Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding

Subash Chandar G, Mahesh Mehendale
Texas Instruments India Ltd.
Bangalore, India
{subba,m-mehendale}@ti.com

R. Govindarajan
Indian Institute of Science
Bangalore, India
govind@serc.iisc.ernet.in

Abstract— In this paper, we propose a reconfiguration mechanism that allows multiple instruction compression to reduce both code size, which in turn reduces the cost, and (instruction fetch) power, which enhances the battery lifetime, two key considerations in embedded DSP systems. We enhance Texas Instruments DSP core TMS320C27x to incorporate this mechanism and evaluate the improvements on code size and instruction fetch energy using real life embedded control application programs. We show that even with minimal hardware overhead, we can improve code size by over 10% and instruction fetch energy by over 40%.

1. Introduction

In embedded control applications, system cost and power are important considerations. The chip area, for such applications, is dominated by the program memory. Hence, reducing program memory is very important for reducing system cost. The processors used in these applications use several techniques like variable length instructions [2] [1], complex instructions and many addressing modes [2] to reduce code size.

Several compression techniques have been proposed for general purpose and application specific architectures [13] [7] [4]. The mechanism proposed in [13], uses the instruction cache as a decompression buffer that stores uncompressed copies of recently used blocks of instructions. Code compression using operand factorization is proposed in [4].

All these work focussed on using short variable length code words to represent a list of instructions. Huffman coding is commonly used to achieve higher compression ratios [13] [7]. However, this increases the latency of decompression. A system with less hardware overhead was proposed in [5]. [8], [9] noted that most compression on DSP architectures can be attributed to single instruction patterns. Compressing only single instructions helps in a simpler decompression unit.

In all these work, the encoding of a sequence or a single instruction to a smaller code word happens once in the beginning of running an application. In this paper, we propose a mechanism to use short encodings effectively by allowing the meaning of the encodings to change during different phases in running an application program. Also all these earlier work focussed only on code size reduction and the re-usability of the same mechanism for power reduction was either not applicable or not evaluated.

Fetching Instructions from the memory consumes considerable part of the total power. Code compression has also been used to target power reduction [11] [10]. [6] trades off programmability to reduce power using instruction sub-setting. Power reduction by reducing number of toggles on address bus using Gray code is discussed in [12]. Our experimental results indicate that the number of toggles on program data bus amounts to 80% of the total number of toggles Hence we focus on reducing toggles on program data bus.

We propose a common reconfiguration mechanism that can be used for both code size and power reduction. The proposed mechanism encodes (or compresses) instructions to reduce code size and power, unlike any other approach discussed in related work.

2. Re-configurable Instruction Encoding for Code Size Reduction

In our re-configurable architecture, the encodings of instructions are not fixed and can be re-mapped to different encoded values. This re-mapping can be done either statically, once prior to running a given application (static configuration), or possibly many times while running a given application program (dynamic configuration). An instruction re-mapped to a shorter length encoding is referred as a compressed instruction.

2.1 Static Configuration

In case of static configuration, we encode the instructions to binary values of different lengths, once prior to running a given application with the objective of reducing the code size for that particular application. Thus each application has fine tuned encodings for the instructions. Hence, this method achieves better code size compared to having the same fixed encodings for the instructions for all applications.

In this method, each of the application program is individually profiled to get instruction usage information. Using this information, we identify top m most frequently used instructions and configure (map) them to codes of shorter length. This configuration is done by adding the configuration sequence at the beginning of the code. (refer to Section 3 for details) After the configuration, m most commonly used instructions have smaller width for their binary representation and hence occupy less space in program memory. In other words, now the code size has reduced.

In computing the code size for the compressed code, we take into account the configuration overhead (re-map table size and the instructions added for configuring the re-map table) and the overhead due to the additional bit for each uncompressed instruction (refer to Section 3.1 for details).

2.2 Dynamic Configuration

Inside each given application, the instruction usage may vary across different segments of code. Static configuration cannot exploit this property to reduce code size. In dynamic configuration, the instructions can be re-mapped to smaller encodings on the fly, possibly many times, while running a given application program. Thus, dynamic configuration exploits varying instruction usage inside a given application, to achieve better code size. Every re-configuration has an overhead associated with it. An overall improvement in code size is achieved when the code size of the given application with dynamically configured instructions that includes the re-configuration overhead is smaller than the original code size.

The application program is first divided into several segments. Each segment is individually profiled to get instruction usage information and the top most m commonly used instructions are compressed. In other words, each code segment can now be viewed as being configured statically. The code size after compression is obtained by adding the compressed code size of each of the code segment. The code size for a segment is computed in a manner similar to that in static configuration.

