Published at ICCD 1999, International Conference on Computer Design,
October 10-13, 1999, Austin Texas, pp. 593-598.

TriMedia CPU64 Application Development Environment

E.J.D. Pol, B.J.M. Aarts, J.T.J. van Eijndhoven, P. Struik, F.W. Sijstermans*,
M.J.A. Tromp*, J.W. van de Waerdt*, P. van der Wolf
Philips Research Laboratories Eindhoven, The Netherlands
**Philips Semiconductors, Sunnyvale, California*
evert-jan.pol@philips.com

Abstract

The architecture of the TriMedia CPU64 is based on the TM1000 DSPCPU. The original VLIW architecture has been extended with the concepts of vector processing and superoperations. The new vector operations and superoperations need to be supported by the compiler and simulator to make them accessible to application programmers. It was our intention to support these new features while remaining compliant with the ANSI C standard. This paper describes the mechanisms which were implemented to achieve this goal. Furthermore, the optimization of applications needs to address the vectorization of the functions to be implemented. Some general guidelines for producing efficient vectorized code are given.

1. Introduction

Media processors are a relatively new kind of processors. In many ways, they take some characteristics from general-purpose CPUs on the one hand, and from DSPs on the other. Usually, the architecture and instruction set of media processors are RISC-style, clearly following general-purpose CPU designs, and breaking away from the DSP style of incorporating irregular hardware and instruction set characteristics such as special-purpose registers and irregular connection networks. This regularity of the architecture of media processors allows them to be programmed efficiently in C, rather than in assembly.

Although the architecture of a media processor is quite regular, usually a host of special-purpose operations are supported in the instruction set. This aspect of media processors is more akin to the DSP-style of architecture. The special-purpose operations are designed to support specific processing that is common in the application domain of media processing, and are referred to as custom operations. The presence of custom operations is one of the factors that allow media processors to achieve a high performance level for media processing tasks. See [1] for a more detailed overview of general VLIW characteristics.

The TriMedia CPU64 was designed as a successor to the DSPCPU as implemented in the TM1000. The TM1000 contains many modules, such as input/output coprocessors, a variable-length decoder, an image coprocessor, and, most importantly, a 32 bit wide DSPCPU. This processor is called a DSPCPU to emphasize the fact that on the one hand the processor is not a general-purpose CPU, but designed to perform embedded signal processing, while on the other hand it does offer the high-level language programmability that is common for general-purpose CPUs. As is true for the TM1000, the target domain of applications is digital media processing, including tasks such as video and audio decoding, encoding and transcoding, and 3D graphics. References [2] and [3] describe in detail the application domain and architecture of the CPU64, while references [4] and [5] provide additional information on the TriMedia architecture.

The application development environment for the TriMedia line of processors includes an optimizing C compiler. This compiler supports many features that are generic to VLIW compilers. This paper will not elaborate on these generic VLIW compiler features, but will concentrate on the issues that we needed to deal with to support the specific architectural features of the CPU64. The reader is referred to [6] for information on the TriMedia compilation system and for further references on general VLIW compiler technology.

Section 2 explains some basic concepts that have been implemented in hardware and have been made available to the application programmer. Section 3 discusses a basic and central source of information for the compiler and simulator: the machine description file. Section 4 elaborates on the specific support in the CPU64 compiler for the new vector types and custom operations. Section 5 explains how the application programmer is supported in the issues related to vector programming. Section 6 shows how this new functionality is supported both in the CPU64 development environment and in standard ANSI C development

environments. Section 7 offers some details on the development environment that we use to simulate the CPU64. Section 8 presents guidelines on application program optimizations, specifically regarding vectorizing application code. Section 9 finishes the paper by summarizing the important issues that have been discussed.

2. Parallelism and instruction set

The basic architecture of the CPU64 is a VLIW machine, which implies that, like superscalar RISC processors, it can perform several operations in parallel, and that, unlike with superscalar RISC processors, these operations are scheduled at compile time. This type of parallelism is called instruction-level parallelism or Multiple Instruction/ Multiple Data (MIMD) parallelism. VLIW systems can exploit this type of parallelism well.

