DESIGN & IMPLEMENTATION OF A REAL-TIME, SPEAKER-INDEPENDENT, CONTINUOUS SPEECH RECOGNITION SYSTEM WITH VLIW DIGITAL SIGNAL PROCESSOR ARCHITECTURE

by

Wai-Ting Ng

B.S. in Engineering, University of Pittsburgh, 2004

Submitted to the Graduate Faculty of

School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Wai-Ting Ng

It was defended on

July 21, 2006

and approved by

Alex K. Jones, PhD, Assistant Professor

Steve P. Levitan, PhD, Professor

Committee Chair: Raymond R. Hoare, PhD, Assistant Professor

DESIGN & IMPLEMENTATION OF A REAL-TIME, SPEAKER INDEPENDENT, CONTINUOUS SPEECH RECOGNITION SYSTEM USING A VLIW DIGITAL SIGNAL PROCESSOR ARCHITECTURE

Wai-Ting Ng

University of Pittsburgh, 2006

This thesis explores the feasibility of mapping a real-time, continuous speech recognition system onto a multi-core Digital Signal Processor architecture. While a pure hardware solution is capable of implementing the entire recognition process in real-time, the design process can be lengthy and inflexible to changes. However, a low-end embedded processor such as ARM7 is insufficient to execute in real-time. As a result, a more flexible and powerful DSP solution with Texas Instruments' C6713 multi-core DSP is used to exploit the instruction level parallelism within the speech recognition process. By exploiting the parallelism using 7 optimization techniques, the performance of the recognition process can be real-time on a 300 MHz DSP for a 1000 word vocabulary.

At its core, continuous speech recognition is essentially a matching problem. The recognition process can be divided into four major phases: Feature Extraction, Acoustic Modeling, Phone Modeling and Word Modeling. Each phase is analyzed in detail to identify performance issues. In short, the major issues are its massive computations and large memory bandwidth. After applying various optimizations, the overall computational performance has improved from about 15 times slower than real-time to 1.6 times faster than real-time with the hardware. Through utilization of Direct Memory Access and larger cache memory, the memory bandwidth problem can be solved. The conclusion is that a multi-core DSP running at 300 MHz would be sufficient to implement a 1000 word Command & Control type application using the optimization techniques described in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Voice communication is one of the most primary and fundamental interactions between human beings in everyday life. The technological curiosity to build a machine that understands humans and the desire to simplify daily life through automation have driven the speech research community for the past few decades. Speech recognition began as people tried to allow computers to recognize isolated words. As progress has been made though the years, researchers have shifted focus to continuous speech recognition. With the increased understanding of human speech, significant improvements have been made within different areas of Automatic Speech Recognition (ASR). Recent advancements in computer hardware has enabled more practical software-based speech recognition applications to emerge. Commercial software running on desktop computers is slowly becoming more mature and usable. Most cellular phones are equipped with a simple recognition process that recognizes names and digits fairly accurately. However, speech-enabled applications have not yet been integrated into our daily life due to the fact that the efficiency level of embedded speech recognition applications is still low. In order to incorporate more speech-enabled applications into human life and to automate daily tasks, recognition processes must run on a portable device, and these processes must also be able to recognize continuous speech with high accuracy in real-time.

1.1 THE PROBLEM

Simply stated, the problem of continuous Automatic Speech Recognition is that the process is computationally intensive. Many approaches have been taken to solve this problem but none have yet tried to map the entire recognition process on a multi-core DSP.

Speech recognition started with the attempt to decode isolated words. As time progressed, research focus has shifted to continuous speech. With years of research effort, the start-of-the-art recognition systems are capable of recognizing continuous speech with acceptable accuracy. Furthermore, recognition systems that are built for Command and Control type applications have achieved accuracy greater than 90% [2]. However, most of these recognition systems are designed to run on a computer system with a high-clock-rate General Purpose Processor (GPP) and a large amount of memory bandwidth available.

The difficulties of recognizing continuous speech can be summarized into the following areas: large variable nature of speech signal, difference in speakers, dictionary size and environment distortions. Through years of study, researchers have obtained a better understanding of speech signals. Statistical models trained by people for thousands of hours are used to compensate for different speakers. Environment distortions can be minimized using various signal processing techniques. Among these difficulties, dictionary size has the most important effect on the system performance.

At its core, speech recognition is a matching problem between observed signals and a set of pre-defined words, or word library. The larger set of the pre-defined words implies that more computations are required. More importantly, the time available to complete all computations remains the same regardless of the dictionary size. State-of-the-art systems are mostly software based that are designed to run on computer platforms with the General Purpose Processor architecture. One limitation of using these GPP architectures is the lack of multiple processing units to take advantage of the instruction parallelism within the recognition process. Furthermore, speech recognition process requires massive amounts of memory bandwidth between the processing unit and memory. The cache architectures that are designed to provide quick access to recently accessed data can take advantage of the locality for parts of the recognition process where data accesses are more sequential. However, some parts of the process inherit a highly irregular memory access pattern where cache would be less useful. For the GPP systems, these limitations are usually compensated by the high clock rate processor and the large amount of cache memory available.

These software-based systems produce relatively good results as long as a high-clock-rate processor is available. However, such a machine might not always be available or even desired for certain applications. For example, mobile phones and PDAs are platforms where hardware resources are limited. Various approaches have been taken to scale down the number of computations. Using a smaller sized dictionary reduces the number of computations but also reduces the usability of the application. Another approach is to reduce the bit precision of the calculation.

1.2 THE SOLUTION

The process of ASR is generally divided into multiple stages. The first stage typically involves extracting useful information from the voice signal. The next few stages take the extracted information and perform word matching at different levels. The differences in functional characteristics and requirements of each stage creates an uneven amount of processing cycle consumption. One study [1] has shown that the Acoustic Modeling stage can potentially consume up to 95% of the entire process. As a result, more effort is spent investigating and optimizing areas that consume the majority of processing time.

A close examination reveals that speech algorithms inherit substantial instruction-level parallelism (ILP) at various stages of the process. It has been shown that custom designed hardware with a deep pipeline is capable of implementing a 1000 word Command and Control based application in real-time [1]. However, custom designed hardware lacks the flexibility for changes due to the evolutionary process of speech recognition. Also, the design cycle of such custom architecture is rather lengthy. This thesis investigates a more flexible software approach with a multi-core Digital Signal Processor (DSP) architecture. This thesis utilizes the Texas Instruments TMS320C6713 DSP Starter Kit as a demonstration that real-time Automatic Speech Recognition is possible an embedded DSP processor.

DSP technology has improved significantly in recent years. Advanced architecture such as Very Long Instruction Word (VLIW) enables a single DSP to execute multiple instructions simultaneously. The VLIW architecture is designed specially for computationally intensive applications such as speech recognition. Furthermore, these DSPs are usually equipped with highly efficient and flexible cache architecture along with multiple channel Direct Memory Access (DMA). With careful design and optimization, the cache system and DMA can be used together to solve the memory bandwidth issue.

This research focuses on the exploitation of the parallelism within speech recognition for multi-core DSP and predicts the resources requirement for real-time. The result has shown that by taking advantage of the multiple execution units and by using various compiler optimizations, a single DSP running at 300 MHz with at least 3MB of on-chip memory would be sufficient to implement a 1000 word Command and Control based application running in real-time. This research will expand the applicability of speech recognition from high clock rate general purpose computers to low-power, portable devices such as cellular phones and PDAs.

1.3 CONTRIBUTIONS & ORGANIZATION OF THE THESIS

The major contributions of this thesis are:

- Redesigned and mapped a real-time, speaker independent, command and control based speech recognition system onto a multi-core DSP architecture. Also examined and optimized each stage of the recognition process including Feature Extraction, Acoustic Modeling, Phone & Word Modeling.
- Optimized Feature Extraction using various optimization techniques including Memory Aliasing, Software Pipeline, Loop Unrolling and Packed Data Memory Access.
- Investigated Acoustic Modeling in two different approaches, Component Approach and Dimension Approach.
- Optimized the Component Approach using various optimization techniques.
- Optimized the Dimension Approach using various optimization techniques.
- Resolved the memory bandwidth problem in Acoustic Modeling by eliminating the delay effect of accessing external memory using DMA and Cache Buffering technique.
- Examined and implemented Phone & Word Modeling and determined that they are computationally simple but require a large memory bandwidth.

1.3.1 Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 reviews recent works pertaining to Automatic Speech Recognition. Different approaches including pure software and dedicated custom hardware architecture had been taken to implement speech recognition. The characteristics and performances of these different approaches will be presented and discussed briefly.

Chapter 3 provides an overview of the Automatic Speech Recognition process. In addition, the system design specifications and requirements will be presented. The specifications of the DSP including the VLIW architecture, the memory system and the DMA will be described in detail along with the real-time requirements. Finally, the basic performance measurements are presented. These measurements will be used to determine how well an algorithm is optimized.

With all the pieces in place, Chapter 4 begins to focus on the detail implementation and optimization of the first stage of the ASR process, Feature Extraction. This section will provide the background theory of Feature Extraction along with its operations performed. The implementation will then be presented. After applying various optimizations, the performance result improved 332% from 231,577 cycles to 69,684 cycles. Furthermore, the implementation is functionally verified against the Sphinx 3 system using the data samples extracted from Sphinx 3.

Chapter 5 focuses on the next phase, Acoustic Modeling. This chapter will discuss the background theory, performance issues and different implementations and the performance result. As shown in [1], AM is the most computational intensive phase of the entire recognition process. As a result, two different approaches, Component Approach and Dimension Approach, are implemented. Each approach is optimized with numerous different optimization techniques to obtain the best performance possible with the given DSP architecture. After comparing the optimal performances, the Dimension approach has the best performance of 0.75 million cycles, an improvement of 4500% from 33.8 million cycles without any optimization. Another performance problem for AM is that it requires a large amount of data that must be stored in the external memory. However, accessing the external memory requires significant amounts of cycle delay. One solution is to use Direct Memory Access as the detail will be presented in (section.5.2) All implementations are functionally verified against the original Sphinx 3 system using the same input samples from Sphinx 3.

Finally, Charter 6 discusses both Phone Modeling and Word Modeling. PM and WM are relatively insignificant from the computation perspective. However, a memory bandwidth problem similar to AM exists and the same solution used for AM is not applicable in this case. The best solution is that a traditional cache system large enough for all required data for PM and WM is sufficient. Similar to the previous sections, the theory, functions and implementations of Phone Modeling will be provided in detail.

In conclusion, all performance results from different phases will be summarized. Each phase is implemented and functionally verified against Sphinx 3. Although the integration of all phases of the 1000 word command and control application doesn't execute in real-time on the 225 MHz DSP. However, it a multi-core DSP running at 300 MHz would be sufficient for real-time implementation.

2.0 LITERATURE REVIEW

Speech recognition has been a popular research topic for the past several decades. Various methods of implementation have been applied to solve the problem of Automatic Speech Recognition. These implementations range from pure software designed to run on GPP to dedicated custom hardware architectures.

With the recent advancement of computer hardware, more promising software-based solutions have been developed not only in the research community, but also in the commercial market. Carnegie Mellon University's SPHINX [3] and University of Colorado's SONIC [11] are two examples of successful research systems. For the commercial systems, IBM's ViaVoice [12] and Nuance' Dragon Naturally Speaking [13] are two popular recognition software. However, all these software-based systems are designed to run on computer systems with high clock rate GPP. They also require a large memory footprint for data storage.

Various works have been done in the research community to characterize speech recognition system and to investigate the implementation of speech recognition in embedded applications. Agaram, et al [15] characterized the speech recognition process in 2001. Aside from the characterization, their research group also performed an extensive analysis on the effect of various cache sizes. The investigated result shows that speech recognition processes substantially exercise the memory system and exhibit a low level of Instruction Level Parallelism. They then proposed various methods to improve the throughput of the system. As a result, they were able to increase the instruction per cycle (IPC) from 0.64 to 3.55.

Binus, et al [10] profiled and characterized Sphinx 3 system in the block levels and their results indicate that the majority of computations in speech recognition is spent in Acoustic Modeling.

Hagen, et al [16] characterized the speech recognition process in hand-held mobile devices. Their work evaluated the performance of the SONIC [11] recognition system from the University of Colorado on a PDA-like architecture. Their work focused on optimizing the code using a strategic set of compiler optimizations. The result shows that real-time recognition is close to being possible with a 500 MHz processor with moderate levels of ILP and cache resources.

The results of the above work conclude that speech recognition is extremely computational intensive where special attention should be paid to Acoustic Modeling. Significant amounts of memory bandwidth is required and recognition algorithms exhibits moderate level of ILP.

2.1 SPHINX 3

A software-based speech recognition system developed by Carnegie Mellon University called Sphinx [3] is used as the basis for this research. Specifically, version 3 of Sphinx is used. Sphinx 3 is a speaker independent, continuous speech, software recognition system. Sphinx 3 had shown that it is capable of archiving 10x more than real-time in a broadcasting news transcription system [7]. Ten times real-time is considered a fairly solid result due to the fact that the system was implemented in 1999 and the word library used contains 64,000 words.

In Sphinx 3, all words are decomposed into sub-word units, called *phonemes*. Specifically in Sphinx 3, all phonemes are decomposed into sub-phoneme units called *senones* [8]. More information about the modeling of phoneme and senone will be presented in later sections. The word library used for this research is based on the DARPA's Resource Management (RM1) corpus [6], which contains 1000 words used in command and control type tasks. All algorithms used in this research are either partially or fully derived from Sphinx 3.

2.2 DIGITAL SIGNAL PROCESSORS

Digital Signal Processors (DSPs) are special-purpose microprocessors designed with specialized architectures very suitable for different type of signal processing applications. The flexibility though reprogramming and the power efficiency provided by the DSP made it very suitable for most embedded applications. Compared to most general purpose processors (GPP), DSPs offer the instruction sets that typically reduce the number of instructions needed to perform the same operations. For example, most signal processing algorithms perform multiple-and-add (MAC) operations. Typically, a MAC takes a few cycles to complete in GPP but only takes one cycle to execute in VLIW DSP. The ability to execute multiple instructions is enabled by the advanced Very Long Instruction Word (VLIW) architecture. At the same time, the use of VLIW architecture increases the memory bandwidth. For parts of the recognition process where all required data fit on the on-chip memory, no significant performance decrease will be imposed from the memory access latency. However, if external memory is needed, the performance will decrease considerably due to the long latency of external memory accesses. If external memory is necessary, it is possible to reduce the latency effect through careful use of the on-chip memory along with Direct Memory Access, or DMA.

2.2.1 Very-Long-Instruction-Word Architecture

Most state-of-the-art DSPs are designed based on the VLIW architectures [9]. VLIW architectures are developed with the purpose to exploit and to increase the instruction-level parallelism (ILP) in programs. These processors contain multiple functional units that are capable of executing multiple instructions simultaneously. The instruction sets of these VLIW architectures are usually simpler than GPP instructions. In order to take advantage of these architectures, enough ILP is necessary. State-of-the-art compilers can exploit ILP through code scheduling techniques such as Software Pipelining [28] and generate codes that group together independent instructions that can run in parallel. A Very-Long-Instruction Word, that contains multiple independent instructions, can then be fetched from the instruction cache together and dispatched to the different functional units in parallel. The DSP used for this research has eight functional units: two floating-point multipliers, two floating-point/fixed-point ALUs and four fixed-point ALUs.

2.2.2 Memory Architecture

Most state-of-the-art DSPs are equipped with two level cache architecture along with a large sized on-chip fast memory, typically SRAM. More often, the second level cache is configurable to either as another cache or on-chip SRAM. For example, the targeted DSP used for this research has 192 Kbytes of on-chip SRAM along with a 2-level cache system. The capacity of the Level 1 (L1) cache is 4 Kbytes organized into 2 direct mapped, or 2-way associative, caches. The capacity of the Level 2 (L2) cache is 64 Kbytes. Furthermore, the Level 2 cache can be configured to be all on-chip SRAM, part SRAM and part cache, or all cache.

Typically a cache hit in L1 cache does not impose any latency while a cache miss in L1 will result in a few cycles delay if the data is resided in L2 cache. If the data is not in L2, then more cycles are needed to access the data from the external memory.

2.2.3 Direct Memory Access

DMA is a mechanism used to transfer data between peripherals and memory or between different memory sections without the interference of the processor. In other words, data movement and data processing can be executed in parallel. With this capability and the configured L2 cache as all on-chip SRAM, the latency effect of external memory access can be completely eliminated. This can be accomplished by configuring L2 as a double buffer so while the processor is using the data of one buffer; the other buffer be being filled with new data by the DMA. One last requirement is that the two buffers are located in separate memory banks so simultaneous accesses by the processor and DMA are possible.

2.3 DEVELOPMENT PLATFORM SPECIFICATIONS

The development platform used in this research is the TMS320C6713 DSP Starter Kit (DSK) from Texas Instruments. The DSK features a TMS320C6713 DSP running at 225 MHz. Other important peripherals available with this DSK are a 24-bit stereo codec, a 32-bit External Memory Interface (EMIF) and 16 Mbytes of external SDRAM. A basic block diagram taken from the TI TMS320C6713 DSK technical reference [27] is shown in Figure 2.1.

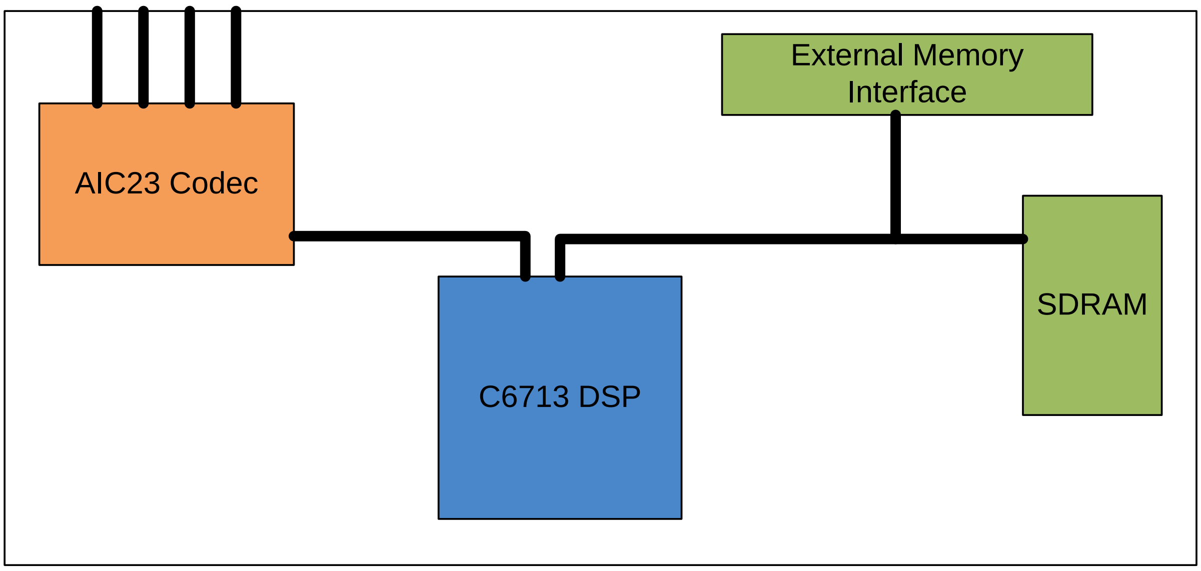
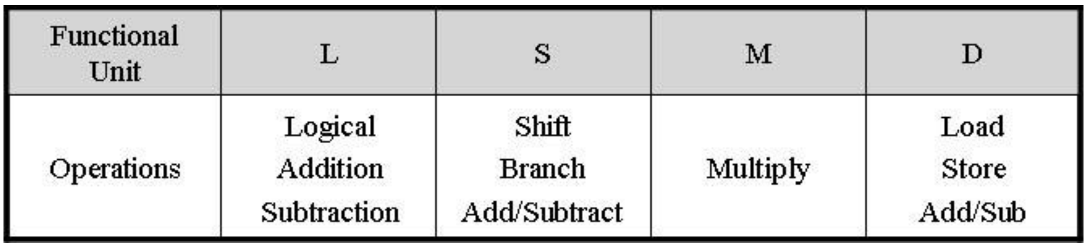


Figure 2.1: A simplified block diagram of the C6713 DSK showing the key components

2.3.1 Texas Instruments C6713 DSP

The C6713 DSP is the high performance floating-point DSP from Texas Instruments. It has 32 32-bit registers and is capable of loading either a 32-bit single word or a 64-bit double word per cycle. The DSP is designed based on an advanced VLIW architecture with eight functional units: 2 fixed-point ALUs (.D units), 4 floating-point/fixed-point ALUs (.L and .S units) and 2 floating-point multipliers (.M units). Figure 2.2 lists the type of operation(s) dedicated to each type of the functional units.



Functional Unit	L	S	M	D
Operations	Logical Addition Subtraction	Shift Branch Add/Subtract	Multiply	Load Store Add/Sub

Figure 2.2: Illustrates the operation(s) support by different functional unit

Eight functional units enable 8 independent instructions to be executed in parallel. These functional units are divided into two separate clusters where each cluster has 4 units each. The 32 registers are organized into 2 register files, each has 16 registers. Each cluster has its own register file, and each functional unit has its own access port to its corresponding register file. To pass data from one cluster to the other, a cross path (.X) must be used. A detailed architectural diagram [27] is shown in Figure 2.3 illustrating the functional units' arrangement.

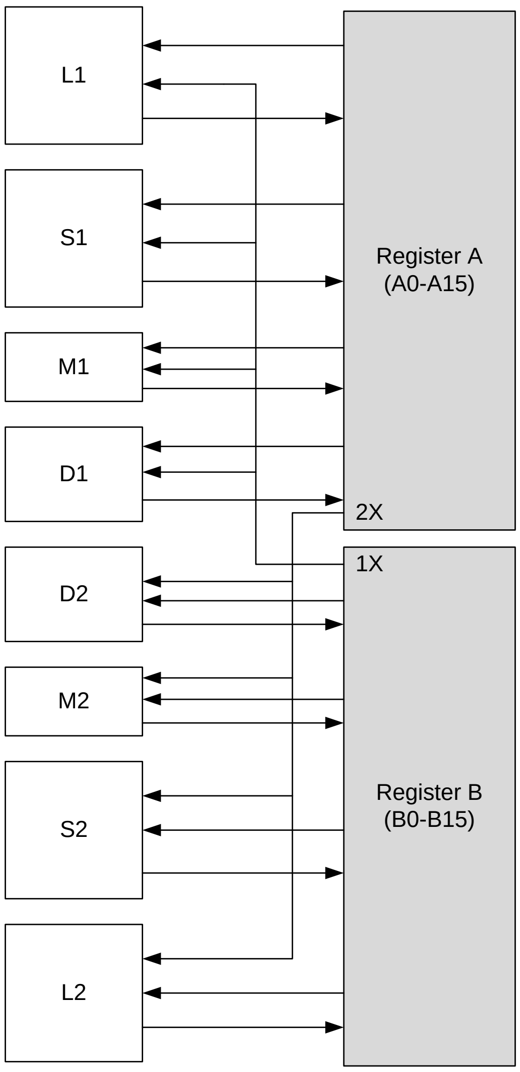
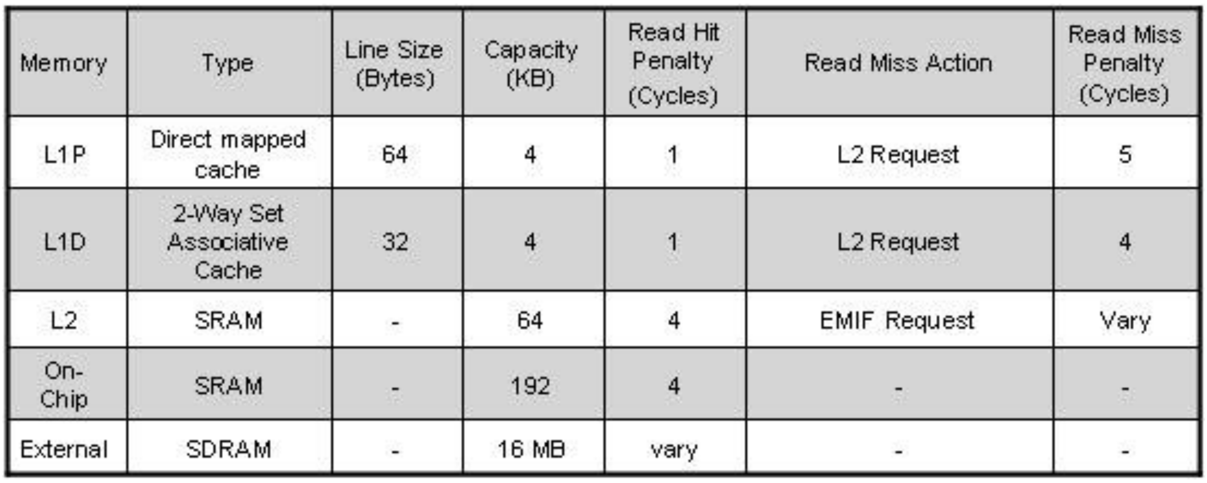


Figure 2.3: A architectural diagram showing the functional units and registers arrangement

2.3.2 Memory System

Three memory systems are available on the TMS320C6713 DSK, cache system, on-chip SRAM and external SDRAM. The C6713 DSP utilizes a two-level cache memory system. Level 1 (L1) is a two-way set-associative cache with the total capacity of 4 KB. The capacity of the level 2 (L2) cache is 64 KB, which can be configured to be all cache, all SRAM, or part cache and part SRAM. In addition to the 2 levels cache, 192 KB of on-chip SRAM is also available for both data and programming storage. If more memory space is needed, there are 16 MB of Synchronous DRAM available through the external memory interface. Table 2.1 summarizes the memory systems available and the configurations used for this research:

Table 2.1: Memory systems and configurations summary



Memory	Type	Line Size (Bytes)	Capacity (KB)	Read Hit Penalty (Cycles)	Read Miss Action	Read Miss Penalty (Cycles)
L1P	Direct mapped cache	64	4	1	L2 Request	5
L1D	2-Way Set Associative Cache	32	4	1	L2 Request	4
L2	SRAM	-	64	4	EMIF Request	Vary
On-Chip	SRAM	-	192	4	-	-
External	SDRAM	-	16 MB	vary	-	-

In addition to the memory type and the capacity, Table 2.1 also summarized other characteristics. Line Size is the number of bytes allocated from the next level memory when a read miss occurs. Read Hit Penalty is the number of cycles for the processor to access the data if the data exists in that particular memory section. Read Miss Action is the process that occurs when the data requested is not available. Finally, Read Miss Penalty is the number of stall cycles when a miss occurs. Note that for this research, L2 is configured as all SRAM.

2.4 OPTIMIZATIONS OVERVIEW

The overall performance of a speech recognition system depends mostly on the efficiency of the algorithms and the mapping of the algorithm onto the available hardware. There are numerous compiler optimization techniques available to help improve the performance of the algorithm. This section will describe the different optimization techniques available and how they are applied.

2.4.1 Variable Registering

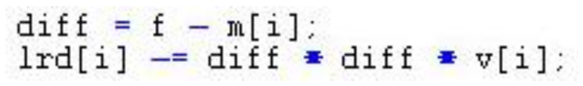
Variable Registering is one of the basic optimization techniques. Instead of reading and writing data from memory outside of the processor for every instruction, registers are used to temporarily store data until the data is no longer needed. This optimization reduces the number of memory accesses significantly and resulting major performance improvement.

2.4.2 Constant Propagation

Constant Propagation is an optimization that replaces local copies of the global variable with the actual value. This technique increases the code size slightly but also improves the code performance.

2.4.3 Data Dependency

Data dependency occurs when the next instruction cannot be executed because it requires the result from the previous instruction. An example is illustrated in Figure 2.4.



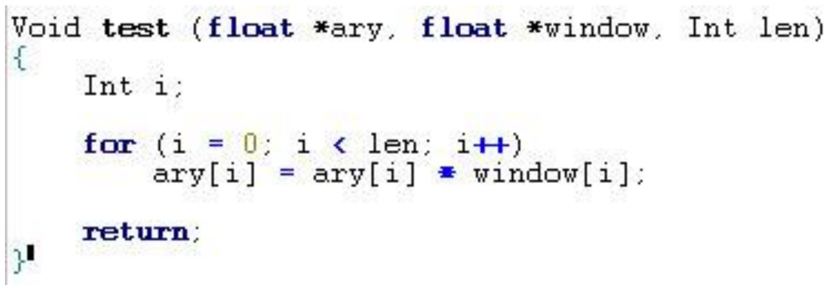
```
diff = f - m[i];
lrd[i] -= diff * diff * v[i];
```

Figure 2.4: An example of data dependency

As illustrated, the second calculation cannot be executed until *diff* is computed by the previous instruction. If the software code is executed sequentially, data dependency would not be an issue. However, almost all modern processors are capable of executing multiple instructions in parallel in order to improve performance. As a result, data dependency can post a significant performance limitation on parallel executed code.

2.4.4 Loop Carried Dependency and Memory Aliasing

Loop carried dependency occurs when the next iteration of the loop cannot start until the previous iteration is completed. This is an issue that only exists in software code executed in parallel. An example is shown in Figure 2.5.

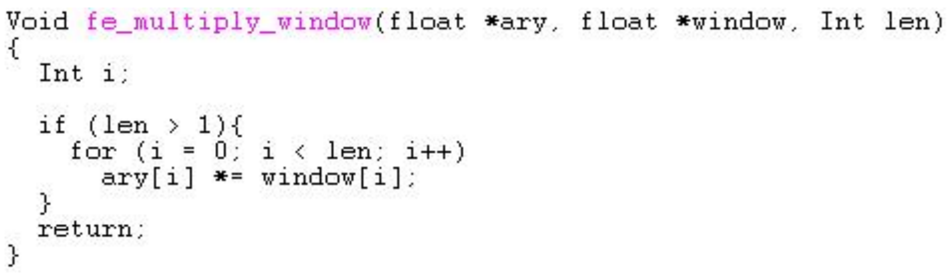


```
Void test (float *ary, float *window, Int len)
{
    Int i;

    for (i = 0; i < len; i++)
        ary[i] = ary[i] * window[i];

    return;
}
```

Figure 2.5: A code sample illustrating loop carried dependency

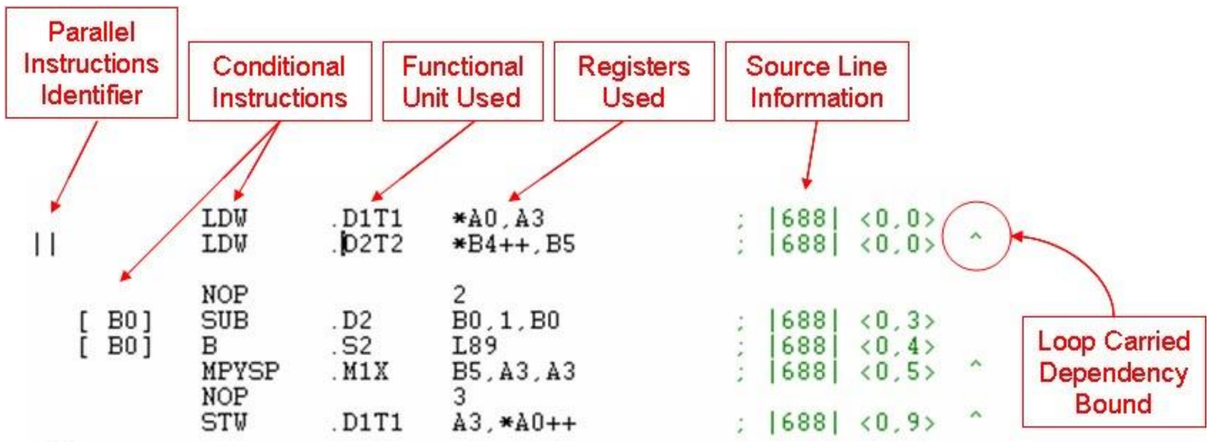
It is not immediately obvious why multiple iterations cannot be executed in parallel since each iteration involves only one element from the each of the *ary* and the *window* array and the result of the multiplication is stored back to the *ary* array. From the compiler prospective, it has no idea if any part of the *ary* and *window* arrays are overlapped in the memory, or *memory aliasing*. If some parts of the two arrays are overlapped, then executing this code in parallel would produce incorrect results.

One way to identify loop carried dependency is to use a dependency graph if the source code is available. A simpler way to find the dependency is by using the compiler generated assembly. Figure 2.6 displays a sample code and Figure 2.7 shows the generated assembly from the sample code.



```
Void fe_multiply_window(float *ary, float *window, Int len)
{
  Int i;

  if (len > 1){
    for (i = 0; i < len; i++)
      ary[i] *= window[i];
  }
  return;
}
```

Figure 2.6: Sample source code



```
          LDW     .D1T1   *A0,A3        ; |688| <0,0>
||        LDW     .D2T2   *B4++,B5      ; |688| <0,0>  ^
          NOP             2
   [ B0]  SUB     .D2     B0,1,B0       ; |688| <0,3>
   [ B0]  B       .S2     L89           ; |688| <0,4>
          MPYSP   .M1X    B5,A3,A3      ; |688| <0,5>  ^
          NOP             3
          STW     .D1T1   A3,*A0++      ; |688| <0,9>  ^
```

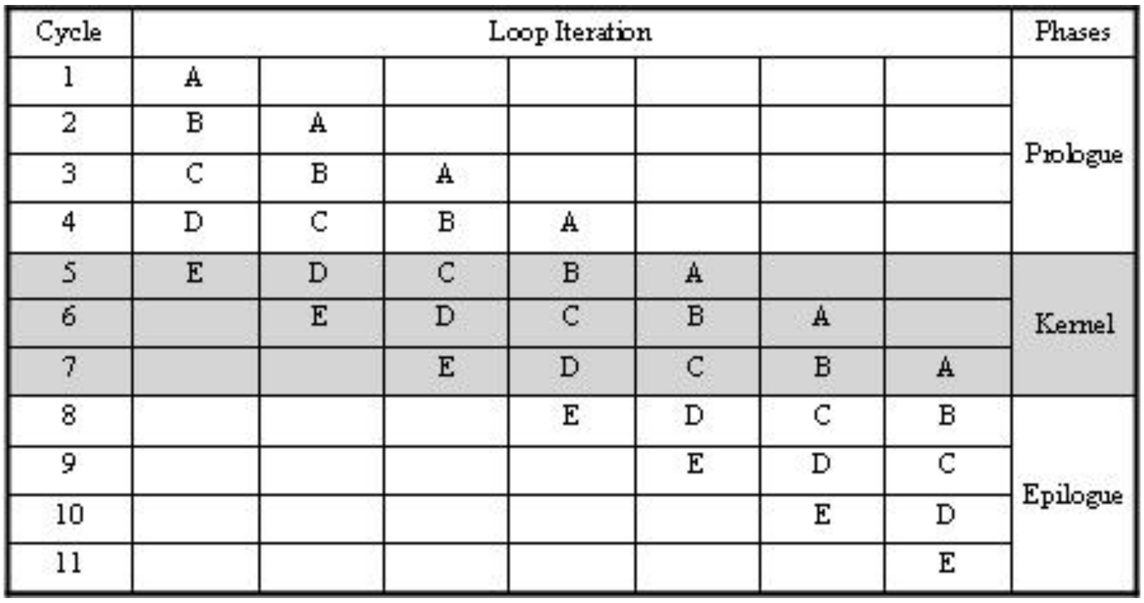
Figure 2.7: Generated assembly from source code in Figure 2.6

Instructions with dependency are indicated by (^) symbol as shown in Figure 2.7. Clearly, the problem exists between the LDW (load word) and the STW (store word) instructions. Figure 2.6 shows that the core calculation requires two loads, a multiply and a store. Within the scope of the local function, the compiler cannot assume that pointers *ary* and *window* do not overlap. The compiler cannot execute another load instruction for the next iteration until the store instruction from the previous iteration is completed.

Assuming that the two input pointers do not overlap in memory, the dependency can be removed by explicitly informing the compiler that there is no aliasing using the *restrict* keyword [25]. The *restrict* keyword is a type qualifier that represents a guarantee by the programmer that within the scope of the pointer declaration the object pointed to can only be access by that pointer. It is applied to the variable in the function declaration as shown in the following code sample: void function_name (int * restrict variable). For more information on *restrict* type qualifier, please refer to [25].

2.4.5 Software Pipeline

Software Pipelining [28] significantly improves code performance by executing multiple iterations of the loop in parallel. This is enabled by the advanced VLIW architecture. Figure 2.8 illustrates a software pipelined loop. A, B, C, D and E are five independent stages of a loop.

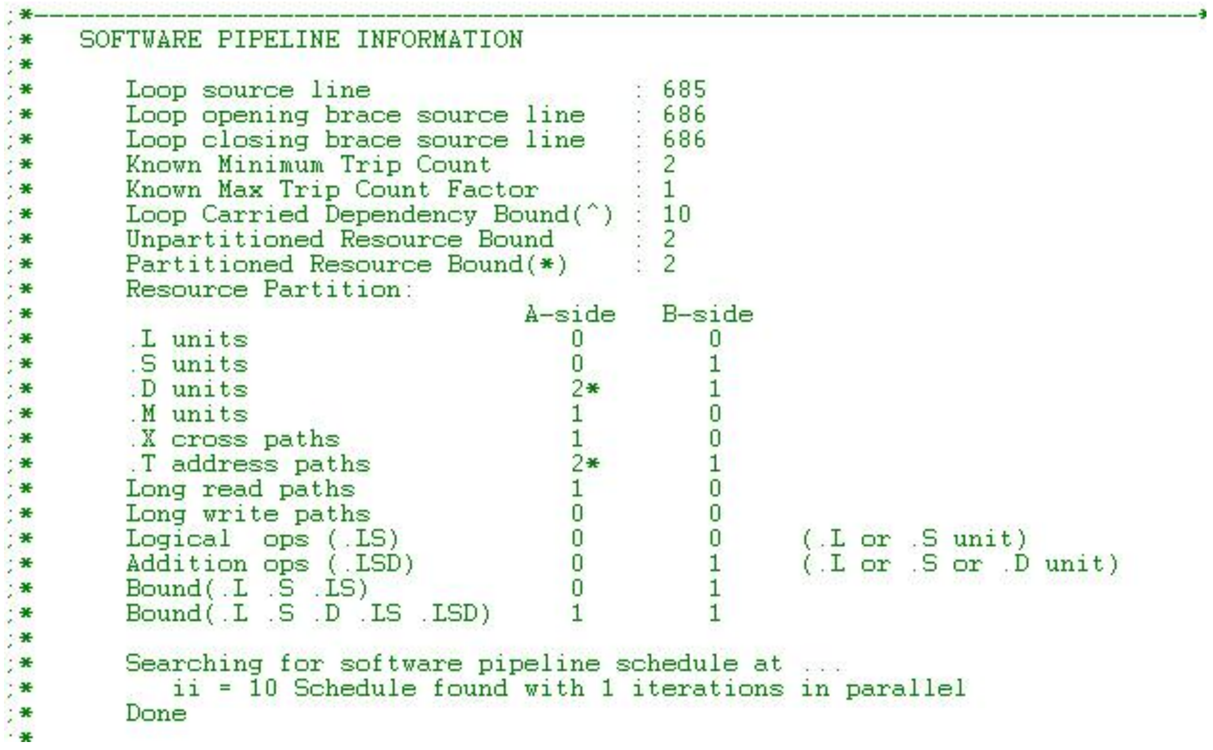


Cycle	Loop Iteration							Phases
1	A							Prologue
2	B	A						
3	C	B	A					
4	D	C	B	A				
5	E	D	C	B	A			Kernel
6		E	D	C	B	A		
7			E	D	C	B	A	
8				E	D	C	B	Epilogue
9					E	D	C	
10						E	D	
11							E	

Figure 2.8: Example of a software pipelined loop

In Figure 2.8, cycle 5 to 7 are known as the kernel. All five stages of instructions from five different iterations are executed in parallel. Cycles 1 to 4 are known as the prologue, which set up the software pipeline. Cycles 8 to 11 are known as the epilogue, which finishes the pipeline. Since all five stages are executed in parallel during the kernel, the iteration interval (ii) is only one cycle. Software Pipelining has the most advantage when the kernel can continue to execute with a minimum iteration interval and all functional units are fully utilized.

