2016 IEEE 19th International Symposium on Real-Time Distributed Computing

Increasing the Code Density of Embedded RISC Applications

H. Lozano and M. Ito
Electrical and Computer Eng. Department
University of British Columbia
Vancouver, Canada

***Abstract*—Cache misses are problematic in real-time systems because they are unpredictable. Therefore, smaller codes that can fit in on-chip fast memories will help get rid of caches. However, reducing RISC applications code size is not an easy task because traditional RISC instruction sets have fixed length instructions compared to the higher density CISC instruction sets that use variable length instructions. A solution used in modern RISC processors is to combine mixed length instruction sets to achieve code densities comparable to if not higher than CISC without any performance degradation. To further enhance the code density of mixed length RISC instruction sets we add a number of 8-bit instructions to a 16-bit RISC instruction set and use novel techniques such as static compression of immediate values to compensate for the limited number of bits available for instruction encoding. The addition of 8-bit instructions combined with the proposed novel techniques reduces RISC code size by as much as 30% without degrading performance while using less than 5% additional hardware. Total processor power is reduced by 10%.**

Keywords—Code density; RISC embedded processors efficiency; Cache-less systems

I. INTRODUCTION

Denser codes generate smaller programs that can fit in smaller memories. Smaller memories not only cost less but also consume less dynamic and static power which is an important feature in embedded systems. Smaller memories also have faster access time; this allows the code to execute from on-chip tightly coupled memories instead of a combination of slower off-chip memories and caches, which are problematic in real-time systems because of the unpredictability of cache misses. Denser codes also consume less dynamic energy to fetch and decode instructions which translates to longer battery life [14].

Prior works reported that mixed length CISC instruction sets yield anywhere from 25% to 35% denser code than fixed length RISC instruction sets [1], [12]. However, modern RISC instruction sets combine different length instructions to yield code size comparable to if not higher than CISC without any significant performance loss [11]. Actually, replacing 16-bit instructions with a mix of 16-bit and 32-bit instructions in [11] generated a 1.07x increase in performance as a result of the reduced cache misses. Mixed length RISC instruction sets have the potential to produce denser code than CISC while maintaining RISC performance advantage.

To further enhance the code density of mixed length RISC instruction sets we propose to add 8-bit instructions to 16-bit and 32-bit RISC instruction sets. The 8-bit instructions require relatively minor modifications to the decoder compared to if we were to increase instructions length beyond 32 bits. The instruction decoder is also expected to use less power to fetch and decode 8-bit instructions compared to 16-bit instructions because there are fewer bits to decode and less information to extract from the instruction. The downside of using shorter instructions is that they can only be used to encode simple operations due to the limited number of available bits. For example, Intel x86 8-bit instructions are used only to encode simple operations that use a single operand such as pushing/popping a register on/off the stack and index increment/decrement (www.intel.com).

In order to expand the usability of 8-bit instructions, we need to increase the range of values that can be encoded using limited number of bits. We propose to: (a) statically compress immediate operands by a fixed amount in order to increase the max values that can be encoded using a limited number of bits and (b) replace explicit operand registers with implicit register aliases in order to free up additional bits for encoding the remaining operands. The proposed techniques increase the pool of candidate operations that can be implemented using just eight bits by up to 60% thus increasing the code density by 20% on average without incurring any performance loss. The reduction in code size increases the potential for embedded programs to be run from on-chip tightly coupled memories instead of a combination of external memories plus caches. Simulation of a number of embedded benchmarks using an FPGA prototype show that although the decoder size increased by approximately 5%, total processor power consumption decreased by approximately 10% mainly from reducing access to program memory.

In section 2 we present the benchmarks and simulation environment. In section 3 we discuss the motivation for improving code density. In this section, we compare the code density of a number of embedded MiBench and DSP benchmarks using three different target processors, one RISC and two CISC. In section 4 we present the 8-bit instructions and introduce the proposed techniques to increase the usability of 8-bit instructions. Results are discussed in Section 4 and Section 5 describes related work. Finally, the conclusion is presented in Section 6.

2375-5261/16 $31.00 © 2016 IEEE
DOI 10.1109/ISORC.2016.33

II. SIMULATION ENVIRONMENT

For code size comparison we selected three different embedded processors: the 32-bit ARM Cortex-M4 RISC processor, the 32-bit Intel i386 CISC processor and the 32-bit Freescale m528x CISC processor. The ARM Cortex-M4 processor uses a combination of 16-bit Thumb and 32-bit Thumb2 instruction sets. The Intel x86 processor uses the CISC x86 instruction set with instruction length that varies from 8-bit to 120-bit. The Freescale m528x processor uses the variable length Coldfire V2 CISC instruction set which is based on the popular Motorola m68k instruction set and includes 16-bit, 32-bit and 48-bit instructions.

