An ASIP Design Methodology for Embedded Systems

Kayhan Küçükçakar[†]

Escalade Corporation
2475 Augustine Drive
Santa Clara, CA 94086
kayhan@escalade.com

Abstract

A well-known challenge during processor design is to obtain the best possible results for a typical target application domain that is generally described as a set of benchmarks. Obtaining the best possible result in turn becomes a complex tradeoff between the generality of the processor and the physical characteristics. A custom instruction to perform a task can result in significant improvements for an application, but generally, at the expense of some overhead for all other applications. In the recent years, Application-Specific Instruction-Set Processors (ASIP) have gained popularity in production chips as well as in the research community. In this paper, we present a unique architecture and methodology to design ASIPs in the embedded controller domain by customizing an existing processor instruction set and architecture rather than creating an entirely new ASIP tuned to a benchmark.

1. Introduction

Many embedded systems have tight constraints for product cost. The cost shows itself in two major types of characteristics: code size and processor cost. Using off-the-self processors and memories is a common way to achieve the best time-to-market and the lowest cost. Core-based design will further decrease the cost by merging these off-the-shelf components on a single die. It is generally easy for designers to find the right processor with average performance and power for an application. But, **some parts** of the application generally become **performance bottlenecks**. If such bottlenecks can not be avoided through software reorganization, then either ASIC co-processor solutions are used or the application needs to upgrade to a higher performance processor which would generally have a higher cost and energy consumption.

Semiconductor industry generally attempts to address this problem by designing processors that are tuned for a special application domain such as DSP chips. DSP chips are regular processors with special architectural features and instructions that are tuned for digital signal processing. But, designing domain-specific processors do not necessarily solve the problem. There would be still applications that fit the general focus area of a processor, but a more expensive processor would **have to** be used due to a **small** number of performance bottlenecks.

This problem results in introduction of many commercial processors that target a general application domain such as embedded control, but also target a more specialized area such as fuzzy logic [1]. Even in such cases, more specialized instructions are desired on an individual application basis. There are also existing processors such that unused instructions can be removed from the hardware at the time of synthesis and from the software compiler to improve an application's characteristics.

In general terms, ASIP design task is the creation of a new processor, whose instruction set and architecture are customized for a targeted set of applications.

Creation of a new ASIP is tightly related to instruction-set design, backward compatibility issues, software compilation, design methodology, test and debugging tools and methodology. Primarily due to lack of tools in this area and the magnitude of work involved to integrate a new ASIP into design environments, the use of ASIPs is rather limited at this time [2].

1.1 Problem Statement

Our goal is to be able to optimize the code efficiency and the performance of a given application on a customizable processor architecture such that

- system efficiency (cost, code size, performance, and power) stays acceptable for embedded systems,
- customization should be as local as possible
- changes to the software environment should be as minimal as possible (backward compatibility for the bulk of the software should be preserved),

[†] The work was performed while the author was with Motorola, Inc.

- variable degrees of manufacturing flexibility is possible (custom, mask programmable, field programmable)

The related work in instruction-set selection and retargetable compilers can be used to further our approach. But, in this paper we are limiting ourselves to discuss architectural and methodological issues.

2. Related Work

Prior work in ASIP area is restricted to custom processor design such as MC68HC12 and DSP processors, or to approaches which combine instruction set definition (or selection), architecture creation and instruction mapping onto a newly created architecture.

Alomary, et. al. combines instruction-set and architecture design [3]. The first step is to pass the software through a profiler which provides an "importance" factor for each operation. The architecture consists of a kernel which contains the primitive RTL operators, register file, multiplexers, control and buses. In addition to the kernel, application-specific ALUs can be added to the architecture, based on an area constraint. The architecture is fairly simple and restrictive.

Van Praet, et. al. provides an interactive approach to combined instruction-set definition and instruction selection [4]. An analysis tool is used to extract operations and operation sequences from an application. Instruction set is described in nML. Datapath parts are created manually, based on observations from the analysis tool. Then, the operations are bundled to create instruction formats such as loop instructions, and these formats define encoding restrictions for the instructions.