Before applying any optimization techniques to improve the code performance, problems must first be identified. A very useful way to identify performance issues, is by using the software pipeline feedback table. This feedback information is generated along with the assembly by the compiler. An example of a feedback table generated using the source code showing in Figure 2.6 is shown in Figure 2.9.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 685
;*      Loop opening brace source line   : 686
;*      Loop closing brace source line   : 686
;*      Known Minimum Trip Count         : 2
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 10
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                     2*       1
;*      .M units                     1        0
;*      .X cross paths               1        0
;*      .T address paths             2*       1
;*      Long read paths              1        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     1        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 10 Schedule found with 1 iterations in parallel
;*      Done
;*
```

Figure 2.9: A feedback table generated with the source code showing in Figure 2.6

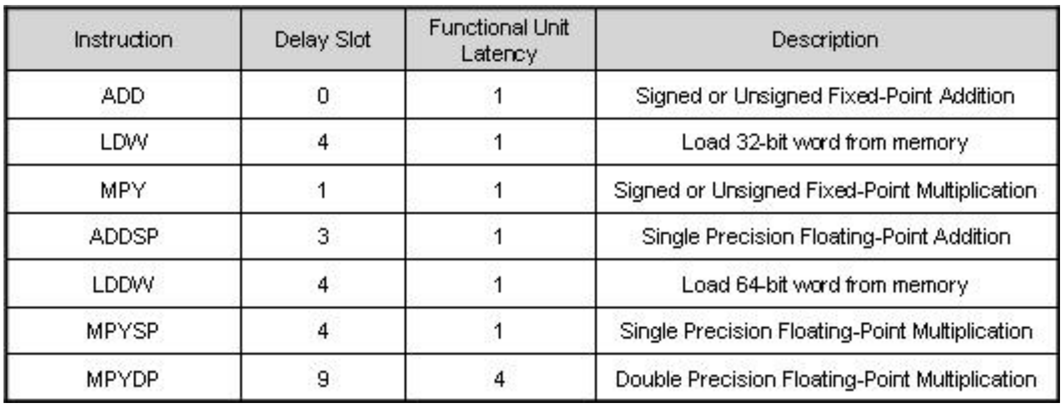
It is very important to understand the information from the feedback table showing in Figure 2.9 as many performance issues can be identified using this information.

- *Loop source line:* The line number of loop declaration in the source code.
- *Loop opening brace source line:* The line number for the loop opening brace, "{".
- *Loop closing brace source line:* The line number for the loop closing brace, " }".

- *Known Minimum Trip Count:* The minimum number of loop iterations determined by the compiler.
- *Known Max Trip Count Factor:* The number of times the loop can be unrolled as determined by the compiler. This factor must be evenly divided into the *Known Minimum Trip Count.*
- *Loop Carried Dependency Bound (^):* The number of cycles needed to execute one iteration of the loop if a loop carried dependency existed. This line can be used to identify any loop carried dependency.
- *Unpartitioned Resource Bound:* The maximum number of times that any particular resource is being used for one iteration. This figure is determined by the compiler before instructions are partitioned to two resource clusters.
- *Partitioned Resource Bound (*):* The maximum number of times that any particular resource is being used for one iteration after instructions are partitioned by the compiler to the two resource clusters. The resource that is used most is indicated by (*) on the A- and B- side resource listing. This information can also be used to identify any uneven partitioning between two resource clusters.
- *.L units:* Floating-point arithmetic execution units. Two units total are available, one on *side-A* and one on *side-B*. The number under each side indicates the number of times this unit is used for one iteration. For example, a “1” under *side-A* for *.L units* means that this floating-point unit on *side-A* is used by one instruction only for one iteration.
- *.S units:* Fixed-point arithmetic execution units. There are total of two, one on *side-A* and one of *side-B*. The number under each side indicates the number of times this unit is used for one iteration.
- *.D units:* Address calculation units. There are total of two, one on *side-A* and one of *side-B*. The number under each side indicates the number of times this unit is used for one iteration.

- *.M units:* Floating-point multiplier units. There are total of two, one on *side-A* and one of *side-B*. The number under each side indicates the number of times this unit is used for one iteration.
- *.X cross path:* Cross path used for transfer data from one side of the register file to the other side for execution. There are two paths total. One path between *side-A* execution units and *side-B* register file and the other path between *side-B* execution units and *side-A* register file.
- *.T address path:* Address path used to pass the calculated address from *.D units* to its associated register file.
- *ii = xx:* Iteration interval. This is the number of cycles needed for one loop iteration determined by the compiler. From the example in Figure 2.9, 10 cycles are required for one loop iteration because a loop carried dependency exists.
- *x iterations in parallel:* This indicates the number of iterations executed in parallel.

2.4.5.1 Instruction Latency Instruction latency is an important factor that affects the performance of the code. Although an instruction can be issued to a functional unit every cycle, the result usually is not available until a few cycles later, this is the instruction latency. The number of cycles required after the execution and before the result is ready is called the delay slot. Different instructions have varied amounts of delay slots. Figure 2.10 shows the delay slot and the functional latency for some common C67x instructions. To minimize the effects of long instruction latency, optimization techniques like Loop Unrolling and Software Pipelining are used. For a full list of C67x instructions delay slot and functional latency, please refer to [24].



Instruction	Delay Slot	Functional Unit Latency	Description
ADD	0	1	Signed or Unsigned Fixed-Point Addition
LDW	4	1	Load 32-bit word from memory
MPY	1	1	Signed or Unsigned Fixed-Point Multiplication
ADDSP	3	1	Single Precision Floating-Point Addition
LDDW	4	1	Load 64-bit word from memory
MPYSP	4	1	Single Precision Floating-Point Multiplication
MPYDP	9	4	Double Precision Floating-Point Multiplication

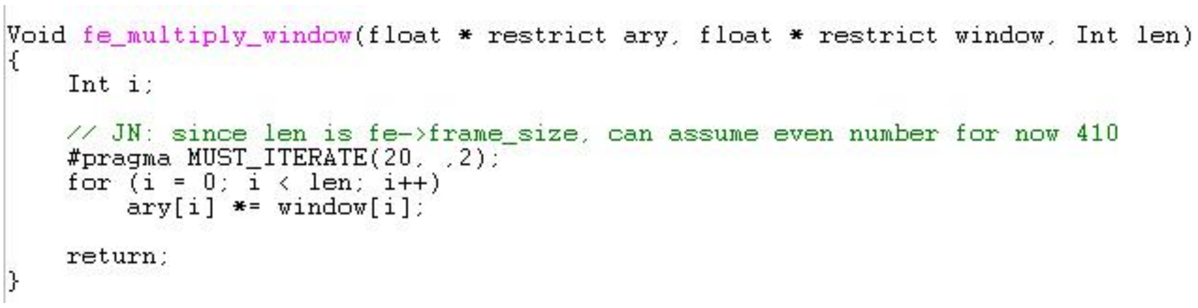
Figure 2.10: Example of delay slot and functional unit latency of some C67x instructions

2.4.6 Loop Unrolling To Balance Resources

The DSP architecture and the number of functional units available limit the performance of a software pipelined loop. The C6713 DSP used in this research has eight functional units divided into two separate channels. Each channel has four units, one type of functional unit each, with two cross path between the channels. The number of functional units available is known as the resource constraint. An example would be a loop with too many load and store operations but too few multiplications. The two-channel architecture also presents aproblem of resource partitioning. For example, if a loop contains three load/store operations, the two channels will not be balanced since one channel would have two operations while the other channel only has one.

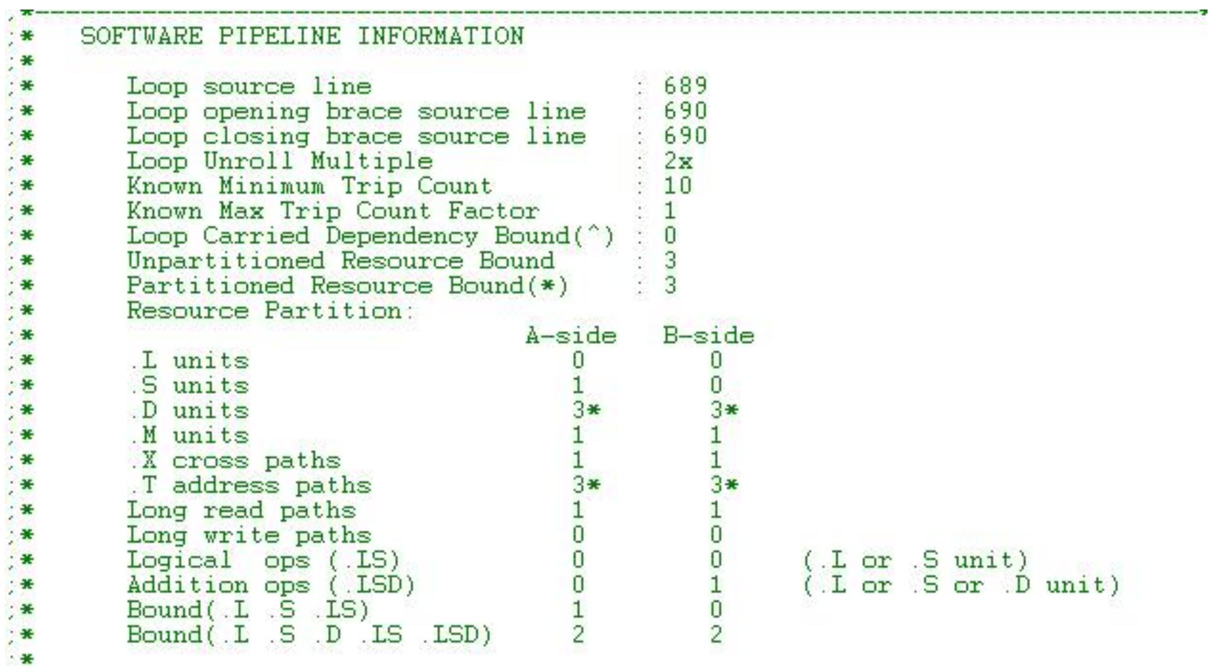
The software pipeline feedback table is a good tool to identify any unbalanced utilization of resources. The line Partitioned Resource Bound (*) from Figure 2.9 shows that there are unbalanced resources between the two channels. The reason is that the loop has an odd number of instructions, two loads and one store that would use both the .D and .T functional units.

To balance the resources for this loop, loop unrolling can be considered. Notice that to perform loop unrolling, the loop count must be a multiple of the unrolling factor. For example, to unroll the loop by a factor of 2 correctly without doing extra calculations, the loop count must also be even. Similarly, to unroll a loop by a factor of 3, the loop count must be a multiple of 3. In Figure 2.6, the compiler does not know the *len* value from the local view of the function so the compiler does not perform loop unrolling. A *MUST_ITERATE (MIN LOOP COUNT, MAX LOOP COUNT, UNROLL FACTOR)* pragma can be used to pass the minimum loop count, maximum loop count (optional) and the unrolling factor information to the compiler. With the information explicitly stated, the compiler can perform loop unrolling for our example loop. Figure 2.11 shows the modified source code and Figure 2.12 shows the newly generated feedback table.



```
Void fe_multiply_window(float * restrict ary, float * restrict window, Int len)
{
    Int i;

    // JN: since len is fe->frame_size, can assume even number for now 410
    #pragma MUST_ITERATE(20, ,2);
    for (i = 0; i < len; i++)
        ary[i] *= window[i];

    return;
}
```

Figure 2.11: Modified source code with restrict type qualifier



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 689
;*      Loop opening brace source line   : 690
;*      Loop closing brace source line   : 690
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 10
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     3*       3*
;*      .M units                     1        1
;*      .X cross paths               1        1
;*      .T address paths             3*       3*
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        0
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
```

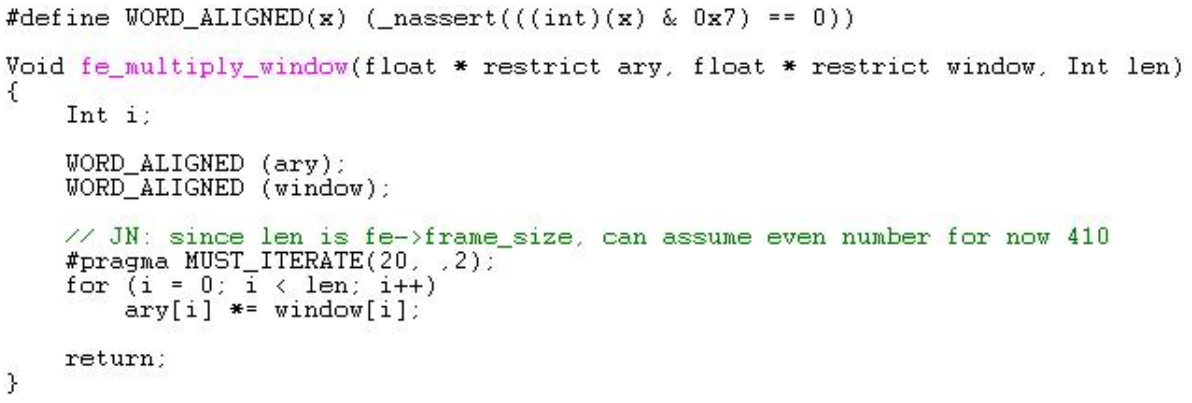
Figure 2.12: Feedback table with modified source code

Looking at the new feedback information, the loop now is unrolled by a factor of 2x (Loop Unroll Multiple). Also, both A-side and B-side channels have the same amount of instructions scheduled.

2.4.7 Packed Data Memory Access

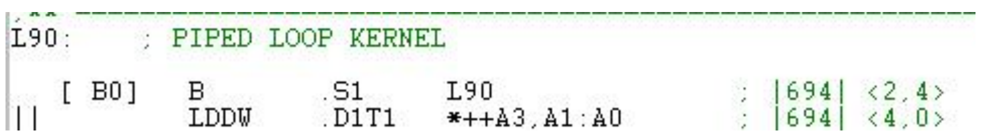
This optimization technique can also help balancing resources for certain situations. The C6713 DSP is capable of loading a single 32-bit word or a 64-bit double word per cycle. An example of the situation where this optimization is suitable would be the example loop shown in Figure 2.6. The loop contains two loads, one multiply and one store. Without unrolling the loop, there will be unbalanced usage on the D function units. Packed data optimization reduces the number of load instruction from two to one. Notice that in order to apply this optimization technique, the data in the memory must be continuous and double-word-aligned.

Double-word-aligned means the lower 3 bits of the address are zero. To explicitly instruct the compiler that data are double-word-aligned, a function is used to assert that data are double-word-aligned. Figure 2.13 shows the modified code from the previous example and Figure 2.14 shows the newly generated assembly with two 32-bit loads is now combined to one 64-bit load.



```
#define WORD_ALIGNED(x) (_nassert(((int)(x) & 0x7) == 0))

Void fe_multiply_window(float * restrict ary, float * restrict window, Int len)
{
    Int i;

    WORD_ALIGNED (ary);
    WORD_ALIGNED (window);

    // JN: since len is fe->frame_size, can assume even number for now 410
    #pragma MUST_ITERATE(20, ,2);
    for (i = 0; i < len; i++)
        ary[i] *= window[i];

    return;
}
```

Figure 2.13: Modified code with new function to assert data are double-word-aligned



```
L90:     ; PIPED LOOP KERNEL

   [ B0]    B       .S1     L90                ; |694| <2,4>
||          LDDW    .D1T1   *++A3,A1:A0        ; |694| <4,0>
```

Figure 2.14: Assembly showing two LDWs combined into one double load LDDW

3.0 OVERVIEW OF AUTOMATIC SPEECH RECOGNITION AND SYSTEM DESIGN SPECIFICATIONS

Automatic Speech Recognition (ASR) is essentially a matching problem. The goal is to find the best match between a set of existing words and an observed speech. Continuous speech recognition extends the matching from single word to a series of words. Figure 3.1 shows the basics of an ASR system in block levels:

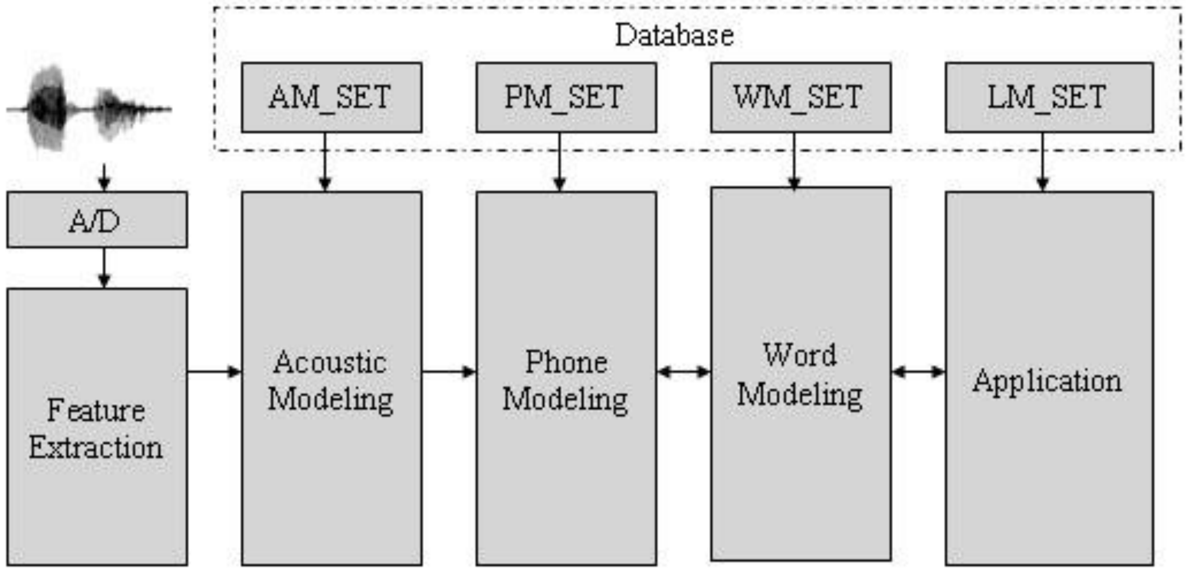


Figure 3.1: Basic ASR system in block levels

The speech signal is first sampled by an analog-to-digital converter. The digital representation of the speech is then processed by the Feature Extraction (FE) block. In FE, characteristics of the speech are extracted to form a vector of 39 features. Each feature corresponds to the characteristics of a different frequency band. A new feature vector is generated every 10 millisecond interval, which is the basic requirement for real-time speech processing. The next three blocks, Acoustic Modeling (AM), Phone Modeling (PM) and Word Modeling (WM) combine to perform word matching at different levels. The feature vector is matched against each word available from the word library. The Word Modeling block keeps track of words that have the probabilities higher than a certain threshold and eliminates words that have probabilities below that threshold. Once a word is recognized by the WM block, it will be passed onto the Application block. Usually the duration of a spoken word requires multiple 10 ms frames. As a result, several frames are needed before a word can be recognized. This is the basic idea of word recognition in ASR. Continuous speech recognition is archived at the Application block. As words are observed by the WM block, a model of inter-words probability from the LM_SET of the Database can be used to determine how these recognized words are related, hence, forming a continuous speech. This task is done by the Application block.

3.1 FORMAL DEFINITION

The goal of ASR is to transcribe speech into words and sentences. From a statistical point of view, the goal is that given an acoustic observation $X = X_{1\ldots} X_n$, find the corresponding word sequence $W = W_{1\ldots} W_n$ that maximizes the posterior probability $P(W|X)$. This expression implies finding the probability of every word within the given word library. For large sized word library, it is very impractical. Instead, Bayes' rule can be applied to decompose the posterior probability, $P(W|X)$, into two components:

$$P(W|X) = \frac{P(X|W)P(W)}{P(X)} \qquad \text{Eq. 3.1}$$

P(W|X) is the probability of the word sequence *W* given the observed sequence *X*. *P(X|W)* is the probability of the observed sequence *X* given the word sequence *W*. *P(W)* is the probability of the word sequence *W*. *P(X)* is the probability of the observed sequence. Equation 3.1 can be further simplified by removing *P(X)* because the probability of the observed sequence is random for every case. Furthermore, the purpose of speech recognition is to find the best match between the observed sequence and the word library**.** Hence, the modified equation looks like the following:

$$\max P(W|X) = \max P(X|W)P(W) \qquad \text{Eq. 3.2}$$

Equation 3.2 indicates that finding the best match is to maximize the product of *P(X|W)* and *P(W)*. The probability of each word, *P(W)*, is generated by comparing the relative occurrence frequency against other words in the dictionary. For example, the word "the" has a much higher probability of occurrence than other words like "coefficients" or "Gaussian". *P(W)* is used by the Word Modeling block and is pre-generated and stored in the WM_SET of the Database. *P(X|W)* is determined by the Acoustic Modeling and the Phone Modeling blocks.

3.2 BLOCK LEVEL OVERVIEW

This section provides a functional overview for each processing block in Figure 3.1. From Figure 3.1, speech is first sampled then propagated from the FE block through AM, PM, WM and Application blocks. This system model is simple and straightforward since data flow sequentially and all words are evaluated against the sampled speech. However, evaluating all words is sometime impossible to achieve in real-time for certain applications. All words needed to be evaluated due to the assumptions that any word is possible to be the starting word of a sentence, and any word can transit into another word. However, in reality, the probability of certain words being the starting word is very small relative to other words. For example, the word "he" has a higher probability of being the start of a sentence than the word "only". Hence, the number of computations can be reduced by only evaluating those probable, or "active", words at any given frame. Tracking these active words is the responsibility of the Application block.

For a word library that only contains a few words, it may be possible to recognize each word as a whole. However, for a large word library that contains similar words, it is no longer possible to recognize each word as a whole. In order to distinguish similar words; all words are further divided into multiple sub-word units called *phonemes*. With this new word definition, ASR can be viewed as phoneme recognition instead of word recognition. While the Application block is responsible for word tracking, WM block is responsible for phoneme tracking. Similar to word tracking, phoneme tracking is necessary because not all phonemes can be the starting part of a word and not all phonemes are allowed to be followed by any phonemes. Only those active phonemes are evaluated by the Phone Modeling block at any given frame. The list of active phonemes passed from WM to PM is referred as Feedback. The system model with Feedback is more complex as extensive data management is required. However, from the computational stand point, Feedback helps improve the overall system performance by reducing the required number of evaluations. A modified block diagram with Feedback is shown in Figure 3.2. Pseudo-code of the data flow with Feedback is also provided in Figure 3.3.

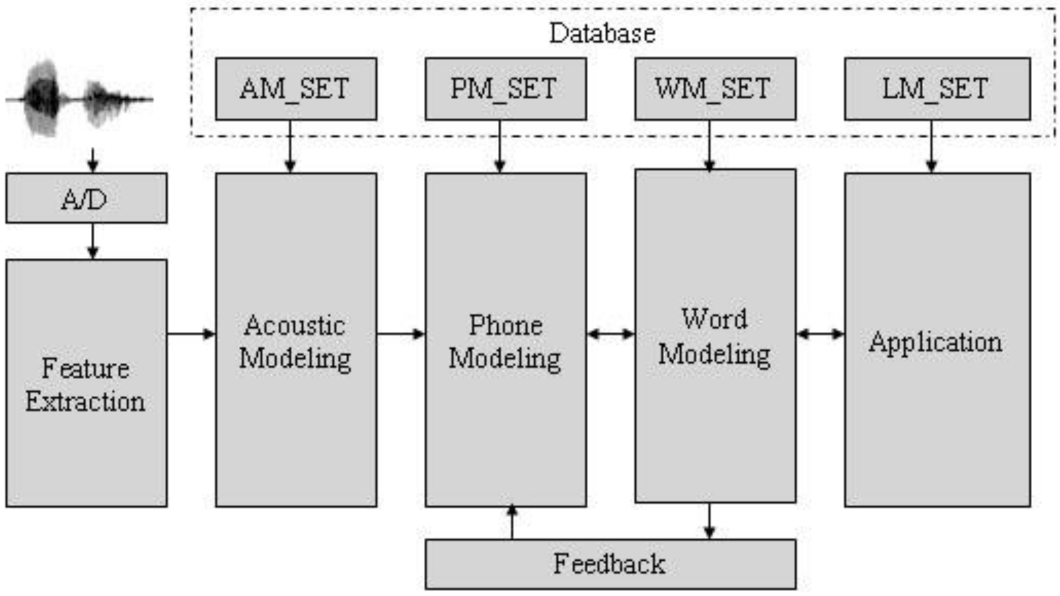
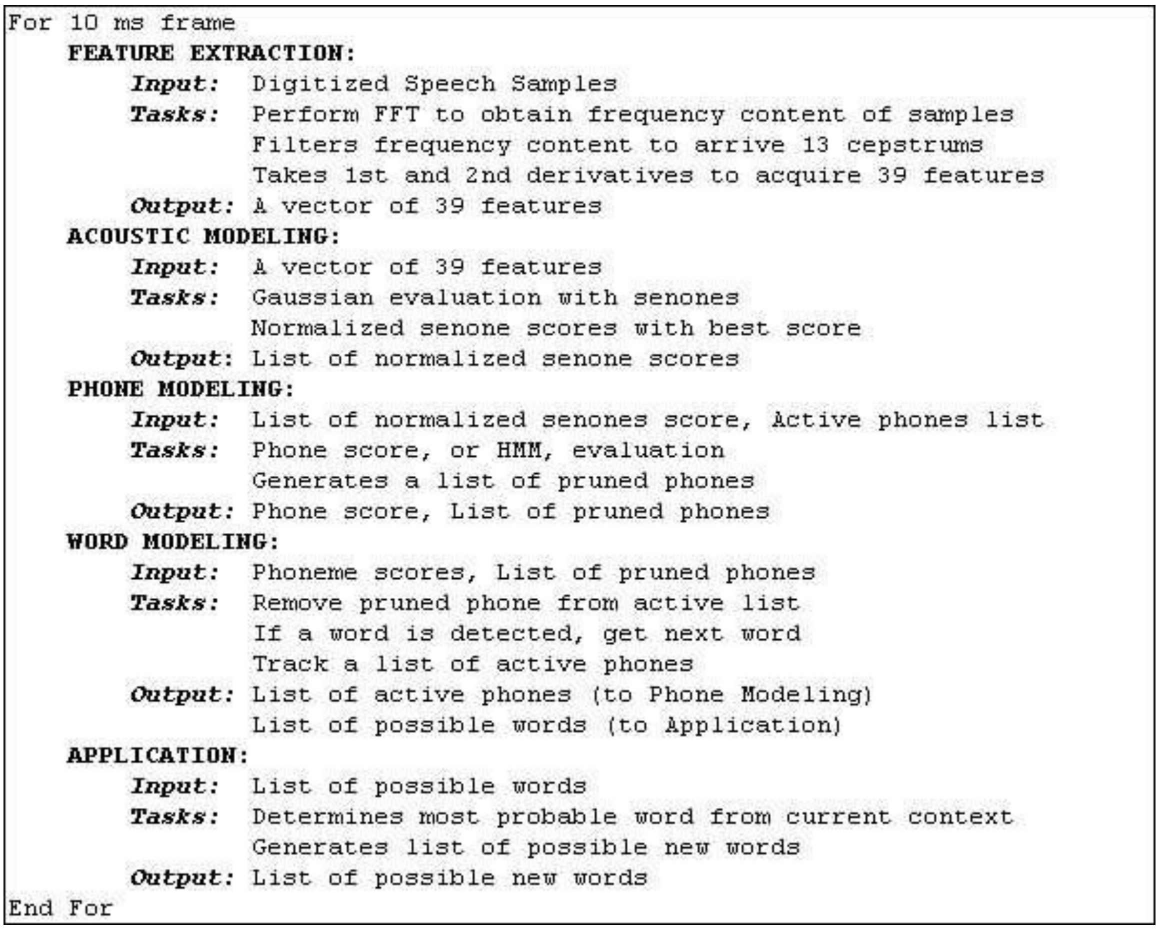


Figure 3.2: Block diagram for ASR with Feedback



```
For 10 ms frame
    FEATURE EXTRACTION:
        Input:   Digitized Speech Samples
        Tasks:   Perform FFT to obtain frequency content of samples
                 Filters frequency content to arrive 13 cepstrums
                 Takes 1st and 2nd derivatives to acquire 39 features
        Output: A vector of 39 features
    ACOUSTIC MODELING:
        Input:   A vector of 39 features
        Tasks:   Gaussian evaluation with senones
                 Normalized senone scores with best score
        Output: List of normalized senone scores
    PHONE MODELING:
        Input:   List of normalized senones score, Active phones list
        Tasks:   Phone score, or HMM, evaluation
                 Generates a list of pruned phones
        Output: Phone score, List of pruned phones
    WORD MODELING:
        Input:   Phoneme scores, List of pruned phones
        Tasks:   Remove pruned phone from active list
                 If a word is detected, get next word
                 Track a list of active phones
        Output: List of active phones (to Phone Modeling)
                 List of possible words (to Application)
    APPLICATION:
        Input:   List of possible words
        Tasks:   Determines most probable word from current context
                 Generates list of possible new words
        Output: List of possible new words
End For
```

Figure 3.3: Block level pseudo-code for data flow

The pseudo-code in Figure 3.3 shows the inputs, outputs along with all the tasks performed for each of the processing block.

The Database divides the word library into four datasets, one for each processing block as shown in Figure 3.2. Each dataset contains the necessary information needed for each block to complete its tasks. For example, the AM dataset stores the means and covariants needed for Gaussian evaluations.

3.3 Performance Characteristics

The differences in functionality have created different performance characteristics and memory bandwidth requirements for each processing block. The performance of these speech recognition processes had been characterized and profiled in [10]. It shows that FE takes up less than 1% of the overall computation cycles while AM takes up about 55%-95% and PM combined with WM consumes about 5%-45%. The actually percentages depend mainly on the size of the word library used. The larger the dictionary size, the more the performance of AM becomes significant over the other blocks.

The most computation intensive function in Feature Extraction is the Fast Fourier Transform (FFT), which converts the data samples from the time scale to the frequency scale. Utilizing the hand optimized signal processing library [26] available along with the DSP, the amount of computational cycles required to perform FFT is reduced to almost negligible relative to other parts of the recognition process, which corresponds to [10]. Also, all data required with FE can be stored within the on-chip SRAM which eliminates the performance effect that will be caused by accessing to the external memory.

The main function of Acoustic Modeling is to take the sample characteristics and match them against its own library. Part of this process is the Gaussian evaluation. Each of these evaluations includes two multiplications and two subtractions. The size of the dictionary dictates the number of Gaussian evaluations required. For the 1000 word dictionary used in this research, approximately 604,000 evaluations are performed every 10 milliseconds, which translates into 1.2 million multiplications and subtractions each. In Sphinx 3, all of these computations are evaluated in float point precision. Other research [1] had shown that reducing the bit precision does not impose significant decrease in system performance; however, the number of computations does not change.

While it is a challenge to perform such massive amounts computations with a limited amount of computing cycles, the bigger problem is feeding these computations with the required data. Each Gaussian evaluation takes three inputs and produces one output. Depending on the method of implementation, the required number of input data can be varied. However, regardless of the implementations, a minimum of two inputs are necessary. With each input data stored with 4 bytes each regardless of fixed-point or floating-point, approximately 4.83 Megabytes of data bandwidth is needed for every 10 ms. This amount of data is larger than most of the cache available on any system. As a result, these data will be stored in the external memory.

The primary function of Phone Modeling is to evaluate all of the active phonemes within a 10 ms frame. Phonemes are made active based on the previous frame of data by the Word Modeling block. In Sphinx 3 and for this research, each phoneme is modeled as a 3 states Hidden Markov Model (HMM). More information about HMM will be discussed in (section.6.0) Computational wise, each HMM evaluation consists of 9 additions and 3 comparisons. All operations are done in integer form. From the study of [1], at most approximately 4000 phonemes can be active at any given frame for the 1000 words dictionary, which are significantly fewer computations compared to AM. On the other hand, the amount of data required per HMM evaluation is more than Gaussian evaluation. A total of 15 integers are needed for each HMM evaluation. Further, unlike AM where data are accessed in a regular manner, data required for PM are from multiple different locations and these locations are more random and irregular. This type of irregular memory access pattern presents a different type of challenge than AM.

The final stage considered in this research is Word Modeling. WM keeps track of all phonemes, removing unpromising ones and adding new ones to the active list. Since the operation of WM is tightly coupled with PM, they will be analyzed together from this point and on.

Summarizing these performance characteristics, it should be clear that AM should be the main focus since AM requires the most processing power and the largest memory bandwidth.

3.4 TIMING REQUIREMENT

The standard technique used for speech processing is based on frame processing. Sphinx 3 processes speech data at 10 millisecond intervals. In order to be real-time, all computations must be completed within the 10 ms interval. The number of computation cycles available, however, depends on the speed of the processor. For example, if the processor is operating at 100 MHz, that means all calculations must be done in 1 million cycles, 1/100 of a second. For this research, the DSP operates at 225 MHz. As a result, the cycle budget available for this research to satisfy the real-time requirement is 2,250,000 cycles.

3.5 PERFORMANCE MEASUREMENTS

As presented in the (section 3.3), different processing blocks have different computational and memory requirements. Feature Extraction is the least computational intensive relative to other parts of the recognition process. Also the memory footprint is small enough to be stored entirely in the on-chip SRAM. However, in embedded environment where a high speed processor isn't available, FE remains a performance bottleneck. As a result, FE must also be optimized using several compiler optimizations. The optimized performance of FE improved 332% from 231,000 cycles to 69,000 cycles, which is 3.07% of the real-time budget.

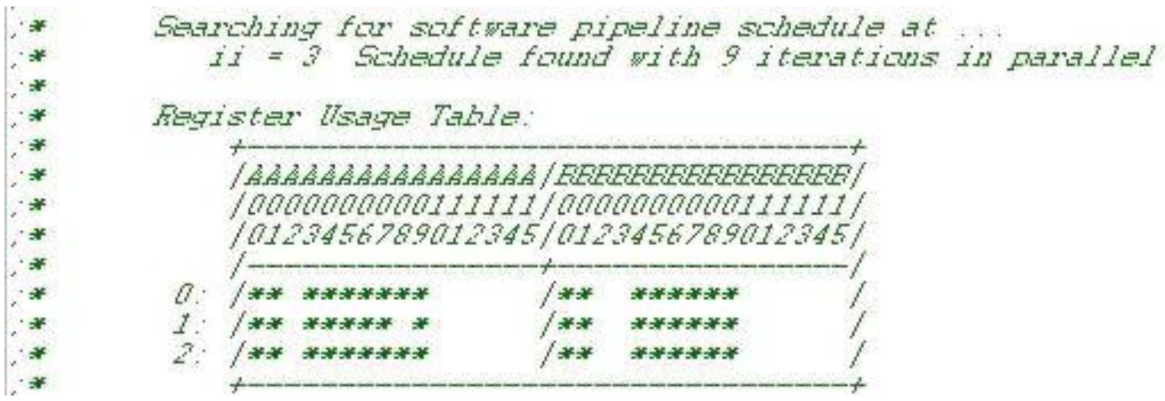
Acoustic Modeling is the most computational intensive and requires the largest memory bandwidth in the entire recognition process. For these reasons, most efforts are spent trying to optimize AM. AM implementation presents two separate problems: a large number of computations and massive memory bandwidth. AM performs 1.2 million multiplications and subtractions each. For a fair comparison with Sphinx 3, all operations are performed in floating-point precision. With eight functional units available where two of them are float-point multipliers and two are floating-point ALUs, it seems possible to execute a single Gaussian evaluation per cycle. However, data dependency and functional unit latency prevent the achievement of such performance. Data dependency exists due to the fact that not all 4 operations are independent. In other word, the output of one operation is the input for the other operations. Further, although all functional units are capable of executing most instructions in a single cycle, but there exists some latency before the result is available. This latency is due to the deep pipeline architecture of these functional units.

Although it is impossible to archive one cycle per Gaussian evaluation with the available DSP architecture, it is possible to get close to it. Two different implementations methods were analyzed in this research where each of them is optimized through software pipeline and various compiler techniques. The best performance obtained is 1.25 cycle per Gaussian evaluation resulting an approximately 754,000 total cycles. This result is obtained based on the assumption that all data are available to the processor without any latency. In reality, this is impossible since the L1 cache size is too small. The DSP and cache architecture used in this research impose a 4 cycles delay whenever L1 misses and L2 hits, regardless of L2 being on-chip SRAM or cache. Further, if L2 misses and external memory is accessed, over 100 cycles of delay is possible. To eliminate this memory latency effect, the L2 cache is configured to be used as all on-chip SRAM where it is divided into two different buffers. While one buffer is being accessed by the processor, the other buffer can be filled with new data using DMA. This method will completely eliminate the effect of external memory latency. Since the line size of the L1 cache is 8 32-bits words, the actual performance would now be the total number of data required divided by 8 and multiples by 4 cycles. With 603,000 total evaluations and 3 pieces of float-point data each, total of 1.81 million float-point data is needed. Dividing by 8 and multiplying by 4 resulting 905,000 cycles. With the processor running at 225 MHz, a 10 ms frame would have 2.25 million real-time cycle budget. After optimizing AM, the overall cycle performance would be 1,659,000 (computation + memory latency, 754,000 + 905,000), which is about 73.7% of the real-time cycle budget.

Finally, for PM and WM, all computations are integer based with no data dependency. The only problem is that PM and WM also require a fairly large memory bandwidth and the access pattern is more irregular. Due to the irregularity, DMA would not be as useful in this case since data are coming from different locations of the memory rather than accessed as a block in sequential fashion. The only solution would be to have all the required data to be stored in the on-chip SRAM. However, since PM and WM do not present any computational issues, they are only implemented for the completeness of the project.

3.5.1 Register Utilization

Registers are temporary stores within the processor. Their purpose is to allow the processor to have quick access to often used variables. Certain complex algorithms require storing more temporary variables than the number of registers available. Whenever the processor runs out of register, external memory is accessed. In other word, the number of registers available limits the performance of execution of an algorithm. The registers also play an important role on the efficiency of a software pipeline code since the number of registers along with the number of execution units available limits how many iterations of a loop can be execute in parallel. An example of a register utilization table generated by the compiler is shown in Figure 3.4.



```
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 9 iterations in parallel
;*
;*      Register Usage Table:
;*          +---------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: |** *******      |**  ******      |
;*       1: |** ***** *      |**  ******      |
;*       2: |** *******      |**  ******      |
;*          +---------------------------------+
```

Figure 3.4: An example of a compiler generated register utilization table

As mentioned before, C6713 DSP has 32 registers that are divided into 2 register files, each has 16 registers. The example showing in Figure 3.4 indicates that each loop takes 3 cycles to execute (ii = 3). As a result, the Register Usage Table shows 3 cycles of usage, 0-2. Each (*) indicates that register is used in that particular cycle. For example, registers A00, A01, A03-A09 are used in cycle 0. Register utilization is used as a measurement to determine how well an algorithm is optimized. In other word, a 100% utilization of registers for every cycle is best while more unused register means more possible optimizations can be applied.

4.0 FEATURE EXTRACTION

This section discusses the theory, implementation and optimizations used for Feature Extraction (FE). Although FE takes up less than 1% of the overall cycle budget on systems with high speed processors, FE can become a significant performance issue in embedded environments. As a result, FE must also be optimized using various compiler optimization techniques. The optimized performance of FE improved 332% from 231,000 cycles to 69,000 cycles, which is 3.07% of the real-time budget.