TABLE I lists a summary of the three target processors, the instruction set used by each processor, the compiler version and the compiler command line used to compile the benchmarks for each target processor. Intel uses a native GNU GCC compiler that runs on an Intel based host machine whereas ARM and Freescale use a GNU GCC cross-compiler, the *arm-none-eabi-gcc* and the *m68k-elf-gcc* compilers respectively. All benchmarks are statically linked and compiled with full optimization (-O3). The low count of floating point operations in the simulated benchmarks does not justify the use of dedicated hardware floating point units. Instead, software floating point emulation is used. The MiBench benchmarks are linked using a semi-hosted configuration and the DSP benchmarks, which do not reference any external library functions, are linked using a baremetal configuration without OS support. The semi-hosted and the ARM baremetal linker scripts are copied from the corresponding GCC distribution and modified to model the same processor configuration used for linking the remaining DSP benchmarks. The Intel and Freescale baremetal linker scripts are custom scripts that we created to model a basic baremetal system with the same processor configuration common between all three target platforms. The main characteristic of the common processor configuration is a 2-MB combined instruction/data memory.

TABLE I. SUMMARY OF PROCESSOR AND COMPILER INFO.

	ARM	**Intel**	**Freescale**
Processor	Cortex-M4	i386	m528x
Instruction Set	Thumb/Thumb2	x86	Coldfire V2
Instruction Length	16-bit 32-bit	8-bit to 120-bit	16-bit 32-bit 48-bit
Compiler Version	GNU GCC 4.9.3	GNU GCC 4.4.3	GNU GCC 4.1.1
Command Line Options (semi-hosted)	-O3 --specs=rdimon.specs -mfloat-abi=soft -mthumb -mcpu=cortex-m4	-O3 -static -nostdlib -msoft-float -march=i386 -mtune=i386 -Tcygmon.ld	-O3 -nostdlib -msoft-float -m528x -Tm5282evb-ram.ld
Command Line Options (baremetal)	-O3 --specs=nosys.specs -mfloat-abi=soft -mthumb -mcpu=cortex-m4	-O3 -static -nostdlib -msoft-float -march=i386 -mtune=i386 -Tnosys.ld	-O3 -nostdlib -msoft-float -m528x -Tnosys.ld

TABLE II. BENCHMARK LIST SHOWING TOTAL NUMBER OF INSTRUCTIONS EXECUTED AND ELF FILE SIZE.

Benchmark	**Category**	**ELF Executable Size (bytes)**		
		ARM	***Intel***	***Freescale***
basicmath	automotive	80,492	86,940	65,304
bitcount	automotive	71,560	62,060	48,740
qsort	automotive	92,524	96,760	59,092
susan.smoothing	automotive	96,808	100,588	61,957
susan.edges	automotive	96,808	100,588	61,957
susan.corner	automotive	96,808	100,588	61,957
dijkstra	network	91,620	95,800	58,080
patricia	network	97,476	102,349	67,408
blowfish.encode	security	54,888	37,316	31,632
blowfish.decode	security	54,888	37,316	31,632
rijndael.encode	security	103,044	92,652	84,136
rijndael.decode	security	103,044	92,652	84,136
sha	security	81,100	61,964	47,920
ADPCM.encode	telecom	70,892	60,156	46,596
ADPCM.decode	telecom	70,892	60,156	46,596
CRC32	telecom	72,772	62,732	48,820
2K Radix-2 FFT	telecom	76,968	71,788	57,168
2K Radix-2 IFFT	telecom	76,968	71,788	57,168
stringsearch	office	84,004	72,748	59,452
jpeg.encode	consumer	156,740	218,936	160,076
jpeg.decode	consumer	157,772	203,608	151,960
4K Radix-2 FFT	DSP	68,700	51,128	48,004
4K Radix-4 FFT	DSP	68,972	51,480	48,280
4K Radix-8 FFT	DSP	69,948	52,568	49,280
1K x 1K Matrix Multiply	DSP	12,035,836	12,018,168	12,015,072
1K x 1K Matrix Add	DSP	12,035,868	12,018,104	12,015,020
1K x 1K Matrix Transpose	DSP	12,035,780	12,018,136	12,015,036
FIR Filter	DSP	39,924	22,264	19,172
MiBench median size		84,004	86,940	59,092
DSP median size		69,948	52,568	49,280
All median size		82,552	79,844	58,586

TABLE II lists the benchmarks used in the evaluation as well as the ELF executable size per benchmark for each target processor. All debug and unneeded symbols are removed from the final ELF executables using the *strip* utility. TABLE II results interestingly show that the median size of an Intel DSP benchmark is 25% smaller than the median size of an ARM DSP benchmark which matches previous results that show CISC executables are smaller in size than RISC executables. In contrast, the median size of an Intel MiBench benchmark is approximately 4% larger than the median size of an ARM MiBench benchmark which makes the overall median size of an Intel benchmark just 3% smaller than ARM. These results are surprising because we expect Intel that has the highest number of options for instruction length to produce the smallest binaries, which is not the case. Actually, Freescale, which has only three different instruction sizes produced the smallest binaries of all three processors. Freescale benchmarks are not only 30% smaller than ARM benchmarks but also 30% smaller than Intel benchmarks. In the Motivation section we will explore in detail the difference in binary size between the three different platforms. At this point, we would like to point out that ELF executable size vary widely from one platform to another based on several factors such as target processor and, therefore, is not a reliable metric to measure code size especially when targeting embedded systems.

