANCE) --see pkg_rloc for definition
            port map(
        clk     => clk,
        ce      => '1',
        d       => pm(j),
        q       => pm_reg(j),
        reset => '0',
        set     => '0');
```

```
end generate;

--5) Select (mux) the best metric based on the sign bit of the
    subtraction
mux_A: entity work.mux2(xprim)
generic map(
  REG    => true,
  INIT   => ZERO_SM,
  PLACE  => PLACE_SUB_INSTANCE)
      port map(
  clk    => clk,
  sel    => keep_pm1,
  a      => pm_reg(0),
  b      => pm_reg(1),
  ans    => pm01,
  ce     => '1',
  reset  => '0',
  set    => '0');

mux_B: entity work.mux2(xprim)
generic map(
  REG    => true,
  INIT   => ZERO_SM,
  PLACE  => PLACE_SUB_INSTANCE)
      port map(
  clk    => clk,
  sel    => keep_pm3,
  a      => pm_reg(2),
  b      => pm_reg(3),
  ans    => pm23,
  ce     => '1',
  reset  => '0',
  set    => '0');
--pipeline the decision bits - used later in the pipeline
keep_pm1_pipe1_flop: FDRSE
generic map ( INIT => '0')
port map( C   => clk,
          D   => keep_pm1,
          Q   => keep_pm1_pipe1,
          CE  => '1',
          R   => '0',
          S   => '0');
keep_pm3_pipe1_flop: FDRSE
generic map ( INIT => '0')
port map( C   => clk,
          D   => keep_pm3,
          Q   => keep_pm3_pipe1,
          CE  => '1',
          R   => '0',
          S   => '0');

--6) Peform the subtraction comparison between the best 2 path
    metrics
subtract_final: entity work.adder_sub( xprim )
generic map(
  REG             => false,
  REG_CARRY_ONLY=> true,
  INIT            => ZERO_SM,
  CHECK_ANSWER    => CHECK_ANSWER,
  PLACE           => PLACE_SUB_INSTANCE)
      port map(
  clk             => clk,
```

```
    subtract        => '1',
    a               => pm01,
    b               => pm23,
    ans             => pm01_minus_pm23,  --ans = a - b
    ce              => '1',
    reset           => '0',
    set             => '0');
--If sign bit of result is 1, then -ve result and so keep pm23.
    Register it.
keep_pm23 <= pm01_minus_pm23(pm01_minus_pm23'high);
--      keep_pm23_flop: FDRSE
--      generic map ( INIT => '0')
--      port map( C    => clk,
--                D    => pm01_minus_pm23(pm01_minus_pm23'high),
--                Q    => keep_pm23,
--                CE   => '1',
--                R    => '0',
--                S    => '0');
--Pipeline the pms and decision bits
--pms
reg_pm01: entity work.reg(xprim)
generic map(
  INIT    => ZERO_SM,
  PLACE   => PLACE_SUB_INSTANCE) --see pkg_rloc for definition
      port map(
  clk     => clk,
  ce      => '1',
  d       => pm01,
  q       => pm01_reg,
  reset => '0',
  set     => '0');
reg_pm23: entity work.reg(xprim)
generic map(
  INIT    => ZERO_SM,
  PLACE   => PLACE_SUB_INSTANCE) --see pkg_rloc for definition
      port map(
  clk     => clk,
  ce      => '1',
  d       => pm23,
  q       => pm23_reg,
  reset => '0',
  set     => '0');

--decision bits
keep_pm1_pipe2_flop: FDRSE
generic map ( INIT => '0')
port map( C    => clk,
          D    => keep_pm1_pipe1,
          Q    => keep_pm1_pipe2,
          CE   => '1',
          R    => '0',
          S    => '0');
keep_pm3_pipe2_flop: FDRSE
generic map ( INIT => '0')
port map( C    => clk,
          D    => keep_pm3_pipe1,
          Q    => keep_pm3_pipe2,
          CE   => '1',
          R    => '0',
          S    => '0');
--7)Finally, select (mux) the best metric, and output the
    decision bits indicating which of the four
```