Many of the applications of the targeted domain also exhibit another type of parallelism, called data parallelism, or Single Instruction/Multiple Data (SIMD) parallelism. Media streams typically contain many instances of data that need similar processing, such as pixels in a stream of digital video. SIMD parallelism becomes increasingly important when the register width of the processor is increased. When expanding the register width from 32-bits registers in the TM1000 to 64-bits registers in the CPU64, it becomes more important to pack separate data elements into vectors of data to be processed simultaneously. Once the stream of data is available as vectors, it is necessary to have vector functions available that operate on these vectors of data. Thus, to efficiently exploit the availability of wide registers in the CPU64, the system needs to support vectors of various types plus a rich set of operations that process these vectors.

This support needs to start at the application programming level to unleash the full processing power of the CPU64 processor. The TriMedia method of application programming is a best-of-both-worlds approach to standard C and assembly programming. If the application programmer can only use standard C to implement the desired functionality, the compiler is burdened by finding a suitable vectorization, and recognizing the available hardware operations in the arithmetic expressions. These tasks are notoriously difficult, and lead (at the present level of compiler technology) to suboptimally generated code. Programming the functionality in assembly leads (in principle) to optimal code, but results in unreadable, weakly maintainable, and probably unreliable code, not to mention the severe headaches on the part of the application programmer.

The strengths of both the compiler and the application programmer can be combined by allowing the application programmer direct access to the hardware operations from C. The compiler then takes care of the correct translation of the program constructs, scoping rules for variables, stack handling, register allocation, and operation scheduling. Especially these last two issues are highly complex for VLIW processors. The application programmer can thus concentrate on choosing appropriate data representations, and calling the appropriate custom operations to accomplish the necessary calculations.

The set of operations that is available to the application programmer at the C level is somewhat different from the set of operations that is supported in hardware. The reason for this difference is that we want the C level programmer's interface to be both orthogonal and independent of the actual hardware-supported operations. The independence of the API and the hardware instruction set allows better portability of code over processors with changing instruction sets. The application-level instruction set is orthogonal in the sense that any regular operation is available for any vector type. The orthogonality of the API ensures that the application programmer does not need to consult the processor hardware manual to see if some specific combination of operation and type is supported. If the specific combination is not available as a single hardware operation, the compiler will translate the C-level custom operation into a sequence of operations supported in hardware that together implement the desired function.

3. Machine Description

The development of the CPU64 involved benchmark experiments with a large number of processor variations, as described in [7]. To support these benchmarking experiments, a retargetable compiler and simulator were created. These retargetable tools read in a machine description file at run time, so that they adapt their behavior to the specific instance of the processor to be tested. These processor instances were described in a format that was designed specifically for this purpose. The format allows the specification of machine parameters, including register files, functional units, and instruction set. The following will focus on the function units and instruction set.

Firstly, for each type of functional unit, the machine description file specifies how many times it is instantiated, in which issue slots it is available, and which operations the functional unit supports. Figure 1 shows an example declaration of some functional units. This piece of machine description file defines two types of functional units, one multiplier unit for byte vectors, and a similar one for byte vector superoperations. The regular multiplier is instantiated in slots 2 and 3, the two superunits occupy slots 1+2 and 3+4, respectively. Both types of units support operations with a latency of three clock cycles. These operations

are implemented for signed and unsigned byte vectors; the regular operations produce only the lower halves of the products, the superoperations produce the full double precision products.

```
FUNCTIONAL UNITS

    bmul
        SLOT    2 3
        LATENCY 3
        OPERATIONS
            mul_bs,mul_bu;

    super_bmul
        SLOT    1+2 3+4
        LATENCY 3
        OPERATIONS
            dblmul_bs,dblmul_bu;
```

Figure 1. Example machine description file declaration of functional units

Secondly, the machine description file specifies how custom operations available at the C level map onto operations supported in hardware. Figure 2 shows an example mapping of multiply operations available at the C level to multiply operations available at the hardware level. The `$x` variables indicate operation arguments and results.

```
MAPPING

    $0 sb_mul $1 $2 =
            $0 mul_bs $1 $2;

    $0 $1 sb_dblmul $2 $3 =
            $0 $1 dblmul_bs $2 $3;
```

Figure 2. Example machine description file mapping of custom operations

Thirdly, the machine description file also contains information on individual operations. For example, it describes per operation how many arguments and results the operation has. This level of flexibility is necessary to describe the so-called superoperations. These superoperations are implemented in functional units that straddle two (possibly even more) issue slots, and thus have access to the doubled (possibly even more) number of read and write ports on the register file. In this way, it is possible to implement operations that require more than two arguments and/or produce more than one result.