Huang and Despain describe an approach which combines instruction set and architecture design, and instruction mapping from an application and architecture template [5]. The architecture template defines the pipeline structure which consists of fetch, decode, register read, ALU operation, memory access and register write phases.

Each of these approaches primarily creates a new instruction set and a new architecture which are tuned for a set of applications. This kind of approach creates extreme fluidity in the processor and requires very effective hardware synthesis and retargetable software compilers. Also all approaches ignore control aspect of design. This might be acceptable for DSP applications but not for general embedded systems. It should be noted all prior art (just as ours) have significant interactivity built into the process.

Our approach, as described in the next section, is targeting a different problem and area than ASIP research so far.

3. Approach

Our approach attempts to customize an existing processor rather than synthesizing a brand new processor with a new instruction set and architecture that are optimized for a group of benchmark applications. In this section, we will provide a brief description of the problem. Let

I_i be an existing instruction i,

L_i be the number of clock cycles I_i takes to execute

$NI_i = \{I_a, I_b, \ldots\}$ be a new instruction which is composed of an ordered set of existing instructions

NL_i be the number of clock cycles to execute NI_i

G_i be the performance gain provided by NI_i

C_i be a 0-1 variable and set to 1 if NI_i is used

F_i be the use frequency of NI_i

Then,

$$G_i = \Sigma_j L_j - NL_i \qquad \text{where } j \in NI_i$$

The general goal is to maximize the overall performance gain

$$\Sigma_i F_i G_i C_i \qquad \text{where } i \text{ is over } NI_i$$

subject to

$$ControlAreaConstraint \geq TotalControlArea$$

$$DatapathAreaConstraint \geq TotalDatapathArea$$

But, the control and datapath costs are not straightforward to formulate except obtained through trial implementation. When multiple instructions are mapped to the same programmable logic area, logic sharing and optimization affect the total area (cost). The clock frequency is generally dictated by many other factors and is not subject to general area-performance tradeoff.

3.1 Design Flow

The design flow is shown in Figure 1. Given a software (firmware) implementation using traditional methods, performance bottlenecks are identified with the aid of a profiler program. Then, the processor is customized through addition of application-specific instructions. Finally, the firmware is updated to use new instructions.

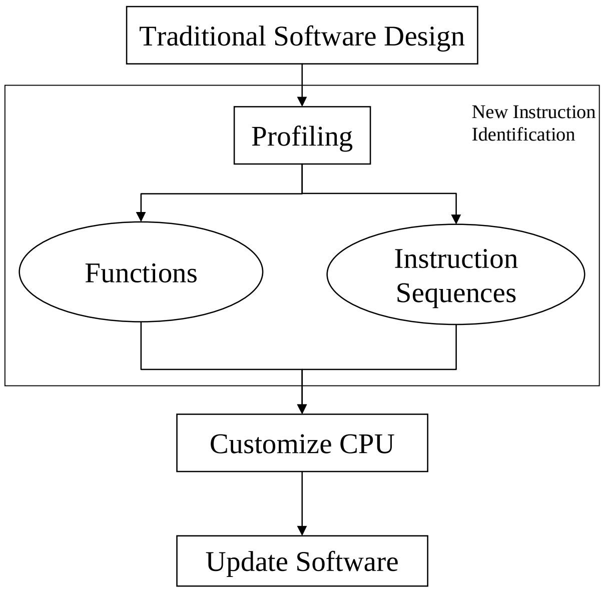

Figure 1. ASIP Design Flow

3.2 Identification of New Instructions

The new instructions generally have one of the two characteristics. They are either a subroutine in the firmware that is used frequently or a sequence of instructions that are common in the application. Examples of subroutines include device drivers, basic elements of computation, timer operations, and operating system primitives such as schedulers. Examples of frequently used sequence of instructions include shift-and-add sequence used in digital filtering operations, zero-overhead loops, data type conversion, data formatting for I/O, and signal polling.

Any sequence of instructions, whether detected manually or automatically, can be used in our methodology. But, we focus our efforts to identifying low-level subroutines as candidates for new instructions without loosing any generality.