Feature extraction, often referred to as the front-end processing, generates a set of 39-dimension features representing the important characteristics of the digitized speech samples. This is accomplished by dividing the input speech samples into blocks and derives a smoothed spectral estimate from each divided block. The typical spacing of each block is 10 milliseconds, resulting 100 frames per second. To obtain the required spectral estimates, numerous processes have been developed [17]. However, the most standard method, Mel-Frequency Cepstral Coefficients (MFCC) [18], is used by Sphinx 3. MFCC is a representation of a windowed signal derived from the FFT of that signal. The process of MFCC is divided into six stages as shown in Figure 4.1 and is described in detail in the following sections.

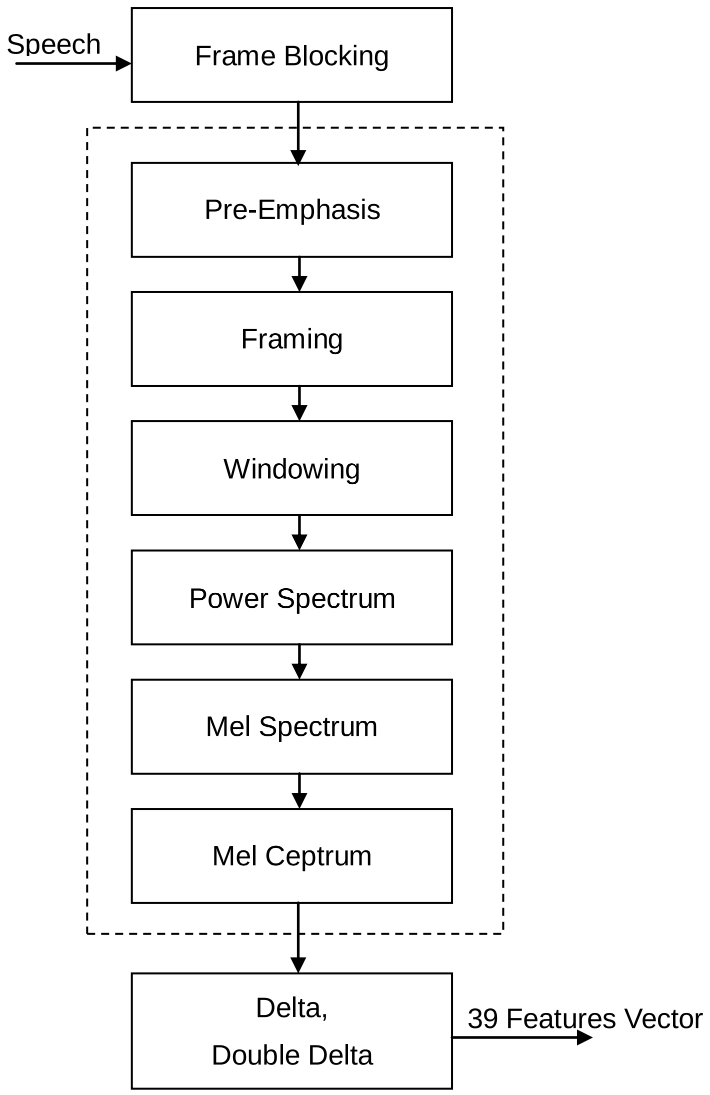


Figure 4.1: Different stages of Mel Frequency Cepstral Coefficients

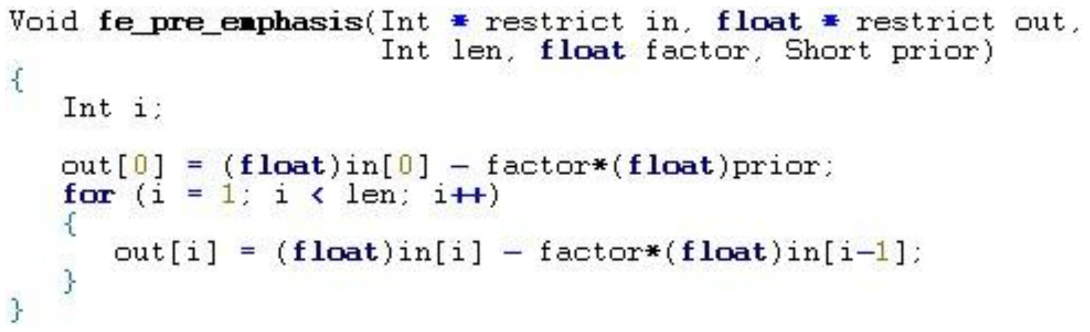
4.1 PRE-EMPHASIS

In the spectrum of a human speech signal, the energy of the signal decreases as the signal frequency increases. The pre-emphasis process is applied to the input speech samples by using a first order FIR filter to increase the signal energy inversely proportional to its signal's frequency. This will equalize the power across all frequencies. The computation performed in Sphinx is show in Equation 4.1.

$y[n] = x[n] - \alpha x[n-1]$, where $0.9 <= \alpha <= 1$ Eq. 4.1

In this research, α is set to be 0.97, same as Sphinx. This operation is performed on every speech sample.

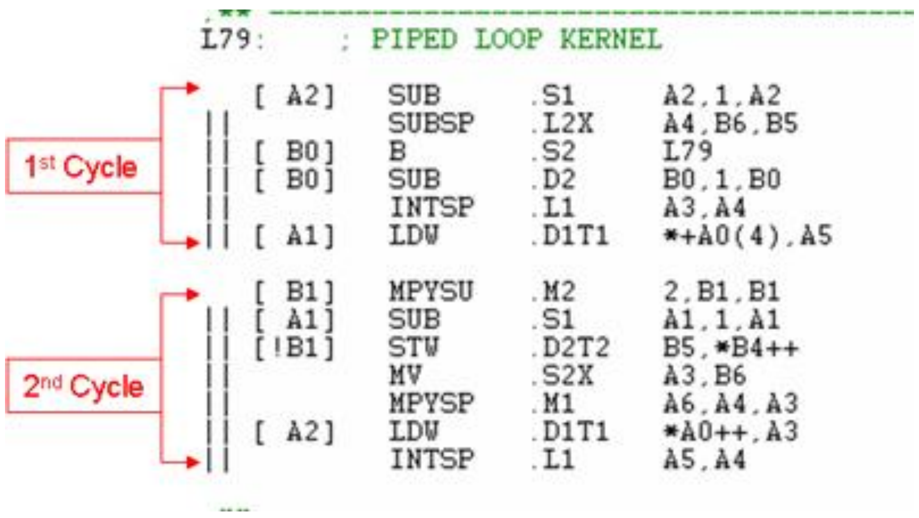
Pre-emphasis is implemented as a for-loop when each loop iteration performs a multiplication and an addition. The source code is shown in Figure 4.2.



```
Void fe_pre_emphasis(Int * restrict in, float * restrict out,
                     Int len, float factor, Short prior)
{
   Int i;

   out[0] = (float)in[0] - factor*(float)prior;
   for (i = 1; i < len; i++)
   {
      out[i] = (float)in[i] - factor*(float)in[i-1];
   }
}
```

Figure 4.2: Implementation code for pre-emphasis

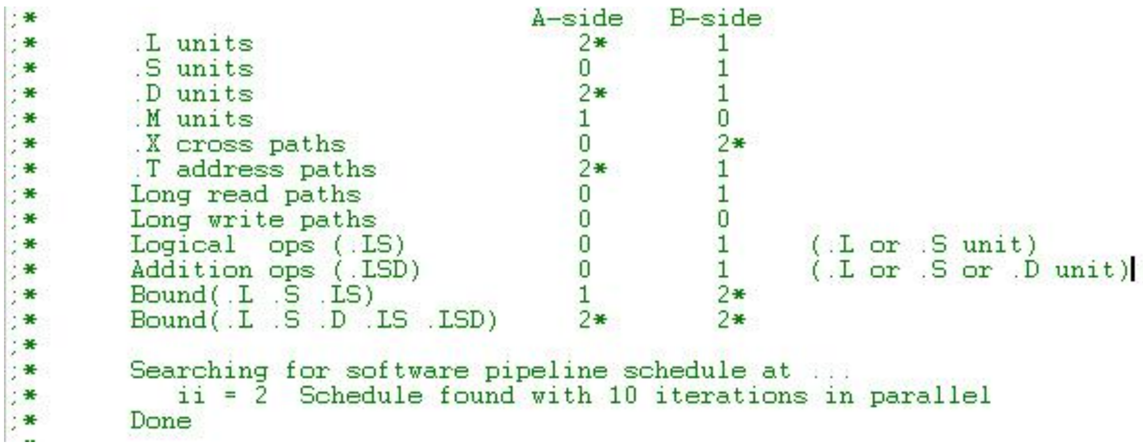
Since all iterations are independent of each other, there is no data dependency issue. Further, *restrict* keyword is used to guarantee that *in* and *out* are non-overlapping arrays. However, Loop unrolling, Packed Data Load and other techniques are not applicable due to the fact that the loop count can be varied depending on the sampling rate. Finally, this loop is further optimized with software pipeline. The optimized assembly is shown in Figure 4.3.



```
;** ----------------------------------------
L79:      ; PIPED LOOP KERNEL

        [ A2]   SUB     .S1     A2,1,A2
||              SUBSP   .L2X    A4,B6,B5
||      [ B0]   B       .S2     L79
||      [ B0]   SUB     .D2     B0,1,B0
||              INTSP   .L1     A3,A4
||      [ A1]   LDW     .D1T1   *+A0(4),A5

        [ B1]   MPYSU   .M2     2,B1,B1
||      [ A1]   SUB     .S1     A1,1,A1
||      [!B1]   STW     .D2T2   B5,*B4++
||              MV      .S2X    A3,B6
||              MPYSP   .M1     A6,A4,A3
||      [ A2]   LDW     .D1T1   *A0++,A3
||              INTSP   .L1     A5,A4
```

Figure 4.3: Optimized assembly showing pipelined instruction

Figure 4.3 shows that the loop kernel is made up of two sets of parallel instructions. During the 1st cycle, six instructions are executed including 1 data load (LDW) and 1 floating-point subtraction (SUBSP). The 2nd cycle executes 7 instructions including another LDW and a floating-point multiplication. Finally the optimized result is shown in the software pipeline table in Figure 4.4.



```
;*                                A-side   B-side
;*      .L units                    2*       1
;*      .S units                    0        1
;*      .D units                    2*       1
;*      .M units                    1        0
;*      .X cross paths              0        2*
;*      .T address paths            2*       1
;*      Long read paths             0        1
;*      Long write paths            0        0
;*      Logical  ops (.LS)          0        1     (.L or .S unit)
;*      Addition ops (.LSD)         0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            1        2*
;*      Bound(.L .S .D .LS .LSD)    2*       2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 10 iterations in parallel
;*      Done
```

Figure 4.4: Optimized result showing in the software pipeline table

As indicated in Figure 4.4, it takes 2 cycles to complete an iteration (ii = 2) and 10 iterations are executed in parallel. In other words, the overall performance of this loop is *2n* cycles. The *n* in this research is set to be 2000 and so the overall performance would be 4000 cycles with the assumption that all data is available in the L1 cache. However, this is not the case since all data will reside in the L2 SRAM. The cache miss effect would be the total number of samples divided by the L1 cache line size, multiplied by L1 cache miss penalty. Since each sample is a short type integer, the total effect would be 2000 sample x 2 bytes / 32 bytes cache line x 4 cycles penalty = 500 cycles. The combined result is 4500 cycles. Table 4.1 summarizes the performance gain for Pre-emphasis.

Table 4.1: Performance summary for Pre-Emphasis

Pre-emphasis	Performance	Improvement
*Standard Optimizations	14,500 cycles	-
Optimized with MA & SP	4500 cycles	322%

*Standard optimizations are Variable Registering and Constant Propagation
** MA = Memory Aliasing; SP = Software;

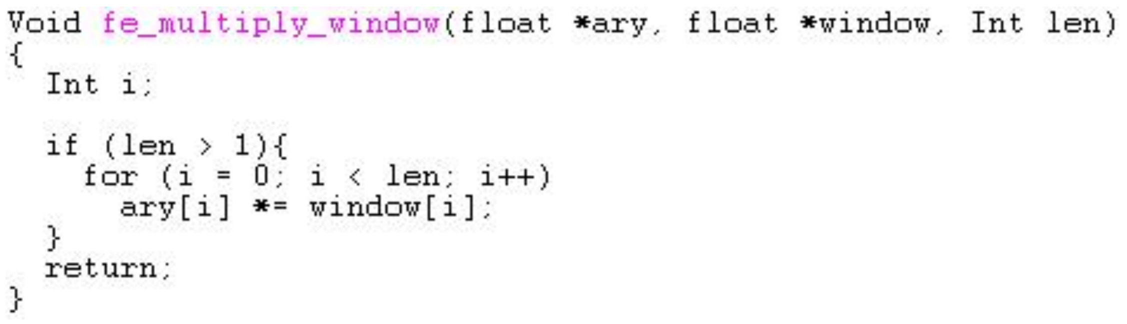
4.2 WINDOWING

The remaining operations are performed on a frame basis. Each frame consists of 2000 speech samples with 160 samples overlapped from the previous frame.

Each of the 2000 sample frames are multiplied with a Hamming Window to minimize the effect of discontinuities at the edges of the frame during Fast Fourier Transform performed in the later stage. The Hamming Window is shown in Equation 4.2 and this operation is performed 2000 times, same as the size of a frame.

$$w[n] = 0.54 - 0.46 \cos \frac{2 \quad n}{N \quad 1} \text{, where N = length of the frame} \qquad \text{Eq. 4.2}$$

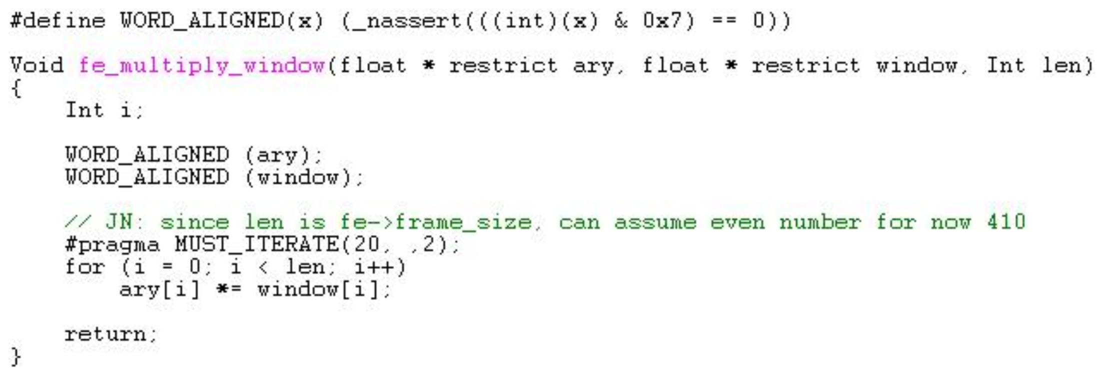
The Windowing function simply multiplies the Pre-Emphasis data samples with a set of Hamming Windows coefficients. It is implemented as a simple for-loop. The Sphinx version of the source code without any optimizations is shown in Figure 4.5.



```
Void fe_multiply_window(float *ary, float *window, Int len)
{
  Int i;

  if (len > 1){
    for (i = 0; i < len; i++)
      ary[i] *= window[i];
  }
  return;
}
```

Figure 4.5: Sphinx version source code for Windowing function

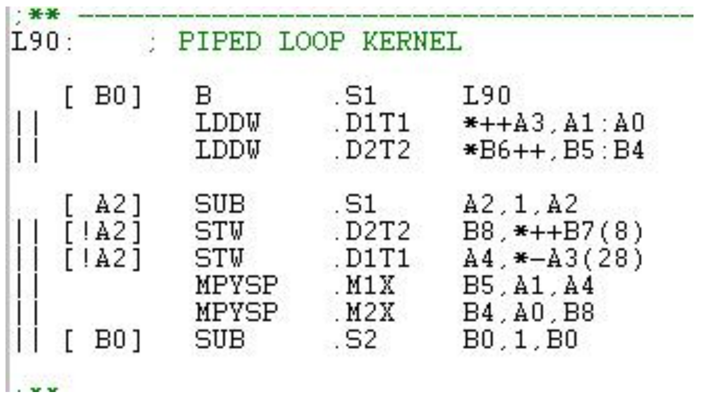
Unlike Pre-Emphasis, where the loop count is variable, the loop count (*len*) for the Windowing function is fixed at 410 for this research. Further, knowing that the loop count is an even number, Loop Unrolling and Packed Data Load optimizations can be applied. It is also clear that there is no data or loop carried dependency and the compiler should be guaranteed that no memory aliasing between the input and output arrays use the *restrict* keyword. The optimized source code is shown in Figure 4.6.



```
#define WORD_ALIGNED(x) (_nassert(((int)(x) & 0x7) == 0))

Void fe_multiply_window(float * restrict ary, float * restrict window, Int len)
{
    Int i;

    WORD_ALIGNED (ary);
    WORD_ALIGNED (window);

    // JN: since len is fe->frame_size, can assume even number for now 410
    #pragma MUST_ITERATE(20, ,2);
    for (i = 0; i < len; i++)
        ary[i] *= window[i];

    return;
}
```

Figure 4.6: Optimized source code for Windowing function

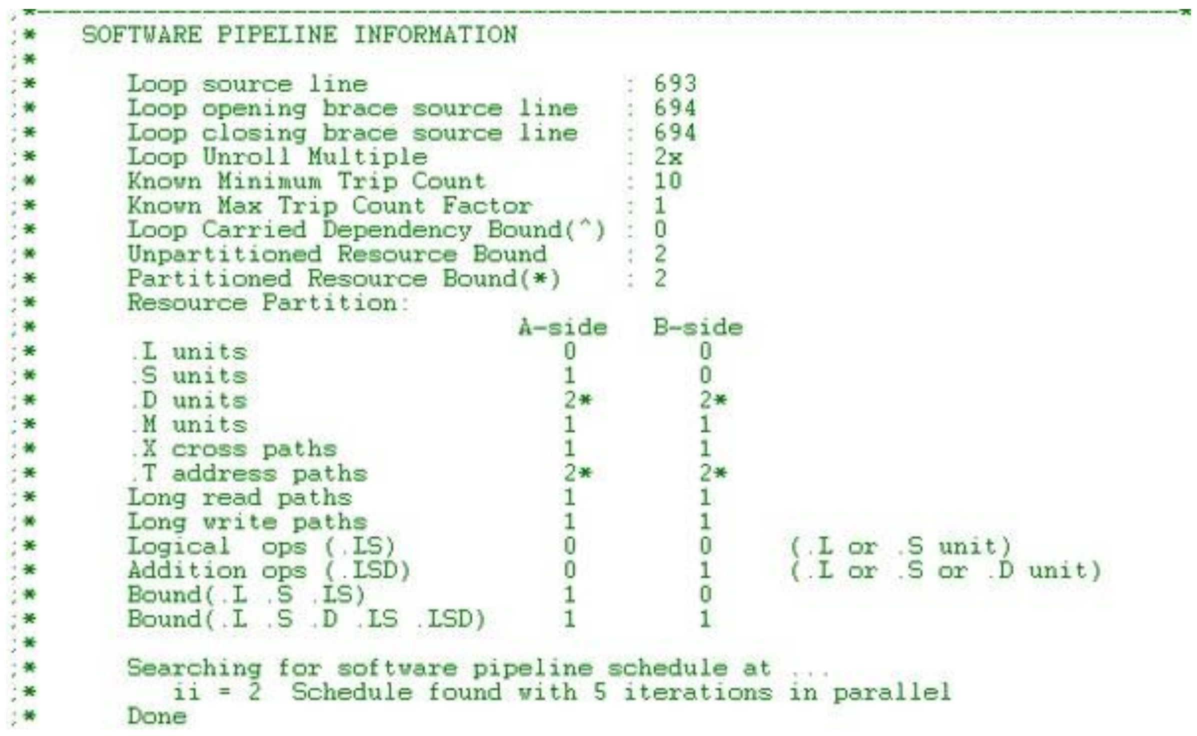
As shown in Figure 4.6, each array is guaranteed that there are no memory overlaps exist. Further improvement can be made by unrolling the loop by a factor of 2. This is done by using the MUST_ITERATE pragma. With the loop unrolled by 2 means that each cycle would require 4 pieces of input data instead of 2. Packed Data Load can be used to load 4 32-bit words as long as the compiler is guaranteed that data is aligned in 64-bit. The WORD_ALIGNED macro is used to explicitly instruct the compiler that data is double-word (64 bits) aligned so that double-word load is safe. Finally, software pipeline is also enabled so that multiple instructions can be executed in parallel. The final optimized assembly is shown in Figure 4.7.



```
;** --------------------------------------
L90:      ; PIPED LOOP KERNEL

   [ B0]   B       .S1     L90
||         LDDW    .D1T1   *++A3,A1:A0
||         LDDW    .D2T2   *B6++,B5:B4

   [ A2]   SUB     .S1     A2,1,A2
|| [!A2]   STW     .D2T2   B8,*++B7(8)
|| [!A2]   STW     .D1T1   A4,*-A3(28)
||         MPYSP   .M1X    B5,A1,A4
||         MPYSP   .M2X    B4,A0,B8
|| [ B0]   SUB     .S2     B0,1,B0
;**
```

Figure 4.7: Optimized assembly for Windowing function

Illustrated in Figure 4.7 is that instructions are executed in parallel (indicated by ||) and that all 4 pieces of data are loaded in the same cycle (indicated by LDDW). The final software pipeline feedback table is shown in Figure 4.8.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 693
;*      Loop opening brace source line   : 694
;*      Loop closing brace source line   : 694
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 10
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     2*       2*
;*      .M units                     1        1
;*      .X cross paths               1        1
;*      .T address paths             2*       2*
;*      Long read paths              1        1
;*      Long write paths             1        1
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        0
;*      Bound(.L .S .D .LS .LSD)     1        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 5 iterations in parallel
;*      Done
```

Figure 4.8: Software pipeline feedback table for Windowing function

The final result shows that (by ii) each loop rotation takes 2 cycles to complete while 5 iterations are executed in parallel. Again, assuming all data is available in L2 SRAM with 4 cycles of penalty for every L1 cache miss, the overall performance would be 410 x 2 = 820 cycles plus cache miss penalty, which are 412 x 2 / 8 x 4 = 410 cycles. Total cycles would be 1230 cycles. Table 4.2 summarizes the performance of Windowing.

Table 4.2: Performance summary for Windowing

Windowing	Performance	Improvement
*Standard Optimizations	2,050 cycles	-
Optimized with MA, SP, LU & PD	1,230 cycles	167%

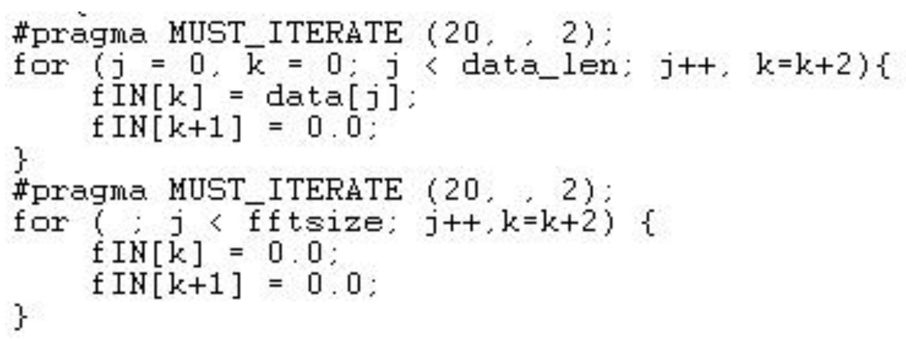
*Standard optimizations are Variable Registering and Constant Propagation
** MA = Memory Aliasing; SP = Software; LU = Loop Unrolling; PD = Packed Data Memory Access

4.3 POWER SPECTRUM

The power spectrum process obtains the frequency domain representation of the time domain windowed speech samples. This is accomplished by performing Fast Fourier Transform (FFT). Before FFT can be applied, the 2000 samples are zero-padded to the length of a power of 2. Instead of using the FFT algorithm that is used in Sphinx, a hand optimized FFT algorithm from Texas Instruments (TI) is used. This hand optimized algorithm is part of the signal processing library made available from TI to be used on their DSPs. Once the frequency spectrum is acquired, the square of the magnitude is then computed to obtain a real result instead of the complex output from FFT. Equation 4.3 shows the computation of the square magnitude.

$$S[k] = (\mathrm{real}(X[k]))^2 + (\mathrm{img}(X[k]))^2, \text{ where } 0<n<=N/2 \qquad \text{Eq. 4.3}$$

Power Spectrum is implemented in 3 different stages: Pre-FFT, FFT and Square Magnitude. Pre-FFT is basically data rearrangement. It is made up of 2 FOR loops. After eliminating Memory Alias and performing Loop Unrolling, the source codes looks like in Figure 4.9:

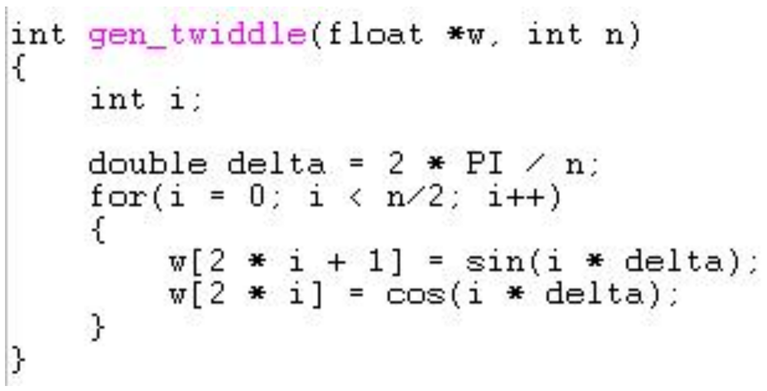


```
#pragma MUST_ITERATE (20, , 2);
for (j = 0, k = 0; j < data_len; j++, k=k+2){
    fIN[k] = data[j];
    fIN[k+1] = 0.0;
}
#pragma MUST_ITERATE (20, , 2);
for ( ; j < fftsize; j++,k=k+2) {
    fIN[k] = 0.0;
    fIN[k+1] = 0.0;
}
```

Figure 4.9: Pre-FFT stage source code for Power Spectrum function

The performance of the first loop is 1.5 cycle/iteration and the second loop is 1 cycle/iteration. With *data_len* = 410 and *fftsize* = 512, the computation performance is 1.5 x 410 = 615 cycles plus 512 cycles, total of 1127 cycles. L1 cache miss penalty would be 410 / 8 x 4 = 105 cycles.

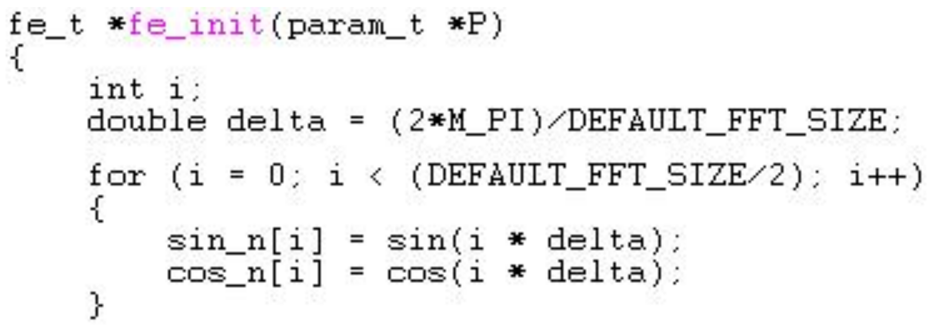
The FFT section is further divided into 4 sub-stages: generates twiddle-factors, bit reverses the twiddle factors, 512 point single precision floating point FFT and Bit reverses the FFT result. The *gen_twiddle* function is extracted from the TI C6713's Signal Processing library. The source code is shown in Figure 4.10.



```
int gen_twiddle(float *w, int n)
{
    int i;

    double delta = 2 * PI / n;
    for(i = 0; i < n/2; i++)
    {
        w[2 * i + 1] = sin(i * delta);
        w[2 * i] = cos(i * delta);
    }
}
```

Figure 4.10: Generates twiddle-factors source code from TI

Due to the function calls (sin and cosine) inside the for-loop, none of the optimization techniques can be applied. One way to improve the run-time performance of this loop is to pre-calculate the sin and cosine values during setup time and performing a table lookup during run-time. However, this optimization is desired only if the run-time performance is more important than the size of the memory footprint. The pre-calculations source code is listed in Figure 4.11.



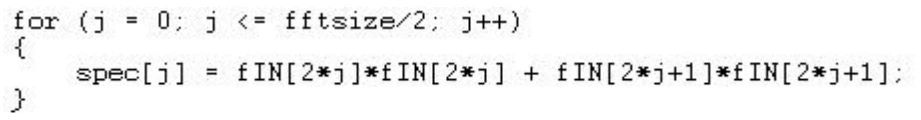
```
fe_t *fe_init(param_t *P)
{
    int i;
    double delta = (2*M_PI)/DEFAULT_FFT_SIZE;
    for (i = 0; i < (DEFAULT_FFT_SIZE/2); i++)
    {
        sin_n[i] = sin(i * delta);
        cos_n[i] = cos(i * delta);
    }
```

Figure 4.11: Pre-calculate the sin and cosine values during initialization

After eliminating the function calls, optimization techniques including Software Pipeline, eliminating Memory Aliases and Loop Unrolling can now be applied. The optimized performance is 2 cycles per iteration resulting 512 / 2 x 2 = 512 cycles total. Cache miss penalty is 256 samples x 2 (sin & cosine) / 8 (cache line size) x 4 (cycles penalty) = 256 cycles.

The bit reversal function can only be optimized using software pipelining. The conditional complexity makes it very difficult to apply other optimizations. The profiled performance for the pre- and post-FFT bit reversal functions are 12,000 and 24,000 cycles, respectively. As for the FFT function itself, it is hand optimized from TI and the performance is 11,500 cycles.

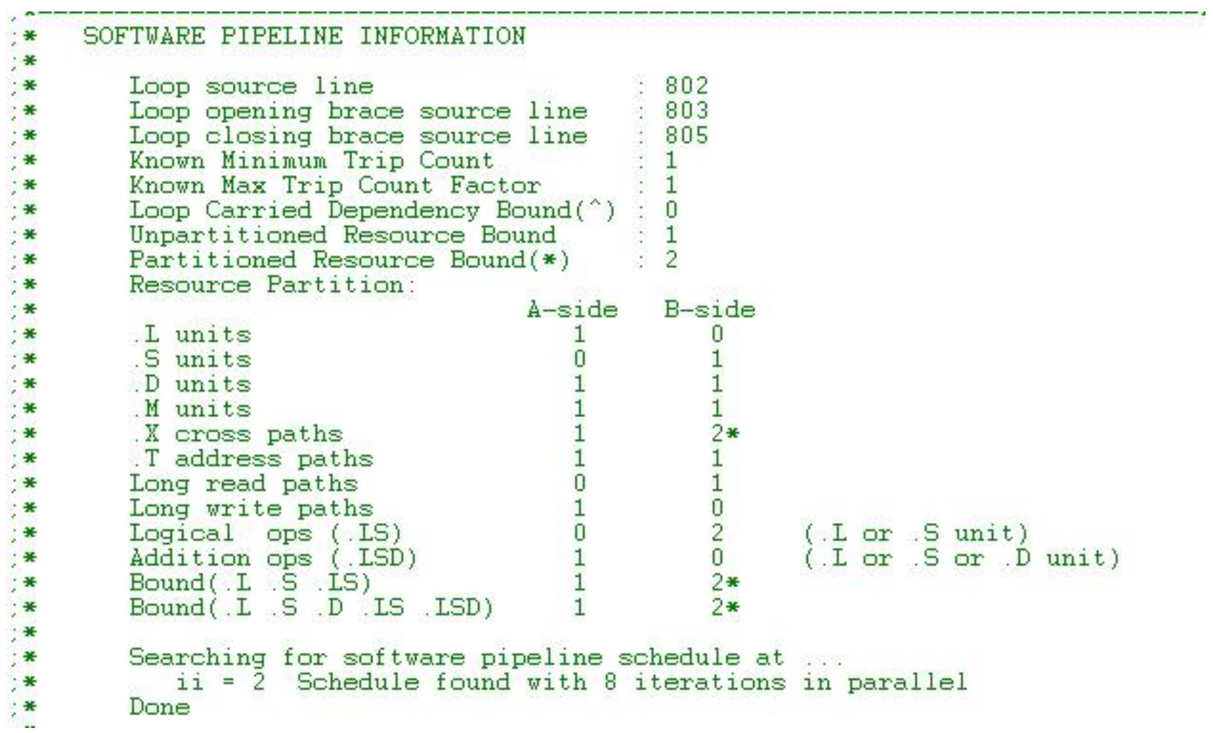
The final stage is Square Magnitude. The source code is displayed in Figure 4.12.



```
for (j = 0; j <= fftsize/2; j++)
{
    spec[j] = fIN[2*j]*fIN[2*j] + fIN[2*j+1]*fIN[2*j+1];
}
```

Figure 4.12: Source code for Square Magnitude

Each iteration requires 2 input data and performs 2 multiplications and 1 addition. Since the loop count is *fftsize* (512) / 2 = 256, Software Pipeline, Memory Aliases and Packed Data Optimization are applied. The software pipeline feedback table is shown in Figure 4.13. An overall of 512 cycles are needed for computation with 2 cycles per iteration. Cache miss penalty would be 512 / 8 x 4 = 256 cycles.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 802
;*      Loop opening brace source line   : 803
;*      Loop closing brace source line   : 805
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     1        0
;*      .S units                     0        1
;*      .D units                     1        1
;*      .M units                     1        1
;*      .X cross paths               1        2*
;*      .T address paths             1        1
;*      Long read paths              0        1
;*      Long write paths             1        0
;*      Logical  ops (.LS)           0        2     (.L or .S unit)
;*      Addition ops (.LSD)          1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        2*
;*      Bound(.L .S .D .LS .LSD)     1        2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 8 iterations in parallel
;*      Done
;*
```

Figure 4.13: Software pipeline feedback after applying different optimizations

The overall computation performance for power spectrum is the sum of all sections: Pre-FFT (1,232 cycle), Generate Twiddle Factor (768 cycles), Bit Reversal of Twiddle Factor (12,000 cycles), FFT (11,500 cycles) and Bit Reversal of FFT result (24,000 cycles), Square Magnitude (768 cycles). Total of 50,268 cycles. Table 4.3 summarizes the optimization result.

Table 4.3: Performance summary for Power Spectrum (FFT)

Power Spectrum (FFT)	Performance	Improvement
*Standard Optimizations	193,210 cycles	-
Optimized with MA, SP, LU & FFT Lib	50,268 cycles	384%

*Standard optimizations are Variable Registering and Constant Propagation
** MA = Memory Aliasing; SP = Software; LU = Loop Unrolling;
FFT Lib = Optimized FFT function from Texas Instruments Signal Processing Library

4.4 MEL SPECTRUM

Mel spectrum attempts to model the human perceptual system. In principle, human auditory system performs frequency analysis on different frequency components of sounds. The cochlear, part of the inner ear, acts as if it was a set of overlapping filters. The bandwidths of these filters are modeled by a non-linear scale, called mel-scale [19]. This frequency scale is linear up to 1000 Hz and logarithmic afterward. The use of mel-scale attempts to replicate the fact that the sensitivity of the human ear does not seem to be linear across all frequencies.

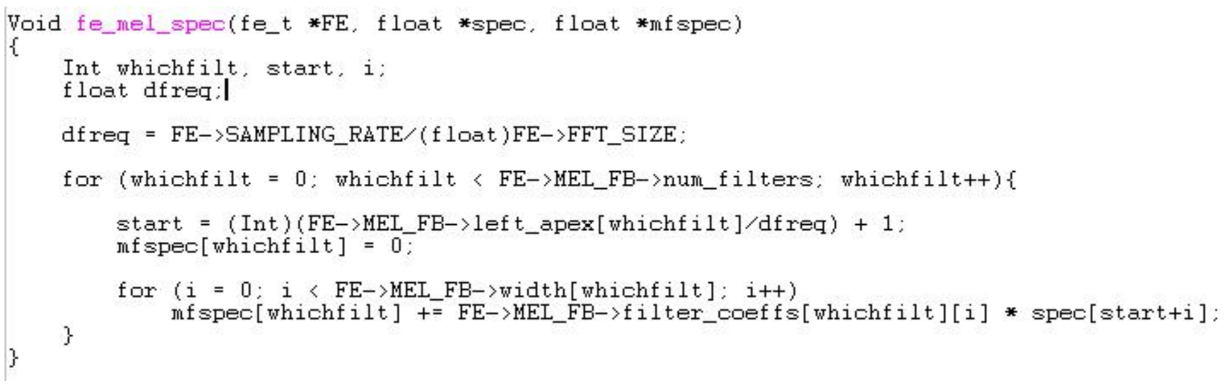
In Sphinx, a set of L triangular bandpass filter banks is used to approximate the frequency resolution of the human ear. These filters mimic the frequency analysis that take place in cochlear. Mel spectrum is computed by multiplying the power spectrum by each of the filters and integrating the result [20]. Equation 4.4 shows the function that transforms linear frequencies to mel frequencies while Equation 4.5 displays the computation to obtain the mel spectrum,

Mel[f] = $2595 \ \log 1 \quad \frac{f}{700}$ Eq. 4.4

$S'[l] = \overset{\frac{N}{2}}{\underset{l \ 0}{}} \ (S[k]mel_l[k])$, where $l = 0, 1, \dots, L-1$ Eq. 4.5

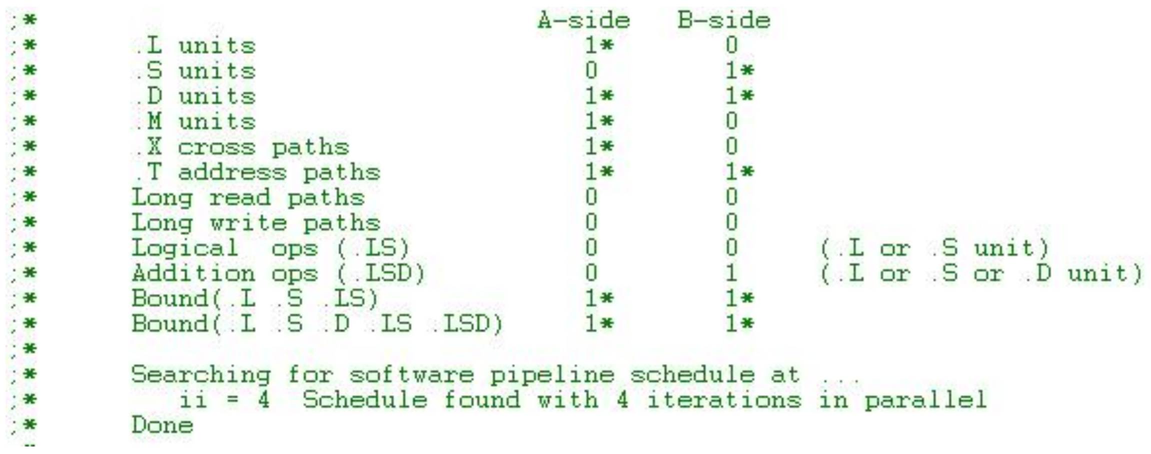
where N is the length of the DFT and L is the total number of triangular mel filters. In Sphinx and this research, 40 mel filters are used.