The exact specification in the machine description file of a custom (possibly super-) operation is described by its so-called signature. The signature details the number of arguments and results, and can also specify whether or not the operation has any side effects other than the results produced. Figure 3 shows an example signature of some multiply operations. This signature describes the `dblmul_bs` operation as having a guard, two arguments, and two results. The keyword `PURE` signifies that the operation has no side effects.

```
OPERATIONS

SIGNATURE (r:r,r->r,r) PURE
    dblmul_bs;
```

Figure 3. Example machine description file signature of a custom operation

4. Compiler

The TriMedia TM1000 compiler was extended to deal with the vectors and custom operations used in the processor. However, great care was taken to ensure that the application programs that actually use these vectors and custom operations can also be compiled and run in a standard application development environment.

The supported vector types include arrays of signed and unsigned integers of 8, 16, and 32 bits long, and 32 bit floating point vectors. These vectors are built into the CPU64 compiler, so that the types can be used directly in C. However, these vector types can also be defined as structures containing the appropriate integers for use by standard C compilers. Structures are needed, as they are used by value for function arguments and results, as opposed to plain arrays which are used by reference. Thus, a special include file is available that defines these vectors as structures for standard C compilers, while for the CPU64 compiler this definition is not read, and the built-in definitions for vector types are used. This way, the vector types can be used both in the CPU64 development environment and in a standard C development environment.

Figure 4 shows an example vector type definition. In this fragment of a C header file, the preprocessor flag `CPU64` switches between compilation by the cpu64 compiler and a standard C compiler. The vec_8sb type is built into the cpu64 compiler; from the compiler's point of view, the additional types are scalar.

```
#ifdef CPU64
typedef vec_8sb vec64sb;
#else
typedef struct
    {int8 sb[8];} vec64sb;
#endif
```

Figure 4. Example C definition of a vector type

The CPU64 compiler of course needs to know many details about the vector types, conversions between them, equivalence of certain operations on these types, and so on. We have endeavored to supply as much as possible of that information in the machine description file. For example, the many custom operations that work on these vectors are described in the machine description file, which the compiler reads each time an application is compiled. All these operations are thus known to the compiler, and some information about the semantics of a few of them is built into the compiler. This allows the compiler to generate efficient code. Again, the definitions of custom operations are also available in include files, for use by standard C compilers. Figure 5 shows an example custom operation definition.

```
#ifndef CPU64
#define custom_op extern
#endif

custom_op vec64sb
   sb_mul(vec64sb x,vec64sb y);
```

Figure 5. Example C definition of a custom operation

For most custom operations, the CPU64 compiler only needs to know of their existence, and the number and types of arguments and results. The compiler does not need to know about the actual function implemented by most of the custom operations. Therefore, the semantics of the custom operations are not described in the machine description file. The next section will elaborate on the implementation of the semantics of the custom operations.

5. Vector memory model and register model

The CPU64 instruction set only supports vector load and store operations on properly aligned memory addresses. This means that addresses of vectors must be a multiple of eight. In turn this implies, for example, that the compiler must properly align statically allocated vector variables and vector arrays in memory. Also, the ANSI C function `malloc()` must return an address that is now a multiple of eight, as the definition of `malloc()` requires that its result may be cast to any pointer type.

The situation is complicated further by an additional design constraint that we felt significantly increased the accessibility of vector processing for application programmers. The design constraint is that an array of data must be accessible both by scalar load and store instructions as well as by vector load and store instructions. Now the C language requires that appropriate address arithmetic results in the desired element pointer, and therefore elements of increasing array index must be stored on similarly increasing addresses. This in effect fixes the ordering of vector elements in memory.

In addition to the ordering between elements of vectors, the storage of individual vector elements must be the same as the storage of the corresponding scalar elements. This implies that the same endianness rules are applied to both scalars and individual vector elements. As is the case for the TM1000, the CPU64 supports both types of endianness.

The above paragraphs dealt with the memory model of vectors. Next we will discuss the register model of vectors. When a vector is loaded from memory into a register, the current processor endianness determines how the data corresponding to individual vector elements are interpreted and ordered. However, we still have a choice to the order in which the elements of a vector are to be stored in the register. We may choose to store the element with the lowest array index at the least significant end or at the most significant end of the register. Thus, in principle we could introduce at this point a second type of endianness: register vector endianness.