3.3 Processor Architecture

Our overall approach depends on the processor architecture that has a fixed set of instructions and datapath, but also allows addition of new control and datapath logic through use of programmable hardware. But, this is accomplished without changing the overall processor architecture. Two styles of such customizable processor architectures are shown in Figure 2 and Figure 3. The decoder block shown in both figures include sequencers hence should be considered as the entire control block.

These architectures do not impose any restrictions on instruction decoding, sequencing or datapath. Therefore, these architectures are applicable to most industrial processors.

The architecture is chosen to have programmable logic to reduce the time-to-market. A custom implementation can still be used for high-volume applications. Implementing the entire processor on programmable hardware is possible but not preferred since it brings fluidity to the instruction set, hence creates maintenance problems.

Application-specific computation can also be performed on a peripheral implemented by programmable logic. The disadvantage of such an approach is the lack of efficient access to processor internals.

3.3.1 Static Decode Architecture

This type of architecture (shown in Figure 2) only allows a set of predetermined opcodes to be used for new instructions, thus having an efficient opcode decoding structure. If any of such opcodes is fetched and decoded, the signal $start_i$ which is dedicated for the opcode *i* is activated. At the same time, the processor relinquishes the control of the datapath control signals to the programmable section. When opcode *i* is completed, it activates another dedicated signal $done_i$, hence giving control back. Intrinsic instructions do not have access to the functional units implemented via programmable logic. This architecture has a better implementation efficiency, but at the expense of presetting number of new instructions that can use the programmable logic.

3.3.2 Dynamic Decode Architecture

This type of architecture is more flexible and allows new opcodes to be defined per application basis. The decoding of new instructions are performed by the programmable logic. When such an instruction is fetched, the processor can not decode it, hence it issues a trap by sending a signal to the programmable logic section. If the programmable logic can decode the instruction, then it is executed. If the instruction is not known to the programmable logic, then it activates trap' which instructs the fixed logic to start illegal instruction handling procedures.

3.4 ASIP Implementation

The new ASIP implementation phase can be carried out using RTL synthesis or behavioral synthesis. Our preference is to use behavioral synthesis to reduce the effort: reuse of code segments from existing instructions in definition of the new instructions and easy partitioning of new instruction logic from the existing implementation. Only prerequisite for using behavioral synthesis in this phase is the ability of behavioral synthesis to produce competitive (commercial grade) processor implementations which was previously shown in [6].

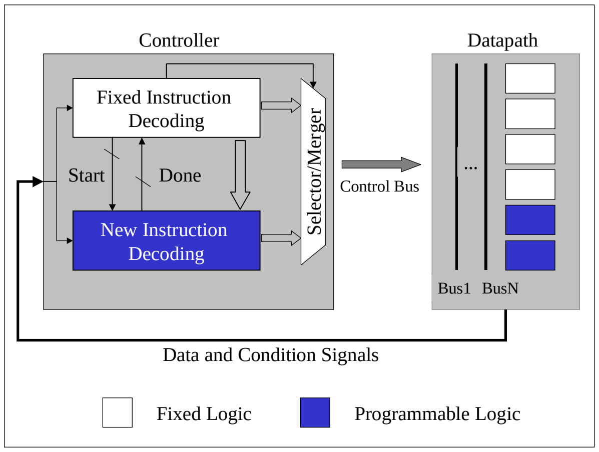

Figure 2. Static-decode Architecture

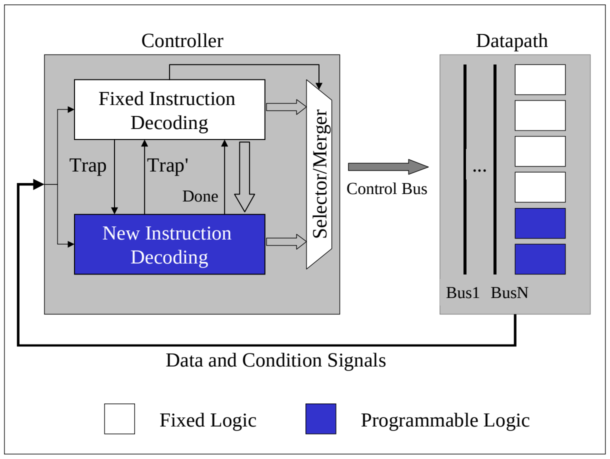

Figure 3. Dynamic-decode Architecture

3.5 Firmware Modification

Once new instructions are implemented in the processor, the embedded firmware needs to be modified to use these new instructions. Depending on software methodology used, there are three ways of performing this task.