The source code from Sphinx for Mel Spectrum contains a nested for-loop as shown in Figure 4.14. Since both loop counts for the outer loop *(FE->MEL_FB->num_filters, 40)* and the inner loop *(FE->MEL_FB->width[whichfilt], vary)* is small, none other than Software Pipeline optimization is applied. The feedback table generated is shown in Figure 4.15.



```
Void fe_mel_spec(fe_t *FE, float *spec, float *mfspec)
{
    Int whichfilt, start, i;
    float dfreq;

    dfreq = FE->SAMPLING_RATE/(float)FE->FFT_SIZE;

    for (whichfilt = 0; whichfilt < FE->MEL_FB->num_filters; whichfilt++){

        start = (Int)(FE->MEL_FB->left_apex[whichfilt]/dfreq) + 1;
        mfspec[whichfilt] = 0;

        for (i = 0; i < FE->MEL_FB->width[whichfilt]; i++)
            mfspec[whichfilt] += FE->MEL_FB->filter_coeffs[whichfilt][i] * spec[start+i];
    }
}
```

Figure 4.14: Source code for Mel Spectrum



```
;*                                A-side   B-side
;*       .L units                  1*       0
;*       .S units                  0        1*
;*       .D units                  1*       1*
;*       .M units                  1*       0
;*       .X cross paths            1*       0
;*       .T address paths          1*       1*
;*       Long read paths           0        0
;*       Long write paths          0        0
;*       Logical  ops (.LS)        0        0     (.L or .S unit)
;*       Addition ops (.LSD)       0        1     (.L or .S or .D unit)
;*       Bound(.L .S .LS)          1*       1*
;*       Bound(.L .S .D .LS .LSD)  1*       1*
;*
;*       Searching for software pipeline schedule at ...
;*          ii = 4  Schedule found with 4 iterations in parallel
;*       Done
```

Figure 4.15: Software pipeline feedback table for Mel Spectrum

Note that the feedback result is generated only for the inner loop. The result shows that 4 cycles/iteration and the overall performance depends on the summation of all the filter widths. By inspection, this summation is 370 and the performance would be 4 x 370 + cache misses penalty. Since 3 input data are needed per iteration, a total of 3 x 370 = 1110 input data are loaded from the L2 SRAM. L1 cache miss penalty would be 1110 / 8 x 4 = 555 cycles. The overall performance would be 2035 cycles. Table 4.4 summarizes the performance.

Table 4.4: Performance summary for Mel Spectrum

Mel Spectrum	Performance	Improvement
*Standard Optimizations	3,145 cycles	-
Optimized with SP	2,035 cycles	155%

*Standard optimizations are Variable Registering and Constant Propagation

** SP = Software Pipeline;

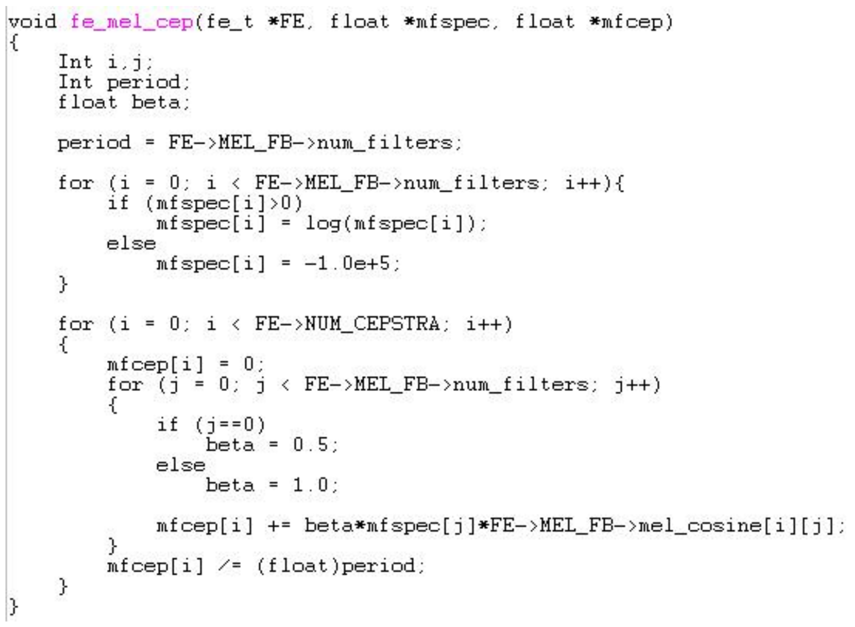
4.5 MEL CEPSTRUM

Mel Cepstrum, the static feature of speech signal, is obtained by applying DCT on the natural logarithm of mel spectrum. Equation 4.6 shows the operations performed:

$$c[n] = {}_{n\ 0}^{L\ 1} \ln(S'[i])\cos((\pi n/2L)(2i+1)), \text{ where } c = 0, 1, \ldots, C-1 \qquad \text{Eq. 4.6}$$

As Specified in Sphinx [20], 13 mel cepstra are produced from the above operation. By taking the 1st and 2nd derivative of the 13 mel cepstra, 26 more dynamic features are obtained forming a vector of 39 features, which is the output of FE and the input to the AM.

The source code for Mel Cepstrum is displayed in Figure 4.16.



```
void fe_mel_cep(fe_t *FE, float *mfspec, float *mfcep)
{
    Int i,j;
    Int period;
    float beta;

    period = FE->MEL_FB->num_filters;

    for (i = 0; i < FE->MEL_FB->num_filters; i++){
        if (mfspec[i]>0)
            mfspec[i] = log(mfspec[i]);
        else
            mfspec[i] = -1.0e+5;
    }

    for (i = 0; i < FE->NUM_CEPSTRA; i++)
    {
        mfcep[i] = 0;
        for (j = 0; j < FE->MEL_FB->num_filters; j++)
        {
            if (j==0)
                beta = 0.5;
            else
                beta = 1.0;

            mfcep[i] += beta*mfspec[j]*FE->MEL_FB->mel_cosine[i][j];
        }
        mfcep[i] /= (float)period;
    }
}
```

Figure 4.16: Source code for Mel Cepstrum

The source code for Mel Spectrum contains two for-loops with one being a nested loop. The first for-loop contains a log function call if the condition is satisfied. No optimizations are applied due to the function call. As for the nested for-loop, the outer loop count would be 40 (*num_filters*) and the inner loop count is 13 (*num_cepstra*). Again, the loop count of the core loop is relatively small, as a result, software pipelining only improves the performance slightly. The performance obtained after optimization of the inner for-loop is 2 cycle per iteration. With the total number of rotations to be 40 x 13 = 520, the total computation cycles required would be 1040 cycles. Cache penalty would be 520 x 3 / 8 x 4 = 780 cycles. Table 4.5 summarizes the performance for Mel Cepstrum.

Table 4.5: Performance summary for Mel Cepstrum

Mel Cepstrum	**Performance**	**Improvement**
*Standard Optimizations	12,580 cycles	-
Optimized with SP	7,900 cycles	159%

*Standard optimizations are Variable Registering and Constant Propagation
** SP = Software Pipeline;

4.6 DYNAMIC FEATURE: DELTA, DOUBLE DELTA

Acoustic Modeling assumes that each cepstrum is unrelated to its predecessors and successors. However, since speech is continuous, this assumption may not be correct. This is usually compensated by appending the first and second derivation of the cepstrum.

The final stage for FE is to take the 13 generated cepstras and to calculate the 1st and 2nd derivatives. The source code is relatively large and complex. It is attached with this thesis in APPENDIX. The overall performance is measured using the Texas Instruments profiler and the result is 3,751 cycles. Table 4.6 shows the performance summary for Dynamic Feature.

Table 4.6: Performance summary for Dynamic Feature

Dynamic Feature	Performance	Improvement
*Standard Optimizations	6,092 cycles	-
Optimized with SP	3,751 cycles	162%

*Standard optimizations are Variable Registering and Constant Propagation
** SP = Software Pipeline;

4.7 PERFORMANCE SUMMARY

All optimizations used for each FE stage is summarized in Table 4.7 while Table 4.8 provides the summary of the computation performance along with the cache miss penalty for each FE stages and the overall result. The overall performance for FE with cache miss penalty is about 69,600 cycles, which is 3.1% of our real-time cycle budget of 2,250,000 cycles.

Table 4.7: Optimization summary for FE

Stage	**VR**	**CP**	**MA**	**SP**	**LU**	**PD**	**SPL**
Pre-Emphasis	*	*	*	*			
Windowing	*	*	*	*	*	*	
Power Spectrum	*	*	*	*	*		* (FFT)
Mel Spectrum	*	*		*			
Mel Cepstrum	*	*		*			
Dynamic Features	*	*		*			

*VR = Variable Registering; CP = Constant Propagation; MA = Memory Aliasing; SP = Software Pipelining; LU = Loop Unrolling; PD = Packed Data Memory Access; SPL = Signal Processing Library

Table 4.8: Performance summary for FE

Section	Computation Performance (cycles)	Cache Miss Penalty (cycles)	Optimized Performance (cycles)	Unoptimized Performance	Improvement
Pre-Emphasis	4000	500	4500	14,500	322%
Windowing	820	410	1,230	2,050	167%
Power Spectrum	-	-	50,268	193,210	384%
Mel Spectrum	1,480	555	2,035	3,145	155%
Mel Cepstrum	7,120	780	7,900	12,580	159%
Dynamic Features	-	-	3,751	6,092	162%
FE (Overall)	-	-	69,684*	231,577	332%

*This result is obtained WITHOUT counting any loop setup overhead. Actual performance might be slightly longer.

5.0 ACOUSTIC MODELING PERFORMANCE

This chapter will discuss the background theory, performance issues, different implementations and the performance result. As shown in [1], Acoustic Modeling is the most computational intensive phase of the entire recognition process. As a result, two different approaches, Component Approach and Dimension Approach, are implemented. Each approach is optimized with numerous different optimization techniques to obtain the best performance possible with the given DSP architecture. After comparing the optimal performances, the Dimension approach has the best performance of 0.75 million cycles, an improvement of 4500% from 33.8 million cycles without any optimization. Another performance problem for AM is that it requires large amounts of data that must be stored in the external memory. However, accessing the external memory requires significant amounts of cycle delay. One solution is to use Direct Memory Access, as the detail will be presented later in (section.5.2)

As mentioned in Section 3.1, the main purpose of Acoustic Modeling along with Phone Modeling is to find *max P(X|W)*, the maximum probability of the observed input given a set of words. In other words, AM tries to determine the best match between the observed speech and the set of pre-defined words. Acoustic Modeling computes these probabilities at the senone level while PM completes the task at the phone level. Senone level probability evaluation means the evaluation of Gaussian distributions used to model senones. Aside from Gaussian evaluation, Acoustic Modeling also performs other functions including logarithm addition, relative scoring of senone scores and senone scores normalization.

Speech signal varies tremendously depending on the speakers. Due to that same reason, a single Gaussian distribution is found to be insufficient to represent a senone. To more accurately model a wide data distribution, a mixture of Gaussian distributions is used to model each senone.

Typically, speech recognition systems use between 2 to 64 distributions. For Sphinx 3, 8 Gaussian distributions are used and they are referred as Components. Since the weights of these distributions depend upon the nature of the data in the corpus, a weighting factor known as *Mixture Weight* is used to adjust the contribution of each distribution toward the overall component score. The senone score is computed as shown in Eq. 5.1.

$$\text{Senone\_Score}_{\text{s}} = \overset{C}{\underset{c\ \ 1}{}} (\text{Mixture\_Weight}_{\text{s,c}} * \text{Component\_Score}_{\text{s,c}}) \qquad \text{Eq. 5.1}$$

Furthermore, since the output of the FE block is a vector of 39-dimension features, multi-dimension Gaussian distribution is used instead of 1-D distribution to represent each component. The equation used to calculate multi-dimension Gaussian probability for Senone, *s*, and Component, *c*, is shown in Equation 5.2:

$$\text{Component\_Score}_{\text{s,c}} = \frac{1}{\sqrt{2 \quad ^{d} \; | \quad ^{2}_{s,c} \; |}} \; e^{\overset{D}{\underset{d\ \ 1}{}} \frac{(x_d \quad _{s,c,d})^{2}}{2 \quad ^{2}_{s,c,d}}} \qquad \text{Eq. 5.2}$$

where,

x = Feature Input

μ = Mean

σ = Variance

$|\sigma^{2}_{s,c}| = \sigma^{2}_{x,c,1} * \sigma^{2}_{x,c,2} * \ldots \; \sigma^{2}_{x,c,D}$

Notice that the exponential term implies multiplications in the above equation, which imposes a significant amounts of complexity to the calculation. A logarithmic transformation can be applied [1] to convert these multiplications into additions. The transformed equation is shown below:

$$\mathrm{Senone\_Scr}_{s} = \mathrm{LOG} \overset{8}{\underset{c\ \ 1}{\quad}} [W_{s,c} + f * \mathrm{GAUS\_DIST}] \qquad \text{Eq. 5.3}$$

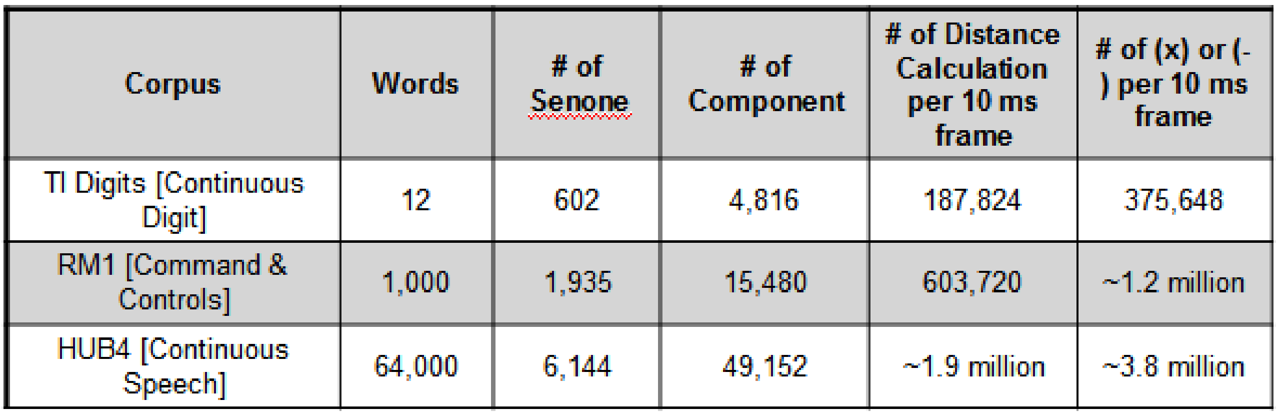
where,

- $Senone\_Scr_{s} = logs_{s}3[Senone\_Score_{s}]$
- $W_{s,c} = logs3(Mix\_Wt_{s,c}) = f * log_{e}(Mix\_Wt_{s,c})$
- $f = log_{s3}\ e = 3333.83$
- $K_{s,c} = log_{e} \dfrac{1}{\sqrt{2^{39} \mid {}^{2}_{s,c,d} \mid}}$
- $\sigma'^{2}_{s,c,d} = \dfrac{1}{2\ {}^{2}_{s,c,d}}$
- $GAUS\_DIST = K\mathrm{s,c} - \overset{39}{\underset{d\ \ 1}{\quad}} [(x_{d} - \mu_{s,c,d})^{2} * \sigma'^{2}_{s,c,d}]$

By applying logarithmic transformation, Equation 5.3 allows Gaussian probability to be calculated in log-domain. The core computation lies in the *GAUS_DIST* calculation where the distance between the observed input, x_d, and Gaussian mean, $\mu_{s,c,d}$, scaled by the variance, $\sigma'_{s,c,d}$. The only input, x_d, to this equation is the 39-dimension features that are obtained from FE. Each *GAUS_DIST* is then scaled by the scaling factor, *f*, and the mixture weight factor, *Ws,c*. The result is accumulated in the log domain, referred as log-add, to get the final *Senone_Scr*$_s$.

From a computational prospective, *GAUS_DIST* calculations are very computationally intensive. Each *GAUS_DIST* calculation performs 39 distance calculations, namely $(x\text{-}m)^2 * v$, and each distance calculation requires 2 multiplications and 2 subtractions. For RM1 corpus used for this research, there are 1935 senones. A senone is modeled with a mixture of 8 components and each component is a 39-dimension distribution. The total number of multiplication or subtraction would be well over 1.2 million (1935x8x39x2). To worsen the problem, most speech recognition systems perform these operations in floating point to maintain the accuracy of the overall system. A table is presented below showing the calculation statistics for various speech corpuses.

Table 5.1: Calculation statistic in AM for various speech corpuses



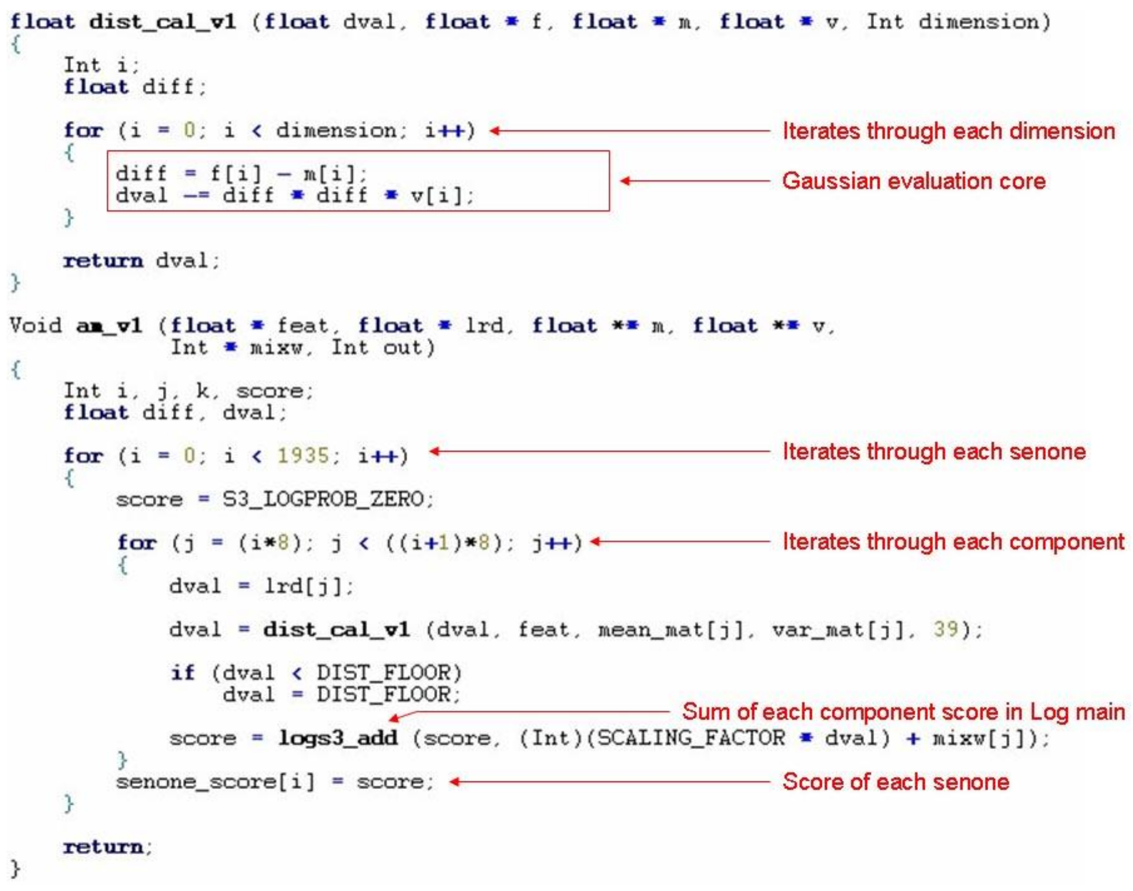
Corpus	Words	# of Senone	# of Component	# of Distance Calculation per 10 ms frame	# of (x) or (-) per 10 ms frame
TI Digits [Continuous Digit]	12	602	4,816	187,824	375,648
RM1 [Command & Controls]	1,000	1,935	15,480	603,720	~1.2 million
HUB4 [Continuous Speech]	64,000	6,144	49,152	~1.9 million	~3.8 million

5.1 IMPLEMENTATIONS OF GAUSSIAN EVALUATION

Gaussian evaluation is performed 39 times per component. For the RM1 corpus, there are total of 1,935 senones with 8 components each. As a result, a total of 603,720 evaluations are required to be completed in a frame of 10 milliseconds. These large amounts of calculations consume between 55-95% of the overall computation cycle [1]. It is extremely important that all evaluations are completed in the most efficient way. To ensure that the optimal performance is archived, two different approaches are investigated.

5.1.1 Individual Component Approach

In the individual component approach, the score of each senone is computed individually. Eight components of each senone are evaluated sequentially and each component score is added in the log domain. Finally, for each component, 39 Gaussian evaluations are performed and the running sum is recorded. The overall calculation is shown in Eq. 5.2 while the core Gaussian evaluation is highlighted in Eq. 5.3 . The source code for this implementation is also given in Figure 5.1.



```
float dist_cal_v1 (float dval, float * f, float * m, float * v, Int dimension)
{
    Int i;
    float diff;

    for (i = 0; i < dimension; i++)
    {
        diff = f[i] - m[i];
        dval -= diff * diff * v[i];
    }

    return dval;
}

Void am_v1 (float * feat, float * lrd, float ** m, float ** v,
            Int * mixw, Int out)
{
    Int i, j, k, score;
    float diff, dval;

    for (i = 0; i < 1935; i++)
    {
        score = S3_LOGPROB_ZERO;

        for (j = (i*8); j < ((i+1)*8); j++)
        {
            dval = lrd[j];

            dval = dist_cal_v1 (dval, feat, mean_mat[j], var_mat[j], 39);

            if (dval < DIST_FLOOR)
                dval = DIST_FLOOR;

            score = logs3_add (score, (Int)(SCALING_FACTOR * dval) + mixw[j]);
        }
        senone_score[i] = score;
    }

    return;
}
```

Iterates through each dimension
Gaussian evaluation core
Iterates through each senone
Iterates through each component
Sum of each component score in Log main
Score of each senone

Figure 5.1: Source code for Individual Component Approach

This source code contains 3 nested for-loops. The outermost for-loop represents the iterations of all 1,935 senones. A senone score is obtained at the end of each iteration. The first inner for-loop represents the 8 components of each senone. Each component score acquired from the innermost loop is added in the log domain. Finally, the innermost loop represents the 39 Gaussian dimensions for each component. An iteration of the innermost loop executes a Gaussian evaluation core. From the software prospective, the core requires 3 variables to perform 2 floating-point subtractions and 2 floating-point multiplications. A data-flow diagram is shown in Figure 5.2 to illustrate the calculation of the core.

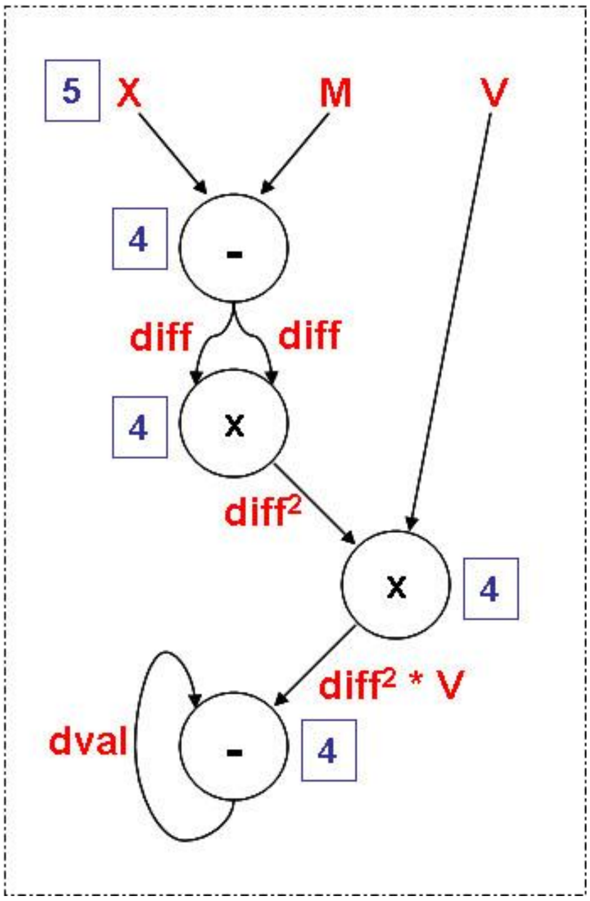


Figure 5.2: Data Flow Diagram for the Gaussian Evaluation core

There are four implications associated with the above data flow diagram. The first implication is data dependency. Each of the operations must be completed in the exact order from the top to the bottom because the input of the next operation is the output of the previous operation. This dependency significantly limits the performance of the software pipeline since none of these operations can be executed out of order.

The second implication is memory access. Three variables, x, m and v, are needed to perform the calculation. That means these variables must be stored in the registers, the on-chip memory, or the external memory. For each 10 milliseconds frame, there are total of 39 features, 603,720 means and 603,720 variances. It is clear that register is not an option since only 32 registers are available for the 6713 DSP. As a result, three memory accesses are required for each of the core calculations regardless if storing location is the on-chip memory or the external memory. However, storing these data in the external memory would further worsen the performance since pipeline stalls would occur whenever there is a cache miss.

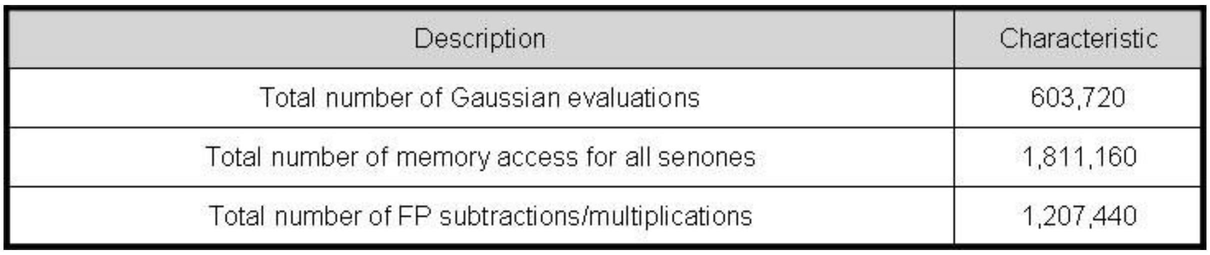
The third implication is register usage. If the code was executed sequentially from the top to the bottom, only four registers (x, m, v, and dval) would be required. However, if the code was to be executed in parallel, each input variable and all intermediate results would need to be stored separately. In other words, each arc in Figure 5.2 would represent a register result minimum of 7 registers. The register usage is important because the total number of registers available from the hardware architecture would dictate how many iterations of this core calculation can be executed in parallel.

The last implication is instruction latency. The rectangular box in the data flow diagram indicates how many cycles are required to complete each of the operations. E.g. 5 cycles are required to load the x, m, or v from the memory. Although an instruction can be issued to the functional unit every cycle, the result would require a few cycles before it is available due to the deeply pipelined hardware architecture. This effect can be minimized by using optimization techniques such as loop unrolling and software pipeline. In summary, each of the implications above must be considered carefully before the optimal code performance can be archived.

5.1.1.1 Algorithm Characteristics As discussed in the previous section, the Gaussian Evaluation core consists of 3 memory accesses, 2 FP subtractions and 2 FP multiplications. Clearly, the limiting factor is the number of memory accesses. Without applying any optimization, the DSP can execute 2 memory accesses per cycle. As a result, the ideal number of cycles required to complete 603,720 Gaussian evaluations would be 905,580 (603,720 calculations * 3 memory access / 2 functional units) cycles. On the other hand, if the Packed

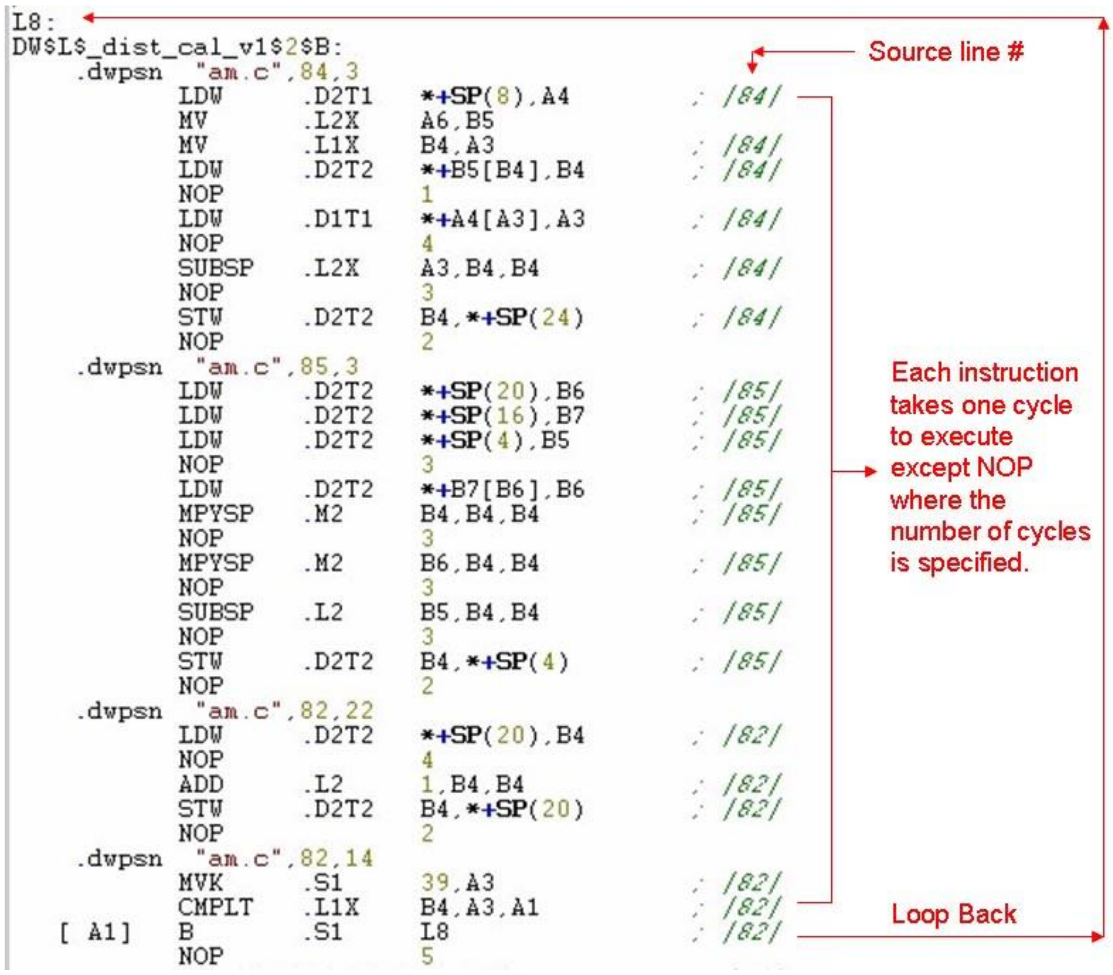
Data optimization is applied, 4 32-bit words, 2 words per functional unit, can be loaded per cycle. The new ideal number of cycles would be limited by the number of FP subtractions/multiplications instead. With 2 FP subtractions/multiplications per core calculation, a total of 1,207,440 would be required. The new ideal performance would be 603,720 (603,720 calculations * 2 multiplications / 2 functional units) cycles. However, other factors such as data dependency, instruction latency and registers availability, would limit the possibility of archiving such ideal performance. A list of important characteristics is summarized in Table 5.2.

Table 5.2: A list of important characteristics of the Gaussian Evaluation



Description	Characteristic
Total number of Gaussian evaluations	603,720
Total number of memory access for all senones	1,811,160
Total number of FP subtractions/multiplications	1,207,440

5.1.1.2 Baseline Performance Before applying any optimizations, the baseline performance of this implementation is measured. The source code in Figure 5.1 is compiled without any optimizations and the generated assembly is shown in Figure 5.3.



```
L8:
DW$L$_dist_cal_v1$2$B:
      .dwpsn    "am.c",84,3
              LDW      .D2T1    *+SP(8),A4              ; |84|
              MV       .L2X     A6,B5
              MV       .L1X     B4,A3                   ; |84|
              LDW      .D2T2    *+B5[B4],B4             ; |84|
              NOP               1
              LDW      .D1T1    *+A4[A3],A3             ; |84|
              NOP               4
              SUBSP    .L2X     A3,B4,B4                ; |84|
              NOP               3
              STW      .D2T2    B4,*+SP(24)             ; |84|
              NOP               2
      .dwpsn    "am.c",85,3
              LDW      .D2T2    *+SP(20),B6             ; |85|
              LDW      .D2T2    *+SP(16),B7             ; |85|
              LDW      .D2T2    *+SP(4),B5              ; |85|
              NOP               3
              LDW      .D2T2    *+B7[B6],B6             ; |85|
              MPYSP    .M2      B4,B4,B4                ; |85|
              NOP               3
              MPYSP    .M2      B6,B4,B4                ; |85|
              NOP               3
              SUBSP    .L2      B5,B4,B4                ; |85|
              NOP               3
              STW      .D2T2    B4,*+SP(4)              ; |85|
              NOP               2
      .dwpsn    "am.c",82,22
              LDW      .D2T2    *+SP(20),B4             ; |82|
              NOP               4
              ADD      .L2      1,B4,B4                 ; |82|
              STW      .D2T2    B4,*+SP(20)             ; |82|
              NOP               2
      .dwpsn    "am.c",82,14
              MVK      .S1      39,A3                   ; |82|
              CMPLT    .L1X     B4,A3,A1                ; |82|
   [ A1]      B        .S1      L8                      ; |82|
              NOP               5
```

Source line #
Each instruction takes one cycle to execute except NOP where the number of cycles is specified.
Loop Back

Figure 5.3: Generated assembly with no optimizations

The generated assembly shows that 56 cycles are needed to complete one Gaussian evaluation. The total number of cycles required for all senones would be approximately 33.81 million (1935 senones x 8 components x 39 Gaussians x 56 cycles). This result is about 15x the real-time constraint, which is 2.25 million cycles with a 225 MHz DSP. This baseline performance will be compared against by the results obtained after a series of optimizations are applied. Optimizations will be applied are as follows:

- Variable Registering
- Constant Propagation
- Software Pipeline

- Removing Memory Aliasing
- Loop Unrolling by 3x
- Packed Data Memory Access
- Loop Unrolling by 2x

5.1.1.3 Optimization: Variable Registering The first optimization applied is variable registering. For the baseline code, all input variables for each instruction are loaded from the memory and all output values including all intermediate calculated results are written back to the memory. This large amount of memory accesses impose significant amounts of overhead into the code. This optimization is applied simply by switching a flag in the compiler option to indicate to the compiler that registers should be used wherever appropriate. In the case of the TI compiler used in this research, an option of multiple levels of optimizations is available in the compiler option. By changing the optimization level from none to zero, the compiler will try to use registers whenever possible. This simple optimization improves the performance over the baseline performance from 56 cycles to 51 cycles on average per Gaussian evaluation. The overall performance is reduced to 30.79 million cycles and a speedup of 1.1x over the baseline performance.

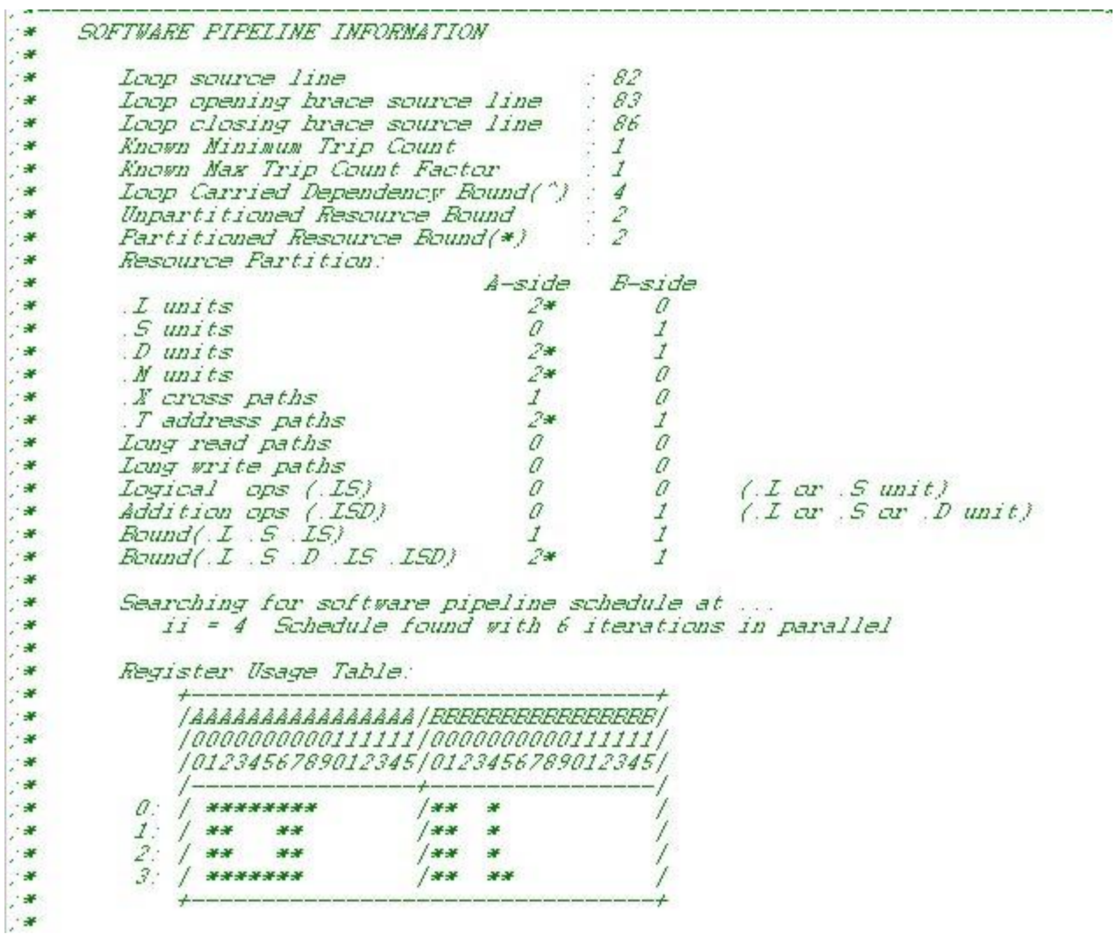
5.1.1.4 Optimization: Constant Propagation Constant propagation is a technique that replaces constants into equations in compiler time. This technique further reduces the amounts of memory accesses required. The result shows that 37 cycles are required per Gaussian evaluation, which produces the overall cumulative performance of 22.34 million and a cumulative speedup of 1.51x over the baseline.

5.1.1.5 Optimization: Software Pipeline Software pipeline enables multiple instructions to be executed by multiple functional units at the same time. By applying software pipeline, a considerable amounts of performance improvement can potentially be obtained. To accurately measure the performance of the code and to determine if the optimal performance is obtained, two other measurements are used in addition to the number cycle per Gaussian evaluation. These two measurements are register utilization and functional unit utilization. Register utilization measures how many registers are used from the total number of register available for one iteration of the loop. For example, if a single Gaussian evaluation takes 2 cycles to complete, iteration is defined to be 2 cycles. Every cycle there are 32 register available, totaling 64 registers for 2 cycles. If a total of 10 registers are used during the evaluation, then the register utilization defined as 10 out of 64, or 15.63%.

Functional unit utilization is a similar measurement to register. This measurement is further divided into 4 sub-measurements. One measurement for each type of functional unit. For example, 2 floating-point multipliers are available per cycle. If an iteration takes 2 cycles, then a maximum of 4 multiplications can be performed. If only one multiplication is performed during the 2-cycle period, then the utilization for the FP multiplier is 1 out of 4, or 25%.

An optimal performance is said to be achieved when the resource utilizations are maximized. In other word, if all functional units are operating and all registers are used every cycle, resource utilizations are maximized.