```
--candidate path metrics was selected.
mux_final: entity work.mux2(xprim)
generic map(
  REG   => true,
  INIT  => ZERO_SM,
  PLACE => PLACE_SUB_INSTANCE)
     port map(
  clk   => clk,
  sel   => keep_pm23,
  a     => pm01_reg,
  b     => pm23_reg,
  ans   => dest_state_metric_slv,
  ce    => '1',
  reset => '0',
  set   => '0');
dest_state_metric(i) <= signed(dest_state_metric_slv);

--Output the decision bits based on which path metric was kept.
    There are two decison bits for
--each ACS unit. Encoded as follows.
--pm0 - "00"
--pm1 - "01"
--pm2 - "10"
--pm3 - "11"
decision_bits_msb: FDRSE
generic map ( INIT => '0')
port map( C    => clk,
          D    => keep_pm23,
          Q    => decision_bits(1 + 2*i),
          CE   => '1',
          R    => '0',
          S    => '0');

decision_bits_lsb: entity work.mux2(xprim)
generic map(
  REG   => true,
  INIT  => "0",
  PLACE => PLACE_SUB_INSTANCE)
     port map(
  clk          => clk,
  sel          => keep_pm23,
  a            => vectorize(keep_pm1_pipe2),     --function
     required to map std_logic to 1 bit slv
  b            => vectorize(keep_pm3_pipe2),
  ans          => decision_bits_lsb_dummy,
  ce           => '1',
  reset        => '0',
  set          => '0');
decision_bits(0 + 2*i) <= decision_bits_lsb_dummy(0);  --needed
    to map 1 bit slv to std_logic
  end generate;
 end generate;
end rtl;
```

D.2.4 Viterbi Trellis Package

matlab/fpga/vhdl/pkg_trellis.vhd

D.2.5 Testbench

/fpga/src/cpm_viterbi_decoder_tb.vhd

D.2.6 Test Vectors Package

/matlab/fpga/vhdl/pkg_matlab_test_vectors_viterbi.vhd

D.3 Branch Metrics Filter Bank and Viterbi Trellis Path Metrics

D.3.1 Synthesis Top Level

/fpga/src/cpm_viterbi_decoder_and_bms_synth.vhd

D.3.2 Top Level

/fpga/src/cpm_viterbi_decoder_and_bms.vhd

D.3.3 Testbench

/fpga/src/cpm_viterbi_decoder_and_bms_tb.vhd

D.4 Placed Primitives Modules

D.4.1 Adder with Subtract and Clear Controls

/fpga/src/xilinx_primitive_encapsulation/adder_sub_clr.vhd

D.4.2 Adder with Subtract and 2 Input Operand Mux

/fpga/src/xilinx_primitive_encapsulation/adder_sub_mux.vhd

D.4.3 16 Deep ROM using Distributed Ram

/fpga/src/xilinx_primitive_encapsulation/lut_rom.vhd

D.4.4 2 Input Mux

/fpga/src/xilinx_primitive_encapsulation/mux2.vhd

D.4.5 Shift Register

/fpga/src/xilinx_primitive_encapsulation/shift_reg.vhd

D.4.6 Register

/fpga/src/xilinx_primitive_encapsulation/register.vhd

D.4.7 Relative Location Constraint(RLOC) Helper Package

/fpga/src/pkg_rloc.vhd

D.5 Sundry Packages

D.5.1 Key Project Constants

/fpga/src/pkg_project.vhd

Appendix E

Matlab Source Code

This appendix lists the Matlab source code referenced in this thesis. Please see the CD associated with this thesis for the complete listing of Matlab source code and Simulink models in electronic format.

E.1 Analytical CPM Code Performance

E.1.1 CPM Code Minimum Euclidean Distance Upper Bound

`/matlab/minimum_distance/dmin_upper_bound.m`

E.1.2 CPM Code Baseband Double Sided Bandwidth

`/matlab/minimum_distance/baseband_spectrum.m`

E.2 Floating and Fixed Point M-file Models

`/matlab/cpm_modulator/`

`/matlab/cpm_demodulator/`

`/matlab/cpm_trellis/`

`/matlab/phase_pulses/`

`/matlab/system_model/`

`/matlab/system_model_simulink/`

E.3 VHDL Trellis Representation and Test Vector Generation

`/matlab/fpga/`

E.3.1 Export Matlab Data to VHDL Writer

/matlab/fpga/VHDL_TEST_export.m

E.3.2 VHDL Writer

/matlab/fpga/vhdl_writer.m

E.4 Miscellaneous

/matlab/misc/

E.4.1 Hekstras Method Bound Calculation

/matlab/misc/hekstra_pm_bound.m

Appendix F

Implementation Results

F.1 Branch Metric Filter Bank

