Running Linux on a DSP? Exploiting the Computational Resources of a programmable DSP Micro-Processor with uClinux

Michael Durrant *Jeff Dionne* *Michael Leslie*

1 Introduction

Many software developers in recent years have turned to Linux as their operating system of choice. Until the advent of uClinux, however, developers of smaller embedded systems, usually incorporating microprocessors with no MMU, (Memory Management Unit) could not take advantage of Linux in their designs. uClinux is a variant of mainstream Linux that runs on "MMU-less" processor architectures. Perhaps a DSP? If a general purpose DSP has enough of a useable instruction set why not!

Component costs are of primary concern in embedded systems, which are typically required to be small and inexpensive. Microprocessors with on-chip MMU hardware tend to be complex and expensive, and as such are not typically selected for small, simple embedded systems, which do not require them.

Benefits

Using Linux in devices, which require some intelligence, is attractive for many reasons:

- It is a mature, robust operating system
- It already supports a large number of devices, filesystems, and networking protocols
- Bug fixes and new features are constantly being added, tested and refined by a large community of programmers and users
- It gives everyone from developers to end users complete visibility of the source code
- A large number of applications (such as GNU software) exist which require little to no porting effort
- Linux's very low cost

2 uClinux System Configurations

Embedded systems running uClinux may be configured in many ways other than that of the familiar UNIX-like Linux distribution. Nevertheless, an example of a system running uClinux in this way will help to illustrate how it may be used.

Kernel / Root Filesystem

The Arcturus uCdimm is a complete computer in an so-DIMM form factor, built around either a Motorola 68VZ328 "DragonBall" microcontroller, the latest processor in a family widely popularized by the "Palm Pilot" or a Motorola CF5272 "ColdFire" microcontroller. It is equipped with 2M of flash memory, 8M of SDRAM, both synchronous and asynchronous serial ports, and an Ethernet controller. There

is a custom resident boot monitor on the device, which is capable of downloading a binary image to flash memory and executing it. The image that is downloaded consists of a uClinux kernel and root filesystem. In UNIX terms, the kernel makes a block device out of the memory range where the root filesystem resides, and mounts this device as root. The root filesystem is in a read-only UNIX-like format called "ROMFS". Since the DragonBall runs at 32MHz, the kernel and optionally user programs execute in-place in flash memory. Faster systems like the MCF5272 ColdFire running at 48 MHz or 66 MHz benefit from copying the kernel and root filesystem into RAM and executing there.

The Micro Signal Architecture MSA developed by ADI (and Intel) is very interesting. While a DSP processor, it possesses enough general-purpose processor instruction support to allow operating systems like uClinux to be deployed. The ADI BlackFin is one such processor.

Other embedded systems may be inherently network-based, so a kernel in flash memory might mount a root filesystem being served via nfs (Network File System). An even more network-centric device might request its kernel image and root filesystems via dhcp (Dynamic Host Configuration Protocol) and bootp. Note that drivers for things like IDE and SCSI disk, CD, and floppy support are all still present in the uClinux kernel.

User Space

The contents of the root filesystem vary more dramatically between embedded systems using uClinux than between Linux workstations.

The uClinux distribution contains a root filesystem which implements a small UNIX-like server, with a console on the serial port, a telnet daemon, a web server, nfs (Network File System) client support, and a selection of common UNIX tools.

A system such as an mp3 (MPEG layer 3 compressed audio) CD player might not even have a console. The kernel might contain only support for a CD drive, parallel I/O, and an audio DAC. User space might consist only of an interface program to drive buttons and LEDs, to control the CD, and which could invoke one other program; an MPEG audio player. Such an application specific system would obviously require much less memory than the full-fledged uClinux distribution as it is shipped.

3 Development Under uClinux

Development Tools

Developing software for uClinux systems typically involves a cross-compiler toolchain built from the familiar GNU compiler tools. Software that builds under gcc (GNU C Compiler) for x86 architectures, for example, often builds without modification on any uClinux target.

Debugging a target via gdb (GNU debugger) presents a debugging interface common to all the platforms supported by gdb. The debugging interface to a uClinux kernel on a target depends on debugging support for that target. If the target processor has hardware support for debugging, such as IEEE's JTAG or Motorola's BDM, gdb may connect non-intrusively to the target to debug the kernel. If the processor lacks such support, a gdb "stub" may be incorporated into the kernel. gdb communicates with the stub via a serial port, or via Ethernet.

uClibc

uClibc, the C library used in uClinux, is a smaller implementation than those which ship

with most modern Linux distributions. The library has been designed to provide most of the calls that UNIX-like C programs will use. If an application requires a feature that is not implemented in uClibc, the feature may be added to uClibc, it may me linked in as a separate library, or it may be added to the application itself.

Differences Between uClinux And Linux

Considering that the absence of MMU support in uClinux constitutes a fundamental difference from mainstream Linux, surprisingly little kernel and user space software is affected. Developers familiar with Linux will notice little difference working under uClinux. Embedded systems developers will already be familiar with some of the issues peculiar to uClinux.

Two differences between mainstream Linux and uClinux are a consequence of the removal of MMU support from uClinux. The lack of both memory protection and of a virtual memory model are of importance to a developer working in either kernel or user space. Certain system calls to the kernel are also affected.

Memory Protection

One consequence of operating without memory protection is that an invalid pointer reference by even an unprivileged process may trigger an address error, and potentially corrupt or even shut down the system. Obviously code running on such a system must be programmed carefully and tested diligently to ensure robustness and security.

Virtual Memory

There are three primary consequences of running Linux without virtual memory. One is that processes, which are loaded by the kernel, must be able to run independently of their position in memory. One way to achieve this is to "fix up" address references in a program once it is loaded into RAM. The other is to generate code that uses only relative addressing (referred to as PIC, or Position Independent Code). uClinux supports both of these methods.