III. MOTIVATION

Modern RISC instruction sets use a mix of different length instructions, similar to CISC, to increase RISC programs code density. In [11], 16-bit instructions are combined with 32-bit instructions to reduce RISC programs code size by 33% compared to using just fixed length 32-bit instructions. In this section, we compare the code size performance of ARM mixed length RISC instruction sets to Intel x86 and Freescale Coldfire CISC instruction sets. Based on the code size comparison results we highlight what is needed to increase the code density of variable length RISC instruction sets. First, we explore the difference in binary image size between the different platforms and how to accurately measure code size.

A. Binary Image Size

The ELF executable file structure varies significantly from one platform to another depending on the OS, compiler, etc. For example, ARM ELF files have three program headers and fifteen section headers whereas Intel ELF files have four program headers and eight section headers and Freescale ELF files have one program header and six section headers [15]. The ELF sections contain system specific debug and symbol data in addition to program instructions which can inaccurately exaggerate the binary file size. In production releases, all unnecessary data are stripped from the ELF executable before the file is loaded into Flash memory or ROM.

Similarly, in semi-hosted systems the size of external libraries that are linked with applications vary widely from one processor platform to another which can skew the code size measurement results. Therefore, to get an accurate measure of the actual code size independent of the ELF and OS overheads we need to measure the code size without linking. In the case of the MiBench benchmarks, code is distributed among several files and, therefore, cannot be measured without linking which complicates the process as some benchmarks have multiple files distributed among several directories. Therefore, we only measure the size of the custom DSP benchmarks which do not require any linking. The UNIX *objcopy* utility with the stripping of debug and symbols option turned on is used to generate the file binary image. Results are plotted in Figure 1.

In a sharp contrast to TABLE II results that show ARM ELF executables are larger than Intel and Coldfire executables, Figure 1 results clearly show that ARM code is actually 15% and 10% smaller than Intel and Freescale code respectively. The only exceptions are the Matrix Add and FIR benchmarks and, to a lesser degree, the Matrix Multiply benchmark. A detailed analysis of the Matrix Add benchmark reveals that the total number of bytes per instructions is actually smaller in ARM than Intel. The major difference is that ARM uses 16 bytes to store the local variables in program memory instead of data memory which increases the total code size by 16 bytes. Although Intel and Freescale use complex CISC instructions that decrease the average number of instructions per operation, the byte count per loop iteration is lower in ARM than Intel and Freescale. The reason is that Intel and Freescale complex load-operate instructions are at least eight and six bytes long respectively whereas ARM combination of load and operate instructions use only four bytes.

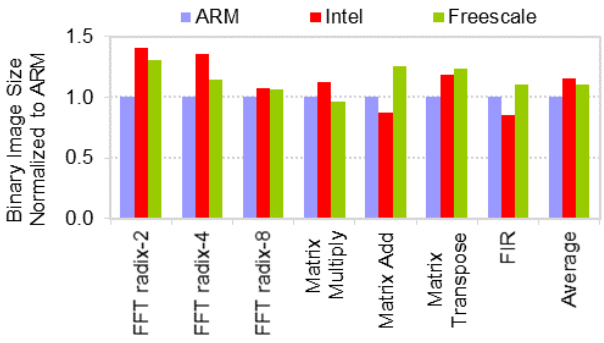

Figure 1. Unlinked code size normalized to ARM binary images size using a baremetal configuration.

B. Finding Candidate Instructions for Size Reduction

The previous results indicate that a variable length RISC instruction set has the potential to yield smaller code than CISC. However, the difference is just 15% which shows that there is room for further improvement. In this section we investigate the potential to further increase the code density of mixed length RISC instruction sets by implementing as many simple operations as possible using 8-bit instructions instead of 16-bit or 32-bit instructions.

However, not any operation can be implemented using just eight bits due to the limited number of bits available for encoding all operands as well the limited number of bits per operand which limits the range of values that can be represented by each operand. For example, a complex ALU operation such as multiply-accumulate uses at least three operands and, therefore, cannot be encoded with just eight bits. Following a detailed analysis of the most common operations in the simulated benchmarks, we identified three main categories of candidate operations that can be encoded using eight bits:

- **ALU operations that use only two operands:** the source operand can be either a register or an immediate value and the destination operand must always be a register. Examples of operations in this category are: value compare, register move and signed/unsigned bit extend.
- **ALU operations that use three operands and the destination register is same as one of the source registers:** the second operand can be either a register or an immediate value. Examples of operations in this category are: add, subtract, multiply and bit logical operations that take two operands.
- **Load and store operations that implicitly use special registers as a base address register:** an immediate offset is added to the PC or SP to generate the effective address of the memory location where a data array pointer is stored or the effective address of the array location in memory where the actual data is stored. Examples of operations in this category are: load PC-relative and load/store SP-relative operations.