```
Release 10.1.03 Map K.39 (nt)
Xilinx Mapping Report File for Design 'branch_metrics_dafir_synth'

Design Information
------------------
Command Line   : map -ise
C:/project/massey_stratex_cpm/demosvn/abr_masters_cpm/fpga/ise10_1/
    branch_metric
s/branch_metrics.ise -intstyle ise -p xc3sd1800a-cs484-4 -cm area -
    detail -pr
off -k 4 -c 100 -o branch_metrics_dafir_synth_map.ncd
branch_metrics_dafir_synth.ngd branch_metrics_dafir_synth.pcf
Target Device  : xc3sd1800a
Target Package : cs484
Target Speed   : -4
Mapper Version : spartan3adsp -- $Revision: 1.46.12.2 $
Mapped Date    : Sat Mar 14 10:28:10 2009

Design Summary
--------------
Number of errors:      0
Number of warnings:  264
Logic Utilization:
  Number of Slice Flip Flops:         7,460 out of  33,280   22%
  Number of 4 input LUTs:             4,896 out of  33,280   14%
Logic Distribution:
  Number of occupied Slices:          3,863 out of  16,640   23%
    Number of Slices containing only related logic:   3,863 out of
        3,863 100%
    Number of Slices containing unrelated logic:          0 out of
        3,863   0%
      *See NOTES below for an explanation of the effects of unrelated
          logic.
  Total Number of 4 input LUTs:       4,896 out of  33,280   14%
  Number of bonded IOBs:                 31 out of     309   10%
  Number of BUFGMUXs:                     1 out of      24    4%

  Number of RPM macros:            5
Peak Memory Usage:  310 MB
Total REAL time to MAP completion:  1 mins 30 secs
Total CPU time to MAP completion:   1 mins 25 secs
```

```
--
------------------------------------------------------------------------------

Release 10.1.03 par K.39 (nt)
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

ANDREW-K64:: Sat Mar 14 10:29:46 2009

par -w -intstyle ise -ol std -t 1 branch_metrics_dafir_synth_map.ncd
branch_metrics_dafir_synth.ncd branch_metrics_dafir_synth.pcf

Constraints file: branch_metrics_dafir_synth.pcf.
Loading device for application Rf_Device from file '3sd1800a.nph' in
    environment
C:\Aps\xilinx_webpack_10_1\ISE.
"branch_metrics_dafir_synth" is an NCD, version 3.2, device xc3sd1800a,
     package cs484, speed -4

Initializing temperature to 85.000 Celsius. (default - Range: 0.000 to
    85.000 Celsius)
Initializing voltage to 1.140 Volts. (default - Range: 1.140 to 1.260
    Volts)

Device speed data version: "PRODUCTION 1.33 2008-07-25".

Overall effort level (-ol):   Standard
Placer effort level (-pl):    High
Placer cost table entry (-t): 1
Router effort level (-rl):    Standard

+--
    ---------------------+----------------+-------+-------+-------------+--------------+

|        Clock Net     |    Resource    |Locked|Fanout|Net Skew(ns)|Max
    Delay(ns)|
+--
    ---------------------+----------------+-------+-------+-------------+--------------+

|                  clk  |  BUFGMUX_X2Y0|  No    |  3863  |   0.371      |
    1.859        |
+--
    ---------------------+----------------+-------+-------+-------------+--------------+


Timing Score: 10

--
    --------------------------------------------------------------------------------------

  Constraint                         |  Check    | Worst Case |   Best Case  |
      Timing  |    Timing
                                     |           |    Slack    | Achievable  |
                                          Errors  |    Score
--
    --------------------------------------------------------------------------------------