Software pipeline is usually applied by changing a flag in the compiler option. This is the case for TI's compiler. The source code in Figure 5.1 is compiled with software pipeline optimization, and the generated assembly is shown in Figure 5.4.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 82
;*      Loop opening brace source line   : 83
;*      Loop closing brace source line   : 86
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2*        0
;*      .S units                     0         1
;*      .D units                     2*        1
;*      .M units                     2*        0
;*      .X cross paths               1         0
;*      .T address paths             2*        1
;*      Long read paths              0         0
;*      Long write paths             0         0
;*      Logical  ops (.LS)           0         0     (.L or .S unit)
;*      Addition ops (.LSD)          0         1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1         1
;*      Bound(.L .S .D .LS .LSD)     2*        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 6 iterations in parallel
;*
;*      Register Usage Table:
;*          +-----------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: | ********       |**  *           |
;*       1: | **   **        |**  *           |
;*       2: | **   **        |**  *           |
;*       3: | *******        |**  **          |
;*          +-----------------------------------+
;*
```

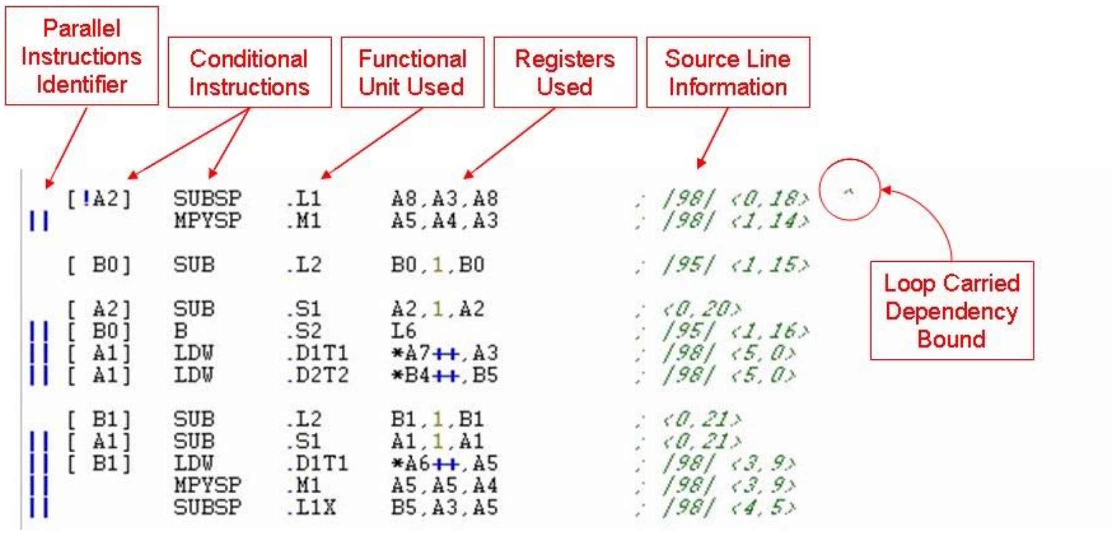
Figure 5.4: Software pipeline feedback table

As indicated by the line *ii = 4*, 4 cycles are required per iteration, resulting a cumulative performance of 2.41 million cycles and a cumulative speedup of 14x over the baseline.

With software pipeline enabled, register utilization can also be used as a performance measurement. Recall that 3 memory accesses, 2 FP multiples and 2 FP subtractions are performed during the evaluation. These operations match the functional units used, 3 (.D), 2(.M) and 2(.L), during each iteration. From these statistics, each functional unit’s utilization can be obtained. The load/store unit is 3 out of 8, or 37.5%. As for the FP multiplier and FP ALUs, the utilizations are 2 out of 8, or 25% while the Fixed-Point ALUs is only used for operations not directly related to the core calculation. So the utilization of Fixed-Point ALU is 0%. As for the register utilization, the register table shown at the bottom of the feedback table indicated that only 36 are used out of 128 (32 registers x 4 cycles), or 28.13%. It should be clear that since all resource utilizations are fairly low, more optimizations can be applied to achieve better performance.

5.1.1.6 Optimization: Removing Memory Aliasing

A closer examination of the feedback table in Figure 5.4 reveals that a Loop Carried Dependency Bound (^) of 4 is the limiting factor of the code performance. An examination of the generated assembly would be required to identify such dependency. This assembly is shown in Figure 5.5.



```
   [!A2]  SUBSP  .L1    A8,A3,A8       ; |98| <0,18>
||        MPYSP  .M1    A5,A4,A3       ; |98| <1,14>

   [ B0]  SUB    .L2    B0,1,B0        ; |95| <1,15>

   [ A2]  SUB    .S1    A2,1,A2        ; <0,20>
|| [ B0]  B      .S2    L6             ; |95| <1,16>
|| [ A1]  LDW    .D1T1  *A7++,A3       ; |98| <5,0>
|| [ A1]  LDW    .D2T2  *B4++,B5       ; |98| <5,0>

   [ B1]  SUB    .L2    B1,1,B1        ; <0,21>
|| [ A1]  SUB    .S1    A1,1,A1        ; <0,21>
|| [ B1]  LDW    .D1T1  *A6++,A5       ; |98| <3,9>
||        MPYSP  .M1    A5,A5,A4       ; |98| <3,9>
||        SUBSP  .L1X   B5,A3,A5       ; |98| <4,5>
```

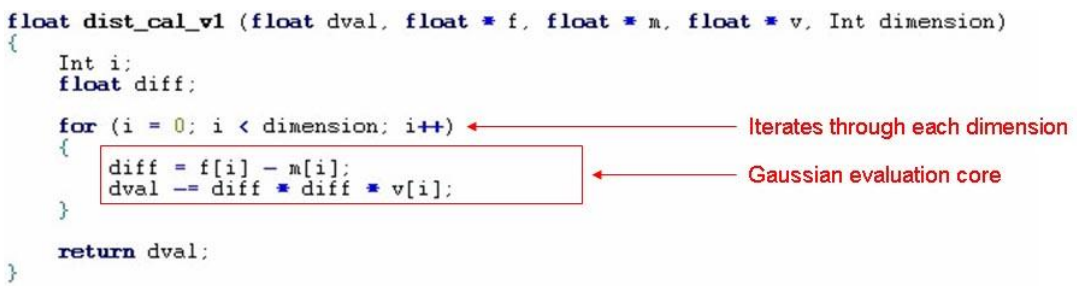
Figure 5.5: Generated assembly with the Loop Dependency Bound identifier

As identified by the (^) symbol in the assembly, there is a dependency bound with the FP subtraction instruction and the multiplication instruction. Typically a carried dependency bound is caused by the load and store to memory instructions. However, in this case, the dependency is caused by the delay slot that is required for those two floating-point operations. As mentioned before, both FP subtraction and multiplication are 4-cycle type instruction which takes one cycle to execute and 3 cycles of delay slots. The cycles of delay slot is caused by the deeply pipeline hardware architecture of the functional unit itself. As a result, nothing can be done to remove such dependency since 4 cycles are needed to perform both subtraction and multiplication.

It is clear that the minimum number of cycles to execute a single Gaussian evaluation is determined to be 4. Since the reduction of the cycle count is no longer possible, the only way to improve the code performance would be to try to do more operations during the 4 cycles. For example, multiple iterations, or Gaussian evaluations, can be executed in parallel. This can be done by performing loop unrolling optimization.

5.1.1.7 Optimization: Loop Unrolling by 3x Loop unrolling allows multiple iterations of the loop to be executed in parallel. This optimization increases the resource utilizations by executing more operations during the 4-cycle minimum period.

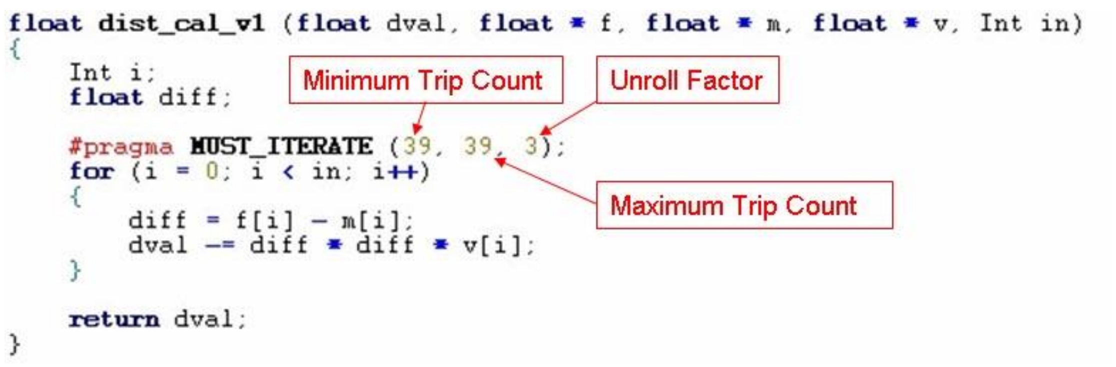
Most compilers, by default, would try to perform loop unrolling without any additional instructions. However, depending on the amount of the loop information available, the compiler might not be able to determine the best unrolling factor. The original source of the core calculation is copied below in Figure 5.6 for convenience.



```
float dist_cal_v1 (float dval, float * f, float * m, float * v, Int dimension)
{
    Int i;
    float diff;

    for (i = 0; i < dimension; i++)
    {
        diff = f[i] - m[i];
        dval -= diff * diff * v[i];
    }

    return dval;
}
```

Figure 5.6: Source code for the core Gaussian evaluation

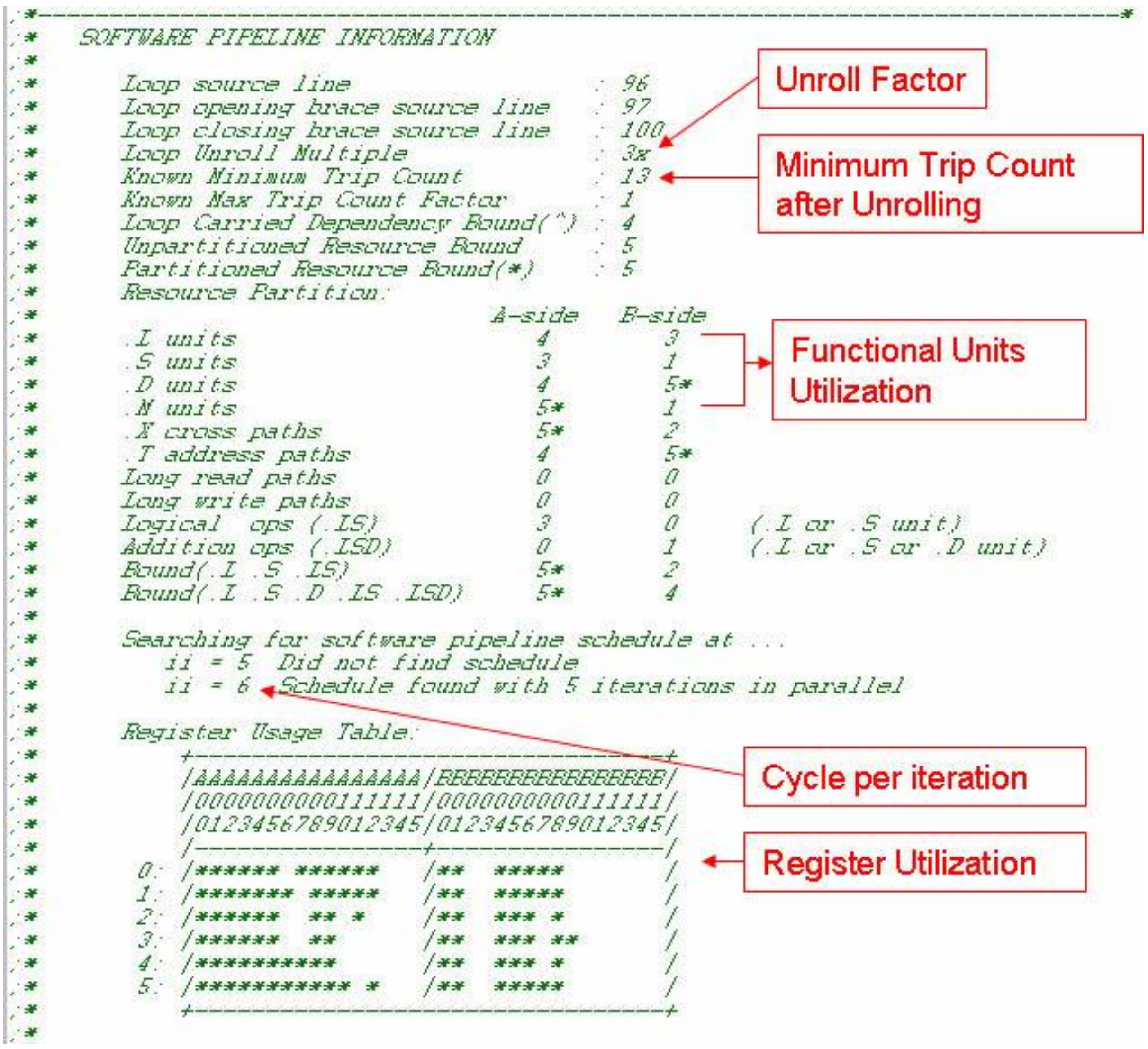
Within the scope of the function itself, the compiler has no information on the minimum or maximum number of the loop iteration. Hence, the compiler cannot unroll the loop safely. As discussed in the previous section, the code designer can use a pragma instruction to pass the trip count information to the compiler. Since the trip count is exactly 39 in this case, the only possible loop unroll factor would be by 3x. The modified source code is shown in Figure 5.7.



```
float dist_cal_v1 (float dval, float * f, float * m, float * v, Int in)
{
    Int i;
    float diff;

    #pragma MUST_ITERATE (39, 39, 3);
    for (i = 0; i < in; i++)
    {
        diff = f[i] - m[i];
        dval -= diff * diff * v[i];
    }

    return dval;
}
```

Figure 5.7: Modified source code with the pragma instruction

As indicated, the MUST_ITERATE pragma passed 3 variables to the compiler: maximum trip count, minimum trip count and an unroll factor that is always divisible by the minimum trip count. With this information, the compiler can now safely unroll the loop by a factor of 3, executing 3 Gaussian evaluations in parallel. The result is shown in Figure 5.8.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                   : 96
;*      Loop opening brace source line     : 97
;*      Loop closing brace source line     : 100
;*      Loop Unroll Multiple               : 3x
;*      Known Minimum Trip Count           : 13
;*      Known Max Trip Count Factor        : 1
;*      Loop Carried Dependency Bound(^)   : 4
;*      Unpartitioned Resource Bound       : 5
;*      Partitioned Resource Bound(*)      : 5
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     4        3
;*      .S units                     3        1
;*      .D units                     4        5*
;*      .M units                     5*       1
;*      .X cross paths               5*       2
;*      .T address paths             4        5*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           3        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             5*       2
;*      Bound(.L .S .D .LS .LSD)     5*       4
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Did not find schedule
;*         ii = 6  Schedule found with 5 iterations in parallel
;*
;*      Register Usage Table:
;*          +---------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: |****** ******   |**  *****       |
;*       1: |******* *****   |**  *****       |
;*       2: |******  ** *    |**  *** *       |
;*       3: |******  **      |**  *** **      |
;*       4: |**********      |**  *** *       |
;*       5: |*********** *   |**  *****       |
;*          +---------------------------------+
;*
```

Unroll Factor
Minimum Trip Count after Unrolling
Functional Units Utilization
Cycle per iteration
Register Utilization

Figure 5.8: Feedback result after performing loop unrolling

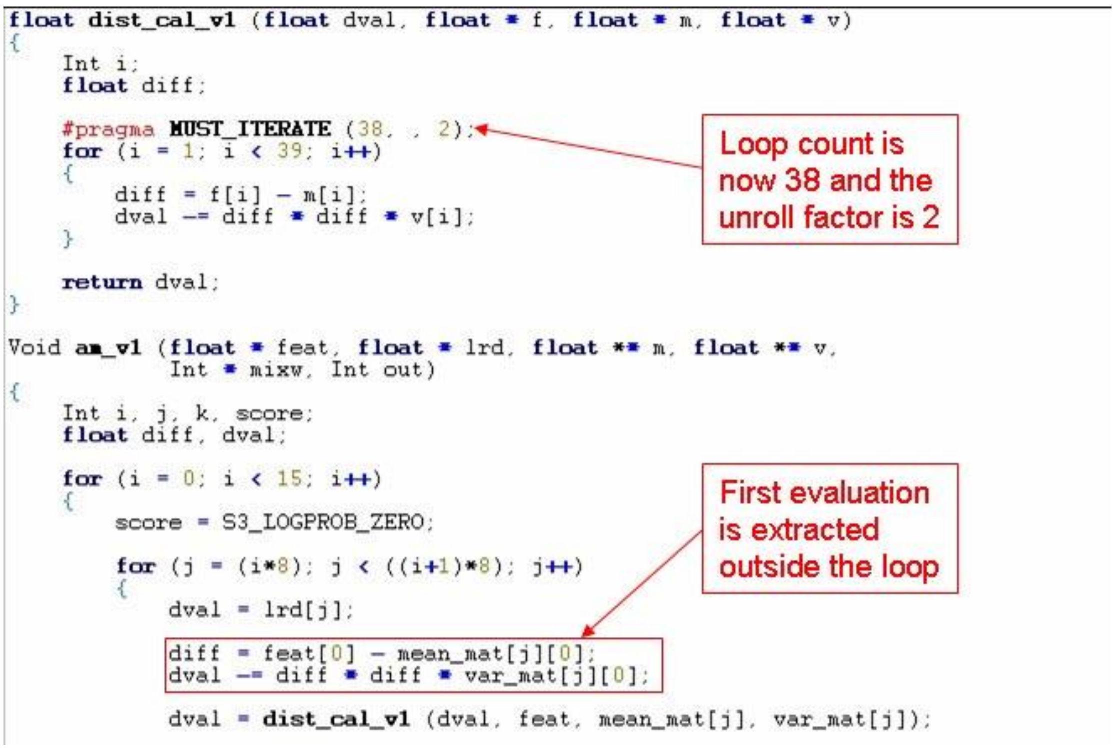
The feedback shows 6 cycles are required to complete an iteration, which evaluates 3 Gaussians. The effective cycle per calculation is 6/3 = 2 cycles per evaluation. The cumulative performance would be 1.22 million cycles and the cumulative speedup over baseline is 28x.

For the load/store (.D) functional unit, the utilization is 9/12, or 74%. As for the FP multipliers and FP ALUs, 6/12, or 50% are used for evaluation operations. Fixed-point unit utilization can be ignored since it is not used to perform any operations directly involved in the evaluation. Finally, the register utilization is 103 out of 192 total, 53.65%. Although all resource utilizations increased compared to the non-loop-unrolled version, the levels of utilizations are still relatively low. It is worth exploring other optimization techniques or another unrolling factor in an attempt to increase the utilization levels and to reduce the effective cycle per iteration.

5.1.1.8 Optimization: Packed Data Memory Access Before applying another unrolling factor, it is worth trying to further reduce the cycle per iteration first. From the feedback result obtained in Figure 5.8, the limiting factor appeared to be the Partitioned Resource Bound as pointed out by the (*). The load/store (.D) unit on the B-side is used 5 times while the A-side is used 4 times per iteration. A total of 9 memory accesses are due to the fact that 3 evaluations being executed in parallel. If the number of memory accesses can be reduced, the effective cycle per iteration can potentially be further reduced.

Packed Data optimization utilizes the load double-word instruction specifically available in this DSP architecture. Each load/store unit is capable of loading 2 32-bit words per cycle. This technique can potentially reduce the number of memory accesses from 9 to 5 if applied appropriately. This optimization is only appropriate if the trip count, or the minimum trip count after loop unrolling, is an even number. If the trip count is an odd number, additional operations will be performed and errors will be produced. For this particular case, the minimum trip count after the loop is unrolled by a factor of 3 is 13. Hence, applying Packed Data could induce undesired errors.

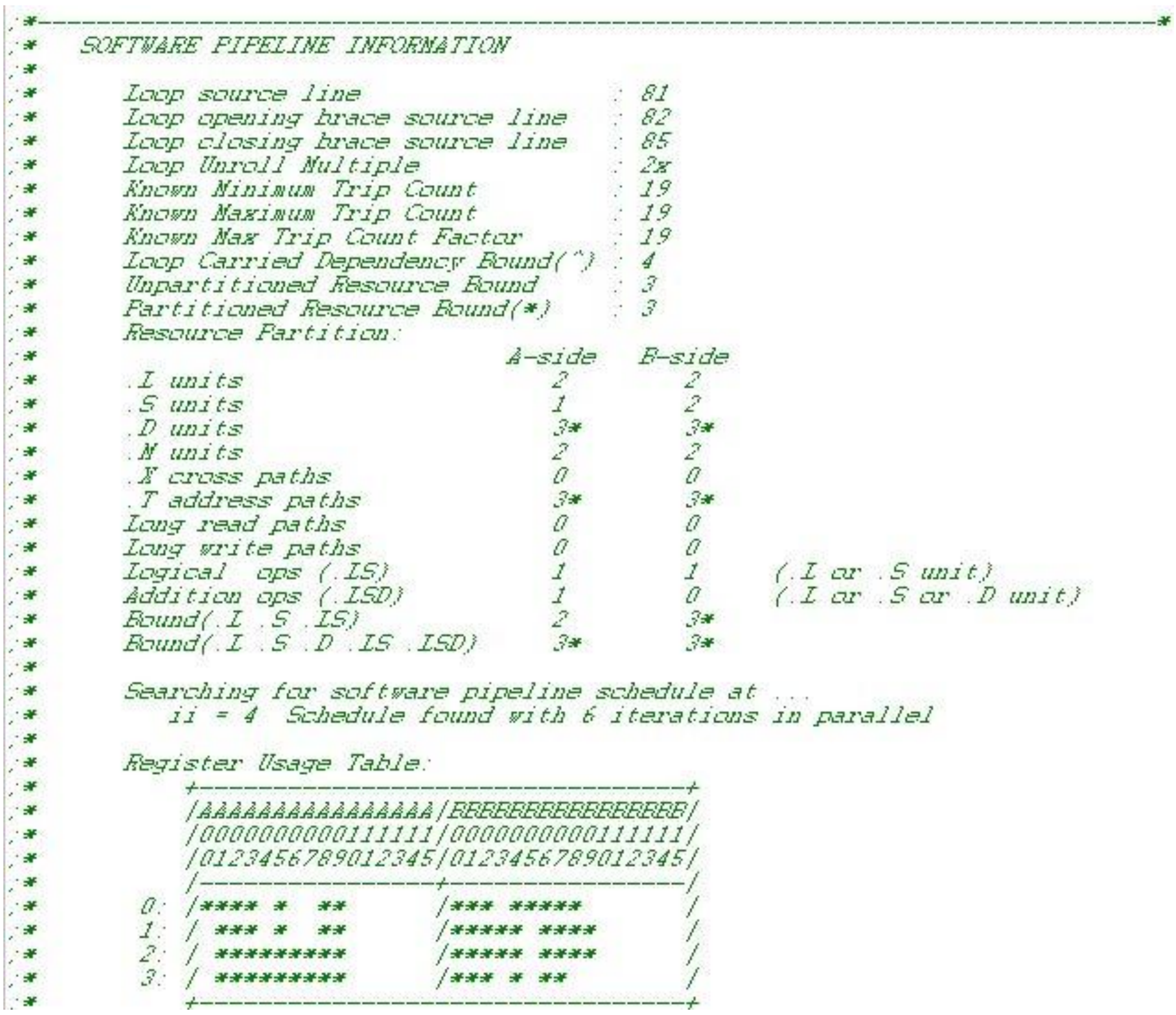
5.1.1.9 Optimization: Loop Unrolling by 2x In an attempt to further increase the resource utilizations, another unrolling factor is investigated. Another possible unrolling factor is 2. The loop can be unrolled by 2, if the first of the 39 evaluations is performed outside of the loop. This will reduce the trip count from 39 to 38, which results in the loop being unrolled by 2. The modified source code and the result are shown in Figure 5.9 and Figure 5.10, respectively.



```
float dist_cal_v1 (float dval, float * f, float * m, float * v)
{
    Int i;
    float diff;

    #pragma MUST_ITERATE (38, , 2);
    for (i = 1; i < 39; i++)
    {
        diff = f[i] - m[i];
        dval -= diff * diff * v[i];
    }

    return dval;
}

Void am_v1 (float * feat, float * lrd, float ** m, float ** v,
            Int * mixw, Int out)
{
    Int i, j, k, score;
    float diff, dval;

    for (i = 0; i < 15; i++)
    {
        score = S3_LOGPROB_ZERO;

        for (j = (i*8); j < ((i+1)*8); j++)
        {
            dval = lrd[j];

            diff = feat[0] - mean_mat[j][0];
            dval -= diff * diff * var_mat[j][0];

            dval = dist_cal_v1 (dval, feat, mean_mat[j], var_mat[j]);
```

Loop count is now 38 and the unroll factor is 2
First evaluation is extracted outside the loop

Figure 5.9: Modified source code with loop unrolling factor of 2



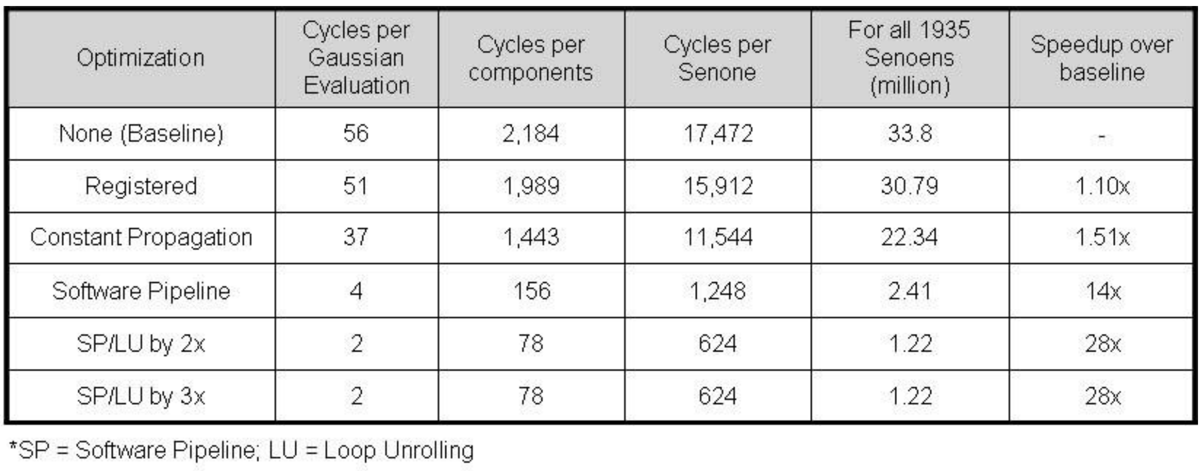
```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 81
;*      Loop opening brace source line   : 82
;*      Loop closing brace source line   : 85
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 19
;*      Known Maximum Trip Count         : 19
;*      Known Max Trip Count Factor      : 19
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2        2
;*      .S units                     1        2
;*      .D units                     3*       3*
;*      .M units                     2        2
;*      .X cross paths               0        0
;*      .T address paths             3*       3*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        1     (.L or .S unit)
;*      Addition ops (.LSD)          1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        3*
;*      Bound(.L .S .D .LS .LSD)     3*       3*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 6 iterations in parallel
;*
;*      Register Usage Table:
;*          +---------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: |**** *  **      |*** *****       |
;*       1: | *** *  **      |***** ****      |
;*       2: | *********      |***** ****      |
;*       3: | *********      |*** * **        |
;*          +---------------------------------+
```

Figure 5.10: The result of unrolling the loop by 2

Unrolling the loop by a factor of 2 produced the same result as the loop being unrolled by 3. The effective cycle per iteration is 4/2, or 2. Load/store unit utilization is 6/8, 75%. FP ALU/multiplier is used 4/8, 50%. Register usage is 64/128 or 50%. Furthermore, Packed Data optimization cannot be applied due to the minimum trip count of 19. Hence, no significant performance increase is acquired by unrolling the loop by 2 instead of 3.

5.1.1.10 Individual Component Approach Performance Summary All optimization results are summarized in Table 5.3 while the resource utilization results are listed in Table 5.4.

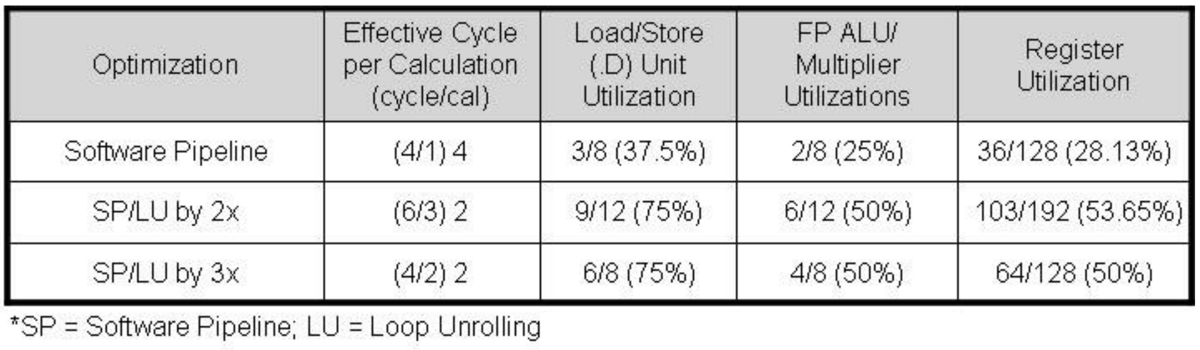
Table 5.3: Summary of different optimization results



Optimization	Cycles per Gaussian Evaluation	Cycles per components	Cycles per Senone	For all 1935 Senoens (million)	Speedup over baseline
None (Baseline)	56	2,184	17,472	33.8	-
Registered	51	1,989	15,912	30.79	1.10x
Constant Propagation	37	1,443	11,544	22.34	1.51x
Software Pipeline	4	156	1,248	2.41	14x
SP/LU by 2x	2	78	624	1.22	28x
SP/LU by 3x	2	78	624	1.22	28x

*SP = Software Pipeline; LU = Loop Unrolling

Table 5.4: Summary of resource utilizations with different optimizations



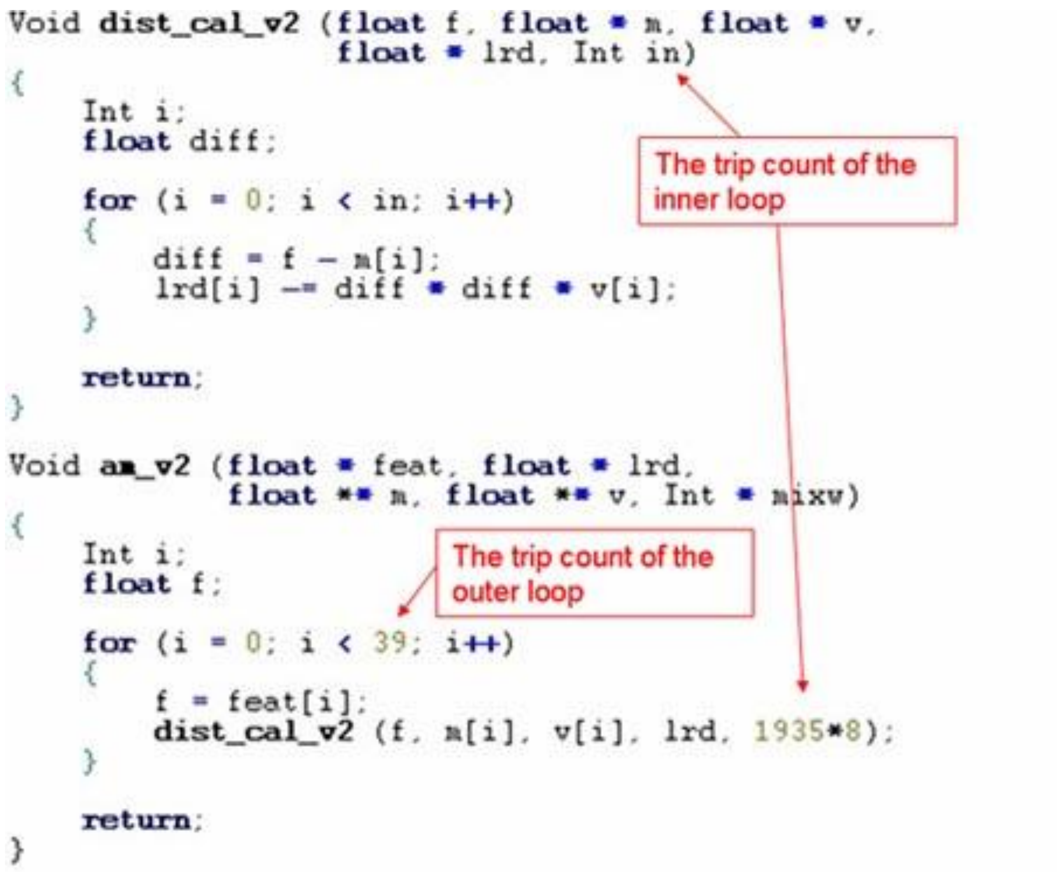
Optimization	Effective Cycle per Calculation (cycle/cal)	Load/Store (.D) Unit Utilization	FP ALU/ Multiplier Utilizations	Register Utilization
Software Pipeline	(4/1) 4	3/8 (37.5%)	2/8 (25%)	36/128 (28.13%)
SP/LU by 2x	(6/3) 2	9/12 (75%)	6/12 (50%)	103/192 (53.65%)
SP/LU by 3x	(4/2) 2	6/8 (75%)	4/8 (50%)	64/128 (50%)

*SP = Software Pipeline; LU = Loop Unrolling

So far, the optimal code performance is produced by applying these following optimizations: Variable Registering, Constant Propagation, Software Pipeline, Removing Memory Aliasing and Loop Unrolling by a factor of 2 or 3. The effective cycle per Gaussian evaluation is 2 cycles, yielding a total of 1.21 million cycles to evaluate all senones. Compared to the baseline performance, the best result produces a speed up of 28x. Although significant performance improve had been made, this is still far from the ideal performance of 603,720 cycles. It is easy to see the problem is that the resource utilizations are fairly low. It is impossible to further increase the utilization levels under the current nested loop structures. Hence, other loop structure must be investigated in attempt to improve the performance.

5.1.2 Individual Dimension Approach

The Individual Dimension Approach essentially merges the senone and component loops and alters the order of the calculation. Since each senone is made up of 8 components, the senone and component loops can be combined to form a larger loop, a loop with trip count of 15,480 (1935 x 8) components. Furthermore, instead of evaluating each component individually, each dimensions for all 15,480 components are evaluated sequentially and separately. For example, the first dimension of all components is evaluated first before the 2 dimension. The advantage of this approach is that the trip count of the inner loop is large enough so that other unrolling factors can be applied. The higher order of the unrolling factor means higher utilization levels across all resources. Furthermore, Packed Data optimization maybe applied if the unrolled trip count is even number. Figure 5.11 shows the source code with for this implementation approach.



```
Void dist_cal_v2 (float f, float * m, float * v,
                  float * lrd, Int in)
{
    Int i;
    float diff;

    for (i = 0; i < in; i++)
    {
        diff = f - m[i];
        lrd[i] -= diff * diff * v[i];
    }

    return;
}
Void am_v2 (float * feat, float * lrd,
            float ** m, float ** v, Int * mixw)
{
    Int i;
    float f;

    for (i = 0; i < 39; i++)
    {
        f = feat[i];
        dist_cal_v2 (f, m[i], v[i], lrd, 1935*8);
    }

    return;
}
```

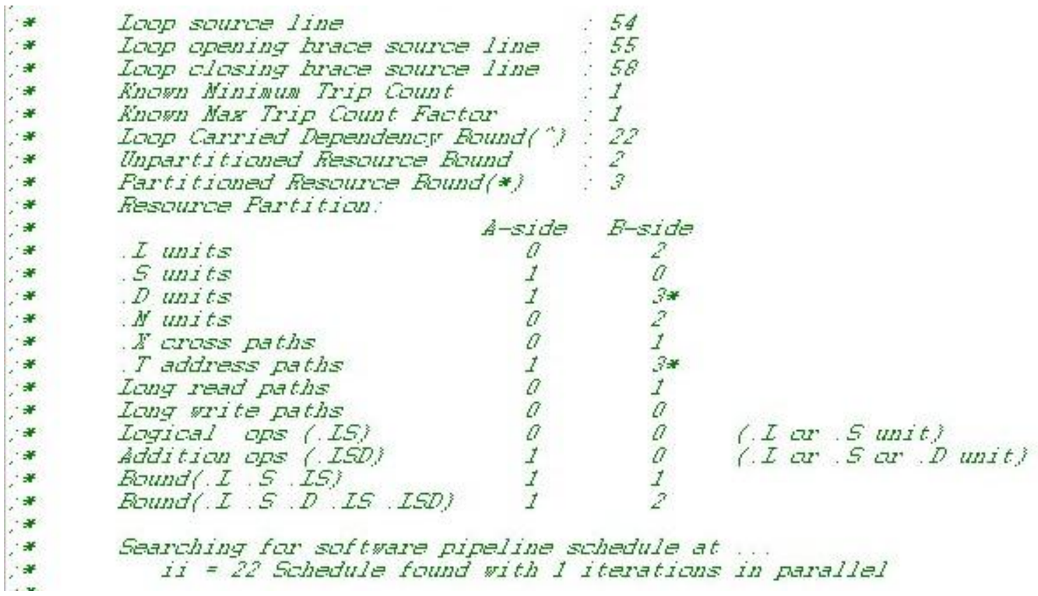
The trip count of the inner loop
The trip count of the outer loop

Figure 5.11: Source code with the Individual Dimension Approach

Beside from the structure differences compared to the previous approach, the computations are also slightly different. Since the feature used to compute the first Gaussian dimensions across all components is the same, a memory access for the feature is no longer needed. However, two additional memory accesses are introduced by the *lrd* vector. Recalls from Eq. 5.3 that a component score is the sum of 39 Gaussian scores. For the individual component approach, the running sum of the 39 Gaussian scores is computed within the inner loop simply using a register. As for this approach, the running sum of the 39 dimension for each component is computed once per iteration of the outer loop. A vector of 15,480 words is required instead to track the running sum of each component. Each position of the vector is loaded from the memory and the new result is stored back on every iteration of the inner loop. Hence, 1 load and 1 store are introduced.

Although the total number of memory accesses actually increased to 4 comparing to 3 in the previous approach, the potential of performance gain is still possible since a higher order of loop unrolling factor can be used. Also, the effect of increased in memory accesses can be offset by applying the Pack Data Memory Access optimization.