Another consequence is that memory allocation and deallocation occurs within a flat memory model. Very dynamic memory allocation can result in fragmentation, which can starve the system. One way to improve the robustness of applications that perform dynamic memory allocation is to replace *malloc*() calls with requests from a preallocated buffer pool.

Since virtual memory is not used in uClinux, swapping pages in and out of memory is not implemented, since it cannot be guaranteed that the pages would be loaded to the same location in RAM. In embedded systems it is also unlikely that it would be acceptable to suspend an application in order to use more RAM than is physically available.

System Calls

The lack of memory management hardware on uClinux target processors has meant that some changes needed to be made to the Linux system interface. Perhaps the greatest difference is the absence of the *fork()* and *brk()* system calls.

A call to *fork()* clones a process to create a child. Under Linux, *fork()* is implemented using copy-on-write pages. Without an MMU, uClinux cannot completely and reliably clone a process, nor does it have access to copy-on-write.

uClinux implements *vfork()* in order to compensate for the lack of *fork()*. When a parent process calls *vfork()* to create a child, both processes share all their memory space including the stack. *vfork()* then suspends the parent's ex-

ecution until the child process either calls *exit()* or *execve()*. Note that multitasking is not otherwise affected. It does, however, mean that older-style network daemons that make extensive use of *fork()* must be modified. Since child processes run in the same address space as their parents, the behaviour of both processes may require modification in particular situations.

Many modern programs rely on child processes to perform basic tasks, allowing the system to maintain an interactive "feel" even if the processing load is quite heavy. Such programs may require substantial reworking to perform the same task under uClinux. If a key application depends heavily on such structuring, then it may be necessary to either re-create the application, or an MMU-enabled processor may also be needed.

A hypothetical, simple network daemon, *hyped*, will illustrate the use of *fork()*. *hyped* always listens on a well-known network port (or socket) for connections from a network client. When the client connects, *hyped* gives it new connection information (a new socket number) and calls *fork()*. The child process then accepts the client's reconnection to the new socket, freeing the parent to listen for new connections.

uClinux has neither an autogrow stack nor *brk()* and so user space programs must use the *mmap()* command to allocate memory. For convenience, our C library implements *malloc()* as a wrapper to *mmap()*. There is a compile-time option to set the stack size of a program.

4 Brief Anatomy Of The uClinux Kernel

This section describes the changes that were made to the Linux kernel to allow it to run on MMU-less processors.

Architecture-Generic Kernel Changes

The architecture-generic memory management subsystem was modified to remove reliance on MMU hardware by providing basic memory management functions within the kernel software itself. For those who are familiar with uClinux, this is the role of the directory */mm-nommu* derived from and replacing the directory */mm*. Several subsystems needed to be modified, added, removed, or rewritten. Kernel and user memory allocation and deallocation routines had to be reimplemented. Support for transparent swapping / paging was removed. Program loaders which support PIC (Position Independent Code) were added. A new binary object code format, named "flat" was created, which supports PIC and which has a very compact header. Other program loaders, such as that for ELF, were modified to support other formats which, instead of using PIC, use absolute references which it is the responsibility of the kernel to "fix up" at run time. Each method has advantages and disadvantages. Traditional PIC is quick and compact but has a size restriction on some architectures. For example, the 16-bit relative jump in Motorola 68k architectures limits PIC programs to 32K. The run-time fix-up technique removes this size restriction, but incurs overhead when the program is loaded by the kernel.

Porting uClinux To New Platforms

The task of adding support for a new CPU architecture in uClinux is similar to doing so in Linux proper. Fortunately, there is a great deal of code in Linux that can be ported with minor adaptations and reused in uClinux. Machine dependent startup code and header files already exist in Linux for MMU versions of processors in the ARM, Motorola 68k, MIPS,

SPARC and other families. This code may be adapted to support non-MMU versions of these processors in uClinux.

Driver code, which already exists in Linux, is often easily portable to run under uClinux. Issues in porting such code may involve endian issues or memory handling code, which assumes the presence of MMU support.

5 The Future of uClinux

Numerous enhancements are in the works for uClinux. The diversity of the innovations that mainstream Linux receives from the community pave a good path for the development of uClinux. The uClinux developer community is very active; enhancements and innovations are frequently made.

Real-Time

Linux is now a platform for hard real-time application development (that is, applications with deterministic latency under varying processor loads). The Linux kernel scheduler already provides non-deterministic, or "soft", real-time, and systems such as RT-Linux and RTAI (Real-Time Application Interface) upgrade the Linux kernel to provide hard (deterministic) real-time support. Real-time applications in Linux have access to the extensive resources of the Linux kernel without sacrificing hard real-time performance. Efforts are underway to provide the RTAI subsystem for use on various MMU-less processors.

Adding DSP support or adding general purpose processor support to a DSP

Processors like the ColdFire MCF5272 are primarily general purpose processors with a RSIC instruction set. Yet Motorola included limited DSP functionality with a multiply and accumulate DSP functions. This is an approach of adding functionality into general purpose processors. In this case the ColdFire processor can and does have uClinux support. Other processors like the Analog Devices BlackFin (MSA DSP), the processor is primarily a DSP with added general purpose processor support. uClinux is portable to the BlackFin and expected to be publicly available in the fall of 2002.