* TS_clk = PERIOD TIMEGRP "clk"  | SETUP     |     -0.010ns|        4.639ns|
           1|        10
  216 MHz HIGH 50%                 | HOLD      |      0.806ns|               |
              0|          0
```

F.2 Viterbi Trellis Path Metrics

```
Release 10.1.03 Map K.39 (nt)
Xilinx Mapping Report File for Design 'cpm_viterbi_decoder_synth'

Design Information
------------------
Command Line   : map -ise
C:/project/massey_stratex_cpm/demosvn/abr_masters_cpm/fpga/ise10_1/
    viterbi_acs/v
iterbi_acs.ise -intstyle ise -p xc3sd1800a-cs484-4 -cm area -pr off -u
    -k 4 -c
100 -o cpm_viterbi_decoder_synth_map.ncd cpm_viterbi_decoder_synth.ngd
cpm_viterbi_decoder_synth.pcf
Target Device  : xc3sd1800a
Target Package : cs484
Target Speed   : -4
Mapper Version : spartan3adsp -- $Revision: 1.46.12.2 $
Mapped Date    : Fri Oct 31 13:50:16 2008

Design Summary
--------------
Number of errors:      0
Number of warnings: 1954
Logic Utilization:
  Number of Slice Flip Flops:        11,338 out of  33,280   34%
  Number of 4 input LUTs:             7,174 out of  33,280   21%
Logic Distribution:
  Number of occupied Slices:          7,303 out of  16,640   43%
    Number of Slices containing only related logic:   7,303 out of
        7,303 100%
    Number of Slices containing unrelated logic:          0 out of
        7,303   0%
      *See NOTES below for an explanation of the effects of unrelated
          logic.
  Total Number of 4 input LUTs:       7,174 out of  33,280   21%
    Number used as logic:             7,173
    Number used as Shift registers:       1
  Number of bonded IOBs:                  1 out of     309    1%
  Number of BUFGMUXs:                     1 out of      24    4%

Peak Memory Usage:  373 MB
Total REAL time to MAP completion:  1 mins 22 secs
Total CPU time to MAP completion:   1 mins 11 secs
```

```
Release 10.1.03 par K.39 (nt)
Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.

ANDREW-K64::  Fri Oct 31 13:51:47 2008

par -w -intstyle ise -ol std -t 1 cpm_viterbi_decoder_synth_map.ncd
cpm_viterbi_decoder_synth.ncd cpm_viterbi_decoder_synth.pcf

Constraints file: cpm_viterbi_decoder_synth.pcf.
Loading device for application Rf_Device from file '3sd1800a.nph'
```

```
in environment C:\Aps\xilinx_webpack_10_1\ISE.
"cpm_viterbi_decoder_synth" is an NCD, version 3.2, device xc3sd1800a,
    package cs484, speed -4

Initializing temperature to 85.000 Celsius. (default - Range: 0.000 to
    85.000 Celsius)
Initializing voltage to 1.140 Volts. (default - Range: 1.140 to 1.260
    Volts)

Device speed data version: "PRODUCTION 1.33 2008-07-25".

Overall effort level (-ol):   Standard
Placer effort level (-pl):    High
Placer cost table entry (-t): 1
Router effort level (-rl):    Standard

+--
    -------------------+---------------+-------+-------+------------+-------------+

|        Clock Net     |    Resource    |Locked|Fanout|Net Skew(ns)|Max
    Delay(ns)|
+--
    -------------------+---------------+-------+-------+------------+-------------+

|                  clk | BUFGMUX_X2Y11| No    | 6343 |   0.348     |
    1.818       |
+--
    -------------------+---------------+-------+-------+------------+-------------+


Timing Score: 751




Release 10.1.03 par K.39 (nt)
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

ANDREW-K64:: Sat Mar 14 10:29:46 2009

par -w -intstyle ise -ol std -t 1 branch_metrics_dafir_synth_map.ncd
branch_metrics_dafir_synth.ncd branch_metrics_dafir_synth.pcf

Constraints file: branch_metrics_dafir_synth.pcf.
Loading device for application Rf_Device from file '3sd1800a.nph' in
    environment
C:\Aps\xilinx_webpack_10_1\ISE.
"branch_metrics_dafir_synth" is an NCD, version 3.2, device xc3sd1800a,
     package cs484, speed -4

Initializing temperature to 85.000 Celsius. (default - Range: 0.000 to
    85.000 Celsius)
Initializing voltage to 1.140 Volts. (default - Range: 1.140 to 1.260
    Volts)

Device speed data version: "PRODUCTION 1.33 2008-07-25".