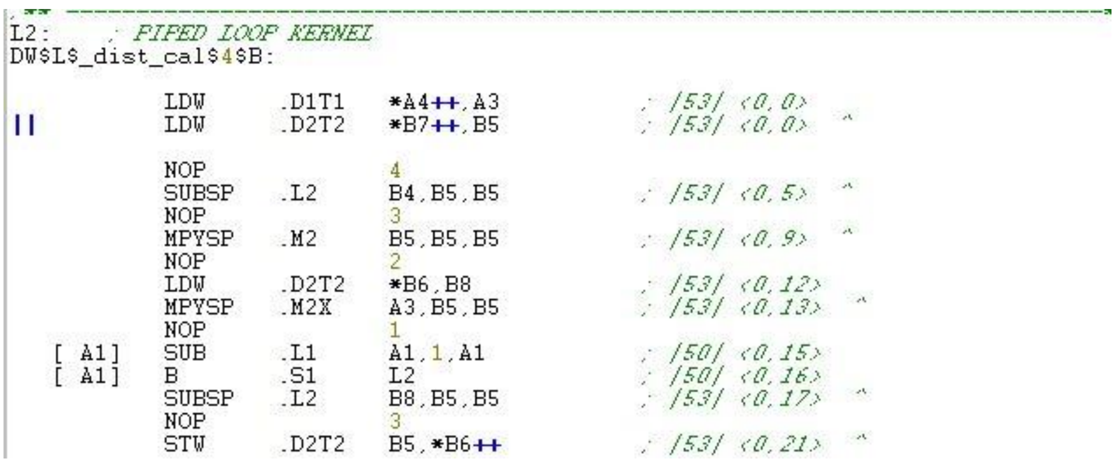
5.1.2.1 Optimization: Software Pipeline The analysis of the second approach begins by applying Variable Registering, Constant Propagation and Software Pipeline since the individual result for the first two optimizations is relatively insignificant at this point. The software pipeline feedback generated by the compiler using the source code in Figure 5.11 is shown in Figure 5.12. The result indicates that 22 cycles is needed per iteration, which only computes one Gaussian evaluation, due to a Loop Carried Dependency Bound (^). The cumulative performance at this point is 13.28 million cycles and the cumulative speedup over baseline is 2.55x. The load/store utilization is 4/44, or 9.09%, while the FP ALUs and multipliers are 2/44, 4.55%. It is obvious that other optimizations are needed to gain better performance.



```
;*      Loop source line                        : 54
;*      Loop opening brace source line          : 55
;*      Loop closing brace source line          : 58
;*      Known Minimum Trip Count                : 1
;*      Known Max Trip Count Factor             : 1
;*      Loop Carried Dependency Bound(^)        : 22
;*      Unpartitioned Resource Bound            : 2
;*      Partitioned Resource Bound(*)           : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .I units                    0        2
;*      .S units                    1        0
;*      .D units                    1        3*
;*      .M units                    0        2
;*      .X cross paths              0        1
;*      .T address paths            1        3*
;*      Long read paths             0        1
;*      Long write paths            0        0
;*      Logical  ops (.IS)          0        0     (.I or .S unit)
;*      Addition ops (.ISD)         1        0     (.I or .S or .D unit)
;*      Bound(.I .S .IS)            1        1
;*      Bound(.I .S .D .IS .ISD)    1        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 22 Schedule found with 1 iterations in parallel
```

Figure 5.12: Software pipeline result with the individual dimension approach

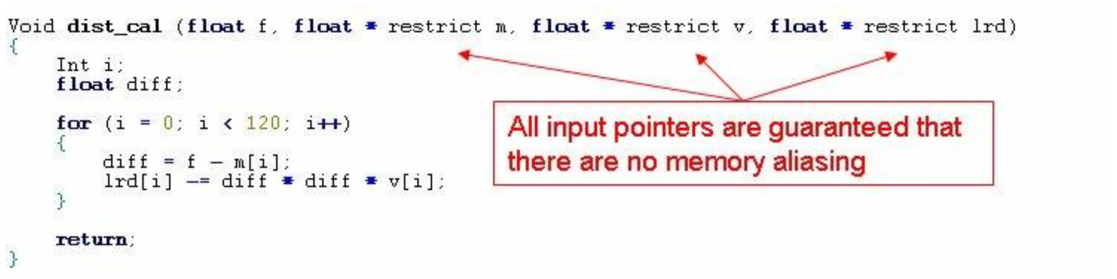
5.1.2.2 Removing Memory Aliasing By examining the generated assembly shown in Figure 5.13, it is clear that exists a dependency path: LDW (5) – SUBSP (4) – MPYSP (4) – MPYSP (4) – SUBSP (4) – STW (1). The number next to each instruction is the cycles required to complete that operation.



```
;** --------------------------------------------------------------------------*
L2:    ; PIPED LOOP KERNEL
DW$L$_dist_cal$4$B:

                LDW     .D1T1   *A4++,A3         ; |53| <0,0>
||              LDW     .D2T2   *B7++,B5         ; |53| <0,0>  ^

                NOP             4
                SUBSP   .L2     B4,B5,B5         ; |53| <0,5>  ^
                NOP             3
                MPYSP   .M2     B5,B5,B5         ; |53| <0,9>  ^
                NOP             2
                LDW     .D2T2   *B6,B8           ; |53| <0,12>
                MPYSP   .M2X    A3,B5,B5         ; |53| <0,13>  ^
                NOP             1
   [ A1]        SUB     .L1     A1,1,A1          ; |50| <0,15>
   [ A1]        B       .S1     L2               ; |50| <0,16>
                SUBSP   .L2     B8,B5,B5         ; |53| <0,17>  ^
                NOP             3
                STW     .D2T2   B5,*B6++         ; |53| <0,21>  ^
```

Figure 5.13: Assembly code showing the Loop Carried Dependency Path

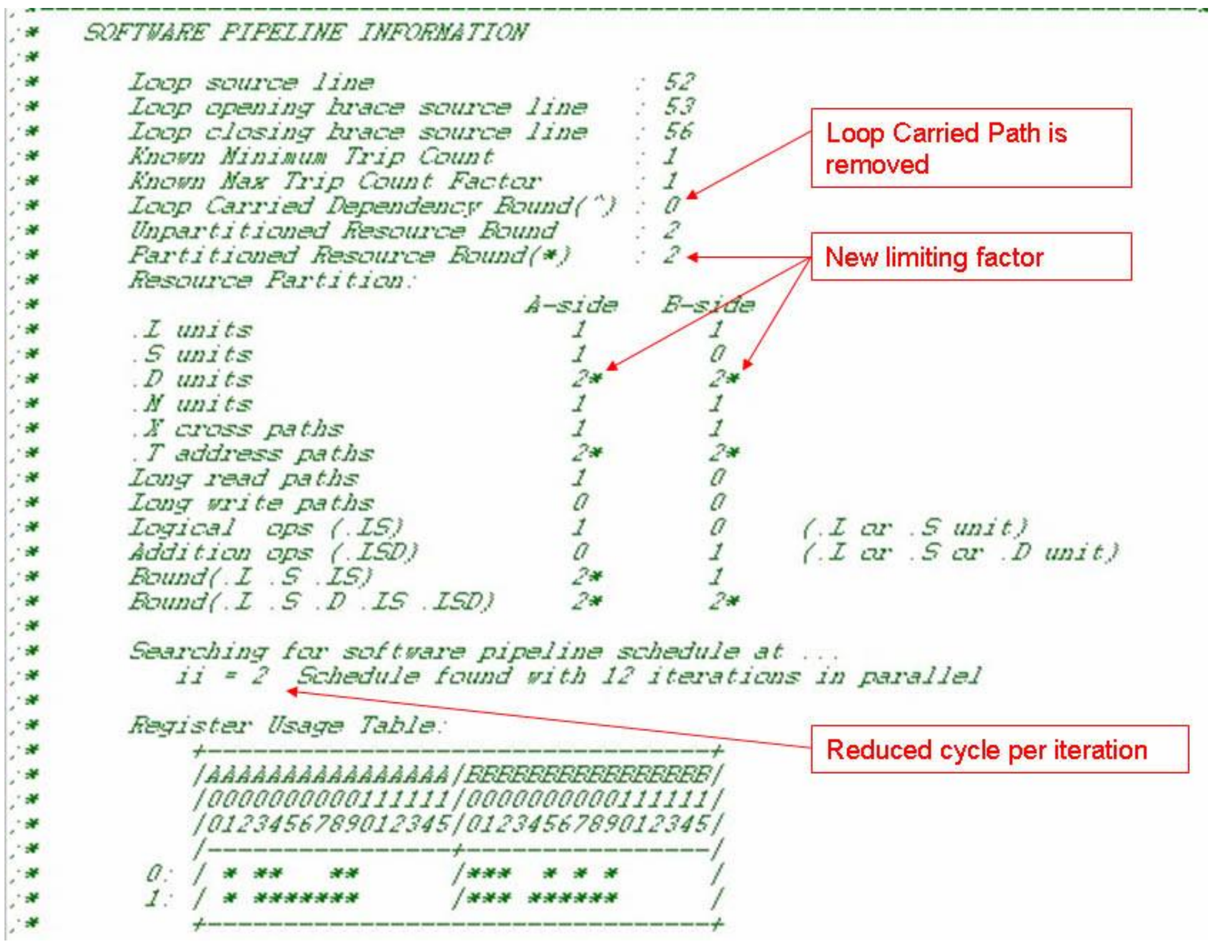
Again, since the compiler has no information about the memory locations of the input pointers from the local scope of the function, the compiler must wait until the result of the current iteration is computed and written back to the memory before the next iteration can be started. On the other hand, if the code designer can guarantee no overlap are between the input pointer arrays, a *restrict* type qualifier can be used to guarantee that there are no memory aliasing among all input pointers. A modified version of the source code is shown in Figure 5.14 and the new feedback is shown in Figure 5.15.



```
Void dist_cal (float f, float * restrict m, float * restrict v, float * restrict lrd)
{
    Int i;
    float diff;

    for (i = 0; i < 120; i++)
    {
        diff = f - m[i];
        lrd[i] -= diff * diff * v[i];
    }

    return;
}
```

All input pointers are guaranteed that there are no memory aliasing

Figure 5.14: Modified source code with the use of restrict keyword



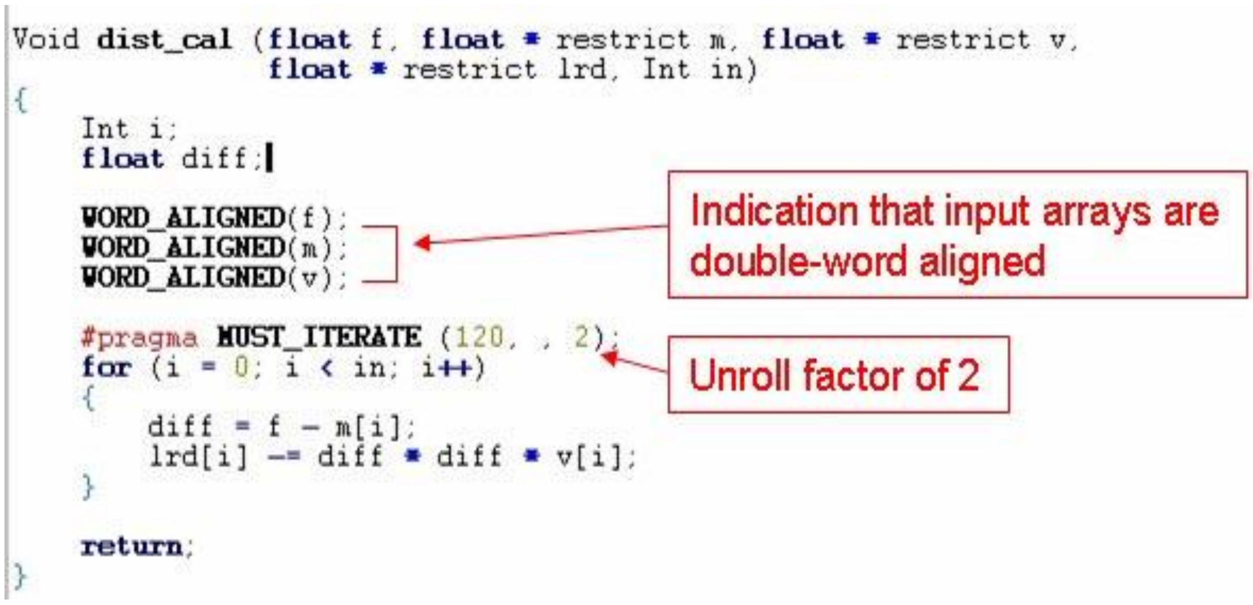
```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 52
;*      Loop opening brace source line   : 53
;*      Loop closing brace source line   : 56
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     1        1
;*      .S units                     1        0
;*      .D units                     2*       2*
;*      .M units                     1        1
;*      .X cross paths               1        1
;*      .T address paths             2*       2*
;*      Long read paths              1        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2*       1
;*      Bound(.L .S .D .LS .LSD)     2*       2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 12 iterations in parallel
;*
;*      Register Usage Table:
;*          +---------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: | * **    **     |***  * * *      |
;*       1: | * *******      |*** ******      |
;*          +---------------------------------+
```

Figure 5.15: Feedback result after removing memory aliasing

With the dependency path removed, 2 cycles are needed per iteration result 1.21 million cycles of cumulative performance and 28x speedup over baseline. Resource utilizations also increased to 100% (4/4) for the load/store while the FP ALUs/Multipliers are raised to 50% (2/2). Register utilization is 28/64, or 43.75%.

5.1.2.3 Optimizations: Packed Data Memory Load and Loop Unrolling By 2x The feedback table in Figure 5.15 also points out the new limiting factor is the Partitioned Resource Bound (*). Since 3 loads and 1 store are required per iteration while subtractions and multiplications are performed twice respectively, it is easy to understand that the load/store unit would become the bottleneck. The Packed Data optimization technique can reduce to number of memory accesses. However, a packed memory load means that 2 consecutive words for the same variable are loaded. For example, m[0] and m[1] can be loaded together but not m[0] and v[0]. Hence, packed data load by itself in this case will not be useful since 3 double-words loads are still required to get the three input variable: m, v and lrd.

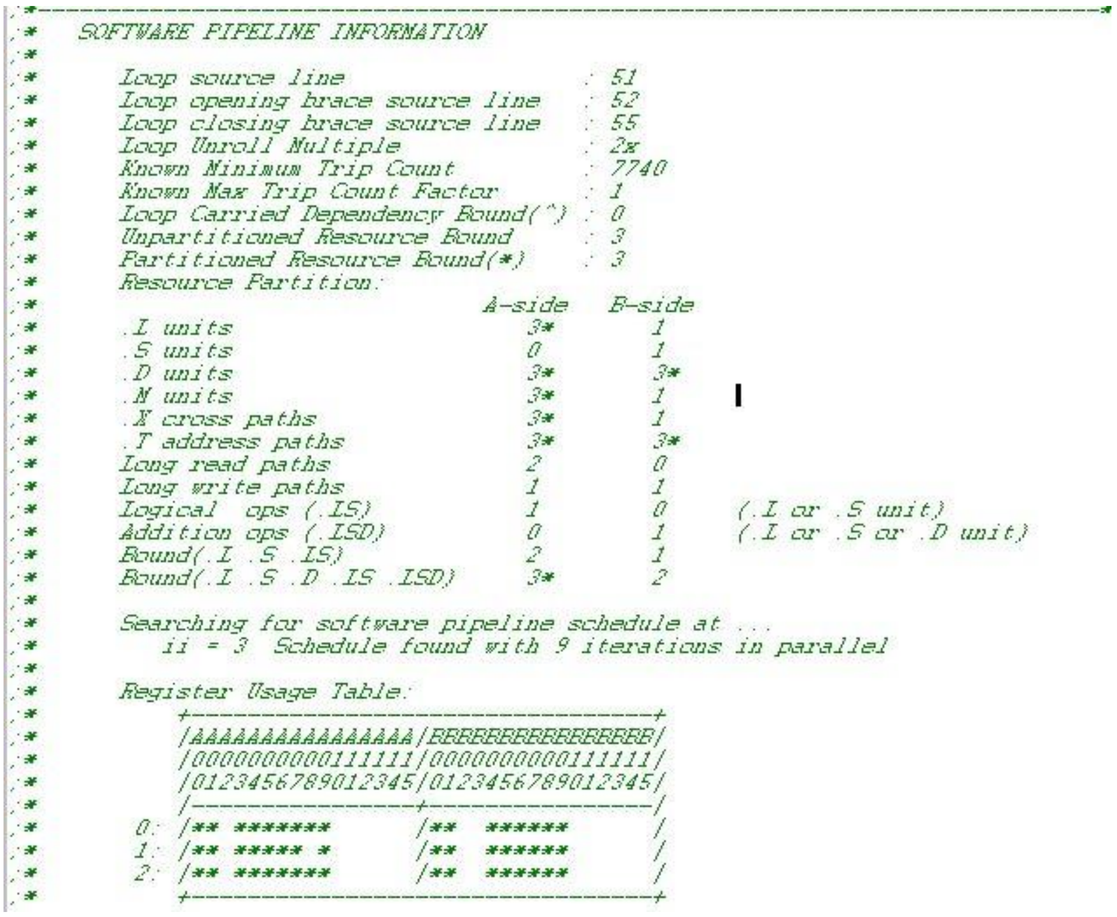
On the other hand, if the loop is unrolled by a factor of 2, double-word load would become very beneficent. Loop unrolling by 2x would mean 6 loads and 2 stores are needed per iteration. Further, 3 pairs of consecutive words are required per input variable. Packed data optimization can be applied here to reduce the number of loads from 6 to 3. The modified source code with both Packed Data optimization and Loop Unrolling by 2x is shown in Figure 5.16.



```
Void dist_cal (float f, float * restrict m, float * restrict v,
               float * restrict lrd, Int in)
{
    Int i;
    float diff;

    WORD_ALIGNED(f);
    WORD_ALIGNED(m);
    WORD_ALIGNED(v);

    #pragma MUST_ITERATE (120, , 2);
    for (i = 0; i < in; i++)
    {
        diff = f - m[i];
        lrd[i] -= diff * diff * v[i];
    }

    return;
}
```

Figure 5.16: Modified source code with double-word load and loop unrolling of 2x

Note that if either of the two optimizations applied here was used alone, no improvement would be gained. Double-word load alone would not reduce the number of loads while loop unrolling would only increase the cycle per iteration by the effective cycle per calculation would reminds at 2. The performance result is shown in Figure 5.17.



```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 51
;*      Loop opening brace source line   : 52
;*      Loop closing brace source line   : 55
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 7740
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     3*       1
;*      .S units                     0        1
;*      .D units                     3*       3*
;*      .M units                     3*       1
;*      .X cross paths               3*       1
;*      .T address paths             3*       3*
;*      Long read paths              2        0
;*      Long write paths             1        1
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     3*       2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 9 iterations in parallel
;*
;*      Register Usage Table:
;*          +-----------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: |** *******       |**  ******      |
;*       1: |** ***** *       |**  ******      |
;*       2: |** *******       |**  ******      |
;*          +-----------------------------------+
```

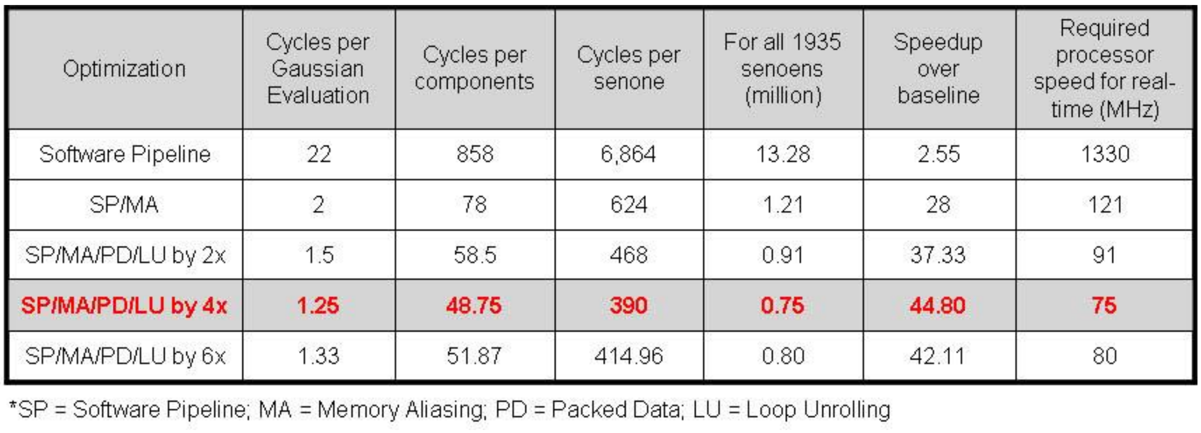
Figure 5.17: Performance after applying double-word load and loop unrolling

Surprisingly, the result obtained at this point is already better than the optimal result obtained using the Individual Component Approach. As shown in the feedback table, each iteration, which evaluates two Gaussians, takes 3 cycles. This produces the effective cycle per calculation of 3/2, or 1.5. With 1.5 cycles per evaluation, only 905,580 cycles are required to compute all senone scores. This result also produces a 37.3x speed up over the baseline performance obtained from the previous approach. As for the resource utilization levels, load/store units are used 6/6, 100%. FP ALUs and multipliers are used 4/6, or 66.7%. Finally, the register usage is at 53/96, or 55.21%.

With the performance archived so far, it is clear that the Individual Dimension Approach is a better. Yet, more performance gain is seemed to be possible since the resource utilization levels are still low, only 66.7% for the FP ALUs/multiplier and 55.21% for the register usage. Clearly, increasing the utilization levels would be ideal if all possible. This can be done by using a higher loop unrolling factors.

5.1.2.4 Optimization: Loop Unrolling by 4x, 6x And Summary Further performance analysis had been done by using higher loop unrolling factors. The overall performance results are summarized in Table 5.5 and the resource utilization results are listed in Table 5.6.

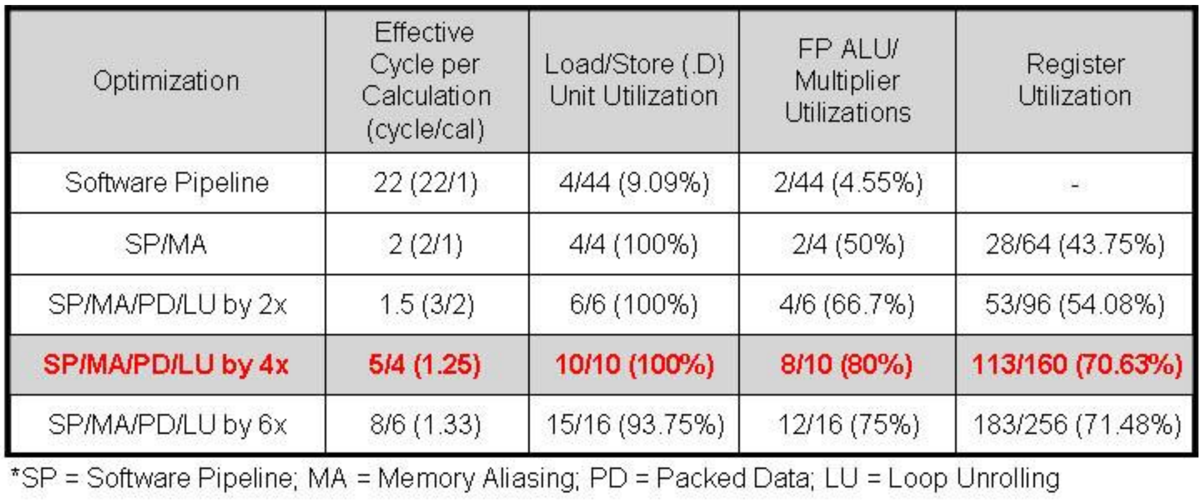
Table 5.5: Performance Summary for the Individual Dimension Approach



Optimization	Cycles per Gaussian Evaluation	Cycles per components	Cycles per senone	For all 1935 senoens (million)	Speedup over baseline	Required processor speed for real-time (MHz)
Software Pipeline	22	858	6,864	13.28	2.55	1330
SP/MA	2	78	624	1.21	28	121
SP/MA/PD/LU by 2x	1.5	58.5	468	0.91	37.33	91
SP/MA/PD/LU by 4x	**1.25**	**48.75**	**390**	**0.75**	**44.80**	**75**
SP/MA/PD/LU by 6x	1.33	51.87	414.96	0.80	42.11	80

*SP = Software Pipeline; MA = Memory Aliasing; PD = Packed Data; LU = Loop Unrolling

Table 5.6: Resource Utilization Summary



Optimization	Effective Cycle per Calculation (cycle/cal)	Load/Store (.D) Unit Utilization	FP ALU/ Multiplier Utilizations	Register Utilization
Software Pipeline	22 (22/1)	4/44 (9.09%)	2/44 (4.55%)	-
SP/MA	2 (2/1)	4/4 (100%)	2/4 (50%)	28/64 (43.75%)
SP/MA/PD/LU by 2x	1.5 (3/2)	6/6 (100%)	4/6 (66.7%)	53/96 (54.08%)
SP/MA/PD/LU by 4x	**5/4 (1.25)**	**10/10 (100%)**	**8/10 (80%)**	**113/160 (70.63%)**
SP/MA/PD/LU by 6x	8/6 (1.33)	15/16 (93.75%)	12/16 (75%)	183/256 (71.48%)

*SP = Software Pipeline; MA = Memory Aliasing; PD = Packed Data; LU = Loop Unrolling

As highlighted in the two summary tables, the maximum performance archived is 1.25 cycles per Gaussian evaluation. This is obtained by performing the following optimizations: Software Pipeline, Removing Memory Aliasing, Packed Data Memory Load, and Loop Unrolling by 4. Furthermore, load/store utilization is at 100% while FP ALUs/multiplier is at 80%. Although register utilization is not the highest, however, the overall performance is best when the loop is unrolled by 4.

5.2 MEMORY BANDWIDTH ISSUE

5.2.1 The Problem

Beside the enormous amount of computations required, Acoustic Modeling also presents another problem with the memory bandwidth. As mentioned in previous sections, AM performs about 60,000 Gaussian evaluations where each evaluation requires 3 32-bit words as the input. In other words, almost 2 Mbytes of memory bandwidth would be required for every 10 ms frame. The computational performances obtained in previous sections are all based on the assumption that all data are available to the processor without any delay. In reality, 2 Mbytes of data is much larger than the size of the L1 cache. It is inevitable that these data must be stored in the external memory. The issue is that the processor could takes well over 100 cycles to access the external memory. The actual number of cycle delay depends mainly on the ratio between the processor speed and the speed of the external memory interface. It is easy to see that it is completely infeasible to allow data to be accessed from the external memory.

5.2.2 The Solution

One solution is to find a processor with large enough cache size that fit all the required data. Another solution is to setup the L2 SRAM into a double buffer and use Direct Memory Access to eliminate the direct access of the external memory by the processor. Since the L2 SRAM is divided into 4 banks and two different banks can be accessed in the same cycle, the processor can access the data in one buffer while the other buffer is being filled with new data by the DMA. Although this setup removes the external memory penalty, however, L1 cache misses would still occur since all data are loaded from the L2 SRAM. With 4 cycles L1 cache miss/L2 cache hit penalty, total memory latency can be calculated. A L1 cache miss would allocates 8 words from the L2 SRAM so the total number of L1 cache miss would be 603,000 x 3 / 8 = 226,125 misses. Each miss takes 4 cycles so the total memory latency would be 226,125 x 4 = 905,500 cycles. Combining the computation requirement and the memory latency, the overall best optimized performance obtained is 750,000 + 905,500 = 1,655,500 cycles, which is about 73.7% of the total cycle budget.

6.0 PHONE AND WORD MODELING

This chapter covers both Phone Modeling and Word Modeling since both processing blocks works hand to hand together. The theory for both processing blocks will first be discussed and then the implementations and results.

Aside from continuous speech recognition, automatic speech recognition basically is finding the most probable match from a set of pre-defined words given an observed speech input. This set of pre-defined words is given as the dictionary for the system is chosen. The two key functions of the Word Modeling block are to track a list of active words and to prune the words that seem non-promising.

For a small sized dictionary, each word is different enough to be recognized based on word matching. However, for large sized dictionary, words like “capacity” and “capacities” can be difficult to differentiate. As a result, words are decomposed into multiple sub-word units called phonemes, or phones. A phoneme is a unique sound and there are 45 base phonemes RM1 corpus. Figure 6.1 illustrates an example of a word decomposed into multiple phones. With this new sub-word definition, the process of speech recognition can be seemed as phone matching instead of word matching.

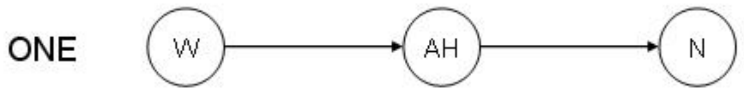


Figure 6.1: An example of word decomposed into multiple phones

In continuous speech, there isn't a clear boundary between phones. As a result, a concept of *context-dependent* phone or *triphone* is introduced. A triphone is made up of a base phone, a left context phone and a right context phone. With 45 base phones, well over 90,000 of tri-phones can be formed. However, not all triphones occur in real life, RM1 corpus only uses 30,080 tri-phones. Further reduction is archived by grouping triphones that are very similar. As a result, 5605 triphones are used in Sphinx 3. From this point on, the term phone will refer to all base phones and triphones.

The basic of ASR can be described as traversing each of the lists of phones and finding the best match. Each word in the word dictionary is represented by a composite of multiple phones. Table 6.1 shows a few words and their phonetic representations.

Table 6.1: Example of words in phonetic representation

Word	Phonetic Representation
ONE	W-AH-N
CALEDONIA	K-AE-L-IX-D-OW-N-IY-AY
CALIFORNIA	K-AE-L-AX-F-AO-R-N-Y-AX

In order to efficiently traverse through the large list of phones, words with the same beginning phone are grouped together in the dictionary. The entire dictionary structure can be visualized as the composition of multiple subtree structures where each subtree represents a group of words that share the same beginning phone. Figure 6.2 displays an example of a subtree.

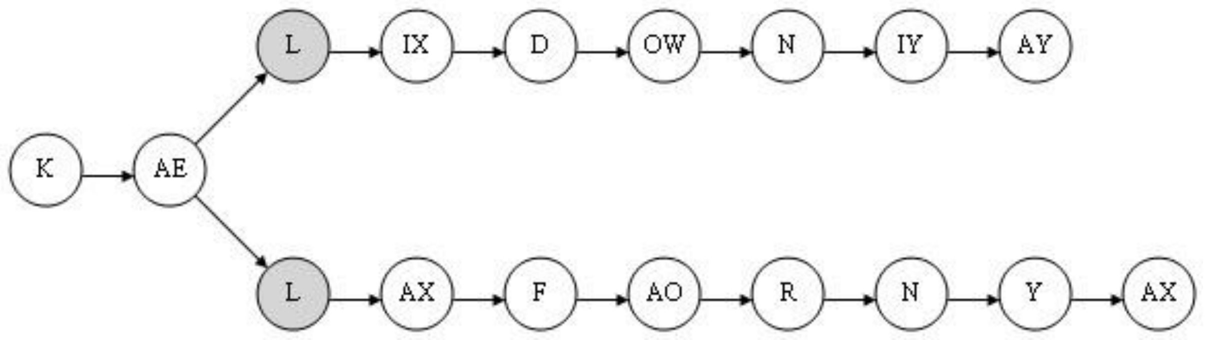


Figure 6.2: An example of a subtree with two words.

Notice that for continuous speech recognition, typically a triphone model is used. In other words, two phones are not considered to be the same if their previous and their next phones are not the same. For example, the phone /L/ in Figure 6.2 is not considered the same since the next phones of each /L/ phone is different.

Tracking a list of active phones for a large dictionary can be very challenging even with the optimized tree structure. According to [15], the number of active phones at a given time can be well over 10,000. Keeping track of such a large amount of data implies that substantially memory bandwidth is needed to operate efficiently. The cache structure of the system also plays a significantly role on the overall performance of the system [15].

The function of Word Modeling is word tracking. Since each word is modeled by multiple HMM nodes, word tracking means that the determination of which HMM node should be evaluated or eliminated. The complexity of WM is determined by the organization used to represent all HMM nodes from the words library. The total number of HMM nodes for the full dictionary of the RM1 corpus is 6305.

6.1 PHONE MODELING THEORY

Each phone is a unique representation of certain frequency characteristic. However, a phone spoken by different people would have slightly different characteristics. This variation can be caused by number of factors such as age, gender or speaking rate. To account for these variations, a statistical model called Hidden Markov Model (HMM) [21,22] is used to model phonemes. A phone is further broken down into multiple of states in HMM model. The HMM topology used in Sphinx 3 is a simple 3-states HMM where phones are divided into 3 states representing the beginning, middle and the end of a phone. Figure 6.3 displays a simple 3-state HMM.

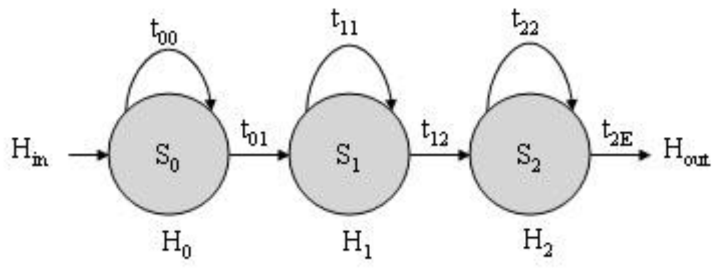


Figure 6.3: A simple 3-state HMM

Aside from the states, each HMM state (H_n) is also associated by two transition probabilities ($t_{n,n}$ and $t_{n,n+1}$), one transition back to the same state and the other transition to the next state.

In Sphinx 3, the states of a HMM is referred as Senones [8]. Each senone represents different part of the frequency characteristics of a phone. In a 3-state HMM, 3 senones are used to model the beginning, middle and the end of a phone. Each senone is modeled by a mixture of Gaussian Probability Density Distributions. The evaluations of these Gaussian distributions are done in Acoustic Modeling as the details will be discussed in section.5.0

Composite senones can be described as an optimization used to minimize the effects of the problem created by the use of context-dependent phones. The problem arises between the transitions of words. At the end of a word, any words are possible to be spoken. To represent this fact at the context-dependent phone level, the left context represents the end of the previous word. The middle and the right contexts, which represent the beginning of the next word, can be a significant amount of combinations. In other words, the number of phone evaluations needed increases exponentially as word is transited to the next. This problem is simplified by using a special phone instead of a tri-phone to represent these word transitions. The senones of this special phone is modeled not by one mixture Gaussian distribution but the composite of all mixture Gaussians distributions that are possible in different context positions, hence, formed Composite Senones. Furthermore, the number of senones that makeup a composite senone can vary depends on the dictionary. When composite senones are evaluated at AM, the score is said to be the best score among all senones that makeup such composite senone.

Phone Modeling is basically divided into 3 sections: HMM Evaluation, Finding Local Maximum and Node Pruning. HMM evaluation calculates the probability of each HMM nodes while Finding Local Max determines the maximum state score from all current active HMM nodes. The maximum score is then used as the basis for Node Pruning.

6.2 PHONEMES EVALUATION

There are two functions performed in PM. First is to compute the probability of the input frame given a phone model, this is referred to as Phone Score calculation. The other computation involves finding the phone models that are below certain threshold.

The following equations are performed during Phone Score calculation.

$H_{in}(t)$ = Negative Infinity Eq. 6.1

$H_0(t) = \text{MAX}\ [H_{in}(t)\ ,\ H_0(t-1) + t_{00}] + S_0(t)$ Eq. 6.2

$H_1(t) = \text{MAX}\ [H_0(t-1) + t_{01}\ ,\ H_1(t-1) + t_{11}] + S_1(t)$ Eq. 6.3

$H_2(t) = \text{MAX}\ [H_1(t-1) + t_{12}\ ,\ H_2(t-1) + t_{22}] + S_2(t)$ Eq. 6.4

$H_{out}(t) = H_2(t) + t_{2E}$ Eq. 6.5

where

- t and t-1 represent the current and the previous frames,
- $H_n(t)$ and $H_n(t-1)$ represent the current and previous state phone scores,
- $H_{in}(t)$ and $H_{out}(t)$ represent the exit phone score for the previous frame and the exit phone score for the current frame**,**
- t_{nn} represents the transition probabilities,
- $S_n(t)$ are senone score obtained from the AM block.

The equations above indicate that the state scores from the previous frame are needed to computer the state scores of the current frames.

Once the current phone scores are calculated, each of them can be evaluated. A phone can be classified into three states: pruned, exit to next phone, exit to next word. Three relative threshold values are calculated and used to determine the phone status. Three pre-defined values are needed to calculate these thresholds: *HMM_BEAM, PHONE_BEAM* and *WORD_BEAM*. The entire evaluation requires several steps of calculations. These steps and equations associated with each step are described below:

- Finding the local best phone score (*BEST_SCORE)*. For every active HMM, the best state score between the HMM states is first determined.

$\text{BEST\_SCORE} = \text{MAX}[H_0(t), H_1(t), H_2(t)]$ Eq. 6.6

- Finding the global best phone scores (*B_HMM*). The best score among all *BEST_SCOREs* is determined.

$\text{B\_HMM} = \text{MAX}[\text{BEST\_SCORE}_n]$ Eq. 6.7

- Perform *beam pruning* [23]. Instead of using absolute threshold, a relative threshold, called *HMM_TH*, is calculated every frame using the *B_HMM* and a pre-defined *HMM_BEAM* to account for variation in speech.

$\text{HMM\_TH} = \text{B\_MM} + \text{HMM\_BEAM}$ Eq. 6.8

- Pruning and generating a dead phone list. If the local *BEST_SCORE* < *HMM_TH*, it is moved to the dead list. This dead list is passed to the WM later so that the WM can track on which phones should keep active. Pruning non-promising phones implies that any branches in the subtree structure associated with these phones are also eliminated. This will effectively improves the performance of the overall system by reducing the required computations.
- Calculate exit to next phone threshold (*PHONE_TH*).

$\text{PHONE\_TH} = \text{B\_HMM} + \text{PHONE\_BEAM}$ Eq. 6.9

- Generate a list of phones that are ready to exit to the next phone. If $H_{out}(t) >= \text{PHONE\_TH}$, that phone is moved to the exit phone list.

- Calculate *B_HMM* among only active and word-exit phones (*B_HMM_WORD*).

  $\text{B\_HMM\_WORD} = \text{MAX}[\text{BEST\_SCORE}_n]$ Eq. 6.10

  where n only for active and word-exit phone

- Calculate exit to next word threshold (*WORD_TH)*.

  $\text{WORD\_TH} = \text{B\_HMM\_WORD} + \text{WORD\_BEAM}$ Eq. 6.11

- Generate a list of phones that are ready to exit to the next word. If $H_{out}(t) >= WORD\_TH$, that phone is moved to the exit word list.

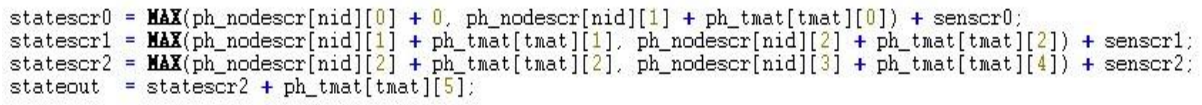
Three lists are generated as the output of PM block. Dead phone list, exit phone list and exit word list. All three lists are passed back to the WM block, which will remove phones that are on the dead list from the active list and generate new phones according to the exit phone list and exit word list.

6.3 WORD ORGANIZATION

In Sphinx and other work [1], all HMM nodes are organized in a tree structure where all similar words are grouped to form a sub-tree. This organization reduces the number HMM evaluations but requires more logic for node tracking and pruning. For this research, however, another organization is used. Each word is modeled as an individual linked list. This organization requires more HMM evaluation since common HMM node between two different words are actually modeled as two different nodes. However, the logic needed to track active node become very simple. Further, the linked list structure offers a very important advantage over the tree structure that will be discussed in the next (section.6.4)

6.4 IMPLEMENTATION AND RESULT

From the computational prospective, HMM evaluation is very simple. Three HMM state scores plus an exit scores are computed for each HMM. The source code for HMM core evaluation is shown in Figure 6.4. Note that the full list of the source code is attached in APPENDIX.



```
statescr0 = MAX(ph_nodescr[nid][0] + 0, ph_nodescr[nid][1] + ph_tmat[tmat][0]) + senscr0;
statescr1 = MAX(ph_nodescr[nid][1] + ph_tmat[tmat][1], ph_nodescr[nid][2] + ph_tmat[tmat][2]) + senscr1;
statescr2 = MAX(ph_nodescr[nid][2] + ph_tmat[tmat][2], ph_nodescr[nid][3] + ph_tmat[tmat][4]) + senscr2;
stateout  = statescr2 + ph_tmat[tmat][5];
```

Figure 6.4: Source code for the core part of the HMM evaluation function

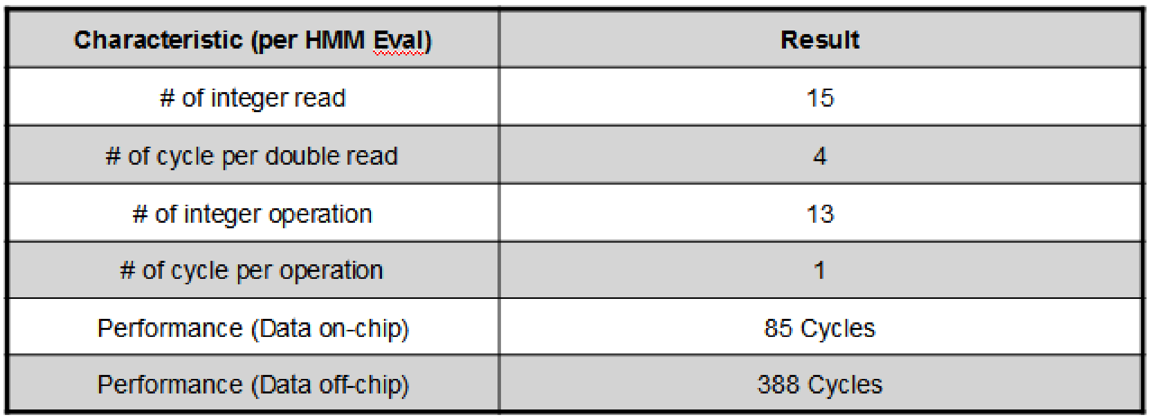
Each state score *(statescr0, statescr1, statescr2)* computation involves 3 integer additions and 1 integer comparison. The HMM exit score *(stateout)* only requires an integer addition. It is clear that an HMM evaluation is simple computational wise. However, similar to Gaussian evaluation, memory bandwidth is the problem. The 1st state score requires 4 integer inputs while the 2nd and 3rd state score requires 5 integers each. Exit score requires another integer. Total 15 integers are needed for each HMM evaluation. It should be clear now that the performance of HMM evaluations are memory bound instead of computational bound.