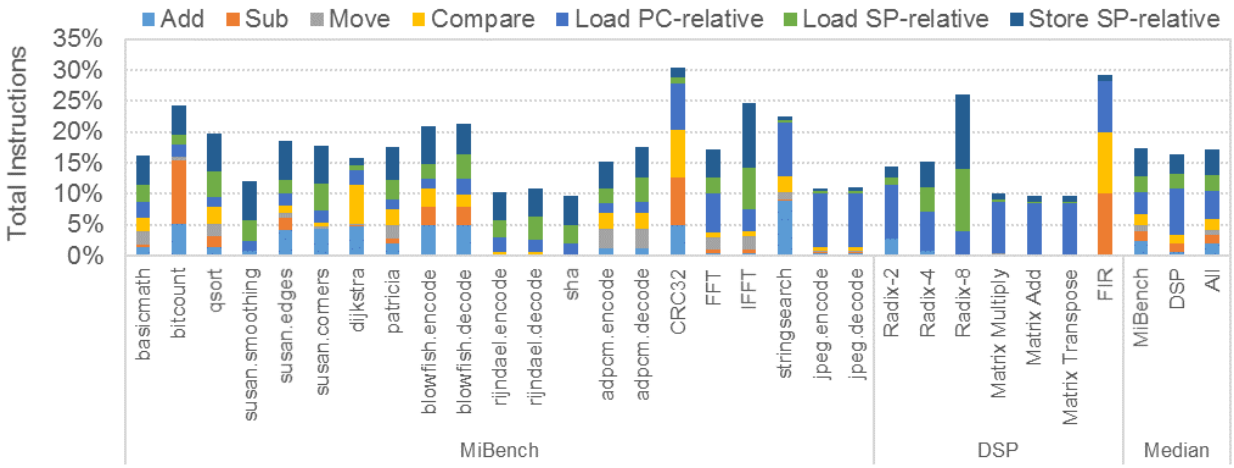

Figure 2. Static distribution of candidate instructions for 8-bit conversion as a percentage of total static 16-bit Thumb instructions.

Figure 2 shows the distribution of candidate 16-bit Thumb instructions that are suitable for 8-bit conversion as a percentage of total 16-bit Thumb instructions. These instructions are selected based on the three categories introduced earlier. The first two categories of candidate instructions that are suitable for 8-bit conversion are ALU operations that have a two to three operands and each operand value is between zero and three so that the operand can be encoded using only two bits. The main observations for these categories of candidate instructions are:

- There are two types of add/subtract instructions: register+register which are used for normal ALU operations and register+immediate which are mostly used to increment/decrement loop counters. The register+immediate instructions are especially suited for 8-bit implementation because the immediate value can be encoded using just one bit, for example incrementing/decrementing a loop counter by one, which frees up more bits for encoding the remaining operands. The median percentage of candidate add/subtract instructions is around 8% but some benchmarks that contain long data accumulating loops such as *bitcount* have twice this percentage.
- Logic instructions which include among others, compare instructions, bit extend instructions, bit logic instructions are not commonly used in embedded benchmarks. The median percentage of logic instructions is 2.5% but can go up to as high as 9% in benchmarks that manipulate individual data bits using short repetitive loops such as *CRC32*.
- Move immediate and move register are good candidate instructions for size reduction because most often operand values used in these instructions are small and can be encoded using just few bits. The median percentage of move immediate and move register instructions is around 10% in MiBench benchmarks and 8% overall.

The remaining category of candidate instructions are load PC-relative and load/store SP-relative instructions that implicitly use special registers PC and SP as a base register. These type of instructions are used to access arrays which are commonly used in embedded applications to store data. The load PC-relative and load/store SP-relative instructions are good candidate for size reduction because the Program Counter (PC) and Stack Pointer (SP) don't need to be explicitly encoded in the instruction, which frees up valuable bits that can be used to encode the remaining operands if needed. The main observations for this category of candidate instructions are:

- The load PC-relative instructions are used to initialize array pointers at the start of the program. In contrast, the load/store SP-relative instructions are used mainly within the loop body to load/store results into an array. The load/store SP-relative instructions are more commonly used in DSP benchmarks that store large arrays of data on the stack, such as Radix-8 FFT.
- The total combined median percentage of load PC-relative and load/store SP-relative operations is approximately 8% the total number of static instructions (17% if the ALU instructions are added). These percentages are relatively low because the generated code is compiled using the standard ARM Thumb/Thumb2 instruction sets; adding support for 8-bit instructions will allow the compiler to optimize instruction scheduling in order to reduce code size.
- Despite that the PC and SP values are not explicitly encoded in the instruction, the total number of bits that can be used to encode the memory address offset is four which can encode a max offset of 16 bytes. The challenge is to find ways to increase the range of max offset values that can be encoded using only two to four bits. The same applies to ALU operations, especially, when one operand is an immediate value. On the other hand, register values for operands can be controlled by the compiler by using a subset of the available registers.

IV. PROPOSED SOLUTIONS

Based on the previous analysis, adding 8-bit instructions to the 16-bit instruction sets is expected to improve code density. However, adding new instructions to an existing instruction set is a very difficult task as most instruction sets have very few, if not any, number of free opcodes that can be used. Thumb has only two unused opcodes (0x0100010100 and 0x11011110). However, these opcodes are eight bits long and therefore cannot be used for 8-bit instructions. Furthermore, some Thumb instructions are either redundant, in the sense that they can be replaced by another instruction without affecting performance, or are rarely used. The following are examples of redundant instructions: (1) "Add Immediate to SP" instruction which can be replaced with the "Generate SP-relative Address" instruction and (2) the "Load Multiple Registers" and the "Store Multiple Registers" instructions that are used to load data from the stack and store data on the stack respectively can be replaced with POP and PUSH instructions. The "Supervisor Call" instruction, which performs a software interrupt, is an example of a rare instruction that is mainly used by the operating system (OS). A number of opcodes can be freed and re-used for 8-bit instructions by reassigning some of the unused, redundant and rarely used instructions in the Thumb instruction set.