2.2.1 *Key Care-abouts in Dynamic Configuration*

In dynamic configuration, same encodings map to different instructions in different parts of the code. Hence, the branch instructions that jump from one code segment to the other need special care, if the two code segments use different configurations. This is taken care of by inserting a configuration instruction at the entry of the branch that executes the configuration sequence for the segment into which the branching has happened.

Subroutines need special care since they can be called from different parts of the code which may have different configurations. For simplicity, we leave the subroutines uncompressed in our analysis.

Branch instructions themselves are not compressed. The branch target addresses have to be re-adjusted after the replacement of original instruction with compressed instructions. We assume that the instructions are assembled after re-map instructions are introduced.

3. Instruction Re-map Table

To allow configuration of instructions, we propose a mechanism — *Instruction Re-map Table (IRT)*. *IRT* can be viewed as a register file. Each entry in this table can hold a valid uncompressed instruction of size equal to 16 bits. Each entry in the table has a unique address which forms the compressed representation of a valid instruction stored in that location. A pair of special instructions are added to the existing instruction set that allows configuration of *IRT*. Configuring the *IRT* involves writing the instructions to be compressed, into it. Once an instruction is written into the table, it can be referenced by the address of it's location. Since the width of the address which is the compressed instruction is smaller than that of the actual instruction, we achieve compression. The details are discussed in the following section.

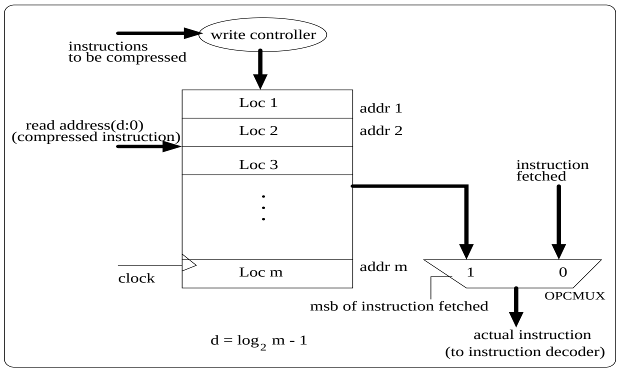

Figure 1: Instruction Re-map Table

3.1 Organization of Re-map Table

The instructions to be compressed are first written into locations Loc 1 to Loc m in the IRT shown in Figure 1. Configuring the re-map table will be discussed in Section 3.2. After the configuration, each of the locations in the re-map table holds a 16 bit instruction. The width of the address of these locations is equal to $log_2 m$ where m is the number of locations in IRT. Each instruction that is written into the table can now be referenced by the address of it's corresponding location. We add one extra bit to this address to indicate that it is a compressed instruction. This added bit forms the most significant bit (msb) of the compressed instruction and is set to '1' to indicate that it is a compressed instruction. The msb of all uncompressed instructions are set to '0'. With this, the program memory is organized as 17 bit wide. For on chip memories, this non-standard width is not an issue.

Now, the instructions that are written into the instruction re-map table have a unique compressed representation. All these instructions in the program memory are now replaced by their compressed representation. Other instructions appear as uncompressed.

Decompression of compressed instructions is achieved as follows. The re-map table is indexed with $log_2 m$ bits following the ms bit of the fetched instruction (compressed or uncompressed). The ms bit of the fetched instruction is used to select either the contents of the indexed location of the re-map table or the fetched instruction (without the ms bit). Thus the output of the multiplexer is the uncompressed encoding of the fetched instruction.

3.2 Configuring the Re-map table

An instruction can be compressed by writing it into the instruction re-map table. A set of instructions that are to be compressed are identified and are written into the instruction re-map table. The process of writing instructions into the instruction re-map table is referred as *configuring the IRT*. Configuration of the IRT is made possible by adding a pair of special instructions to the existing instruction set. We call these instructions as $BCONF$ (Begin CONFiguration) and $ECONF$ (End CONFiguration) This pair of instructions is similar to a $call$ and $return$ instruction pair.

$BCONF < conf.address >$
This instruction is decoded as a $CALL$ instruction by the decoder. In addition, a configuration bit in the re-map table is set. Then the instructions are fetched from the location $< conf.address >$ in the program memory. The configuration bit in the instruction re-map table, when set, shuts off the instruction decoder and enables the write controller of the re-map table. The write controller generates write address to the instruction re-map table starting from the first location. So a set of consecutive locations are written with the instructions that are fetched from $< conf.address >$. This configuration happens till an $ECONF$ instruction is fetched.