The attached paper *"Exploiting the Computational Resources of a Programmable DSP Micro-processor (Micro Signal Architecture MSA) in the field of Multiple Target Tracking" (Hussain et al 2001)*, is a technical representation of the computational requirements for a multiple target acquisition system. Such a system would require a DSP and would greatly benefit from a UNIX like operating system afforded by uClinux. Commercial uses for such a system would include traditional sonar/radar devices allowing for affordable collision detection systems, for robots, and automobiles.

uClinux 2.4

uClinux 2.4, with support for Motorola DragonBall and ColdFire, was released in January of 2001. New ports, including MIPS, Hitachi SH2, ARM, and SPARC, will be made to the uClinux 2.4 tree, which is based on Linux 2.4. but enhancements are also still being made to the uClinux 2.0 tree. uClinux 2.4 will give developers access to many of the new features added to Linux since 2.0, including support for USB, IEEE Firewire, IrDA, and new networking features such as bandwidth allocation, (a.k.a. QoS: Quality of Service) IP Tables, and IPv6.

Since uClinux is Open Source, development effort spent on uClinux will never be lost. Engineering professionals world-wide, are using uClinux to create commercial products and a

significant portion of their work is contributed back to the open source community.

****

References:

“Running Linux on low cost, low power, MMU-less processors”, Michael Durrant, Arcturus Networks Inc.

“Building Low Cost, Embedded, Network Appliance with Linux”, Greg Ungerer, SnapGear Inc.

“Embedded Coldfire-Taking Linux On-Board”, Nigel Dick, Motorola Ltd

“When hard real-time goes soft”, D. Jeff Dionne, Arcturus Networks Inc.

Real-Time Application Interface (RTAI): **http://www.rtai.org**

The uClinux project: **http://www.uClinux.org**

EXPLOITING THE COMPUTATIONAL RESOURCES OF A PROGRAMMABLE DSP MICRO-PROCESSOR (Micro Signal Architecture MSA) IN THE FIELD OF MULTIPLE TARGET TRACKING

Principal Researcher: Dr. D.M. Akbar Hussain
Contributions: Michael Durrant *michael.durrant@ArcturusNetworks.com*
Jeff Dionne *jeff.dionne@ArcturusNetworks.com*

Arcturus Networks Inc. 195 The West Mall, Suite 608, M9C 5K1, Toronto CANADA
Tel: 416 621 0125 Fax: 416 621 0190

Abstract: During the last few decades the improved technology available for surveillance systems has generated a great deal of interest in algorithms capable of tracking large number of objects.. Typical sensor systems, such as radar or sonar using information from one or more sensors can obtain noisy information data returns from true targets and other possible objects. The tracking problem requires the processing of this data to produce accurate estimates of the position and velocity of the targets. There are two types of uncertainties involved with the measurement data, first the position inaccuracy, as the measurements are corrupted by noise, and second the measurement origin since there may be uncertainty as to which measurement originates from which target. These uncertainties lead to a data association problem and the tracking performance depends not only on the measurement noise but also upon the uncertainty in the measurement origin. Therefore, in a multiple target environment extensive computation may be required to establish the correspondence between measurements and tracks every radar scan.

It is also true that tremendous advancement has also been made in the computational capabilities of a processing unit to deal with such demanding tasks. In this paper we present a simulated study of implementing a recursive multiple target tracking (MTT) algorithm using a track splitting filter, study uses a MSA processor from Analog Devices Inc. (ADI) which is a programmable Digital Signal Processor (DSP) with the added functionality to realize many of the programming advancements more normally associated with Micro-controllers. In addition, the study also explores the porting and support of an embedded real time operating system to such an architecture.

Keywords: DSP, RTAI, Kalman Filter, Target Tracking, State Estimation, uClinux.

1. INTRODUCTION

In the ideal situation of tracking a single target, where one noisy measurement is obtained, standard Kalman filter technique can be used at each radar scan In the multi-target case, an unknown number of measurements are received at each radar scan and, assuming no false measurements, each measurement has to be associated with an existing or new target tracking filter. When the targets are well apart from each other forming a measurement prediction ellipse around a track to associate the measurement with the appropriate track is a standard technique [1]. When targets are near to each other, more than one measurement may fall within the prediction ellipse of a filter and prediction

ellipses of different filters may interact. The number of measurements accepted by a filter will therefore be quite sensitive in this situation to the accuracy of the prediction ellipse. Several approaches may be used for this situation [2, 3]. One such approach is called the Track Splitting Filter algorithm. In this algorithm, if *n* measurements occur inside a prediction ellipse, then the filter branches or splits into *n* tracking filters.

This situation, which results in an increased number of filters, requires more processing power and in some cases the system may saturate. A mechanism for restricting excess tracks splitting is required, since eventually this process may result in more than one filter tracking the same target. The first criterion is the support function, which uses the likelihood function of a track as the pruning criterion. The second, similarity criterion, which uses a distance threshold to prune similar filter tracking the same target [4]. The flow chart shown in Fig. 1 depicts the actual processing sequence of a recursive MTT algorithm.

2. TARGET MOTION MODEL AND STATE ESTIMATION

The motion of a target being tracked is assumed to be approximately linear and modeled by the following equations

$$\underline{x}_{n+1} = \Phi \underline{x}_n + \Gamma \underline{\omega}_n \tag{1}$$

$$\underline{z}_{n+1} = H \underline{x}_{n+1} + \underline{\nu}_{n+1} \tag{2}$$

where the state vector

$$\underline{x}_{n+1}^T = (x\ x^\bullet\ y\ y^\bullet)_{n+1} \tag{3}$$

is a four dimensional vector, $\underline{\omega}_n$ is the two dimensional disturbance vector, $\underline{z}_{n+1}$ is the two dimensional measurement vector and $\underline{\nu}_{n+1}$ is the two dimensional measurement error vector. Also Φ is the assumed (4x4) state transition matrix, Γ is the excitation matrix (4x2) and H is the measurement matrix (2x4) and are defined respectively by,

$$\Phi = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$\Gamma = \begin{bmatrix} \Delta t^2/2 & 0 \\ \Delta t & 0 \\ 0 & \Delta t^2/2 \\ 0 & \Delta t \end{bmatrix} \tag{5}$$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6}$$