A. Offset Compression

Single byte instructions contain eight bits shared between the opcode and the operands fields. A minimum of four bits are reserved for the opcode field which leaves just four bits for the operand fields. The four bits are divided between a maximum of two fields and each field can be no longer than two bits which limits the number of registers to four (*r0*, *r1*, *r2* or *r3*). For load and store instructions the Program Counter (PC) and Stack Pointer (SP) are implied and therefore don't need to be hard coded in the instruction. This leaves two bits for the destination/source register and two bits for the immediate address offset. The limited number of registers that can be addressed using two bits may potentially increase the pressure on the RF and hurt performance. However, our simulation results show that the life span of registers in embedded applications is short (50% of registers have a lifespan of four or less cycles). Therefore, a small number of registers can be used without impacting performance. The real challenge is coding immediate values using just two bits.

In order to increase the range of immediate values that can be represented using just two bits we propose to compress each value by four at compile time. The instruction decoder will automatically expand the immediate value by shifting it left by two or discarding the least two significant bits. An analysis of the load PC-relative and load/store SP-relative immediate offset values in the simulated benchmarks shows that 14% of the offsets are 12 bytes or less which can be encoded using two bits plus compression, 27% of the offsets are 28 bytes or less which can be encoded using three bits plus compression and 58% of the offsets are 60 bytes or less which can be encoded using four bits plus compression. Thus by using a 4-bit field to encoded immediate offset values roughly 60% of the load PC and load/store SP relative instructions can be converted to 8-bit instructions.

The drawback of multiplying offset values by four is that offsets can only represent multiple of four values such as 0, 4, 8, 12, 16, etc. For load and store operations this means that we can only access word aligned locations in memory. To access half-word and byte locations the compiler needs to use regular 16-bit or 32-bit instructions. The majority of memory access in the simulated benchmarks (70%) are word access and therefore the loss of precision is not expected to affect the advantage provided by the compression technique.

B. Access Register

To further increase range of immediate offsets used in load PC-relative and load/store SP-relative instructions, we propose to reduce the number of bits used to store the destination/source register from two bits to one bit. This implies that the destination/source register can be either *r0* or *r1*. However, limiting the choice of registers to two may potentially increase the pressure on the RF and hurt performance. In order to reduce the pressure on the RF, we propose to replace the destination/source with a dedicated Access Register (AR). The AR can be either a separate standalone physical register or an alias to one of the general purpose register located in the RF. There are pros and cons for each approach:

- Using a dedicated physical register for AR simplifies the instruction decoding (RF address is not needed) and speeds up the operand read access. This option also reduces the pressure on the RF by bypassing the RF altogether. On the downside, the instruction set needs to be modified to use a dedicated register.
- Using an alias requires some modifications to the instruction decoder to translate the AR to the corresponding RF register. The downside is that the register number that the alias is mapped to must be hard coded in the instruction decoder. However, 16-bit Thumb instructions can only address the first eight registers in the RF (*r0* to *r7*) which provides enough choices to map the AR to a higher register number that is not used by the 16-bit instructions (*r8* to *r14*).

When a single AR is used, there is no need to explicitly encode its value in the instruction as it will be implicitly mapped to the assigned RF register by the decoder. The downside is that the mapping is hardcoded in hardware as mentioned earlier. In this case, the offset field can be expanded to four bits to support a range of values from 0 to 60 bytes in a 4-byte increment (0, 4, 8, …, 60 bytes) with compression and from 0 to 15 bytes without compression. Simulation results show that approximately 60% of the of the 16-bit load PC-relative and load/store SP-relative instructions have an offset value that is within and therefore can be converted to 8-bit. However, embedded benchmarks contain two to three arrays per function on average and each array requires a separate AR per array. When two AR are used, only three bits are left to encode the offset value. A 3-bit offset field can represent a max offset value of 28 bytes. The percentage of offset values ranging from 0 to 28 bytes in the simulated benchmarks is around 30%. This limits the number of candidate instructions by half compared to using a 4-bit offset field.

V. RESULTS

A. Code Density

Five 8-bit instructions, shown in Figure 3, are added to the Thumb instruction set. The four categories of operations that these instructions represent are:

- load PC-relative and store SP-relative with compression and two access registers *ar0* and *ar1* (because of the limited number of 8-bit instructions that can be added to the Thumb instruction set we only chose to implement the store SP-relative which occurred more frequently in the simulated benchmarks)
- compare immediate without compression
- Add immediate without compression
- Move zero to register

The total percentage of the five 8-bit instructions in the simulated benchmarks is around 34%. We chose not to compress add and compare immediate values because most of these values are relatively small and can be encoded using two bits. The three least significant bits of the load PC-relative and store SP-relative instructions hold a compressed immediate offset that gets shifted left by two positions or truncated at decode time. The fourth bit holds the AR number which can be either zero for *ar0* or one for *ar1*. The two least significant bits of add, move and compare instructions hold the immediate operand value. The other two bits hold the destination register number which can be any of the following registers *r0*, *r1*, *r2* or *r3*. In the case of the add instruction the destination and source registers are the same and therefore use a single operand field in the instruction code. The compare instruction has only two operands, an immediate source value and a destination register. The move instruction has an immediate and a register source operand.