It is common knowledge that endianness issues can cause great confusion, and even thinking twice seldomly leads to better understanding. Introducing yet another type of endianness might entirely sabotage the concept of vector processing. Somewhat surprisingly, instead of making the situation more difficult by introducing vector processing, we were able to make vector processing a more intuitive concept for the application programmer by introducing the following abstraction.

The definition of custom operations at C level, as described in the application programmer's manual, does not refer to the register vector model at all. It only refers to the indices of the individual vector elements, and thus hides the actual implementation of the operation at the hardware level.

This abstraction actually makes life easier for the application programmer as compared with the TM1000 situation, where the ordering of vector elements in registers is highly visible. The CPU64 vector programming model releases the application programmer from maintaining a mental image of how the vector elements are stored both in memory and in registers. Experience has shown that this release does not occur without concerted effort on the part of the programmer, but once achieved, it is indeed experienced as a great improvement.

Figure 6 shows an example definition of a vector custom operation from the application programmer's manual. Note that the description only refers to actions on individual elements, and does not specify the order of the elements.

Now that this abstraction has been made for the application programmer, there is in fact no need for the system architect to actually implement both versions of register

```
void sb_dblmul (vec64sh *v, vec64sh *w, vec64sb x, vec64sb y)

Returns the elementwise product of x and y at full precision.
The first result operand will get the low indexed half of the results,
the second result the high indexed half.

Thus, for all i,0<=i<4: *v[i]=x[i]*y[i] and *w[i]=x[i+4]*y[i+4]
```

Figure 6. Example vector custom operation definition

vector endianness. Therefore we simply chose to implement the convention that the element of the lowest index is located in the register at the least significant end. The other choice would have been just as valid, and the application programmer would not have experienced any difference: the C program would not change at all.

Note that this abstraction needs to be supported in the instruction set by introducing load and store operations that are specialized for specific vector types. Depending on the current processor endianness (and the chosen register vector endianness), a load of a vector of 8-bit integers might be different from a load of a vector of 16-bit integers.

6. Supporting libraries

The complete set of custom operations is supported by the development environment. As mentioned before, the compiler is notified of the available hardware custom operations through the machine description file. Thus, the syntax of the hardware custom operations is defined.

The machine description file also contains information for the compiler on how to translate the C-level custom operations to the set of available hardware operations. Only the simulator needs to know about the semantics of the custom operations. As the hardware implementation operates on collections of 64 bits, and the interpretation of these collections of 64 bits is entirely up to the specific operation that executes on this data, the implementation of the semantics of the custom operations is untyped, in the sense that the routines only operate on generic 64-bit containers. Inside the routines that implement the functionality, the contents of the containers are cast to the appropriate vector type, the operation is performed, and the result is cast back into the generic 64-bit container type. This library of routines is referred to as the hardware operations library.

The hardware operations library is also used to support native compilation of application programs that use CPU64 custom operations, but an additional set of routines is necessary to connect the specifically typed C level custom operations with the generically typed hardware functions. This library is called the software operations library, and, in effect, it represents the same mapping functionality as is

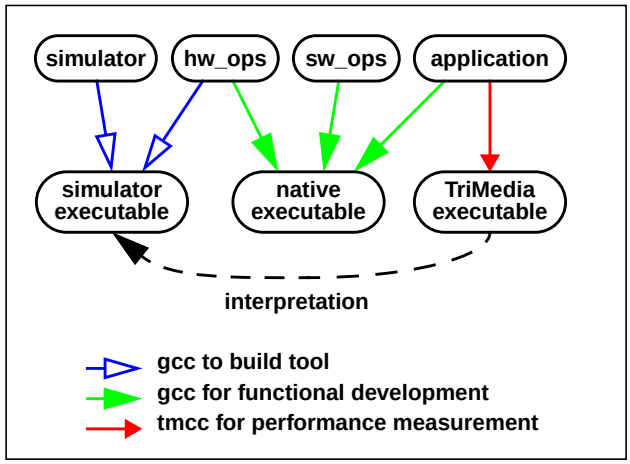

Figure 7. Overview of CPU64 and native compilation trajectories

provided to the CPU64 compiler by the machine description file.