If the firmware is entirely in assembly language or new instructions only relate to the portion of the firmware that is in assembly language, only assembly code is modified.

If the software methodology is using a higher-level language compiler such as C and if the new instructions are applicable to other applications, then they can be added to the compiler.

When retargetable compilers become common tools, a new architectural specification of the processor can be fed into a retargetable compiler that becomes enabled for the new instructions.

4. Results

In this section, we present early results in two application areas. We implemented two application-specific instructions on an MC68HC11 [6] that we have previously implemented. The first example is a hexadecimal-to-binary conversion routine. This is a representative of a typical control-dominated application. The second example is a 16x16 multiplication that is a representative of DSP applications. We have explored different ways using programmable logic on these examples demonstrating the tradeoff between available programmable logic and performance speedup possible.

4.1 Hex-to-binary conversion example

The assembly code for hexadecimal-to-binary conversion routine is shown in Figure 4. For this application, the new instruction was implemented without using any programmable datapath since the application is control oriented. The implementation of a new instruction (partially shown in Figure 5) resulted in about 75% reduction in the number of cycles to execute the hex-to-binary conversion routine as shown in Table 1. This new instruction specification was entirely mapped onto the existing structure of MC68HC11 through Matisse.

The benefits of the new instruction came from 3 sources:

- Elimination of unnecessary instruction fetching
- Elimination of data passing between instructions
- Elimination of creating unused information

The program memory use for the original implementation was 39 bytes and it was reduced to 1 byte due to the use of a new 1-byte opcode.

4.2 16x16 Multiply example

MC68HC11 has an 8-bit multiply instruction. The 16x16 multiplication is performed as a series of 8x8 multiplications with use of scratch memory locations in main memory, as shown in Figure 6.

3 different instruction implementations were explored. “A” instruction only exploits a programmable control section with no datapath extensions. Since the application is not control dominated, the benefit was 33% cycle-count reduction, not as much as the previous example. Since a significant bottleneck in this application is the shift-and-add multiplication, “B” instruction added an 8x8 single-cycle hardware multiplier. In this case, cycle-count reduction reached 58%. To check the higher limit of performance increase without changing the architecture of the MC68HC11, an eight-location register file was added in instruction “C”. Then, the cycle-count reduction reached 71%. The original code size for the multiplication routine was 51 bytes and new instructions enabled 1-byte opcode. Each of these new instructions was implemented by creating an algorithmic specification and then implementing through behavioral synthesis tool Matisse.

```
; Calling Routine
                LDS             #stack
                LDAA            #'D'
                JSR             HEXBIN
…

HEXBIN          PSHA
UPCASE          CMPA            #'a'
                BLT             UCDONE
                CMPA            #'z'
                BGT             UCDONE
                SUBA            #$20
UCDONE          CMPA            #'0'
                BLT             HEXNOT
                CMPA             #'9'
                BLE             HEXNMB
                CMPA            #'A'
                BLT             HEXNOT
                CMPA            #'F'
                BGT             HEXNOT
                ADDA            #$9
HEXNMB          ANDA            #$0F
                INS
                RTS
HEXNOT          PULA
                SEV
                RTS
```

Figure 4. Hexadecimal-to-binary conversion

	Execution Cycles			
	Best Case	**Percent Reduction**	**Worst Case**	**Percent Reduction**
68HC11	30	-	53	-
68HC11+	7	77	13	75

Table 1. Cycle-count reduction by a new instruction

5. Conclusion and Future Directions

We have provided an architecture and a co-design methodology to improve the performance of embedded system applications through instruction-set customization. The customization method through mask-level or field programmability enables designers to optimize performance and code size of applications. We have also demonstrated through both control and data oriented applications that significant benefits are possible.