The add immediate and move zero instructions share the same opcode. When the immediate value is zero the decoder recognizes the instruction as a move zero instruction, otherwise the decoder treats the instruction as an add immediate instruction. An analysis of the simulation results shows that approximately 40% of the move immediate instructions are used to initialize registers to zero. Alternatively, the add immediate instruction can be used as a move immediate instruction by making sure that the destination/source register is cleared before the addition is performed. Compressing the source and/or destination register number allows the compiler to use higher numbered registers for some specific instructions and free up the lower numbered registers for the remaining instructions thus helping to decrease the pressure on the RF in 16-bit instructions.

```
0x1010xxxx     Load PC-relative
0x1101xxxx     Store SP-relative
0x1001xxxx     Compare immediate
0x1111xxxx     Add immediate
0x1111xx00     Move zero to [r0 : r3]
```

Figure 3. List of 8- bit instructions added to the existing ARM Thumb ISA.

```
<main>:
    0:  2300        movs      r3, #0
    2:  4a07        ldr       r2, [pc, #28]
    4:  f832 1013   ldrh.w    r1, [r2, r3, lsl #1]
    8:  4a06        ldr       r2, [pc, #24]
    a:  f832 2013   drh.w     r2, [r2, r3, lsl #1]
    e:  440a        add       r2, r1
   10:  4905        ldr       r1, [pc, #20]
   12:  f821 2013   strh.w    r2, [r1, r3, lsl #1]
   16:  3301        adds      r3, #1
   18:  4a04        ldr       r2, [pc, #16]
   1a:  4293        cmp       r3, r2
   1c:  d1f1        bne.n     2
   1e:  4770        bx        lr

<main>:
    0:  fc          movs      r3, #0
    1:  a7          ldr       ar0, #7 (#28/4)
    2:  f832 1013   ldrh.w    r1, [ar0, r3, lsl #1]
    6:  a6          ldr       ar0, #6 (#24/4)
    7:  f832 2013   ldrh.w    r0, [ar0, r3, lsl #1]
    b:  440a        add       r0, r1
    d:  ad          ldr       ar1, #5 (#20/4)
    e:  f821 2013   strh.w    r0, [ar1, r3, lsl #1]
   12:  dd          adds      r3, #1
   13:  a4          ldr       ar0, #4 (#16/4)
   14:  4293        cmp       r3, ar0
   16:  d1f1        bne.n     2
   18:  4770        bx        lr
```

Figure 4. Assembly code for the Matrix Add benchmark using (top) ARM Thumb and (bottom) the modified ARM Thumb with 8-bit instructions.

Figure 4 illustrates the use of 8-bit instructions by comparing the assembly code for the Matrix Add benchmark generated using (top) the original ARM Thumb instruction set and (bottom) the modified Thumb instruction set with 8-bit instructions. The original Thumb version uses a total of 34 bytes whereas the modified version uses a total of 26 bytes, a 24% reduction in code size. In Figure 4 example, the load PC-relative AR aliases are mapped to RF registers *r6* and *r7* respectively which frees up the lower numbered registers r0 to r7 for regular Thumb instructions encoding. Although IPC performance does not change as instruction latency is the same, power consumption is expected to decrease as 24% fewer bytes are fetched and decoded. Consuming less power while delivering the same performance is expected to reduce the processor total energy consumption.

Code density increased by 20% on average as shown in Figure 5. Benchmarks with a large percentage of load PC-relative and store SP-relative instructions such as *Radix-8 FFT* experienced a 30% increase in code density. Benchmarks with simple arithmetic operations in long loops such as *bitcount and Matrix Add* benefited more from the ALU instructions than the load/store instructions . The lowest increase in code density is around 9%; considering that we have just added five instructions, there is still potential to further increase code density by redesigning the instruction set from scratch to support additional 8-bit instructions. Reducing applications size by 20% to 30% increases the potential for these applications to be moved from slow off-chip memory and loaded entirely into on-chip RAM thus getting rid of caches which are problematic in real-time systems because of the non-deterministic aspects of cache misses. Alternatively, if the application already fits in the on-chip RAM, reducing its size allows the RAM to be downsized to lower cost and power.