Figure 7 shows an overview of the libraries involved in CPU64-specific and native compilation of application programs that contain custom operations. The hardware operations library is used in the construction of the simulator by linking it into the simulator executable. This simulator is then able to execute the custom operations as it encounters them in the instruction stream while interpreting an application compiled by the CPU64 compiler.

For compilation towards a native executable, the application is compiled while the `CPU64` preprocessor flag is inactive. Subsequently, the hardware operations library, the software operations library, and the application are linked into the executable which can run on the native platform.

7. Simulator

The CPU64 simulator contains modules that correspond to the major blocks of a complete TriMedia chip. Examples of these modules are the main memory interface, the highway data transport bus, the DSPCPU itself, etc. The simulator can perform cycle-accurate simulations of the entire chip and external memory. The speed achieved during simulations is about 30K cycles per second on a UNIX work-

station, which corresponds to a factor on the order of 10,000 below real-time performance.

The significant gap in simulated performance versus real-time performance urged us to support native compilation of applications. Thus, application programmers can develop their programs at the functional level by compiling and executing natively on a UNIX workstation. Once the application runs correctly, the application programmer can make specific test runs using the simulator to optimize the code. So, the simulator is used only in the later stages of application development. In this stage of development, the simulator provides crucial information on the run-time behavior of the application. The simulator collects and writes statistical information to a file, detailing where the CPU64 spends its time, and in what way: how many instructions were executed in parallel, how many cycles the processor was waiting on instruction or data cache misses, etc. This information can then be analyzed by the application programmer in order to optimize the performance of the application.

8. Optimization

As is true for all high-performance computing systems, it is easier to improve processor computing bandwidth than memory I/O bandwidth. Therefore, considering the targeted six-fold increase of performance of the CPU64 with respect to the TM1000, it becomes highly important to write and optimize applications to utilize the available memory bandwidth most efficiently. Ideally, the optimization process begins before a single line of code is written. From the outset, it is important to plan the way the data flows through the machine, when to write back data to memory versus when to try to keep it in registers, and in which type of vectors the data needs to be packed. Once this global application structure is in place, the individual functions can be implemented.

Up to this stage the application can be developed using native compilation, enabling a short functional development cycle. Once the application has been implemented in a functionally correct way, the local optimization process can start. The local optimization process is very similar to the optimization process of the TM1000. The optimization is guided by statistical information from simulated runs of the application, possibly supplemented by information from intermediate files that are generated by the compiler, such as assembly code. Thus, although writing assembly is discouraged, it may be necessary to study generated assembly in order to improve the efficiency of generated code.

9. Conclusions

The concepts that were implemented in the CPU64 to increase the performance include vector processing and superoperations. To fully support these new concepts throughout the entire platform, vector types were built into the TriMedia compiler, both the orthogonal C-level instruction set and the hardware instruction set were described in the machine description file, and the semantics of the custom operations were made available in supporting libraries. The resulting platform offers an intuitive approach to vector programming, allows application development with standard C compilers resulting in short development cycles, and supports subsequent optimization of the application based on cycle-accurate simulations.

10. Acknowledgements

The authors would like to thank the many people who have contributed to the construction of the described toolset and support libraries, including the application programmers for triggering the bugs. The engineers and architects at TriMedia are acknowledged for their valuable input and inspiration.

11. References

[1] P. Faraboschi, G. Desoli, J.A. Fisher, “The Latest Word in Digital and Media processing”, *IEEE Signal Processing Mag*., pp. 59-85. March 1998.

[2] A.K. Riemens et al., “TriMedia CPU64 Application Domain and Benchmark suite”, *these proceedings*

[3] J.T.J. van Eijndhoven et al., “TriMedia CPU64 architecture”, *these proceedings*

[4] S. Rathnam, G. Slavenburg, “An architectural overview of the programmable multimedia processor, TM-1”, *proc. Compcon ‘96*, Santa Clara CA, pp. 319-326. Feb. 1996

[5] G. Slavenburg, S. Rathnam, and H Dijkstra, “The TriMedia TM-1 PCI VLIW Mediaprocessor”, *Hot Chips 8*, Stanford CA, Aug. 1996

[6] J. Hoogerbrugge, A. Augusteijn, “Instruction Scheduling for Trimedia”, *J. for Instruction-Level Parallelism*, Vol. 1 (to appear), 1999

[7] G.J. Hekstra et al., “TriMedia CPU64 Design Space Exploration”, *these proceedings*