Here Δt is the sampling interval and corresponds to the time interval (scan interval) assumed constant, at which radar measurement data is received. The system noise sequence $\underline{\omega}_n$ is a two dimensional Gaussian sequence for which

$$\mathrm{E}(\underline{\omega}_n) = 0 \tag{7}$$

where E is the expectation operator. The covariance of $\underline{\omega}_n$ is

$$\mathrm{E}(\underline{\omega}_n \underline{\omega}_n^T) = Q_n \delta_{nm} \tag{8}$$

where Q_n is a positive semi-definite (2x2) diagonal matrix and δ_{nm} is the Kronecker delta defined as

$$\delta_{nm} = \begin{matrix} 0 & n \neq m \\ 1 & n = m \end{matrix}$$

The measurement noise sequence $\underline{\nu}_n$ is a two dimensional zero mean Gaussian white sequence with a covariance of

$$\mathrm{E}(\underline{\nu}_n \ \underline{\nu}^T{}_n) = R_n \delta_{nm} \tag{9}$$

where R_n is a positive semi-definite symmetric (2x2) matrix given by

$$R_n = \begin{bmatrix} \sigma^2{}_x & \sigma_{xy} \\ \sigma_{xy} & \sigma^2{}_y \end{bmatrix} \tag{10}$$

σ_x^2 and σ_y^2 are the variances in the error of the x, y position measurements, and σ_{xy} is the covariance between the x and y measurements errors. It is assumed that the measurement noise sequence and the system noise sequence are independent of each other, that is

$$\mathrm{E}(\underline{\nu}_n \ \underline{\omega}^T{}_n) = 0 \tag{11}$$

The initial state $\underline{x}_0$ is also assumed independent of the $\underline{\nu}_n$ and $\underline{\omega}_n$ sequences that is

$$\mathrm{E}(\underline{x}_0 \ \underline{\omega}^T \, n) = 0 \tag{12}$$

$$\mathrm{E}(\underline{x}_0 \ \underline{\nu}^T \, n) = 0 \tag{13}$$

$\underline{x}_0$ is a four dimensional random vector with mean $\mathrm{E}(\underline{x}_0) = \underline{x}_{0/0}$ and a (4x4) positive semi-definite covariance matrix defined by

$$\mathbf{P}_0 = \mathrm{E}[(\underline{x}_0 - \underline{x}^-{}_0)(\underline{x}_0 - \underline{x}^-{}_0)^{\mathrm{T}}] \tag{14}$$

where $\underline{x}^-{}_0$ is the mean of the initial state $\underline{x}_0$. The Kalman filter is an optimal filter as it minimizes the mean squared error between the estimated state and the true state (actual) provided the target dynamics are correctly modeled.

The standard Kalman filter equations for estimating the position and velocity of the target motion described by eqns. (1) and (2) are

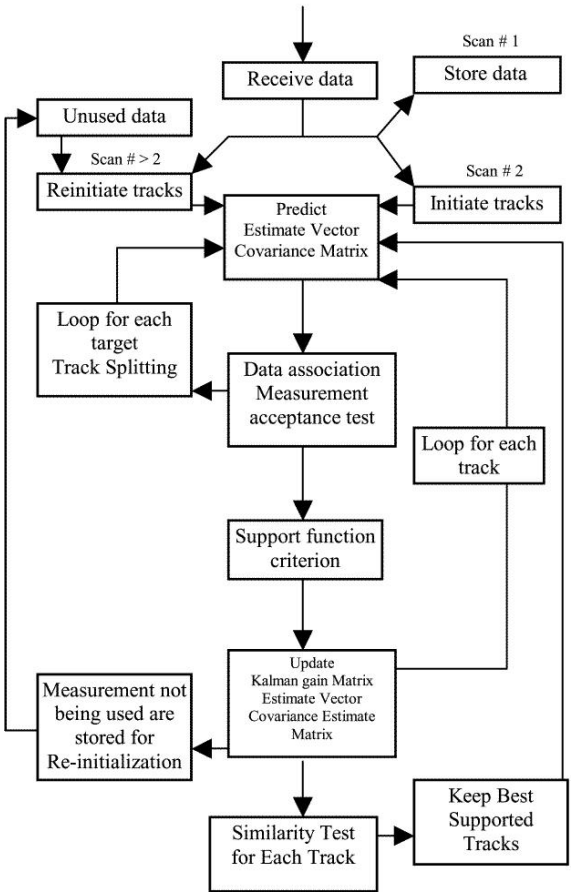

Figure 1: Recursive MTT

$$\underline{x}^{\wedge}_{n+1/n} = \Phi \underline{x}^{\wedge}{}_n \tag{15}$$

$$\underline{x}^{\wedge}_{n+1} = \underline{x}^{\wedge}_{n+1/n} + \mathbf{K}_{n+1} \, \underline{\nu}_{n+1} \tag{16}$$

$$\mathbf{K}_{n+1} = \mathbf{P}_{n+1/n} \mathbf{H}^{\mathbf{T}} \mathbf{B}^{-1}{}_{n+1} \tag{17}$$

$$\mathbf{P}_{n+1/n} = \Phi \mathbf{P}_n \Phi^{\mathbf{T}} + \Gamma \mathbf{Q}_n^{\mathbf{F}} \Gamma^{\mathbf{T}} \tag{18}$$