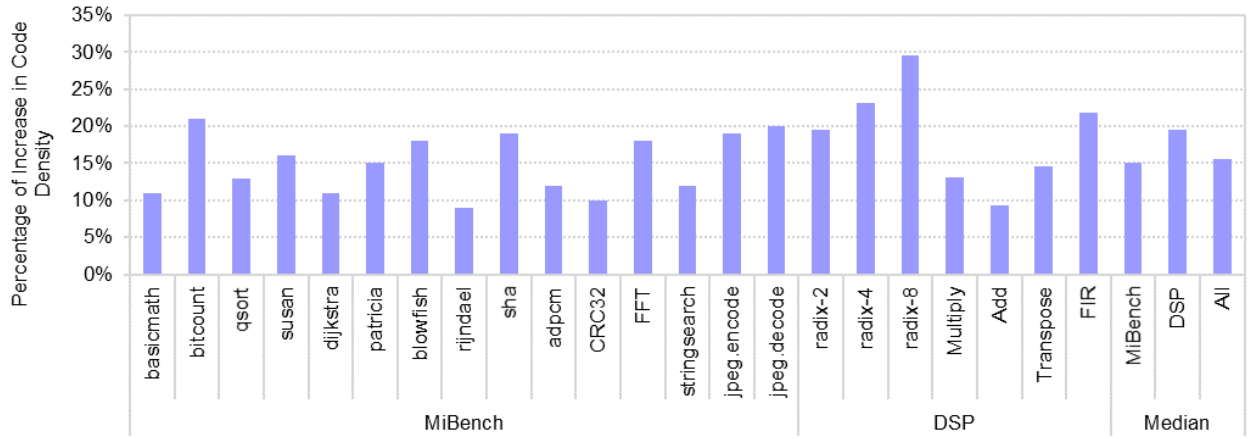

Figure 5. Percentage of increase in code density as a result of using one-byte instructions.

B. *Performance*

Higher code density is often achieved at the expense of performance, especially, when replacing 32-bit long instructions with shorter 16-bit instructions [8], [10]. However, a modest 1.07x improvement in the performance of a dual issue RISC processor is recorded when fixed length 32-bit instructions are replaced with a mix of 16-bit and 32-bit instructions [11]. The performance improvement is a result of the reduction in L1 cache miss rate which, consequently, improves load time. In our evaluation, embedded processors are used in a cache-less configuration and therefore our simulation results show no performance improvement [2]. Also, the newly added 8-bit instructions have the exact same latencies as the 16-bit instructions that they replace and therefore they have no impact on performance.

We followed a very cautionary approach by adding few selected instructions to an existing instruction set that has very few unused opcodes. The drawback is that we may not have exploited the full potential of 8-bit instructions for code size reduction. If we were to take a more aggressive approach and redesign the instruction set from scratch, we would be able to replace entire categories of 16-bit instructions with equivalent 8-bit instruction and increase the opportunity for the compiler to schedule instructions more efficiently to reduce code size and improve performance.

C. *Power Consumption*

A model of the ARM Cortex-M4 processor that supports 8-bit instructions is implemented on an Altera Cyclone-V FPGA. A number of benchmarks are simulated on the FPGA platform to collect power measurements. TABLE III lists a summary of the FPGA implementation results for the base Cortex-M4 processor and the modified Cortex-M4 with support for 8-bit instructions. The last column in TABLE III lists the percentage of change for each entry. Dynamic power estimates were generated using Altera PowerPlay tool which is part of Altera Quartus-II design suite. Signal activity results generated by Altera-Modelsim gate level simulation with back-annotation of a number of selected benchmarks were input into PowerPlay to give accurate power estimates.

The decoder size increased by around 5% and the FPGA global connectivity resources by 3% resulting in an increase of approximately 2% in the processor total number of FPGA logic elements (LE). This is a very modest increase compared to if we were to add support for instructions longer than 32 bits similar to Coldfire 48-bit or Intel 120-bit instructions.

The single issue processor average dynamic power dissipation is approximately 31.2 mW when operated at 60 MHz which gives a power rating equal to 0.52 mW/MHz. As expected, power consumption benefited generously from reducing the code size as average total power consumption decreased by approximately 10% despite a 2% increase in total number of LE. Most of the saving in power consumption was achieved by reducing the number of accesses to the shared instruction/data memory, which is by far the highest power consuming module in the processor (52% of total power). The additional power consumed by the decoder does not impact total power consumption because it consumes less than 10% of the total processor power.

The minimal increase in hardware cost combined with the 20% average reduction in code size and 10% reduction in power consumption shows the significant potential of the proposed techniques.

TABLE III. FPGA IMPLEMENTATION RESULTS (RESOURCES + POWER CONSUMPTION) FOR THE BASE ARM PROCESSOR (ARM) AND THE BASED ARM WITH 8-BIT INSTRUCTIONS (ARM+).

Resource	ARM Cortex-M4	ARM Cortex-NM4 + 8-bit	% Change
Logic elements (LE)	1,673	1,695	+2%
ALU	1024	1024	NC
Decoder	218	229	+5%
RF	85	85	NC
Connectivity	346	357	+3%
Speed (MHz)	60	60	NC
Dynamic power (mW/MHz)	0.52	0.47	-10%

VI. Related Work

A comparison of code density, performance, power and energy between CISC and RISC using a number of commercial high end embedded processors is presented in [16]. The authors conclude that the type of instruction set used has no influence on code density, i.e. will neither harm nor benefit code density. In [12] the authors measured the code density of a very simple program (Linux logo banner) on 21 different processors and concluded that CISC has higher code density than RISC. The code density results published in these works are based on: (1) the size of the ELF binaries on a fully hosted system, i.e. with full OS support and (2) a fixed length RISC instruction set. Our own simulation results have clearly shown that ELF binary size is not a good indication of code density. The text segment size need to be measured independent of the ELF overheads. To the best of our knowledge no code density comparison between a mixed length RISC instruction set and CISC instruction set has ever been published.