As mentioned in the previous section, 6305 HMM nodes are used for the RM1. Each HMM node contains the data for HMM evaluation in addition to all the status information required for node tracking. As a result, a data structure of size about 3 Mbytes is needed where each entry of this data structure represents a node with all the information. Since the size of the cache available is too small, external memory is used instead. As a node is activated by Word Modeling, all the possible branching nodes from the active node must be pre-loaded. In other words, numbers of entries from the Phone data structure must be accessed from the external memory and be loaded into the cache memory to avoid significant delay. As indicated in AM that accesses to the external memory requires long cycle delay and DMA is a solution that can solve this problem. However, under the tree structure, the effect of DMA is very limited. DMA works well to access a block of data that are stored sequentially where different entries of the branching nodes cannot be stored in sequential manner. On the other hand, under the linked list structure, all of the data can be stored sequentially according to the order of the HMM nodes that makes up a word. As a result, when the first node of a word is activated, the rest of the nodes of that active word can be load as a block using DMA. The size of the block being transferred depends on the number of words that are active and then number of nodes associated with each of the active word. As shown in [1], each word is made up of 8 HMM nodes where each entry of the HMM data structure is about 68 bytes (60 bytes of HMM data and 8 bytes of status information) in size. If assuming 10% of all words are active on average, then the average transfer block size would be (6305 x 10%) x 68 bytes x 8 nodes = 342,992 bytes. By using the linked list structure, a significant amount of cycle delay can be avoided enabling the process to be executed in real-time.

From the pure computational perspective, the performance of Phone Modeling can be measured at one HMM evaluation and a sub-function that determine the maximum scores among the 3 state scores and the exit score. If all data are available on-chip, 85 cycles would be required while 388 cycles would be needed if data are stored off-chip.

As for Word Modeling, the implementation source code is listed in APPENDIX. The code is functionally verified with senone scores obtained from Sphinx that represents the word “california” and “capacity”. Table 6.2 summarizes the characteristics and performance.

Table 6.2: HMM Characteristics and Performance



Characteristic (per HMM Eval)	Result
# of integer read	15
# of cycle per double read	4
# of integer operation	13
# of cycle per operation	1
Performance (Data on-chip)	85 Cycles
Performance (Data off-chip)	388 Cycles

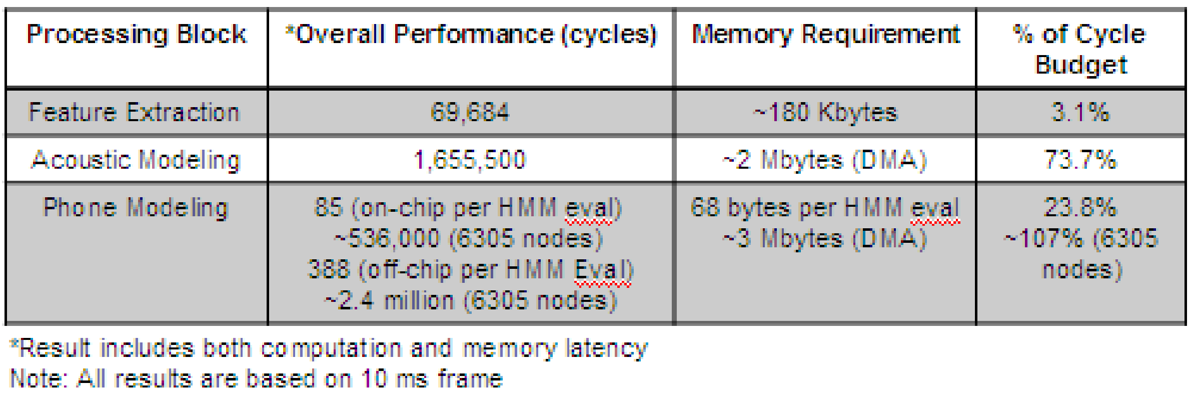
7.0 CONCLUSION

Automatic Speech Recognition is extremely computational intensive. Many design approaches had been taken to solve such problem but none has yet attempted to map this recognition process onto a VLIW DSP. The work of this research is to explore the possibility and the feasibility of mapping this recognition process onto TI's C6713 VLIW DSP.

The work of this research has discovered that the problem is not only the massive amount of computation required but feeding the necessary information from memory to the processor to perform these calculations. One solution is to use DMA and the on-chip memory buffer to allow data to be moved from outside memory while the processor is performing calculations. This method works well for sequential memory accesses such as those in Acoustic Modeling. For Phone and Word modeling where memory accesses are more irregular, the solution is to organize the HMM nodes in the linked list structure instead of the tree structure. The linked list structure enables the use of DMA to transfer HMM data as a block from the external memory where tree structure would not be able to do so.

Although it is not possible to map the entire 1000 words recognition real-time process onto the TI C6713 development platform due to the processor speed, however, each individual processing block is implemented and tested. Table 8.1 summarized the results of each processing block.

Table 7.1: Summary of results of all processing blocks



Processing Block	*Overall Performance (cycles)	Memory Requirement	% of Cycle Budget
Feature Extraction	69,684	~180 Kbytes	3.1%
Acoustic Modeling	1,655,500	~2 Mbytes (DMA)	73.7%
Phone Modeling	85 (on-chip per HMM eval) ~536,000 (6305 nodes) 388 (off-chip per HMM Eval) ~2.4 million (6305 nodes)	68 bytes per HMM eval ~3 Mbytes (DMA)	23.8% ~107% (6305 nodes)

*Result includes both computation and memory latency
Note: All results are based on 10 ms frame

7.1 MAJOR CONTRIBUTIONS

The major contributions of this thesis are as follows:

- Redesigned and mapped a real-time, speaker independent, command and control based speech recognition system onto a multi-core DSP architecture. Also examined and optimized each stage of the recognition process including Feature Extraction, Acoustic Modeling, Phone & Word Modeling.
- Optimized Feature Extraction using various optimization techniques including Memory Aliasing, Software Pipeline, Loop Unrolling and Packed Data Memory Access. The overall optimized performance is 69,684 cycles, which is 3.1% of the real-time cycle budget of 2.25 million cycles.
  - For Pre-Emphasis, performance improved 322% from 14,500 to 45,00 cycles
  - For Windowing, performance improved 167% from 2,050 to 1,230 cycles
  - For Power Spectrum, performance improved 384% from 193,210 to 50,268 cycles
    - Optimized the FFT part of Power Spectrum using Texas Instruments Signal Processing Library

    - For Mel Spectrum, performance improved 155% from 3,145 to 2,035 cycles
    - For Mel Cepstrum, performance improved 159% from 12,580 to 7,900 cycles
    - For Dynamic Feature, performance improved 162% from 6,092 to 3,751 cycles
- Investigated Acoustic Modeling in two different approaches, Component Approach and Dimension Approach, and determined that the Dimension Approach has a better optimal performance of 0.75 million cycles (33.3% of the real-time cycle budget of 2.25 million cycles) than 1.22 million cycles performance of the Component Approach.
- Optimized the Component Approach using various optimization techniques. The baseline performance measured before any applying any optimization is 33.8 million cycles, which is 15x the real-time cycle budget of 2.25 million. The best optimized performance obtained is 1.22 million cycles, 54% of the real-time cycle budget.
    - Applying standard optimizations (Variable Registering and Constant Propagation), performance improved 151% from 33.8 to 22.34 million cycles
    - Plus Software Pipeline, performance improved 1,400% over baseline to 2.41 million cycles
    - Plus Loop Unrolling by 2x or 3x, performance improved 2,800% over baseline to 1.22 million cycles
- Optimized the Dimension Approach using various optimization techniques. The optimal performance archived is 0.75 million cycles, which is 33.3% of real-time.
    - Applying standard optimizations and Software Pipeline, performance improved 255% over baseline to 13.28 million cycles
    - Plus Memory Aliasing, performance improved 2,800% to 1.21 million cycles
    - Plus Packed Data Memory Access and Loop Unrolling by 2x, performance improved 3733% to 0.91 million cycles
    - Or Packed Data Memory Access and Loop Unrolling by 6x, performance improved 4211% to 0.80 million cycles

    - Or Packed Data Memory Access and Loop Unrolling by 4x, performance improved 4480% to 0.75 million cycles
- Resolved the memory bandwidth problem in Acoustic Modeling by eliminating the delay effect of accessing external memory using DMA and Cache Buffering technique. The result is 4 cycles of delay for every cache miss instead of ~100 cycles of delay when external memory is accessed (exact delays depends on the speed of the processor and the speed of the external memory device).
- Examined and implemented Phone & Word Modeling and determined that they are computationally simple but requires a large memory bandwidth. A HMM evaluation would only consumes 85 cycles (data in cache) rather than 388 cycles (data in external memory). The best solution is to organize each word as an individual linked list and to store all the information for each HMM node in the order of the nodes that make up a word. This method will allow the use of DMA to solve the memory bandwidth problem.

7.2 FUTURE WORK

The work of this thesis demonstrated that it is possible to implement a speech recognition process with a fairly large sized word library onto an embedded hardware platform. The next step would be to complete the entire implementation provided that another DSP platform with sufficient on-chip SRAM is available.