$ECONF$
A simple comparator is used to decode $ECONF$ instruction. This instruction resets the configuration bit in the the write controller of the instruction re-map table. This turns on the instruction decoder. This instruction is decoded as a $return$ instruction and returns back to the calling program and starts executing the instructions.

For static configuration, the IRT is configured once in the beginning of the program. For dynamic configuration, IRT gets configured multiple times, to exploit the varying instruction usage within an application program.

We insert an additional configuration instruction at branch target addresses, if the target address is reachable by a branch from a different segment. This will reconfigure the re-map table to the configuration required by the target code segment.

It can be seen that our proposed implementation of configuring the re-map table using $BCONF$ and $ECONF$ is both versatile and efficient. Note that the $BCONF$ instruction enables writing the uncompressed instructions on successive locations in the re-map table starting from Loc 1, until a $ECONF$ instruction is encountered. Thus, it is possible to only re-write a part of the re-map table during each configuration leaving the rest of the re-map table unchanged. Thus, a combination of single (static) and multiple (dynamic) encoding can be achieved by writing the instructions to be mapped statically into locations at higher addresses (that can be left unchanged) and the instructions to be re-mapped dynamically into locations at lower addresses which are re-written multiple times.

3.3 Architectural Support for Instruction Re-map Table

We use Texas Instruments DSP core TMS320c27x [2] as the base architecture to evaluate our proposal. TMS320c27x DSP core has 8 pipeline stages, viz. Initiate-Fetch, Complete-Fetch, Predecode, Decode, Initiate-Read, Complete-Read, Execute and Write stages.

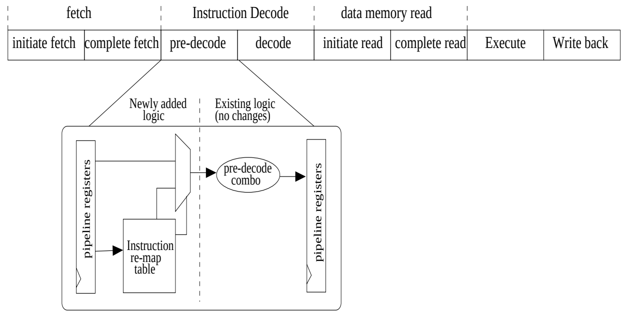

Figure 2: Modifications to CPU due to Instruction re-map table

The address of the instruction to be fetched from program memory is put on the address bus in Initiate-Fetch stage of the pipeline. During Complete-Fetch stage, the memory drives the instruction on the program data bus. The fetched instruction is decoded in two stages - predecode and decode. If the instruction needs data from data memory the same is fetched with a two stage data memory read. Then it is executed in the Execute stage and the results are stored in write stage of the pipeline.

We enhance the DSP core with the inclusion of instruction re-map table. The instruction re-map table fits in the pre-decode stage of the pipeline as indicated in Figure 2. The same can be implemented outside the core. But since memory access time is the bottleneck, we include the re-map table in the pre-decode stage where the access time of small table of size less than 128 entries, can be hidden.

4. Experimental Results: Code Size Reduction

We use two real life embedded control application programs[1] to evaluate the improvement in code size. These benchmarks are named BM1 and BM2. Table 1 summarizes the code size reduction for various re-map table sizes. Columns 2, 3, 4 and 5 in Table 1 represent the percentage reduction in code size for static and dynamic re-configuration for the two benchmarks. In columns 6 and 7, we report the number of unique instructions that get re-mapped during dynamic re-configuration. For static re-configuration, the number of unique instructions compressed is equal to the re-map table size. In computing the reduction in code size, we have taken into account the re-configuration overhead as mentioned in Section 2.

We observe that even with a small table size of 128 entries the gains are over 10% for dynamic configuration. Since we configure only once in static configuration, higher table sizes are needed to cover a good portion of commonly used instructions. In dynamic configuration, since the instructions are configured many number of times, the number of unique instructions compressed are high and this results in good improvements even for smaller table sizes. We also observe that for large table sizes (1024 and 2048), the improvement in code size reduces for both static and dynamic re-configuration.