The benefit of using shorter instructions in order to increase code density is studied in [8] and [11]. In [8] the authors replaced 32-bit ARM instruction set with 16-bit Thumb instruction set to obtain a 50% reduction in code size with minimal performance loss. In [11], replacing 32-bit ARM instruction set with 16-bit+32-bit instruction sets improved performance by 1.07x and decreased code size by 33%. These are simple straightforward solutions that replace an instruction set with another more efficient instruction set. The results published in [11] are particularly promising because they show that code size can be reduced without deteriorating performance.

Code compression is a very popular technique for increasing code density. Either 32-bit instructions [3], [4], [5] [10] or 16-bit instructions [7], [9] are compressed at compile time and decompressed at runtime using dedicated hardware. Compression techniques vary from simple static heuristics [3], [4], [5], [7] to sophisticated dynamic algorithms [9] [10]. However, all compression techniques compress instruction segments, i.e. a block of instructions that vary in length, based on an analysis of the instruction stream; the actual instruction encoding is not affected. In addition, none of the compression techniques can be used in a low cost deeply embedded processor because they all require additional pipeline stages which will increase power consumption. Also, compressed code is usually stored in caches to hide the decompression latency; however, low cost embedded processors do not, or very rarely, contain caches. A dynamic technique for compressing no-operation (*nop*) in static multi-issue processors (VLIW) is proposed in [14]. The technique reduces the code size by 44%. However, the percentage of *nop* in ARM code is less than 1% the total number of executed instructions.

VII. Conclusion

In order to increase the code density of RISC programs, five 8-bit instructions that represent different memory access and ALU operations are added to ARM 16-bit Thumb instruction set. Statically compressing the immediate value at compile time quadruples the range of immediate values that can be encoded in 8-bit instructions thus increasing their usability. In order to further increase the range of immediate values as well as reduce the pressure on the RF, destination/source registers are replaced with register aliases that get mapped to actual RF registers at decode time. The 8-bit instructions in combination with the proposed techniques increase the code density of embedded benchmarks by 20% on average. Programs with frequent loops and large data arrays benefited the most and saw their code size decrease by up to 30%. A number of embedded benchmarks are simulated on a low cost FPGA using a model of an ARM Cortex-M4 embedded processor modified to support 8-bit instructions. Results show that although the total number of FPGA hardware elements increased by approximately 2%, total power consumption decrease by 10% without incurring any performance loss.

References

[1] J. Davidson and R. Vaughan, “The effect of instruction set complexity on program size and memory performance,” in *Proc. 2nd ACM Symp. Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 60–64.

[2] P. Steenkiste, “The impact of code density on instruction cache performance,” in *Proc. 16th IEEE/ACM Int. Symp. Computer Architecture*, 1989, pp. 252–259.

[3] A. Wolfe and A. Chanin, “Executing Compressed Programs on Embedded RISC Architecture”, in *Proc. 25th Ann. Int. Symp. Microarchitecture*, 1992, pp. 81-91.

[4] S. Liao et al., “Code Density optimization for Embedded DSP Processors Using Data Compression Techniques”, in *Proc. 15th Conf. Advanced Research in VLSI*, 1995, pp. 272-285.

[5] C. Lefurgy et al., ‘‘Improving code density using compression techniques”, in Proc. 30th Annu. Int. Symp/ on Microarchitecture, 1997, pp. 194-203.

[6] S. Y. Liao et al., "Code density optimization for embedded DSP processors using data compression techniques," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 1998, pp.601-608.

[7] K. D. Cooper and N. McIntosh, “Enhanced code compression for embedded RISC processors,” in *Proc. Conf. Programming Language Design and Implementation*, 1999, pages 139-149.

[8] A. Halambi et al.,"A design space exploration framework for reduced bit-width Instruction Set architecture (rISA) design," in *Proc. 15th Int. Symp. System Synthesis*, 2002, pp. 120-125.

[9] X. H. Xu, S. R. Jones, C. T. Clarke, "ARM/THUMB code compression for embedded systems," in *Proc. 15th Int. Conf. Microelectronics*, 2003, pp. 32-35.

[10] M. Breternitz, H. Hum, R. Peri, J. Pickett, Y. Wu, "Enhanced code density of embedded CISC processors with echo technology," in *Proc. 3rd IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis*, 2005, pp.160-165.

[11] C. Sudanthi et al., "Performance analysis of compressed instruction sets on workloads targeted at mobile internet devices," in *Proc. IEEE Int. SOC Conference*, 2009, pp.215-218.

[12] V. M. Weaver and S.A. McKee, "Code density concerns for new architectures," in *Proc. IEEE Int. Conf. Computer Design*, 2009, pp.459-464.

[13] V. Guzma et al., "Effects of loop unrolling and use of instruction buffer on processor energy consumption," in Proc. Int. Symp. System on Chip, 2011, pp. 82-85.

[14] J. Helkala et al., "Variable length instruction compression on Transport Triggered Architectures," in *Proc. Int. Conf. Embedded Computer Systems: Architectures, Modeling and Simulation*, 2014, pp.149-155.

[15] SystemV – Application Binary Interface Edition 4.1 (online).

[16] E. Blem et al., "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures," in *Proc. Int. Symp. High Performance Computer Architecture*, 2013, pp. 1-12.