Overall effort level (-ol):   Standard
Placer effort level (-pl):    High
Placer cost table entry (-t): 1
```

```
Router effort level (-rl):    Standard

+--
    ------------------------+------------------+--------+--------+---------------+---------------+
|          Clock Net       |     Resource     |Locked|Fanout|Net Skew(ns)|Max
    Delay(ns)|
+--
    ------------------------+------------------+--------+--------+---------------+---------------+
|                    clk  |   BUFGMUX_X2Y0|  No     |  3863  |    0.371        |
    1.859        |
+--
    ------------------------+------------------+--------+--------+---------------+---------------+

Timing Score: 10

--
    ---------------------------------------------------------------------------------------------------

  Constraint                    |   Check    |  Worst Case  |   Best Case  |  Timing
      |     Timing
                                |            |     Slack    |  Achievable  |  Errors
                                     |      Score
--
    ---------------------------------------------------------------------------------------------------
* TS_clk  =  PERIOD TIMEGRP  |  SETUP    |      -0.240ns|        4.891ns|
    5|            751
  "clk"  215 MHz HIGH 50%   |  HOLD     |       0.750ns|               |
      0|               0
--
    ---------------------------------------------------------------------------------------------------
```

Appendix G

Software Tool Versions

G.1 High Level Modelling: Matlab and Simulink

The work in this thesis used Matlab and Simulink for high level modelling. Table G.1 contains the software versions of Matlab, Simulink and the various blocksets and toolboxes used.

G.2 FPGA Implementation: Xilinx ISE 10.1

Table G.2 contains the software version for the Xilinx synthesis and implementation tools.

G.3 VHDL Simulation: Modelsim

VHDL simulator version: Modelsim Xilinx Edition III 6.0d.

Software	Version
MATLAB	Version 7.5 (R2007b)
Simulink	Version 7.0 (R2007b)
Communications Blockset	Version 3.6 (R2007b)
Communications Toolbox	Version 4.0 (R2007b)
Fixed-Point Toolbox	Version 2.1 (R2007b)
Fixed-Point Toolbox	Version 2.1 (R2007b)
Signal Processing Blockset	Version 6.6 (R2007b)
Signal Processing Toolbox	Version 6.8 (R2007b)

Table G.1: Matlab and Simulink Software Versions

Software Tool	Version
XST	Release 10.1.03 - xst K.39 (nt)
Translate	Release 10.1.03 ngdbuild K.39 (nt)
Map	Release 10.1.03 Map K.39 (nt)
Place and route	Release 10.1.03 par K.39 (nt)
Static Timing Analysis	Release 10.1.03 Trace (nt)

Table G.2: Xilinx Synthesis and Implementation Software Versions

Bibliography

[1] European Telecommunications Standards Institute, "ETSI EN 302 217-2-2 V1.1.3(2004-12) Fixed Radio Systems; Characteristics and requirements for point-to-point equipment and antennas," 2004.

[2] Y.-N. Chang, H. Suzuki, and K. Parhi, "A 2-Mb/s 256-state 10-mW rate-1/3 Viterbi decoder," *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 6, pp. 826–834, Jun 2000.

[3] M. Morelli, U. Mengali, and G. Vitetta, "Joint phase and timing recovery with CPM signals," *Communications, IEEE Transactions on*, vol. 45, pp. 867–876, Jul 1997.

[4] G. Colavolpe and R. Raheli, "Reduced-complexity detection and phase synchronization of CPM signals," *Communications, IEEE Transactions on*, vol. 45, no. 9, pp. 1070–1079, Sep 1997.

[5] B. Sklar, *Digital Communications*. Prentice Hall, 2001.

[6] J. Anderson, T. Aulin, and C. Sundberg, *Digital Phase Modulation*. Plenum Press, 1986.

[7] J. Anderson and A. Svensson, *Coded Modulation Systems*. Kluwer Academic/ Plenum Publishers, 2003.