The methodology presented in this paper enables the use of instruction-set customizable processors for core-based design, such as presented in [8]. More automation in the methodology would enable automatic HW/SW partitioning capability. This would require accurate estimation techniques for the implementation cost of candidate instructions.

6. Acknowledgements

The re-configurable processor architecture described in the paper was developed in collaboration with C.-T. Chen. The MC68HC11, which was used during experiments, was created by T. E. Tkacik.

7. References

[1] *MC68HC12 Reference Manual*, Motorola, Inc.

[2] C. Liem. *Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications*, Kluwer Academic Publishers, 1997.

[3] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint, *In Proceedings of International Conference on Computer-Aided Design*, pages 526-532, 1993.

[4] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction Set Definition and Instruction Selection for ASIPs, *In Proceedings of International Symposium on High-level Synthesis*, pages 11-16, 1994.

[5] I.-J. Huang and A. M. Despain. Generating Instruction Sets and Microarchitectures from Applications, *In Proceedings of International Conference on Computer-Aided Design*, pages 391-396, 1994.

[6] K. Küçükçakar, C.-T. Chen, J. Gong, W. Philipsen, and T. E. Tkacik, Matisse: An Architectural Design Tool for Commodity ICs, *IEEE Design and Test of Computers*, Vol. 15, No. 2, pp. 22-33, April-June, 1998.

[7] MC68HC11 Reference Manual, Motorola, Inc.

[8] K. Küçükçakar. Analysis of Emerging Core-based Design Lifecycle, *In Proceedings of International Conference on Computer-Aided Design*, pages 445-449, 1998.

	Execution Cycles		
Design	**Datapath**	**Cycles**	**% Reduction**
68HC11	-	144	-
68HC11A	-	97	33
68HC11B	8x8 mul	61	58
68HC11C	8x8 mul RF[8]	42	71

Table 2. Cycle-count reduction by new multiplication instructions.

```
….
// Instruction decoding
begin

    // PUSHA
    r_w_N = ‘WRITE;
    addr_bus {SP_hi,SP_lo};
    data_bus = ACCa;
    dec_L(SP_lo,addr_carry,SP_lo);
    dec_H(SP_hi,addr_carry,SP_hi);

    // CMPA #’a’
    r_w_N = ‘READ;
    alub = ~8’h61;
    add(Acca, Alub, 1’b1,ALU_OUT,Cx,V,Hx);
    N = ALU_OUT[7:7};
    If ( ! (N^V)) begin
        // hex number is >= ‘a’
        // CMPA #’z’;
        alub = ~8’h7a;
        add(Acca, Alub, 1’b1,ALU_OUT,Cx,V,Hx);
        N = ALU_OUT[7:7};

        if (Z | (N^V)) begin
             // hex number is <= ‘z’
            alub = ~8’h20;
            add(Acca, Alub, 1’b1,ALU_OUT,Cx,V,Hx);
           AccA = ALU_OUT;
        end
    end
    …
    …
end
```

Figure 5. Algorithmic code for hexadecimal-to-binary conversion to be synthesized

```
; Multiplicand is stored in (Y) - (Y)+1
; Multiplier is in ACCD
; Multiplier Scratch location is in: (X)+4 - (X)+5
; Product is stored in (X) - (X)+3

mpy16           STD             4,X
                LDD             #0
                STD             3,X
                LDA             5,X
                LDB             1,Y
                MUL
                STD             2,X
                LDA             5,X
                LDB             0,Y
                MUL
                ADDD            1,X
                STD             1,X
                LDA             4,X
                LDB             1,Y
                MUL
                ADDD            1,X
                STD             1,X
                ROL             0,X
                LDA             4,X
                LDB             0,Y
                MUL
                ADDD            0,X
                STD             0,X
                RTS
```

Figure 6. 16x16 Multiplication