$$\mathbf{B}_{n+1} = \mathbf{R}_{n+1} + \mathbf{H} \mathbf{P}_{n+1/n} \mathbf{H}^{\mathbf{T}} \tag{19}$$

$$\mathbf{P}_{n+1} = (\mathbf{I} - \mathbf{K}_{n+1}\mathbf{H})\mathbf{P}_{n+1/n} \qquad (20)$$

$$\underline{\nu}_{n+1} = \underline{z}_{n+1} - \mathbf{H}\underline{x}^{\wedge}{}_{n+1/n} \qquad (21)$$

where $\underline{x}^{\wedge}_{n+1/n}$, $\underline{x}^{\wedge}_{n+1}$, $\mathbf{K}_{n+1}$, $\mathbf{P}_{n+1/n}$, $\mathbf{B}_{n+1}$, and $\mathbf{P}_{n+1}$ are the predicted state, estimated state, the Kalman gain matrix, the prediction covariance matrix, the covariance matrix of innovation, and the covariance matrix of estimation respectively. $\mathbf{Q}_n^{\mathbf{F}}$ is the covariance of the measurement noise assumed by the filter which is normally taken equal to $\mathbf{Q}_n$. In a practical situation, however, the value of this covariance is not known so the choice should be such that the filter can adequately track any possible motion of the target. To start the computation an initial value is chosen for $\mathbf{P}_0$. Even if this is a diagonal matrix, then clearly from the above equations the covariance matrices $\mathbf{B}_{n+1}$, $\mathbf{P}_{n+1}$ and $\mathbf{P}_{n+1/n}$ for a given n, do not remain diagonal when $\mathbf{R}_{n+1}$ is not diagonal.

3. MSA

The MSA processor has five independent, full functional computation units: Two Arithmetic/Logic Units (ALU), Two Multiplier/Accumulator Units and a Barrel Shifter, Fig. 2 shows the MSA. The processor units can process 8-bit, 16 bit, 32 bit and 40 bit data, depending upon the type of function being performed.

- ***Data Address Generator (DAG)***

The Micro Signal Architecture processor base is a dual data path, modified Harvard Architecture. The two DAG support the sophisticated operations required in DSP algorithms, such as reversed addressing, circular buffering. In addition, auto increment, auto decrement and base plus immediate offset addressing are possible. These units update dedicated register files, the DAG register file and the pointer register file. In general, DSP mathematical operations involving circular buffers uses the DAG register file. The DAG register file contains four sets of 32 bit Index, Length, Modify and Base registers, for a total of sixteen 32 bit registers. With these two independent DAGs, MSA can generate two 32 bit addresses in a single cycle, fetching or storing two 32 bit or four 16 bit operands. The pointer register file is used for more general operations. It has six general purpose 32 bit addressing registers and two dedicated 32 bit Stack Pointers for stack manipulation. The Micro Signal Architecture processor can access a unified 4 GB linear address space.

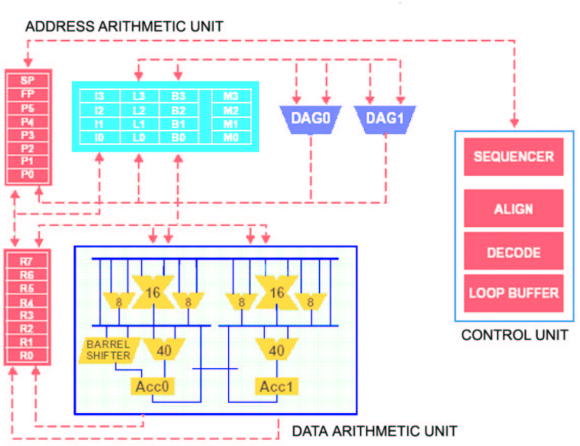

Figure 2: MSA

- ***Memory Management Unit (MMU)***

The memory Management unit (MMU) provides protection to individual tasks that may be operating on the Micro Signal Architecture Processor and may protect system Memory Mapped Registers from unintended access. The architecture includes two Memory Management Units: one for instruction memory and the other one for data memory. These MMUs control accesses to caches, on chip

SRAM and off chip memories. The Micro Signal Architecture Processor supports multiple pages of memory, in four page sizes: 1 K byte, 4 K byte, 1 M byte, 4 M byte. The instruction MMU may designate as many as sixteen distinct memory pages, each with a separate set of criteria governing its cache and protection properties. The Data MMU may designate a further sixteen memory pages.

The Micro Signal Architecture Processor uses a modified Harvard Architecture in combination with a hierarchical memory structure. Memory closest to the Core are referred to as level 1 (L1), and generally have a single cycle, zero latency access by the core. Other memories on chip are referred to as level 2 (L2) and may have multiple-cycle access or latency. Off-chip memories are usually seen at the same hierarchical level as the on chip L2 memory. At the L1 level, the instruction memory holds instructions only and the two data memories hold only data. At L2 level a single unified memory space exists, holding both instructions and data. Also L1 instruction memory and L1 data memories may be configured as either SRAM or caches. The Micro Signal Architecture Processor has a dedicated scratch data memory that is always configured as an additional L1 data SRAM. Scratchpad memory is designed to store stack and local variables.

- ***Program Sequencer***

The program sequencer is used to get instructions from the L1 instruction memory and determine if these instructions are 16 bit 'Control' instructions, 32 bit 'DSP' instructions or 64 bit 'DSP multi-function' instruction. The sequencer manages the control of data through the processor core, insuring that the pipe line is fully interlocked and that zero overhead looping is correctly managed.