In the results presented in Table 1, we have obtained code size reduction by assuming the size of the compressed instruction size to be

[1] Due to proprietary nature of these applications, we do not disclose the names of these applications

Remap	Static config.		Dynamic configuration			
Table	%CSR		%CSR		uniq instr	
Size (m)	BM1	BM2	BM1	BM2	BM1	BM2
8	-0.67	-1.76	2.99	4.49	321	361
16	2.01	0.22	5.72	7.36	593	534
32	4.42	2.06	8.07	9.43	590	606
64	6.66	4.16	9.66	10.21	591	872
128	8.86	5.82	11.11	10.08	608	969
256	10.73	6.96	11.30	9.06	394	1083
512	11.66	6.92	11.66	7.49	512	963
1024	10.88	5.07	10.88	5.07	1024	1024
2048	6.98	0.02	6.98	0.02	2048	2048

Table 1: Code size reduction

$1 + log_2$*(re-map table size)*. However to make the hardware implementation simple, uniform and more practical, we restrict the compressed instruction size to 8 bits for all the table sizes less than or equal to 128. Table 2 summarizes the code size reduction where the size of the compressed instruction is 8 bits irrespective of the table size. With the 8 bit restriction, out of which 1 bit is used to indicate if it is a compressed instruction, we have only 2^7 possibilities for compressed instructions. So we restrict maximum table size to 128 entries in our evaluations. In this, we observe a slight decline in the code size reduction compared to the results presented in Table 1.

Remap	Static config.		Dynamic configuration			
Table	%CSR		%CSR		uniq instr	
Size (m)	BM1	BM2	BM1	BM2	BM1	BM2
16	-0.25	-1.56	0.86	2.17	593	534
32	2.26	0.37	4.19	5.29	590	606
64	5.20	2.97	7.36	7.64	591	872
128	8.86	5.82	11.11	10.08	608	969

Table 2: Code size reduction : Restricted size

5. Re-configuration for Low Power

In this section, we detail the approach to reduce power consumption using re-configurable instructions. In a processor, instruction fetching contributes to a significant portion of the overall power consumption. This power spent depends on the switching activity on the program address bus and that on the program data bus. Our experimental results indicate that program data bus contributes to over 80% of the total number of toggles. Hence we focus on reducing the number of toggles on program data bus.

The number of toggles on program data bus depends on the choice of encodings for the consecutive instructions in the dynamic sequence and the total number of fetches made. With re-configurable instructions, we reduce both the number of fetches needed and the number of toggles between consecutive fetches of the compressed instructions Firstly, we reduce the number of bits fetched by compressing a set of most commonly occurring instructions in the dynamic sequence. Further, using Gray codes for encoding the instructions, we reduce the number of toggles between consecutive fetches of compressed instructions. In this analysis we allow two fixed sizes — 4 bits or 8 bits — for compressed instructions. Table sizes less than 16 locations use 4 bit encodings and table sizes from 16 to 128 locations use 8 bits encodings for the compressed instructions.

5.1 Static Configuration

The entire dynamic sequence of instructions is profiled to get instruction usage information. Based on this information, a set of top most commonly used instructions are chosen and are compressed. Due to this, the number of bits fetched reduces. The compressed instructions, fetched in consecutive accesses, are assigned with Gray coded values to further reduce the number of toggles.

5.2 Dynamic Configuration

IRT Size	% reduction in number of toggles						Energy spent (Normalized)						# of re-configurations (dynamic)		
	static			dynamic			static			dynamic					
	bm-A	bm-B	bm-C	bm-A	bm-B	bm-C	bm-A	bm-B	bm-C	bm-A	bm-B	bm-C	bm-A	bm-B	bm-C
0	–	–	–	–	–	–	1	1	1	1	1	1	0	0	0
4	−3.7	32.4	11.0	6.5	39.7	29.4	1.06	0.71	0.92	0.98	0.65	0.75	27	124	52
8	2.5	36.3	30.3	24.5	47.0	47.9	0.99	0.66	0.75	1.13	0.59	0.63	129	63	75
16	25.9	43.9	40.2	51.0	57.0	57.8	0.79	0.60	0.68	0.65	0.53	0.63	10	32	149
32	39.0	48.2	57.7	65.7	65.8	71.2	0.70	0.59	0.56	0.72	0.78	0.61	10	64	64
64	50.3	54.0	66.3	62.0	70.7	79.8	0.73	0.61	0.58	0.86	0.83	0.53	4	22	9
128	56.8	65.9	68.6	72.5	70.5	68.6	1.04	0.64	0.72	1.32	0.72	0.72	3	4	2

Table 3: Toggle Reduction and Energy savings

The entire dynamic sequence is broken into several equal sized segments. Each segment is then profiled individually to get instruction usage information. Based on this, the most commonly used instructions in each segment are configured to have smaller encodings. For each table size, the partition that results in the least number of toggles is chosen. This partition determines the number of times the IRT is re-configured. All the careabouts listed in Section 2.2.1 are taken care of while dynamically re-mapping the instructions to smaller encodings.