APPENDIX

SOURCE CODE

This section of the appendix contains the C code for the implementations for all phases.

```
#include "c:\\ti\\c6700\\dsplib\\include\\dspf_sp_cfftr2_dit.h"
#include "tw_r2fft.c"

extern far  LOG_Obj trace;
extern Int  INRAM;
extern Int  L2SRAM;
extern Int  EXRAM;
extern Int  EXRAM1;

float *sin_n, *cos_n;

/* FE CONSTANTS */
#define invlogB             ((float) 3333.280269)
#define base                    ((float) 1.000300)
#define invBase             ((float) 0.999700)
#define NEG_INFINITY            ((Int)  -939524096)

/* SPHINX CONSTANTS */
#define MEL_SCALE 1

#define M_PI    (3.14159265358979323846)
```

```
#define CMN_WIN_HWM     800          /* #frames after which window shifted */
#define CMN_WIN        500

#define LIVEBUFBLOCKSIZE       256   /* Blocks of 256 vectors allocated for
livemode decoder */

/* Default feature extraction values */
#define DEFAULT_SAMPLING_RATE       16000.0
#define DEFAULT_FRAME_RATE          100
#define DEFAULT_FRAME_SHIFT         160
#define DEFAULT_WINDOW_LENGTH       0.025625
#define DEFAULT_FFT_SIZE                512
#define DEFAULT_FB_TYPE             MEL_SCALE
#define DEFAULT_NUM_CEPSTRA         13
#define DEFAULT_NUM_FILTERS         40
#define DEFAULT_LOWER_FILT_FREQ     133.33334
#define DEFAULT_UPPER_FILT_FREQ     6855.4976
#define DEFAULT_PRE_EMPHASIS_ALPHA  0.97
#define DEFAULT_START_FLAG          0

/* Defaults for MEL_SCALE for different sampling rates. */
#define BB_SAMPLING_RATE                16000
#define DEFAULT_BB_FFT_SIZE         512
#define DEFAULT_BB_FRAME_SHIFT      160
#define DEFAULT_BB_NUM_FILTERS      40
#define DEFAULT_BB_LOWER_FILT_FREQ  133.33334
#define DEFAULT_BB_UPPER_FILT_FREQ  6855.4976

#define NB_SAMPLING_RATE                8000
#define DEFAULT_NB_FFT_SIZE         512
#define DEFAULT_NB_FRAME_SHIFT      80
#define DEFAULT_NB_NUM_FILTERS      31
#define DEFAULT_NB_LOWER_FILT_FREQ  200
#define DEFAULT_NB_UPPER_FILT_FREQ  3500

#define DEFAULT_BLOCKSIZE 200000
```

```
#define feat_name(f)                ((f)->name)
#define feat_cepsize(f)         ((f)->cepsize)
#define feat_cepsize_used(f)    ((f)->cepsize_used)
#define feat_n_stream(f)        ((f)->n_stream)
#define feat_stream_len(f,i)    ((f)->stream_len[i])
#define feat_window_size(f) ((f)->window_size)

#define MIN(a,b) (((a) < (b))? (a):(b))
#define MAX(a,b) (((a) > (b))? (a):(b))

// MEMORY REQ: 10 x 4 = 40 bytes
typedef struct {

    float SAMPLING_RATE;
    Int   FRAME_RATE;
    float WINDOW_LENGTH;
    Int   FB_TYPE;
    Int   NUM_CEPSTRA;
    Int   NUM_FILTERS;
    Int   FFT_SIZE;
    float LOWER_FILT_FREQ;
    float UPPER_FILT_FREQ;
    float PRE_EMPHASIS_ALPHA;

} param_t;

// MEMORY REQ: Total: 84,344 bytes.
// Variables-24
// filter_coeff (# of filter x FFT size x 4) = 81920
// mel_cosine (# of filter x # of cepstra x 4) = 2080
// left_apex, width (# of filter x 4) = 160
typedef struct {

    float sampling_rate;
    Int   num_cepstra;
    Int   num_filters;
    Int   fft_size;
    float lower_filt_freq;
    float upper_filt_freq;
    float **filter_coeffs;
    float **mel_cosine;
    float *left_apex;
    Int   *width;

} melfb_t;

// MEMORY REQ: Total: 87,672 bytes.
// Variables (12 x 4) = 48 bytes
// overflow_samp (frame_size x 4) = 1640
// MEL_FB = 84,344
// HAMMING_WINDOW (frame_size x 4) = 1640
typedef struct {

    float   SAMPLING_RATE;
    Int     FRAME_RATE;
    Int     FRAME_SHIFT;
    float   WINDOW_LENGTH;
    Int     FRAME_SIZE;
    Int     FFT_SIZE;
    Int     FB_TYPE;
    Int     NUM_CEPSTRA;
    float   PRE_EMPHASIS_ALPHA;
    Int     *OVERFLOW_SAMPS;
    Int     NUM_OVERFLOW_SAMPS;
    melfb_t *MEL_FB;
    Int     START_FLAG;
    Int     PRIOR;
    float   *HAMMING_WINDOW;

} fe_t;

// MEMORY_REQ: 34 bytes
typedef struct feat_s {

    Char *name;             /* Printable name for this feature type */
    Int  cepsize;           /* Size of input speech vector (typically, a
cepstrum vector) */
    Int  cepsize_used;   /* No. of cepstrum vector dimensions actually used (0
onwards) */
    Int  n_stream;          /* #Feature streams; e.g., 4 in Sphinx-II */
    Int  *stream_len;       /* Vector length of each feature stream */
    Int  window_size;       /* #Extra frames around given input frame needed
to compute
                          corresponding output feature (so total =
window_size*2 + 1) */
    Int  cmn;               /* Whether CMN is to be performed on each utterance
*/
    Int  varnorm;           /* Whether variance normalization is to be
performed on each utt;
                          Irrelevant if no CMN is performed */
    Int  agc;               /* Whether AGC-Max is to be performed on each
utterance */
    Void (*compute_feat)(struct feat_s *fcb, float **input, float **feat);
    /* Function for converting window of input speech vector
      (input[-window_size..window_size]) to output feature vector
      (feat[stream][]).  If NULL, no conversion available, the
      speech input must be feature vector itself.
      Return value: 0 if successful, -ve otherwise. */

} feat_t;

typedef struct { float r, i; } complex;

// Condition Check. Die if Condition is not True //
#define xassert(cond) \
  do { if (!(cond)) LOG_printf(&trace, "Assertion failed!  (%s:%i)\n", \
                      __LINE__, __FILE__); } while (0)

/* ==================================================================
*/
/*  GEN_TWIDDLE -- Generate twiddle factors for TI's custom FFTs.         */
/*      The routine will generate the twiddle-factors directly into the    */
/*      array you specify.  The array needs to be N elements long.         */
/* ==========================================================================
*/

int gen_twiddle(float * restrict w, int n)
{
   int i;

   /*
   double delta = 2 * PI / n;
   for(i = 0; i < n/2; i++)
   {
      w[2 * i + 1] = sin(i * delta);
      w[2 * i] = cos(i * delta);
   }
   */

    for (i = 0; i < n/2; i++)
    {
        w[2 * i + 1] = sin_n[i];
        w[2 * i] = cos_n[i];
    }

    return n;
}

// obtain_fe_params: fill a param_t with the correct values
static Void obtain_fe_params(param_t *p)
{
    // initials structure to 0 - JN
    memset(p, 0, sizeof(param_t));

    p->SAMPLING_RATE            = 16000.f;
    p->FRAME_RATE               = 100;
    p->PRE_EMPHASIS_ALPHA   = .97f;
    p->LOWER_FILT_FREQ      = 133.33334f;
    p->UPPER_FILT_FREQ      = 6855.49756f;
    p->NUM_FILTERS          = 40;
}
```

```
Void fe_parse_general_params(param_t *P, fe_t *FE)
{
    #define priv_def_int(x) (FE->x = ((P->x) == 0)? \
                     (DEFAULT_##x) : (P->x))
    #define priv_def_flt(x) (FE->x = ((P->x) == 0)? \
                     ((float)DEFAULT_##x) : (P->x))
    priv_def_int(SAMPLING_RATE);
    priv_def_int(FRAME_RATE);
    priv_def_flt(WINDOW_LENGTH);
    priv_def_int(FB_TYPE);
    priv_def_flt(PRE_EMPHASIS_ALPHA);
    priv_def_int(NUM_CEPSTRA);
    priv_def_int(FFT_SIZE);

    #undef priv_def_int
    #undef priv_def_flt
}

// fe_create_hamming: compute a Hamming filter as an array
// of floats of length len, into the array specified by ary.
Void fe_create_hamming(float *ary, Int len)
{
    Int i;

    if (len > 1) {
    for (i = 0; i < len; i++)
        ary[i] = 0.54 - 0.46*cos(2*M_PI*i/(len-1));
    }
}

// fe_parse_melfb_params: initialize MEL_SCALE parameter
// values from a param_t struct
Void fe_parse_melfb_params(param_t *P, melfb_t *MEL)
{
    MEL->sampling_rate = DEFAULT_SAMPLING_RATE;
    MEL->num_cepstra = DEFAULT_NUM_CEPSTRA;
    // Converted to invert block structure
    if (MEL->sampling_rate == BB_SAMPLING_RATE) {
        MEL->fft_size = DEFAULT_BB_FFT_SIZE;
        MEL->num_filters = DEFAULT_BB_NUM_FILTERS;
        MEL->upper_filt_freq = (float)DEFAULT_BB_UPPER_FILT_FREQ;
        MEL->lower_filt_freq = (float)DEFAULT_BB_LOWER_FILT_FREQ;
    } else if (MEL->sampling_rate == NB_SAMPLING_RATE) {
        MEL->fft_size = DEFAULT_NB_FFT_SIZE;
        MEL->num_filters = DEFAULT_NB_NUM_FILTERS;
        MEL->upper_filt_freq = (float)DEFAULT_NB_UPPER_FILT_FREQ;
        MEL->lower_filt_freq = (float)DEFAULT_NB_LOWER_FILT_FREQ;
    } else {
        MEL->fft_size = DEFAULT_FFT_SIZE;
        if (P->NUM_FILTERS == 0)
          LOG_printf (&trace, "Error initializing MEL_SCALE filter params:
number of necessary filters is undefined!\n");
        if (P->UPPER_FILT_FREQ == 0)
          LOG_printf (&trace, "Error initializing MEL_SCALE filter params:
upper filter frequency is undefined!\n");
        if (P->LOWER_FILT_FREQ == 0)
          LOG_printf (&trace, "Error initializing MEL_SCALE filter params:
lower filter frequency is undefined!\n");
    }

    // allow args file to set the parameters
    if (P->FFT_SIZE != 0)       MEL->fft_size = P->FFT_SIZE;
    if (P->NUM_FILTERS != 0)     MEL->num_filters = P->NUM_FILTERS;
    if (P->SAMPLING_RATE != 0)   MEL->sampling_rate = P->SAMPLING_RATE;
    if (P->NUM_CEPSTRA != 0)     MEL->num_cepstra = P->NUM_CEPSTRA;
    if (P->UPPER_FILT_FREQ != 0) MEL->upper_filt_freq = P->UPPER_FILT_FREQ;
    if (P->LOWER_FILT_FREQ != 0) MEL->lower_filt_freq = P->LOWER_FILT_FREQ;
}

float fe_mel(float x)
{
  return (float)(2595.0*(float)log10(1.0+x/700.0));
}
```

```
float fe_melinv(float x)
{
  return(float)(700.0*((float)pow(10.0,x/2595.0) - 1.0));
}

// Run-Time MEMORY REQ: Total = 228 bytes
// variables (15 x 4 = 60 bytes)
// filt_edge ( (# of filters + 2) x 4 ) = 168 bytes
Int fe_build_melfilters(melfb_t *MEL_FB)
{
    Int i, whichfilt, start_pt;
    float   leftfr, centerfr, rightfr, fwidth, height, *filt_edge;
    float   melmax, melmin, dmelbw, freq, dfreq, leftslope, rightslope;

    /* estimate filter coefficients */
    MEL_FB->filter_coeffs = (float **)xcalloc_2d(INRAM,
(Int)MEL_FB->num_filters, (Int)MEL_FB->fft_size, sizeof(float));

    MEL_FB->left_apex = (float *) MEM_calloc (INRAM,
sizeof(float)*MEL_FB->num_filters, 0);
    if (MEL_FB->left_apex == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: MEL_FB->left_apex");

   MEL_FB->width = (Int *) MEM_calloc (INRAM, sizeof(Int)*MEL_FB->num_filters,
0);
    if (MEL_FB->width == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: MEL_FB->width");

    filt_edge = (float *) MEM_calloc (INRAM,
sizeof(float)*(MEL_FB->num_filters+2), 0);
    if (filt_edge == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: filt_edge");

    dfreq = MEL_FB->sampling_rate/(float)MEL_FB->fft_size;

    melmax = fe_mel(MEL_FB->upper_filt_freq);
    melmin = fe_mel(MEL_FB->lower_filt_freq);
    dmelbw = (melmax-melmin)/(MEL_FB->num_filters+1);
```

```
for (i = 0; i <= MEL_FB->num_filters+1; i++) {
    filt_edge[i] = fe_melinv(i*dmelbw + melmin);
}

for (whichfilt=0;whichfilt<MEL_FB->num_filters; ++whichfilt) {
    /* line triangle edges up with nearest dft points... */
    leftfr   = (float)((Int)((filt_edge[whichfilt]/dfreq)+0.5))*dfreq;
    centerfr = (float)((Int)((filt_edge[whichfilt+1]/dfreq)+0.5))*dfreq;
    rightfr  = (float)((Int)((filt_edge[whichfilt+2]/dfreq)+0.5))*dfreq;

    MEL_FB->left_apex[whichfilt] = leftfr;

    fwidth = rightfr - leftfr;

    /* 2/fwidth for triangles of area 1 */
    height = 2/(float)fwidth;
    leftslope = height/(centerfr-leftfr);
    rightslope = height/(centerfr-rightfr);

    start_pt = 1 + (Int)(leftfr/dfreq);
    freq = (float)start_pt*dfreq;
    i = 0;

    while (freq <= centerfr) {
        MEL_FB->filter_coeffs[whichfilt][i] = (freq-leftfr)*leftslope;

        freq += dfreq;
        i++;
    }
    while (freq < rightfr){
        MEL_FB->filter_coeffs[whichfilt][i] = (freq-rightfr)*rightslope;
        freq += dfreq;
        i++;
    }

    MEL_FB->width[whichfilt] = i;
}
```

```
    MEM_free (INRAM, filt_edge, sizeof(float)*(MEL_FB->num_filters+2));

    return(0);
}

// Run-Time MEMORY REQ: Total = 16 bytes
// variables (4 x 4 = 16 bytes)
Int fe_compute_melcosine(melfb_t *MEL_FB)
{
    float period, freq;
    Int i, j;

    period = (float)2*MEL_FB->num_filters;

    MEL_FB->mel_cosine = (float **) xmalloc_2d (INRAM, MEL_FB->num_cepstra,
MEL_FB->num_filters, sizeof(float));
    if (MEL_FB->mel_cosine == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for MEL_FB->mel_cosine");

    for (i = 0; i < MEL_FB->num_cepstra; i++) {
        freq = 2*(float)M_PI*(float)i/period;
        for (j=0;j< MEL_FB->num_filters;j++)
            MEL_FB->mel_cosine[i][j] = (float)cos((float)(freq*(j+0.5)));
    }

  return 0;
}

// fe_init: initialize a feature extraction object from the parameters
// in P, and set up internal state appropriately.  Parameters that are 0
// are set to default values.
fe_t *fe_init(param_t *P)
{
    int i;
    double delta = (2*M_PI)/DEFAULT_FFT_SIZE;

    fe_t    *FE = (fe_t *)MEM_calloc(INRAM, sizeof(fe_t)*1, 0);
    if (FE == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for FE");

    // JN: added this to reduce cycle count for FFT
    sin_n = (float *)MEM_calloc(L2SRAM, sizeof(float)*DEFAULT_FFT_SIZE, 0);
    if (sin_n == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for sin_n");

    cos_n = (float *)MEM_calloc(L2SRAM, sizeof(float)*DEFAULT_FFT_SIZE, 0);
    if (cos_n == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for cos_n");

    for (i = 0; i < (DEFAULT_FFT_SIZE/2); i++)
    {
        sin_n[i] = sin(i * delta);
        cos_n[i] = cos(i * delta);
    }

    // transfer params to front end
    fe_parse_general_params(P,FE);

    // compute remaining FE parameters
    FE->FRAME_SHIFT = (Int)(FE->SAMPLING_RATE/FE->FRAME_RATE + 0.5);
    FE->FRAME_SIZE = (Int)(FE->WINDOW_LENGTH*FE->SAMPLING_RATE + 0.5);
    FE->PRIOR = 0;

    // establish buffers for overflow samps and hamming window
   FE->OVERFLOW_SAMPS = (Int *) MEM_calloc (INRAM, sizeof(Int)*FE->FRAME_SIZE,
0);
    if (FE->OVERFLOW_SAMPS == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for FE->OVERFLOW_SAMPS");

    FE->HAMMING_WINDOW = (float *) MEM_calloc (INRAM,
sizeof(float)*FE->FRAME_SIZE, 0);
    if (FE->HAMMING_WINDOW == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for FE->HAMMING_WINDOW");

    // create hamming window
    fe_create_hamming(FE->HAMMING_WINDOW, FE->FRAME_SIZE);

    // init and fill appropriate filter structure
    switch (FE->FB_TYPE) {

        case MEL_SCALE: {

            //FE->MEL_FB = (melfb_t *)xcalloc(1,sizeof(melfb_t));
            FE->MEL_FB = (melfb_t *) MEM_calloc (INRAM, sizeof(melfb_t)*1, 0);
            if (FE->MEL_FB == MEM_ILLEGAL)
                LOG_printf (&trace, "Failed to calloc memory for MEL_FB");

            // transfer params to mel fb
            fe_parse_melfb_params(P, FE->MEL_FB);
            fe_build_melfilters(FE->MEL_FB);
            fe_compute_melcosine(FE->MEL_FB);

        } break;

        default:
            LOG_printf (&trace, "Can't initialize FE: invalid filter type\n");
    }

    return FE;
}

// Duplicate a String
static Char *xstrdup (const Char *string)
{
    const Uns len = strlen(string) + 1;
    Char *newstr = MEM_alloc (INRAM, len, 0);
    memcpy (newstr, string, len);
    return newstr;
}

/* feat_s3_1x39_cep2feat: do some math for converting cepstra to
   features. */
static Void feat_s3_1x39_cep2feat(feat_t *fcb, float **mfc, float **feat)
```

```
{
  float *f;
  float *w, *_w;
  float *w1, *w_1, *_w1, *_w_1;
  float d1, d2;
  Int i;

  /* xassert -- DPW */
  xassert (fcb);
  xassert (feat_cepsize (fcb) == 13);
  xassert (feat_cepsize_used (fcb) == 13);
  xassert (feat_n_stream (fcb) == 1);
  xassert (feat_stream_len (fcb, 0) == 39);
  xassert (feat_window_size (fcb) == 3);

  /* CEP; skip C0 */
  memcpy (feat[0], mfc[0]+1, (feat_cepsize(fcb)-1) * sizeof(float));

  /*
   * DCEP: mfc[2] - mfc[-2];
   */
  f = feat[0] + feat_cepsize(fcb)-1;
  w  = mfc[2] + 1;   /* +1 to skip C0 */
  _w = mfc[-2] + 1;

  for (i = 0; i < feat_cepsize(fcb)-1; i++)
    f[i] = w[i] - _w[i];

  /* POW: C0, DC0, D2C0 */
  f += feat_cepsize(fcb)-1;

  f[0] = mfc[0][0];
  f[1] = mfc[2][0] - mfc[-2][0];

  d1 = mfc[3][0] - mfc[-1][0];
  d2 = mfc[1][0] - mfc[-3][0];
  f[2] = d1 - d2;
  /* D2CEP: (mfc[3] - mfc[-1]) - (mfc[1] - mfc[-3]) */
  f += 3;

  w1   = mfc[3] + 1; /* Final +1 to skip C0 */
  _w1  = mfc[-1] + 1;
  w_1  = mfc[1] + 1;
  _w_1 = mfc[-3] + 1;

  for (i = 0; i < feat_cepsize(fcb)-1; i++) {
    d1 =  w1[i] -  _w1[i];
    d2 = w_1[i] - _w_1[i];

    f[i] = d1 - d2;
  }
}

/* feat_init: initialize feature conversion block and return it. */
feat_t *feat_init(void)     /* No arguments now -- DPW */
{
    feat_t *fcb;

    //fcb = (feat_t *)xcalloc(1, sizeof(feat_t));
    fcb = (feat_t *)MEM_calloc (INRAM, sizeof(feat_t)*1, 0);
    if (fcb == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: fcb");

    /* Feature type is always "1s_c_d_dd". -- DPW */
    {
    fcb->name = (Char *)xstrdup("1s_c_d_dd");

    /* 1-stream cep/dcep/pow/ddcep (Hack!! hardwired constants below) */
    fcb->cepsize = 13;
    fcb->cepsize_used = 13;
    fcb->n_stream = 1;
    /* xcalloc -- DPW */
    fcb->stream_len = (Int *) MEM_calloc (INRAM, sizeof(Int)*1, 0);
    fcb->stream_len[0] = 39;
    fcb->window_size = 3;
    /* Added & for clarity -- DPW */
    fcb->compute_feat = &feat_s3_1x39_cep2feat;
    }

    /* cmn is always "current".
     varnorm is always "no".
     agc is always "none";
     -- DPW */
    fcb->cmn = 1;
    fcb->varnorm = 0;
    fcb->agc = 0;

    return fcb;
}

/* fe_start_utt: ready a feature extraction object to accept another utterance.
*/
Void fe_start_utt(fe_t *FE)
{
    FE->NUM_OVERFLOW_SAMPS = 0;
    memset(FE->OVERFLOW_SAMPS, 0, FE->FRAME_SIZE * sizeof(Short));
    FE->START_FLAG = 1;
    FE->PRIOR = 0;
}

/* fe_pre_emphasis: do short->float conversion from in to out of length
   len with pre-emphasis factor factor. */
Void fe_pre_emphasis(Int *in, float *out, Int len, float factor, Short prior)
{
    Int i;

    out[0] = (float)in[0] - factor*(float)prior;
    for (i = 1; i < len; i++)
    {
    out[i] = (float)in[i] - factor*(float)in[i-1];
    }
}
```

```
/* fe_short_to_float: do short->float conversion of length len
   from in to out with no pre-emphasis. */
Void fe_short_to_float(Int *in, float *out, Int len)
{
  Int i;
  for (i = 0; i < len; i++)
    out[i] = (float)in[i];
}

/* fe_multiply_window: multiply the float array ary in-place by the
   window window, with length len. */
Void fe_multiply_window(float * restrict ary, float * restrict window, Int len)
{
    Int i;

    WORD_ALIGNED(ary);
    WORD_ALIGNED(window);

    // JN: since len is fe->frame_size, can assume even number for now 410
    #pragma MUST_ITERATE(20, ,2);
    for (i = 0; i < len; i++)
        ary[i] *= window[i];

    return;
}

/* fe_spec_magnitude: do something or other. */
// JN: data_len = FE->FRAME_SIZE which can be consider constant of 410
void fe_spec_magnitude(float * restrict data, Int data_len, float * restrict
spec, Int fftsize)
{
    Int j,k,wrap;
    float *w, *fIN;

    fIN = (float *) MEM_calloc (L2SRAM, sizeof(float)*2*fftsize, 0);
    if (fIN == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: fIN");
```

```
w = (float *) MEM_calloc (L2SRAM, sizeof(float)*fftsize, 0);
if (w == MEM_ILLEGAL)
    LOG_printf (&trace, "Failed to calloc memory for: w");

if (data_len > fftsize) {
    for (j = 0, k = 0; j < fftsize; j++, k=k+2) {
        fIN[k] = data[j];
        fIN[k+1] = 0.0;
    }
    for (wrap = 0; j < data_len; wrap=wrap+2, j++) {
        fIN[wrap] += data[j];
        fIN[wrap+1] += 0.0;
    }
} else {
    // performance: 1.5 cycle per iteration. Total 1.5*410 = 615 cycles
    #pragma MUST_ITERATE (20, , 2);
    for (j = 0, k = 0; j < data_len; j++, k=k+2){
        fIN[k] = data[j];
        fIN[k+1] = 0.0;
    }
    // performance: 1 cycle per iteration. Total 1*512 = 512 cycles
    #pragma MUST_ITERATE (20, , 2);
    for ( ; j < fftsize; j++,k=k+2) {
        fIN[k] = 0.0;
        fIN[k+1] = 0.0;
    }
}

// performance: 2 cycle per iteration. Total of 2*(512/2) = 512 cycles
gen_twiddle(w, fftsize);
// performance: 12000 cycles from profiler
bit_rev(w, fftsize>>1);
// performance: 11500 cycles from profiler
DSPF_sp_cfftr2_dit (fIN, w, fftsize);
// performance: 24000 cycles from profiler
bit_rev(fIN, fftsize);

// performance: 2 cycle/iteration. Total of 2*(512/2) = 512 cycles
```

```
    WORD_ALIGNED(fIN);
    for (j = 0; j <= fftsize/2; j++)
    {
        spec[j] = fIN[2*j]*fIN[2*j] + fIN[2*j+1]*fIN[2*j+1];
    }

    MEM_free (L2SRAM, fIN, sizeof(float)*2*fftsize);
    MEM_free (L2SRAM, w, sizeof(float)*fftsize);

    return;
}

Void fe_mel_spec(fe_t * restrict FE, float * restrict spec, float * restrict
mfspec)
{
    Int whichfilt, start, i;
    float dfreq;

    dfreq = FE->SAMPLING_RATE/(float)FE->FFT_SIZE;

    for (whichfilt = 0; whichfilt < FE->MEL_FB->num_filters; whichfilt++){

        start = (Int)(FE->MEL_FB->left_apex[whichfilt]/dfreq) + 1;
        mfspec[whichfilt] = 0;

        for (i = 0; i < FE->MEL_FB->width[whichfilt]; i++)
            mfspec[whichfilt] += FE->MEL_FB->filter_coeffs[whichfilt][i] *
spec[start+i];
    }
}

void fe_mel_cep(fe_t * restrict FE, float * restrict mfspec, float * restrict
mfcep)
{
    Int i,j;
    Int period;
    float beta;
    period = FE->MEL_FB->num_filters;

    for (i = 0; i < FE->MEL_FB->num_filters; i++){
        if (mfspec[i]>0)
            mfspec[i] = log(mfspec[i]);
        else
            mfspec[i] = -1.0e+5;
    }

    for (i = 0; i < FE->NUM_CEPSTRA; i++)
    {
        mfcep[i] = 0;

        beta = 0.5;

        // performance: 2 cycle/iteration. Total of (13*40*2) = 1040 cycles
        #pragma MUST_ITERATE(20, , 2);
        for (j = 0; j < FE->MEL_FB->num_filters; j++)
        {
            mfcep[i] += beta*mfspec[j]*FE->MEL_FB->mel_cosine[i][j];
            beta = 1.0;
        }
        mfcep[i] /= (float)period;
    }
}

/* fe_frame_to_fea: Convert frames to features. */
void fe_frame_to_fea(fe_t *FE, float *in, float *fea)
{
    float *spec, *mfspec;

    switch (FE->FB_TYPE) {
        case MEL_SCALE: {
       spec = (float *) MEM_calloc (L2SRAM, sizeof(float)*FE->FFT_SIZE, 0);
            if (spec == MEM_ILLEGAL)
                LOG_printf (&trace, "Failed to calloc memory for: spec");
        mfspec = (float *) MEM_calloc (L2SRAM,
sizeof(float)*FE->MEL_FB->num_filters, 0);
            if (mfspec == MEM_ILLEGAL)
                LOG_printf (&trace, "Failed to calloc memory for: mfspec");

        fe_spec_magnitude(in, FE->FRAME_SIZE, spec, FE->FFT_SIZE);
        fe_mel_spec(FE, spec, mfspec);
        fe_mel_cep(FE, mfspec, fea);

        MEM_free (L2SRAM, spec, sizeof(float)*FE->FFT_SIZE);
        MEM_free (L2SRAM, mfspec, sizeof(float)*FE->MEL_FB->num_filters);
      } break;

      default:
        LOG_printf (&trace, "Error converting frame to features: invalid
filtering type\n");
  }
}

// fe_process_utt: process the given int16 speech data and set
// cep_block to point to the beginning of a newly-allocated (fe_*_2d)
// array of arrays of floats.  Returns the number of frames processed.
// RUM-TIME MEMORY REQ: 21,328 bytes
// temp_spch (frame_size + buf_size) x4 = 9640
// cep (# cepstra x frame_cnt x 4) = 676
// spbuf = 9320 bytes
// fr_data = 1640
// fr_fea = 52
Int fe_process_utt(fe_t *FE, Int *spch, Int nsamps, float ***cep_block)
{
    Int frame_start, frame_count = 0, whichframe = 0;
    Int i, spbuf_len, offset = 0;
    float *spbuf;
    float *fr_data, *fr_fea;
    Int *tmp_spch = spch;
    float **cep = NULL;

    // are there enough samples to make at least 1 frame?
    if (nsamps+FE->NUM_OVERFLOW_SAMPS >= FE->FRAME_SIZE)
    {

        // if there are previous samples, pre-pend them to input speech samps
        if ((FE->NUM_OVERFLOW_SAMPS > 0))
        {
            tmp_spch = (Int *) MEM_alloc (L2SRAM,
sizeof(Int)*(FE->NUM_OVERFLOW_SAMPS + nsamps), 0);
            if (tmp_spch == MEM_ILLEGAL)
                LOG_printf (&trace, "Failed to calloc memory for: tmp_spch");

            memcpy(&tmp_spch[0], FE->OVERFLOW_SAMPS, FE->NUM_OVERFLOW_SAMPS
* sizeof(Int));
            memcpy(&tmp_spch[FE->NUM_OVERFLOW_SAMPS], spch, nsamps *
sizeof(Int));
            nsamps += FE->NUM_OVERFLOW_SAMPS;
            FE->NUM_OVERFLOW_SAMPS = 0; // reset overflow samps count
        }

        // compute how many complete frames can be processed and which samples
correspond to those samps
        frame_count = 0;
        for (frame_start=0; frame_start + FE->FRAME_SIZE <= nsamps; frame_start
+= FE->FRAME_SHIFT)
            frame_count++;

        // 01.14.01 RAH, added +1 Adding one gives us space to stick the last
flushed buffer
        // assume allocation errors always die -- DPW
        cep = (float **) xmalloc_2d (L2SRAM, frame_count+1, FE->NUM_CEPSTRA,
sizeof(float));

        spbuf_len = (frame_count-1)*FE->FRAME_SHIFT + FE->FRAME_SIZE;

        spbuf = (float *) MEM_alloc (L2SRAM, sizeof(float)*spbuf_len, 0);
        if (spbuf == MEM_ILLEGAL)
            LOG_printf (&trace, "Failed to calloc memory for: spbuf");
        // pre-emphasis if needed, and convert from int16 to float
        if (FE->PRE_EMPHASIS_ALPHA != 0.0)
            fe_pre_emphasis(tmp_spch, spbuf, spbuf_len,
FE->PRE_EMPHASIS_ALPHA, FE->PRIOR);
        else
            fe_short_to_float(tmp_spch, spbuf, spbuf_len);

        // frame based processing - let's make some cepstra...
        fr_data = (float *) MEM_calloc (L2SRAM, sizeof(float)*FE->FRAME_SIZE,
0);
        if (fr_data == MEM_ILLEGAL)
            LOG_printf (&trace, "Failed to calloc memory for: fr_data");

        fr_fea = (float *) MEM_calloc (L2SRAM, sizeof(float)*FE->NUM_CEPSTRA,
0);
        if (fr_fea == MEM_ILLEGAL)
            LOG_printf (&trace, "Failed to calloc memory for: fr_fea");

        for (whichframe = 0; whichframe < frame_count; whichframe++)
        {
            for (i = 0; i < FE->FRAME_SIZE; i++)
                fr_data[i] = spbuf[whichframe*FE->FRAME_SHIFT + i];

            fe_multiply_window(fr_data, FE->HAMMING_WINDOW, FE->FRAME_SIZE);
            fe_frame_to_fea(FE, fr_data, fr_fea);

            for (i = 0; i < FE->NUM_CEPSTRA; i++)
                cep[whichframe][i] = (float)fr_fea[i];
        }
        // done making cepstra

        // assign samples which don't fill an entire frame to FE overflow buffer
for use on next pass
        if (spbuf_len < nsamps)
        {
            offset = frame_count * FE->FRAME_SHIFT;
            memcpy(FE->OVERFLOW_SAMPS, &tmp_spch[offset], (nsamps-offset) *
sizeof(Int));
```

```
            FE->NUM_OVERFLOW_SAMPS = nsamps-offset;
            FE->PRIOR = tmp_spch[offset-1];
            xassert(FE->NUM_OVERFLOW_SAMPS < FE->FRAME_SIZE);
        }

        if (spch != tmp_spch)
            MEM_free (L2SRAM, tmp_spch, sizeof(Int)*(FE->NUM_OVERFLOW_SAMPS +
nsamps));

        MEM_free (L2SRAM, spbuf, sizeof(float)*spbuf_len);
        MEM_free (L2SRAM, fr_data, sizeof(float)*FE->FRAME_SIZE);
        MEM_free (L2SRAM, fr_fea, sizeof(float)*FE->NUM_CEPSTRA);

    } else {   // if not enough total samps for a single frame, append new samps
to
             //   previously stored overlap samples

        memcpy(&FE->OVERFLOW_SAMPS[FE->NUM_OVERFLOW_SAMPS], tmp_spch, nsamps
* sizeof(Short));
        FE->NUM_OVERFLOW_SAMPS += nsamps;

        xassert(FE->NUM_OVERFLOW_SAMPS < FE->FRAME_SIZE);
        frame_count = 0;
    }

    *cep_block = cep;

    return frame_count;
}

/* cmn_prior: do CMN stuff. */
Void cmn_prior(float **incep, Int varnorm, Int nfr, Int ceplen, Int endutt)
{
    static float *cur_mean = NULL; /* the mean subtracted from input frames */
    static float *sum = NULL;        /* the sum over input frames */
    static Int   nframe;     /* the total number of input frames */
    static Int   initialize = 1;
    float sf;
    Int   i, j;

    if (varnorm)
        LOG_printf(&trace, "Variance normalization not implemented in live mode
decode\n");

    if (initialize){
        cur_mean = (float *)MEM_calloc (L2SRAM, ceplen*sizeof(float), 0);
   if (cur_mean == MEM_ILLEGAL)
            LOG_printf (&trace, "Failed to calloc memory for: cur_mean");

        /* A front-end dependent magic number */
        cur_mean[0] = 12.0;

        sum     = (float *)MEM_calloc (L2SRAM, ceplen*sizeof(float), 0);
        nframe   = 0;
        initialize = 0;
    }

    if (nfr <= 0)
        return;

    for (i = 0; i < nfr; i++)
  {
        for (j = 0; j < ceplen; j++)
        {
            sum[j] += incep[i][j];
            incep[i][j] -= cur_mean[j];
        }
        ++nframe;
    }

    /* Shift buffer down if we have more than CMN_WIN_HWM frames */
    if (nframe > CMN_WIN_HWM) {
        sf = (float) (1.0/nframe);
        for (i = 0; i < ceplen; i++)
            cur_mean[i] = sum[i] * sf;
        /* Make the accumulation decay exponentially */
        if (nframe >= CMN_WIN_HWM) {
            sf = CMN_WIN * sf;
            for (i = 0; i < ceplen; i++)
                sum[i] *= sf;
            nframe = CMN_WIN;
        }
    }

    if (endutt) {
        /* Update mean buffer */

        sf = (float) (1.0/nframe);
        for (i = 0; i < ceplen; i++)
            cur_mean[i] = sum[i] * sf;

        /* Make the accumulation decay exponentially */
        if (nframe > CMN_WIN_HWM) {
            sf = CMN_WIN * sf;
            for (i = 0; i < ceplen; i++)
                sum[i] *= sf;
            nframe = CMN_WIN;
        }
    }
}

/* feat_s3mfc2feat_block: use the feature control block in fcb to convert
  nfr cepstrum frames from uttcep into a newly allocated array of feature
  vectors stored into *ofeat, with beginutt indicating beginning of
  utterance and endutt indicating end of utterance. */
Int feat_s2mfc2feat_block(feat_t *fcb, float **uttcep, Int nfr, Int beginutt,
Int endutt, float ***ofeat)
{
    static float    **feat = NULL;
    static float    **cepbuf = NULL;
    static Int      bufpos;
    static Int      curpos;
    static Int      jp1, jp2, jp3, jf1, jf2, jf3;
    Int  win, cepsize;
    Int  i, j, nfeatvec, residualvecs;

    float *w, *_w, *f;
    float *w1, *w_1, *_w1, *_w_1;
    float d1, d2;

    xassert(nfr < LIVEBUFBLOCKSIZE);
    win = feat_window_size(fcb);

    if (fcb->cepsize <= 0)
        LOG_printf (&trace, "Bad cepsize: %d\n", fcb->cepsize);

    cepsize = feat_cepsize(fcb);

    if (feat == NULL)
        feat = (float **)xcalloc_2d(L2SRAM, LIVEBUFBLOCKSIZE,
feat_stream_len(fcb,0), sizeof(float));

    if (cepbuf == NULL){
        cepbuf = (float **)xcalloc_2d(L2SRAM, LIVEBUFBLOCKSIZE, cepsize,
sizeof(float));
        beginutt = 1;
    }

    if (fcb->cmn)
    cmn_prior (uttcep, fcb->varnorm, nfr, fcb->cepsize, endutt);

    residualvecs = 0;
    if (beginutt) {
    for (i=0;i<win;i++)
        memcpy(cepbuf[i],uttcep[0],cepsize*sizeof(float));
        bufpos = win;
        bufpos %= LIVEBUFBLOCKSIZE;
        curpos = bufpos;
        jp1 = curpos - 1;
        jp1 %= LIVEBUFBLOCKSIZE;
        jp2 = curpos - 2;
        jp2 %= LIVEBUFBLOCKSIZE;
        jp3 = curpos - 3;
        jp3 %= LIVEBUFBLOCKSIZE;
        jf1 = curpos + 1;
        jf2 %= LIVEBUFBLOCKSIZE;
        jf1 %= LIVEBUFBLOCKSIZE;
        jf2 = curpos + 2;
        jf3 = curpos + 3;
        jf3 %= LIVEBUFBLOCKSIZE;
        residualvecs -= win;
    }

    for (i=0; i<nfr; i++)
    {
        xassert(bufpos < LIVEBUFBLOCKSIZE);
        memcpy(cepbuf[bufpos++],uttcep[i],cepsize*sizeof(float));
        bufpos %= LIVEBUFBLOCKSIZE;
    }

    if (endutt){
        if (nfr > 0) {
            for (i=0;i<win;i++) {
                xassert(bufpos < LIVEBUFBLOCKSIZE);

    memcpy(cepbuf[bufpos++],uttcep[nfr-1],cepsize*sizeof(float));
                bufpos %= LIVEBUFBLOCKSIZE;
            }
        }
        else {
            Short tpos = bufpos-1;
            tpos %= LIVEBUFBLOCKSIZE;
            for (i=0; i<win; i++)
            {
                xassert(bufpos < LIVEBUFBLOCKSIZE);

    memcpy(cepbuf[bufpos++],cepbuf[tpos],cepsize*sizeof(float));
                bufpos %= LIVEBUFBLOCKSIZE;
            }
        }
        residualvecs += win;
    }

    nfeatvec = 0;
    nfr += residualvecs;

    for (i = 0; i < nfr; i++,nfeatvec++)
    {
        memcpy (feat[i], cepbuf[curpos], (cepsize) * sizeof(float));

        f = feat[i] + cepsize;
        w  = cepbuf[jf2];
        _w = cepbuf[jp2];

        for (j = 0; j < cepsize; j++)
        {
            f[j] = w[j] - _w[j];
        }

        f += cepsize;

        w1   = cepbuf[jf3];
        _w1  = cepbuf[jp1];
        w_1  = cepbuf[jf1];
        _w_1 = cepbuf[jp3];

        for (j = 0; j < cepsize; j++)
        {
            d1 =  w1[j] -  _w1[j];
            d2 = w_1[j] - _w_1[j];

            f[j] = d1 - d2;
        }
        jf1++; jf2++; jf3++;
        jp1++; jp2++; jp3++;
        curpos++;
```

```
        jf1 %= LIVEBUFBLOCKSIZE;
        jf2 %= LIVEBUFBLOCKSIZE;
        jf3 %= LIVEBUFBLOCKSIZE;
        jp1 %= LIVEBUFBLOCKSIZE;
        jp2 %= LIVEBUFBLOCKSIZE;
        jp3 %= LIVEBUFBLOCKSIZE;
        curpos %= LIVEBUFBLOCKSIZE;
    }

    *ofeat = feat;

    return(nfeatvec);
}


#include <std.h>
#include <stdlib.h>
#include <math.h>

#include <log.h>
#include <mem.h>

#include <csl.h>
#include <csl_cache.h>

#include <csl_dat.h>
#include <csl_edma.h>

#include "asr_utility.c"
#include "asr_fe.c"

extern far  LOG_Obj trace;
extern Int  INRAM;
extern Int  L2SRAM;
extern Int  EXRAM;
extern Int  EXRAM1;

/*
```

```
// ----- GLOBAL CONSTANTS: FEATURE EXTRACTION -----//
#define FE_BUF_SIZE     ((Int)2000)

// ----- GLOBAL VARIABLES: FEATURE EXTRACTION -----//
Int     *all_buf;
Int     *fe_buf;
param_t params;
fe_t        *fe;
feat_t  *fcb;
float       **fe_feat;

// ----- GLOBAL CONSTANTS: ACOUSTIC MODELING -----//
#define NUM_SENONE      ((Int)1935)
#define NUM_COMP            ((Int)8)
#define NUM_GAUS            ((Int)39)
#define ALL_GAUS            ((Int)603720)
#define ALL_GAUS_PAD    ((Int)604032)
#define ALL_COMP            ((Int)15480)
#define ALL_COMP_PAD    ((Int)15488)

#define L2_BUF_SIZE ((Int)3872)     // 15488/4 = 3872 (which must also
divisible by 4)
#define COMP_ITER_DIV   ((Int)4)            // 15488/3982 = 4
#define L2_PAD_SIZE ((Int)224)      // padding added so buffer align at boundry

#define DIST_FLOOR          ((float)    -281861.7158)
#define S3_LOGPROB_ZERO ((Int)  0xc8000000)
#define SCALING_FACTOR      ((float) 3333.280269)
#define ADD_TBL_SIZE        ((Int)  29350)

// ----- GLOBAL VARIABLES: ACOUSTIC MODELING -----//
float       *am_mean;
float       *am_var;
float       *am_lrd;
float       *am_ksum;
Int     *am_mixw;
float       *am_feat;
Int     *am_add_tbl;
```

```
Short       *am_log_tbl;
Int     *am_senscr;

float       *l2_mbuf0;
float       *l2_pad0;
float       *l2_mbuf1;
float       *l2_pad1;
float       *l2_vbuf0;
float       *l2_pad2;
float       *l2_vbuf1;
*/
// ----- GLOBAL CONSTANTS: PHONE MODELING ----- //
#define SENONE_FRAMES       ((Int) 110) // 79-casulty, 117-capacity,
110-california
#define VALID_OFFSET        ((Int) -200000)
#define EXIT_OFFSET         ((Int) -100000)
#define HMM_BEAM            ((Int) -307006 + VALID_OFFSET)
#define PHONE_BEAM          ((Int) -230254 + EXIT_OFFSET)

// ----- GLOBAL VARIABLES: PHONE MODELING ----- //
Int     **ph_nodescr;
Int     **ph_mdef;
Int     **ph_tmat;
Int     *ph_pal;
Int     *ph_ntype;
Int     **ph_senscr;

// WORD CONSTANT
#define WORD_OFFSET         ((Int) -5000)
#define NUM_SENONE          1978
#define NUM_NODE            6306
#define NUM_TMAT            45

Int     word_threshold      = NEG_INFINITY;
Int     active_node_cnt = 6306;
Int     EXIT_HMM_cnt        = 0;
```

```
//********************************submit_qdma***************************
******
// Submit a QDMA request to transfer the data.
//**********************************************************************
******
Void submit_qdma(Uns src, Uns dst, Uns ele)
{
    EDMA_Config config;
        config.opt = (Uns)
        ((EDMA_OPT_PRI_HIGH << _EDMA_OPT_PRI_SHIFT )
        | (EDMA_OPT_ESIZE_32BIT << _EDMA_OPT_ESIZE_SHIFT )
        | (EDMA_OPT_2DS_NO << _EDMA_OPT_2DS_SHIFT )
        | (EDMA_OPT_SUM_INC << _EDMA_OPT_SUM_SHIFT )
        | (EDMA_OPT_2DD_NO << _EDMA_OPT_2DD_SHIFT )
        | (EDMA_OPT_DUM_INC << _EDMA_OPT_DUM_SHIFT )
        | (EDMA_OPT_TCINT_YES << _EDMA_OPT_TCINT_SHIFT )
        | (EDMA_OPT_TCC_OF(0) << _EDMA_OPT_TCC_SHIFT )
        | (EDMA_OPT_LINK_NO << _EDMA_OPT_LINK_SHIFT )
        | (EDMA_OPT_FS_YES << _EDMA_OPT_FS_SHIFT ));

        config.src = (Uns)src;  // source address
        config.cnt = (Uns)ele;  // element count
        config.dst = (Uns)dst;  // destination address
        config.idx = (Uns)0;        // submit request
        EDMA_qdmaConfig(&config);

    return;
}

//***********************************wait*******************************
******
// Wait until the transfer completes, as indicated by the Interrupt register
//**********************************************************************
******
Void wait()
{
    //while (!(EDMA_getPriQStatus() & EDMA_OPT_PRI_HIGH));
    while (!EDMA_RGET(CIPR));

    IRQ_clear(IRQ_EVT_EDMAINT);
    EDMA_resetAll();

    return;
}

//******************************asr_init_app*****************************
******
// Initiailizes application
//***********************************************************************
******
Void asr_init_app()
{
    CSL_init();         // initial Chip Support Library

    CACHE_enableCaching(CACHE_CE00);    // SDRAM cacheable
    CACHE_setL2Mode(CACHE_0KCACHE); // make L2 64KB SDRAM
}
/*
//******************************asr_init_fe******************************
******
// Initiailizes feature extraction block
//***********************************************************************
******
Void asr_init_fe()
{
    obtain_fe_params (&params);
    fe = fe_init(&params);
    fcb = feat_init();
    fe_start_utt(fe);

    fe_buf      = (Int  *) MEM_calloc (L2SRAM, sizeof(Int)*FE_BUF_SIZE, 0);
    all_buf     = (Int  *) MEM_calloc (EXRAM, sizeof(Int)*21992, 0);

    return;
}
```

```
Void asr_init_am()
{
    Int i;

    am_mean     = (float *) MEM_calloc (EXRAM, ALL_GAUS_PAD, 0);
    am_var      = (float *) MEM_calloc (EXRAM, ALL_GAUS_PAD, 0);
    am_lrd      = (float *) MEM_calloc (EXRAM, ALL_COMP_PAD, 0);
    am_ksum     = (float *) MEM_calloc (INRAM, ALL_COMP_PAD, 0);
    am_mixw     = (Int  *) MEM_calloc (EXRAM, ALL_COMP, 0);
    am_feat     = (float *) MEM_calloc (EXRAM, NUM_GAUS, 0);
    am_add_tbl  = (Int  *) MEM_calloc (EXRAM, ADD_TBL_SIZE, 0);
    am_log_tbl  = (Short    *)  MEM_calloc (INRAM, ADD_TBL_SIZE, 0);
    am_senscr   = (Int  *) MEM_calloc (INRAM, NUM_SENONE, 0);

    l2_mbuf0        = (float    *) MEM_calloc (L2SRAM, L2_BUF_SIZE, 0);
    l2_pad0     = (float    *) MEM_calloc (L2SRAM, L2_PAD_SIZE, 0);
    l2_mbuf1        = (float    *) MEM_calloc (L2SRAM, L2_BUF_SIZE, 0);
    l2_pad1     = (float    *) MEM_calloc (L2SRAM, L2_PAD_SIZE, 0);
    l2_vbuf0        = (float    *) MEM_calloc (L2SRAM, L2_BUF_SIZE, 0);
    l2_pad2     = (float    *) MEM_calloc (L2SRAM, L2_PAD_SIZE, 0);
    l2_mbuf1        = (float    *) MEM_calloc (L2SRAM, L2_BUF_SIZE, 0);

    // copy Int-type log table to Short-type table
    for (i = 0; i < ADD_TBL_SIZE; i++)
        am_log_tbl[i] = (Short) am_add_tbl[i];

    // copy lrd onto temp array
    submit_qdma((Uns) am_lrd, (Uns) am_ksum, ALL_COMP_PAD);
    wait();

    // make the first transfer so AM is ready to go
    submit_qdma((Uns) am_mean, (Uns) l2_mbuf0, L2_BUF_SIZE);
    wait();
    submit_qdma((Uns) am_var, (Uns) l2_vbuf0, L2_BUF_SIZE);
    wait();

    return;
```

```
}
*/

Void asr_init_ph()
{
    Int i, j;

    printmem (INRAM);
    printmem (L2SRAM);
    printmem (EXRAM);
    printmem (EXRAM1);

    ph_nodescr = (Int **) xcalloc_2d (EXRAM, NUM_NODE, 5, sizeof(Int));
    ph_mdef = (Int **) xcalloc_2d (EXRAM, NUM_NODE, 4, sizeof(Int));
    ph_tmat = (Int **) xcalloc_2d (EXRAM, NUM_TMAT, 6, sizeof(Int));
    ph_pal = (Int *) MEM_calloc (EXRAM, NUM_NODE*sizeof(Int), 0);
    ph_ntype = (Int *) MEM_calloc (EXRAM, NUM_NODE*sizeof(Int), 0);
    ph_senscr = (Int **) xcalloc_2d (EXRAM1, SENONE_FRAMES, NUM_SENONE,
sizeof(Int));

    printmem (INRAM);
    printmem (L2SRAM);
    printmem (EXRAM);
    printmem (EXRAM1);

    // initial nodescr table to all NEG_INFINITY
    for (i = 0; i < NUM_NODE; i++) {
        for (j = 0; j < 5; j++) {
            ph_nodescr[i][j] = NEG_INFINITY;
        }
    }

    // make all starting nodes active to begin
    for (i = 0; i < NUM_NODE; i++)
    {
        if ( ph_ntype[i] == 3 )
        {
            ph_pal[i] = 1;
            ph_nodescr[i][0] = -74100;
        }

        if ( ph_ntype[i] == 1 )
        {
            ph_pal[i] = 1;
            ph_nodescr[i][0] = -74100;
        }
    }

    return;
}
/*
Void asr_fe()
{
    float       **features;
    float       **cepstra;
    Int     ncepstra;
    Int     nfeat;
    static const Int maxframes_feat = 128;
    Int     beginutt = 1;
    Int     endutt = 0;
    Int     i, j;
    Int     count = 0;

    while (count < 20000)
    {
        submit_qdma ((Uns) all_buf+count, (Uns) fe_buf, 2000);
        wait();
        count = count + 2000;

        ncepstra = fe_process_utt(fe, fe_buf, FE_BUF_SIZE, &cepstra);


        for (i = 0; i < ncepstra; i += maxframes_feat)
        {
            nfeat = feat_s2mfc2feat_block(fcb, &cepstra[i], MIN(ncepstra-i,
maxframes_feat), beginutt, endutt, &features);
```

```
        }

        beginutt = 0;

        for (i = 0; i < nfeat; i++)
        {
            asr_am(i);
        }
    }

    for (j = 0; j < 1992; j++, count++)
        fe_buf[j] = all_buf[count];

    endutt = 1;

    ncepstra = fe_process_utt(fe, fe_buf, (Int)1992, &cepstra);

    for (i = 0; i < ncepstra; i += maxframes_feat)
    {
        nfeat = feat_s2mfc2feat_block(fcb, &cepstra[i], MIN(ncepstra-i,
maxframes_feat), beginutt, endutt, &features);
    }
}

//**********************************am_dist_eval**************************
******
// DESCRIPTION: Perform Gaussian evaluation n times
// OUTPUT: modified compscr (running sum of component scores
// NOTE: n must be divisible by 4 and the reminder must be even
//**********************************************************************
******
Void am_dist_eval (float f, float * restrict m, float * restrict v,
                         float * restrict compscr, Int n)
{
    Int i;
    float diff;

    WORD_ALIGNED(m);
    WORD_ALIGNED(v);
    WORD_ALIGNED(compscr);

    #pragma MUST_ITERATE (120, , 4);
    for (i = 0; i < n; i++)
    {
        diff = f - m[i];
        compscr[i] -= diff * diff * v[i];
    }

    return;
}

//********************************am_log_add****************************
******
// DESCRIPTION: Performs addition in the log domain
// OUTPUT: a set of senone scores
//*********************************************************************
******
Void am_log_add (float * restrict lrd, Int * restrict mw, Int * restrict senscr)
{
    Int i, j, k, l;
    Int d, r, logq;
    Int score;

    k = 0;
    for (i = 0; i < NUM_SENONE; i++)
    {
        score = S3_LOGPROB_ZERO;

        for (j = 0; j < NUM_COMP; j++)
        {
            l = lrd[k];
            if (l < DIST_FLOOR)
                l = DIST_FLOOR;

            logq = (Int)(SCALING_FACTOR * l) + mw[k];
            if (score > logq) {
                d = score - logq;
                r = score;
            } else {
                d = logq - score;
                r = logq;
            }

            if ((Int)d < ADD_TBL_SIZE)
            {
                r += am_log_tbl[d];
            }
            score = r;

            k++;
        }

        senscr[i] = score;
    }

    return;
}

//***********************************asr_am*****************************
******
// Note: 8x39 additional eval are inserted so that the loop count for dist_eval
//       remains to be divisiable by 4 (loop unroll) and the reminder is
//           divisible by 2 (packed data)
// Note: 1st set of the ping-pong buffer already has the required data!
//*********************************************************************
******
Void asr_am(float * restrict x)
{
    Int i, j;
    Int mv_addr_cnt = 1;
    Int mv_addr_offset = 0;
    Int lrd_cnt = 0;
    Int max_scr = 0;
    for (i = 0; i < NUM_GAUS; i++)
    {
        lrd_cnt = 0;
        for (j = 0; j < COMP_ITER_DIV; j=j+2)
        {
            mv_addr_offset = mv_addr_cnt * L2_BUF_SIZE;
            mv_addr_cnt++;
            // first get the next L2_BUF_SIZE elements Into the 2nd set of the
ping-pong buffer
            submit_qdma((Uns) (am_mean+mv_addr_offset), (Uns) l2_mbuf1,
L2_BUF_SIZE);
            submit_qdma((Uns) (am_var+mv_addr_offset), (Uns) l2_vbuf1,
L2_BUF_SIZE);

            // calculate the data availabled from the 1st set of the ping-pong
buffer
            am_dist_eval (x[i], l2_mbuf0, l2_vbuf0, &am_ksum[j*L2_BUF_SIZE],
L2_BUF_SIZE);
            lrd_cnt += L2_BUF_SIZE;

            if (mv_addr_cnt == (NUM_GAUS*COMP_ITER_DIV))
            {
                mv_addr_offset = 0;
            } else {
                mv_addr_offset = mv_addr_cnt * L2_BUF_SIZE;
                mv_addr_cnt++;
            }
            // then get the next L2_BUF_SIZE elements back to the 1st set of
the ping-pong buffer
            submit_qdma((Uns) (am_mean+mv_addr_offset), (Uns) l2_mbuf0,
L2_BUF_SIZE);
            submit_qdma((Uns) (am_var+mv_addr_offset), (Uns) l2_vbuf0,
L2_BUF_SIZE);

            // calculate data from 2nd set of the buffer
            am_dist_eval (x[i], l2_mbuf1, l2_vbuf1,
&am_ksum[(j+1)*L2_BUF_SIZE], L2_BUF_SIZE);
```

```
            lrd_cnt += L2_BUF_SIZE;
        }
    }

    am_log_add (am_ksum, am_mixw, am_senscr);

    // fill temp lrd with new lrd data from external ram
    submit_qdma((Uns) am_lrd, (Uns) am_ksum, ALL_COMP_PAD);

    // Find max senone scores
    max_scr = am_senscr[0];
    for (i = 1; i < NUM_SENONE; i++)
    {
        if (am_senscr[i] > max_scr)
            max_scr = am_senscr[i];
    }

    // Normalize all senone scores
    for (i = 0; i < NUM_SENONE; i++)
    {
        am_senscr[i] = am_senscr[i] - max_scr;
    }

    return;
}
*/

Void asr_word ()
{
    Int i, j;
    Int WORD_BEAM = word_threshold - 153503 + WORD_OFFSET;
    Int reset_all_start_node = 0;

    if (EXIT_HMM_cnt > 0)
    {
        // go through all exit nodes and active new nodes (end of word and neow)
        for (i = 0; i < NUM_NODE; i++)
        {
            if (ph_pal[i] == 2) // exiting node
            {
                // deactivate the current exiting node
                //ph_pal[i] = 0;

                if ( ph_ntype[i] >= 2 ) // eow type node
                {   // since we assume any word can go into any other word
so...lot of comparisons here
                    if (ph_nodescr[i][4] >= WORD_BEAM)
                    {
                        reset_all_start_node = 1;
                        for (j = 0; j < NUM_NODE; j++)
                        {   // go find all starting nodes
                            if ( ph_ntype[j] == 1 )
                            {   // activate starting node and give it new score
if necessary
                                //ph_pal[j] = 1;
                                if (ph_nodescr[j][0] < ph_nodescr[i][4])
                                    ph_nodescr[j][0] = ph_nodescr[i][4];
                            }
                            if ( ph_ntype[j] == 3 )
                            {   // activate starting node and give it new score
if necessary
                                //ph_pal[j] = 1;
                                if (ph_nodescr[j][0] < ph_nodescr[i][4])
                                    ph_nodescr[j][0] = ph_nodescr[i][4];
                            }
                        }
                        LOG_printf (&trace, "find a word %d", i);
                    }
                }
                else    // neow type node
                {
                    // active the next node id with new score
                    ph_pal[i+1] = 1;
                    // if input score of the branch node is lesser than output
score of current node
                    if (ph_nodescr[i+1][0] < ph_nodescr[i][4])
```

```
                        ph_nodescr[i+1][0] = ph_nodescr[i][4];
                    }
                }
            }
        }

        EXIT_HMM_cnt = 0;

        for (i = 0; i < NUM_NODE; i++)
        {
            if (ph_pal[i] == 2)
                ph_pal[i] = 1;

            if (ph_ntype[i] == 1)
            {
                ph_pal[i] = 1;
            }
            if (ph_ntype[i] == 3)
            {
                ph_pal[i] = 1;
            }
        }

        return;
    }

    Int HMM_eva (Int nid, Int *senscr)
    {
        Int tmat, sid0, sid1, sid2;
        Int senscr0, senscr1, senscr2;
        Int statescr0, statescr1, statescr2, stateout;
        Int max;

        tmat = ph_mdef[nid][0];
        sid0 = ph_mdef[nid][1];
        sid1 = ph_mdef[nid][2];
        sid2 = ph_mdef[nid][3];
        senscr0 = senscr[sid0];     // get first senone score
        senscr1 = senscr[sid1];     // get second senone score
        senscr2 = senscr[sid2];     // get third senone score

        statescr0 = MAX(ph_nodescr[nid][0] + 0, ph_nodescr[nid][1] +
    ph_tmat[tmat][0]) + senscr0;
        statescr1 = MAX(ph_nodescr[nid][1] + ph_tmat[tmat][1], ph_nodescr[nid][2]
    + ph_tmat[tmat][2]) + senscr1;
        statescr2 = MAX(ph_nodescr[nid][2] + ph_tmat[tmat][2], ph_nodescr[nid][3]
    + ph_tmat[tmat][4]) + senscr2;
        stateout     = statescr2 + ph_tmat[tmat][5];

        // reset input score to NEG_INFINITY
        ph_nodescr[nid][0] = NEG_INFINITY;

        // put new scores back to the ph_table (score0, score1, score2, scoreExit)
        ph_nodescr[nid][1] = statescr0;
        ph_nodescr[nid][2] = statescr1;
        ph_nodescr[nid][3] = statescr2;
        ph_nodescr[nid][4] = stateout;

        // find max score
        max = statescr0;
        if (statescr1 > max)
            max = statescr1;
        if (statescr2 > max)
            max = statescr2;
        if (stateout > max)
            max = stateout;

        return max;
    }


    Void asr_ph ()
    {
        Int i, j, z;
        Int max, scr;
        Int B_HMM, B_HMM_prune_scr;
        Int EXIT_HMM;

        for (z = 0; z < SENONE_FRAMES; z++)
        {
            B_HMM = NEG_INFINITY;

            for (i = 0; i < NUM_NODE; i++)
            {
                if (ph_pal[i] == 1)
                    scr = HMM_eva (i, ph_senscr[z]);

                // keep track of the max score among all active node
                if (scr > B_HMM)
                    B_HMM = scr;
            }

            // prune node with max scr < B_HMM+HMM_BEAM
            B_HMM_prune_scr = B_HMM + HMM_BEAM;
            EXIT_HMM = B_HMM + PHONE_BEAM;
            for (i = 0; i < NUM_NODE; i++)
            {
                if (ph_pal[i] == 1)
                {
                    // find exit node: if output score > EXIT_HMM
                    if (ph_nodescr[i][4] >= EXIT_HMM)
                    {
                        ph_pal[i] = 2;
                        EXIT_HMM_cnt++;
                    }

                    // determine word threshold: best stateout score among active
    HMM
                    if ( (ph_ntype[i] == 2) || (ph_ntype[i] == 3) )
                    {
                        if (ph_nodescr[i][4] > word_threshold)
                            word_threshold = ph_nodescr[i][4];
                    }
                    // find local max
                    max = ph_nodescr[i][1];
                    for (j = 2; j < 5; j++)
                    {
                        if (ph_nodescr[i][j] > max)
                            max = ph_nodescr[i][j];
                    }

                    // prune, reset all score: if local max < B_HMM_prune_scr
                    if (max < B_HMM_prune_scr)
                    {
                        ph_pal[i] = 0;
                        ph_nodescr[i][1] = NEG_INFINITY;
                        ph_nodescr[i][2] = NEG_INFINITY;
                        ph_nodescr[i][3] = NEG_INFINITY;
                        ph_nodescr[i][4] = NEG_INFINITY;
                    }
                }
            }

            asr_word();
        }

        return;
    }

    Void main()
    {
        asr_init_app();

        //asr_init_fe();

        //asr_fe();

        //asr_init_am();

        //asr_am(fe_feat);
        asr_init_ph();

        asr_ph();

        return;
    }



    extern far  LOG_Obj trace;
    extern Int  INRAM;
    extern Int  L2SRAM;
    extern Int  EXRAM;
    extern Int  EXRAM1;

    #define WORD_ALIGNED(x) (_nassert(((Int)(x) & 0x7) == 0))

    static Void printmem (Int segid)
    {
        MEM_Stat    statbuf;
        MEM_stat (segid, &statbuf);
        LOG_printf (&trace, "seg %d: 0x%x", segid, statbuf.size);
        LOG_printf (&trace, "\tU 0x%x\tA 0x%x", statbuf.used, statbuf.length);
    }

    static Void **xmalloc_2d(Int segid, Int d1, Int d2, Int elem_size)
    {
        Void *store, **out;
        Int i;

        Uns size = d1 * d2 * elem_size;

        store = MEM_alloc (segid, size, 0);
        if (store == MEM_ILLEGAL)
            LOG_printf (&trace, "Failed to malloc memory for: store!\n");

        out = MEM_alloc (segid, d1*sizeof(Void *), 0);
        if (out == MEM_ILLEGAL)
```

```
        LOG_printf (&trace, "Failed to malloc memory for: out!\n");

    for (i = 0; i < d1; i++)
        out[i] = (Void *)((Char *)store + d2*elem_size*i);

    return out;
}

static Void **xcalloc_2d (Int segid, Int d1, Int d2, Int elem_size)
{
    Void *store, **out;
    Int i;

    store = MEM_calloc(segid, d1*d2*elem_size, 0);
    if (store == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: store!\n");

    out = MEM_calloc(segid, d1*sizeof(Void *), 0);
    if (out == MEM_ILLEGAL)
        LOG_printf (&trace, "Failed to calloc memory for: out!\n");

    for (i = 0; i < d1; i++)
        out[i] = (Void *)((Char *)store + d2*elem_size*i);

    return out;
}
```

BIBLIOGRAPHY

[1] K. Gupta, “Design and Implementation of a Co-Processor For Embedded, Real-Time, Speaker-Independent, Continuous Speech Recognition System-On-A-Chip,” Master Thesis, University of Pittsburgh, 2005.

[2] Medium Vocabulary Test Result, CMU Sphinx, http://cmusphinx.sourceforge.net/MediumVocabResults.html

[3] K. Lee, H. Hon, and R. Reddy, "An overview of the SPHINX speech recognition system," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 38, No. 1, January, 1990, pp. 35 - 45.

[5] Baum, L. E., “An Inequality and Associated Maximization Technique in Statistical Estimation of Probabilistic Functions of Markov Processes,” *Inequalities*, Vol. 3 (1972), pp. 1–8.

[6] Patti Price, William M. Fisher, Jared Bernstein, and David S. Pallett, "The DARPA 1000-word resource management database for continuous speech recognition," *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing (ICASSP)*, New York, USA, Apr. 1988, vol. 1, pp. 651--654.

[7] Ravishankar, M., Singh, R., Raj, B. and Stean, R., “The 1999 CMU 10x Real Time Broadcast news Transcription System,” *Proc. DARPA Workshop on Automatic Transcription of Broadcast News*, Washington DC, May 2000.

[8] M. Y. Hwang, "Subphonetic Acoustic Modeling for Speaker Independent Continuous Speech Recognition," PhD Thesis, Carnegie Mellon University, 1993, CMU-CS-93-230.

[9] Fisher, J. A., Ellis, J. R., Ruttenberg. J. C. and Nieolau. A., “Parallel Processing: A Smart Compiler and a Dumb Machine,” *Proc. ACM SIGPLAN „84 Symp. on Compiler Construction*, Montreal. Canada, June, 1984. pp. 37-47.

[10] Binu Mathew, Al Davis and Zhen Fang, “A Low-Power Accelerator for the SPHINX 3 Speech Recognition System,” *International Conference on Compilers, Architecture and Synthesis for Embedded Systems,* San Jose, CA, 2003.

[11] Bryan Pellom, "SONIC: The University of Colorado Continuous Speech Recognizer,” University of Colorado, tech report #TR-CSLR-2001-01, Boulder, Colorado, March, 2001.

[12] ViaVoice, IBM Inc., http://www-306.ibm.com/software/voice/viavoice/.

[13] Dragon Naturally Speaking, Nuance Inc., http://www.nuance.com/naturallyspeaking/

[15] K. Agaram, S.W. Keckler, and D. Burger, “A Characterization of Speech Recognition on Modern Computer Systems,” *Proceedings of the 4th IEEE Workshop on Workload Characterization*, Dec. 2001.

[16] Hagen A., Connors D.A., and Pellom R.L., "The Analysis and Design of Architecture Systems for Speech Recognition on Modern Handheld-Computing Devices," *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, October 1-3, 2003

[17] Rabiner L. R., Schafer R. W., Digital Processing of Speech Signals. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[18] Davis SB, Mermelstein P., “Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences,” *IEEE Trans ASSP,* Vol 28, No 4, pp357-366, 1980.

[19] Stevens, S.S. and J. Volkman, “The Relation of Pitch to Frequency,” *Journal of Psychology*, 1940, 53, pp.329.

[20] Michael Seltzer, “SPHINX III Signal Processing Front End Specification,” CMU Speech Group, 1999

[21] Rabiner, L.R., “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” *Proc. IEEE*, 77, No.2, 1989, p.257-286.

[22] X. Huang, A. Acero, H.W. Hon, Spoken Language Processing, New Jersey: Prentice Hall, 2001.

[23] Bocchieri, E., “A Study of Beam Search Algorithm for Large Vocabulary Continuous Speech Recognition and Methods of Improved Efficiency,” *Proc. Of Eurospeech*, p1521-1524, Berlin, 1993.

[24] TMS320C6000 CPU and Instruction Set Reference Guide

[25] TMS320C6000 Optimizing Compiler User’s Guide

[26] TMS320C67x DSP Library Programmer's Reference Guide

[27] TMS320C6713 DSK Technical Reference

[28] M. S. Lam., “Software Pipelining: An Effective Scheduling Technique for VLIW Machines,” *In Conference on Programming Language Design and Implementation*, pages 318--328. ACM SIGPLAN, 1988.