[8] I. Sasase and S. Mori, "Multi-h phase-coded modulation," *Communications Magazine, IEEE*, vol. 29, no. 12, pp. 46–56, Dec 1991.

[9] T. Aulin, N. Rydbeck, and C.-E. Sundberg, "Continuous phase modulation–part ii: Partial response signaling," *Communications, IEEE Transactions on [legacy, pre - 1988]*, vol. 29, no. 3, pp. 210–225, Mar 1981.

[10] M. Geoghegan, "Description and performance results for a multi-h CPM telemetry waveform," *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, vol. 1, pp. 353–357 vol.1, 2000.

[11] `http://www.xilinx.com/company/success/novaengr.htm`. retrieved 10th January 2008.

[12] T. Tapp and R. Mickelson, "Turbo detection of coded continuous-phase modulations," *Military Communications Conference Proceedings, 1999. MILCOM 1999. IEEE*, vol. 1, pp. 534–537 vol.1, 1999.

[13] M. Petrov and M. Glesner, "A state-serial Viterbi decoder architecture for digital radio on FPGA," *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 323–324, 11-14 Dec. 2005.

[14] M. Bree, D. Dodds, R. Bolton, S. Kumar, and B. Daku, "A modular bit-serial architecture for large-constraint-length Viterbi decoding," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 2, pp. 184–190, Feb 1992.

[15] C. Shung, H.-D. Lin, R. Cypher, P. Siegel, and H. Thapar, "Area-efficient architectures for the Viterbi algorithm. I. Theory," *Communications, IEEE Transactions on*, vol. 41, no. 4, pp. 636–644, Apr 1993.

[16] P. Black and T. Meng, "A 140-Mb/s, 32-state, radix-4 Viterbi decoder," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 12, pp. 1877–1885, Dec 1992.

[17] T. Zhang, J. Wu, and G. Saulnier, "Efficient coherent detector VLSI design for continuous phase modulation," *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 2, pp. 1663–1666 Vol.2, Nov. 2003.

[18] R. Tessier, S. Swaminathan, R. Ramaswamy, D. Goeckel, and W. Burleson, "A reconfigurable, power-efficient adaptive Viterbi decoder," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 4, pp. 484–488, April 2005.

[19] M. Guo, M. Ahmad, M. Swamy, and C. Wang, "FPGA design and implementation of a low-power systolic array-based adaptive Viterbi decoder," *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, vol. 52, no. 2, pp. 350–365, Feb. 2005.

[20] F. Sun and T. Zhang, "Low-power state-parallel relaxed adaptive Viterbi decoder," *Circuits and Systems I: Regular Papers, IEEE Transactions on [Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, vol. 54, no. 5, pp. 1060–1068, May 2007.

[21] C.-Y. Chang and K. Yao, "Systolic array processing of the Viterbi algorithm," *Information Theory, IEEE Transactions on*, vol. 35, no. 1, pp. 76–86, Jan 1989.

[22] ALTERA, *Megacore: Viterbi Compiler User Guide*, v7.2 ed., October 2007.

[23] XILINX, *Logicore: Viterbi Decoder Product Specification DS247*, v6.2 ed., October 2007.

[24] C. Shung, P. Siegel, G. Ungerboeck, and H. Thapar, "VLSI architectures for metric normalization in the Viterbi algorithm," *Communications, 1990. ICC 90, Including Supercomm Technical Sessions. SUPERCOMM/ICC '90. Conference Record., IEEE International Conference on*, pp. 1723–1728 vol.4, 16-19 Apr 1990.

[25] A. Hekstra, "An alternative to metric rescaling in Viterbi decoders," *Communications, IEEE Transactions on*, vol. 37, no. 11, pp. 1220–1222, Nov 1989.

[26] P. Siegel, C. Shung, T. Howell, and H. Thapar, "Exact bounds for Viterbi detector path metric differences," *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pp. 1093–1096 vol.2, 14-17 Apr 1991.

[27] T. Aulin and C. Sundberg, "Continuous phase modulation–part i: Full response signaling," *Communications, IEEE Transactions on [legacy, pre - 1988]*, vol. 29, no. 3, pp. 196–209, Mar 1981.