- ***Event Controller, Timer and JTAG Interface***

The event controller supports nested and prioritized events. The controller has five basic types of events: Emulation, Reset, Non-maskable Interrupt (NMI), Exception and Interrupts. The programmable interval timer is used to generate periodic interrupts. The 8-bit pre-scale register can be used to set the number of cycles for decrementing a 32 bit counter register. The number of clock cycles per timer decrement may be one to 256. An interrupt is generated when this count register reaches zero. The register may be automatically reloaded from a 16 bit period register and the count resumed.

The JTAG interface provides the method by which the Micro Signal Architecture emulations interact with the processor core. The emulation unit contains an instruction register and data register that is accessed through the JTAG port. These two registers are used to control and interrogate the processor during emulation mode. Additionally the trace unit can be used to store the last 16 non-sequential PC values, which can be used to reconstruct the processor sequence.

- ***Performance Monitor Unit (PMU)***

During the operation of the Micro Signal Architecture Processor, the performance monitor unit can be used to review the efficiency of certain operations, e.g., cache misses, and provide the information for code optimization. The performance unit consists of six instruction address watch points and two data address watch points. These address watch points may be combined in pairs to create address range watch points, which additionally may be associated with various counters for evaluating the performance and profiling the processor code.

- ***Instruction Set***

The Micro Signal Architecture Processor assembly level instruction set employs an algebraic syntax, presenting code to the programmer that is very readable even at the assembly level. The instructions have been specifically tuned to provide a very flexible, yet densely encoded instruction set that will compile to a very small final memory size. The instruction set also provides fully featured multi-function instructions that allow the programmer to use any of the Micro Signal Architecture Processor core resources in a single instruction. Coupled with a variety of enhanced features more often seen on micro-controllers, this instruction set is very efficient when compiling code for C and C++ languages.

- ***Modes of Operation***

The Micro Signal Architecture Processor has five distinct modes of operations: User, Supervisor, Emulation, Idle and Reset. User mode has restricted access to certain system resources, thus providing a protected software environment. Supervisor and Emulation modes have unrestricted access to core resources. Idle and Reset modes prevent software execution, so resource access is not an issue.

4. DSP PROGRAMMING MODEL

The instruction set for the Micro Signal Architecture processor provides two types of instructions: one primarily for micro-controller and general tasks, and those used for DSP oriented computation. Specific Micro Signal Architecture instructions are tuned for their corresponding task, but instructions can easily be combined. Table 1 shows the resources available to the DSP processing. DSP instructions usually read two 32 bit operands from the register file, compute results and either store results back to the register file or accumulate them in the two accumulators as shown in Fig. 3.

Resources	**Description**
Data Execution Unit 0	• 16 x 16 bit MAC unit (MAC 0) • 40 bit accumulator (a0) • 40 bit shifter
Data Execution Unit 1	• 16 x 16 bit MAC unit (MAC 1) • 40 bit accumulator (a1)
Data Register File	8, 32 bit wide, accessible as register halves

Table 1: Execution Units and Register File

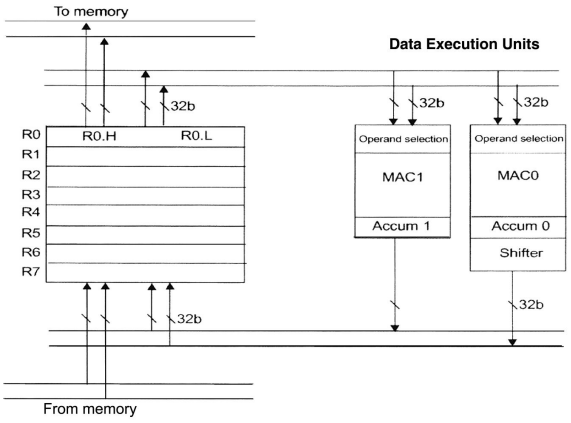

Figure 3: Register Files and Execution Unit

The register file delivers two 32 bit operands to MAC units and accepts two 32 bit results in return. In addition, the register file delivers two 32 bit values to the memory system or it receives two 32 bit values from memory. To perform multiplication, each MAC unit select two 16 bit operands from the two 32 bit words that it receives from the register file as shown in Fig. 4.

It also means that each MAC unit uses four possible combinations of input operands. Therefore, when both MAC units operate in parallel, sixteen possible combinations of 16

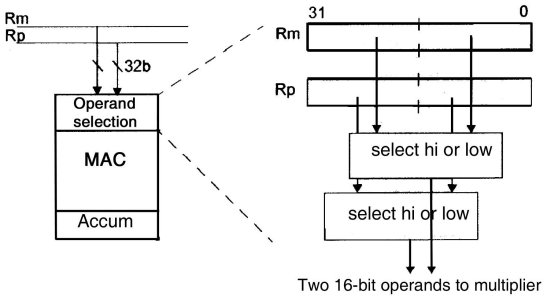

Figure 4: Operand Selection

bit input operand results. Multiply and accumulate operations can be performed on four combinations of input operands, as shown in Fig. 5, for MAC 0. Assembly instruction example of dual MAC operation.

```
A0 += R1.H*R2.L, A0
+=R1.L*R2.L;
```

In addition to performing a multiply accumulate, the multiplier results may optionally be written to the register file, independently of each other. In addition to multiply accumulate, in which the contents of the accumulator are effected, MSA also has multiply instructions which does not effect the contents of the accumulator.