6. Experimental Results: Power Reduction

The reduction in number of toggles for three proprietary benchmarks *bm-A*, *bm-B* and *bm-C* are summarized in columns 2 – 7 in Table 3. For dynamic re-configuration, the minimum number of toggles is achieved for table sizes less than 128 and then the improvement declines. We observe that the reduction in number of toggles is over 70% for all three benchmarks.

To compute the actual savings in energy, we need to include the energy overhead due to *Instruction Re-map Table (IRT)*. The energy spent by *IRT* is due to *writing into the table during configuration* and *reading of compressed instructions from the table*. The table is optimized for low power by gating the clock for the Re-map Table (clock is shut off all the time except when the table is reconfigured), and by indexing the table (changing the address lines) only when the compressed instructions are fetched.

Energy measurements are made as follows. We obtain the switching activity information for the cpu-memory interface and for the IRT using simulations (by fetching the compressed code from the memory that includes re-configuration instructions). This switching activity is then back-annotated on the cells and nets that are part of cpu-memory interface and the IRT We use Synopsys Power Compiler^{TM} [3] to obtain power numbers. The power numbers are then multiplied with the simulation time to get the energy numbers.

Energy reduction for static and dynamic configurations for different table sizes are summarized in columns 8 – 13 in Table 3. Table size 0 corresponds to no configuration (original encoding for instructions). We observe that smaller table sizes provide better energy reduction though the total number of toggles are more compared to bigger table sizes. This is because larger tables consume more energy during read and write operations. The energy consumed is also a function of the number of times the table is re-configured.

We observe that for static configuration, a table size of 32 locations is good enough to yield energy savings of over 40% for *bm-B* and *bm-C* and 30% for *bm-A* . With larger table size, there is a decline in energy reduction. This is due to larger energy overhead for larger re-map table. With dynamic configuration, we are able to achieve energy reduction of 47% for *bm-B* and *bm-C* and 35% for *bm-A* . Even a table size of 16 locations gives very good reduction in energy for dynamic configuration. For *bm-C* , we observe that table size of 16 locations resulted in too many re-configurations (149 times) to achieve good reduction in number of toggles. Due to this high number of re-configurations, the energy overhead due to re-configuration is high. For *bm-C*, table size of 64 gives the best energy savings though the number of toggles is not the lowest, due to lower re-configuration overhead.

7. Conclusions and Future Work

We proposed a mechanism *Instruction Re-Map Table* which acts as a decompression unit with minimal hardware overhead. We explained an incremental re-design of the TMS320c27x CPU to include the instruction re-map table. We showed that multiple (dynamic) encoding helps in achieving higher benefits for both code size and power with smaller tables. In our approach, with small table implemented as a register file inside the CPU, the decompression delay is hidden. We also showed that the same hardware mechanism can be used to target power reduction. We achieved code size improvement of over 10% on an optimized code and about 40% reduction in energy spent in fetching instructions. We used real life application programs as benchmarks in our analysis.

In our future work, we plan to evaluate the impact of compressing subroutines also. We plan to do optimal partitioning for power by taking into account the number of re-configurations.

8. References

[1] Advance RISC Machines Ltd. An introduction to Thumb, March 1995.

[2] Texas Instruments. TMS320C27x DSP CPU and Instruction Set Reference Guide, March 1998.

[3] Synopsys Inc. Power Compiler Reference Manual ver.2000.11, 2000.

[4] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression based on operand factorization. International Symposium on Microarchitecture, 1998.

[5] S. Devadas, S. Liao, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. Adavanced Research in VLSI, 1995.

[6] W. Dougherty, D. Pursley, and D. Thomas. Instruction Subsetting:Trading Power for Programmability. In *Design Automation Conference*, 1998.

[7] M. Kozuch and A. Wolfe. Compression of Embedded System Programs. In *Proc. Int'l Conf. on Computer Design*, 1994.

[8] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving Code Density Using Compression Techniques. Technical Report CSE-TR-342-97, 8 1997.

[9] C. Lefurgy and T. Mudge. Code compression for DSP. Technical Report CSE-TR-380-98. Presented at CASES-98 Workshop, 1998.

[10] H. Lekatsas, J. Henkal, and W. Wolf. Code compression for low power embedded system design. In *Design Automation Conference*, pages 294–299, 2000.

[11] H. Lekatsas, W. Wolf, and J. Henkel. Arithmetic coding for low power embedded system design. In *Data Compression Conference*, pages 430–439, 2000.

[12] M.Mehendale, S.D.Sherlekar, and G.Venkatesh. Extensions to Programmable DSP architectures for Reduced Power Dissipation. In *International Conference on VLSI Design*, 1997.

[13] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. In *Proc. Int'l Symp. on Microarchitecture*, 1992.