[28] T. Svensson and A. Svensson, "Reduced complexity detection of bandwidth efficient partial response CPM," *Vehicular Technology Conference, 1999 IEEE 49th*, vol. 2, pp. 1296–1300 vol.2, Jul 1999.

[29] T. Svensson and A. Svensson, "Empirical model for spectrally efficient continuous phase modulation," *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th*, vol. 1, pp. 696–700 Vol.1, 6-9 Oct. 2003.

[30] T. Svensson and A. Svensson, "Complexity and performance of spectrally efficient continuous phase modulation," *Proceedings Nordic Radio Symposium, Nynshamn, Sweden*, 2001.

[31] D. Asano, H. Leib, and S. Pasupathy, "Phase smoothing functions for continuous phase modulation," *Communications, IEEE Transactions on*, vol. 42, no. 234, pp. 1040–1049, Feb/Mar/Apr 1994.

[32] E. S. Perrins, "Reduced complexity detection methods for continuous phase modulation," *PHD Thesis*, 2005.

[33] A. Svensson, C. Sundberg, and T. Aulin, "A class of reduced-complexity viterbi detectors for partial response continuous phase modulation," *Communications, IEEE Transactions on [legacy, pre - 1988]*, vol. 32, pp. 1079–1087, Oct 1984.

[34] W. Tang and E. Shwedyk, "A CPM receiver based on the walsh signal space," *Communications, Computers, and Signal Processing, 1995. Proceedings. IEEE Pacific Rim Conference on*, pp. 203–206, 17-19 May 1995.

[35] A. Svensson, "Reduced state sequence detection of full response continuous phase modulation," *Electronics Letters*, vol. 26, no. 10, pp. 652–654, 1 May 1990.

[36] A. Svensson, "Reduced state sequence detection of partial response continuous phase modulation," *Communications, Speech and Vision, IEE Proceedings I*, vol. 138, no. 4, pp. 256–268, Aug 1991.

[37] J. Huber and W. Liu, "An alternative approach to reduced-complexity CPM-receivers," *Selected Areas in Communications, IEEE Journal on*, vol. 7, no. 9, pp. 1437–1449, Dec 1989.

[38] P. Laurent, "Exact and approximate construction of digital phase modulations by superposition of amplitude modulated pulses (AMP)," *Communications, IEEE Transactions on [legacy, pre - 1988]*, vol. 34, pp. 150–160, Feb 1986.

[39] U. Mengali and M. Morelli, “Decomposition of M-ary CPM signals into PAM waveforms,” *Information Theory, IEEE Transactions on*, vol. 41, no. 5, pp. 1265–1275, Sep 1995.

[40] G. Kaleh, “Simple coherent receivers for partial response continuous phase modulation,” *Selected Areas in Communications, IEEE Journal on*, vol. 7, pp. 1427–1436, Dec 1989.

[41] S. Simmons, “ACI susceptibility of reduced-state decoding for CPM,” *Communications Letters, IEEE*, vol. 3, no. 11, pp. 305–307, Nov 1999.

[42] February 2009. Proprietary Harris Stratex Documentation.

[43] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, pp. 203–215, Feb. 2007.

[44] XILINX, *Spartan-3A DSP FPGA Family: Datasheet*, v2.1 ed., June 2008.

[45] `http://www.provigent.com/home/index.aspx?lang=1`. retrieved 15th May 2009.

[46] C. Rader, “Memory management in a Viterbi decoder,” *Communications, IEEE Transactions on [legacy, pre - 1988]*, vol. 29, no. 9, pp. 1399–1401, Sep 1981.

[47] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 2001.

[48] R. Andraka and A. Berkun, “FPGAs make a radar signal processor on a chip a reality,” *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 1, pp. 559–563 vol.1, 1999.

[49] March 2009. Email from Jamie Pegg - Xilinx Field Applications Engineer.

[50] S. R. Bullock, *Transceiver and System Design for Digital Communications*. SciTech Publishing Inc, US, 2009.