The ALUs have a different mechanism than MACs, ALU 0 as the primary source of arithmetic and logic operation, it can perform the following operations:

- 32 bit operation on two 32 bit inputs producing one output.
- One 16 bit operation on two 16 bit inputs residing in arbitrary register halves producing one 16 bit output.
- Two 16 bit (dual) operation on four 16 bit inputs (in two registers). Dual 16 bit ALU operations can perform four combinations

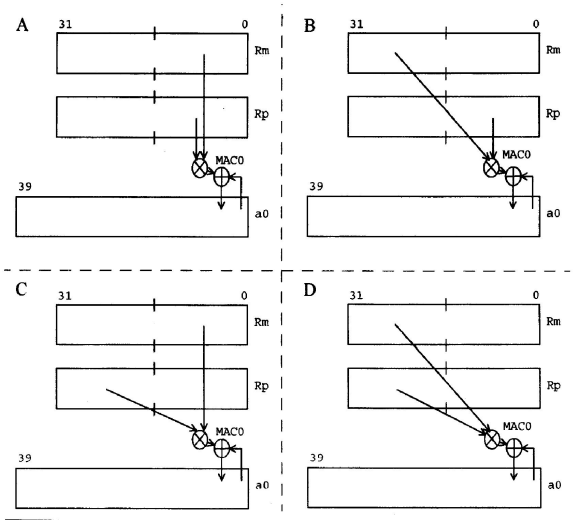

Figure 5: MAC 0 Possible Choices

of addition and subtraction on the input operands as shown in Fig. 6.

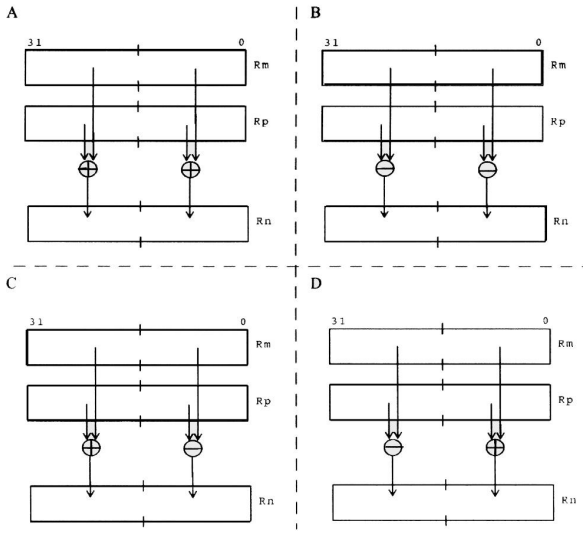

Figure 6: Dual 16-bit ALU Operations

The result from the upper halves are placed into the upper half of the destination register and lower halves to the lower half of the destination register. It also supports the cross option, in which case the order of the two 16 bit result is inverted. In addition to the operations supported on the primary computation unit ALU

0, a small number of instructions support the secondary arithmetic unit ALU 1 in parallel with ALU 0. The Micro Signal Architecture core does not support full dual ALU functionality, because the maximum number of input operands that may be transferred from the register file to the execution units is limited to two 32 bit operands. Therefore, parallel ALU operations on ALU 0 and ALU 1 can be performed only on the same two 32 bit words.

5. IMPLEMENTATION

For the implementation of the target tracking algorithm employing a track splitting filter, a sensor was simulated in two dimensions to generate data for different scenarios. The simulating program is capable of generating up to 20 targets in a predefined scan window, two versions of the program can be used: one, in which initial position, heading, noise and other parameters are taken as default, in the second case all this data can be entered by the programmer. The generated data basically provides the following input information corresponding to each individual target.

x	y	σ_x^2	σ_y^2	σ_{xy}	Flag	CID

Where x, y is the noisy position measurements, σ_x^2, σ_y^2 and σ_{xy} are the measurement noise covariance values, Flag is an index used to see if a measurement has been used by the filter and CID is another index color ID used for identification of each tracking filter/measurement. Basically, first five variables will be available in real time to the tracking algorithm through an interface, in an actual implementation. Here the input data is given to the algorithm through hand coded assembly instructions.

6. RESULT AND CONCLUSION

For the evaluation of our algorithm, a simulator for MSA hardware architecture was written as the actual silicon for MSA is expected in September, 2001. For our evaluation a two crossing target scenario was used, the sensor was moving parallel to the target, towards the targets at a very high speed compared with the targets. The two targets at start up were approximately 30 Km from the sensor and cross each other at 30^{th} scan. After successful initialization of two targets, tracking proceeded with each tracking filter accepting only one measurement as they are well apart. Track splitting (branching) starts at the 24^{th} scan, maximum number of tracks occur at 31^{st} scan. At 38^{th} scan all the redundant tracks are pruned and normal tracking of two targets resume. It should be noted here that lots of detail about tracking is not revealed [5][6] as the actual emphasis is about utilizing the architecture advantages of the processor.

The main concern was to implement the computationally expensive algorithm in such a way to take the advantages of the DSP instruction set for MSA also, one of the reason of selecting Blackfin was its DSP assembly language algebraic syntax which is probably ideal for such an application. As pointed out earlier, the algorithm is hand coded in assembly in order to exploit the DSP. Where ever necessary C is used for the actual implementation. As the architecture supports richly encoded instruction set for such a demanding task, in some cases multiple operations are performed with less number of cycles, this obviously improves real time processing. Here, we just present one particular sequence where the implementation helps in achieving our mission of exploiting its architecture. At each radar scan incoming measurements (returns) are to be tested with each established track (path) for a possible match. The following computation sequence that includes multiplication of three matrices of dimensions [(1x2)*(2x2)*(2x1)] is to be calculated and then compared with a known value to find out if a possible match exits. Therefore, each track should accept the true measurement

from the same target, although this situation changes at the time of multiple measurements existing in the same vicinity. In our case when the target cross each other. Now to evaluate this expression

$$\mathbf{d}^2{}_n = \underline{\nu}_n{}^{\mathbf{T}} \mathbf{B}^{-1}{}_n \underline{\nu}_n$$

suppose $\nu_n^{\mathbf{T}}$ of dimension (1x2) is stored in data register R0 such that:

$$\mathrm{R0.L} = \underline{\nu}_n{}^{\mathbf{T}}(0,0) \quad \mathrm{R0.H} = \underline{\nu}_n{}^{\mathbf{T}}(0,1)$$

(0, 0) represents first row, first column element of the matrix and so on. Similarly $\mathbf{B}_n^{-1}$ is stored in two registers as

$$\mathrm{R2.L} = \mathbf{B}_n^{-1}(0,0) \quad \mathrm{R2.H} = \mathbf{B}_n^{-1}(0,1)$$

$$\mathrm{R3.L} = \mathbf{B}_n^{-1}(1,0) \quad \mathrm{R3.H} = \mathbf{B}_n^{-1}(1,1)$$

By using MSA instruction set we can perform this task in just four steps:

$$\left.\begin{array}{l} \mathrm{A0} = \mathrm{R0.L} * \mathrm{R2.L} \\ \mathrm{A1} = \mathrm{R0.L} * \mathrm{R2.L} \end{array}\right] (\mathrm{Parallel\ \ Op})$$

$$\left.\begin{array}{l} \mathrm{R4.L} = \mathrm{A0+} = \mathrm{R0.H} * \mathrm{R3.L} \\ \mathrm{R4.H} = \mathrm{A1+} = \mathrm{R0.H} * \mathrm{R3.H} \end{array}\right] (\mathrm{Parallel\ \ Op})$$

$$\left.\begin{array}{l} \mathrm{A0} = \mathrm{R4.L} * \mathrm{R0.L} \\ \mathrm{A1} = \mathrm{R4.H} * \mathrm{R0.H} \end{array}\right] (\mathrm{Parallel\ \ Op})$$

$$\mathbf{d}^2{}_n = \mathrm{A0} + \mathrm{A1}$$

The above calculation is just a replica of calculations performed repeatedly in a recursive filter, which basically suite such DSP architecture. Because DSP processors are characterized by tight code loop and in-fact data flow driven. Therefore, this type of loop/calculations can be implemented very efficiently. Actually, the number of times $\mathbf{d}^2{}_n$ is evaluated to find a probable track-measurement pair increases exponentially with increasing number of track splitting. If it is possible to do such calculations quickly the efficiency will increase in terms of time. The simulated study carried out here has logically determined the advantage of using MSA core for such an application. The simulated study after running a number of scenarios and careful evaluation reveals that MSA may provide 20 to 30% improved performance in processing for such a computationally intensive application. Although, no bench mark evaluation criterion is used for such evaluation.

The selection of an operating system suitable for implementation in a target tracking system presents many challenges. The main challenge is selecting a processor and complimentary operating system able to handle the computationally expensive load. The combination of Blackfin architecture and Linux operating system meets this challenge with Linux Kernel. The second being the ability of the operating system to respond in a deterministic manner. This can be achieved by operating system that are designed to operate in Real Time. Linux was selected in this study as its characteristic performance achieves close to a predictable real time response under known loads. However, Linux on its own is not a suitable Real Time Operating System (RTOS) and some characterize Linux's response time as "Soft Real Time", the observed jitter is larger than the jitter associated with running Linux under the control of a real time scheduler such as found in "Real Time Executive in C for DSP

(RTXCDSP)" or the Real Time Application Interface (RTAI) [11].

Recently, efforts are underway to provide the RTAI subsystems for use on various MMU-less processors. If one is to develop an embedded system using such a DSP (MSA) for real time applications, uClinux with uClibc may provide an excellent platform for such real time target tracking implementation. The work is already underway for porting uClinux and uClibc for this processor (MSA).

7. REFERENCES

[1] P. L. Smith and G. Buechler, "A branching algorithm for discriminating and tracking multiple objects," IEEE Transactions Automatic Control, Vol. AC-20, February 1975, pp 101-104.

[2] Y. Bar-Shalom and T. E. Fortmann, "Tracking and Data Association", Academic Press, Inc. 1988.

[3] S. S. Blackman, " Multiple Target Tracking with Radar Applications", Artech House, Inc. 1986.

[4] D. P. Atherton, E. Gul, A. Kountzeris and M. Kharbouch, "Tracking Multiple Targets using parallel processing," Proc. IEE, Part D, No. 4, July 1990, pp 225-234.

[5] D. P. Atherton, D. M. A. Hussain and E. Gul, "Target tracking using transputers as parallel processors," 9th IFAC Symposium on Identification and System Parameter Estimation, Budapest, Hungary July 1991.

[6] M. Kharbouch, "Some investigations on target tracking," D. Phil thesis, Sussex University, 1991.

[7] The Carmel DSP core user's manual, infineon, 1999.

[8] Clifford Liem, Pierre Paulin, Ahmed Jerraya, "Address calculation for retargetable compilation and exploration of instruction set architecture."

[9] The scientist and engineer's guide to Digital Signal Processing.

[10] 1999, DSP architecture directory.

[11] Michael Durrant, Michael Leslie, Using Linux for MMU-less micro-processors. Electronics Engineering UK, Feb. 2001.

8. ACKNOWLEDGEMENTS

The author would like to thank Arcturus Networks Inc. and Lineo Canada Corp for their support in every aspect for doing such a study, and special thanks to Analog Devices Inc., for providing all the technical details, documents etc., about MSA which really encouraged the author to carry out such investigation. The author also likes to extend his appreciation and gratitude to his colleagues, having been very kind to extend their comments at difficult times. Also there is lot of literature available on the Internet about DSP, few of the references are given here for the acknowledgement of this wonderful technology feast, thanks Internet.

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*