Bit Section Instruction Set Extension of ARM for Embedded Applications

Bengu Li
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
libengu@cs.arizona.edu

Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
gupta@cs.arizona.edu

ABSTRACT

Programs that manipulate data at subword level, i.e. bit sections within a word, are common place in the embedded domain. Examples of such applications include media processing as well as network processing codes. These applications spend significant amounts of time packing and unpacking narrow width data into memory words. The execution time and memory overhead of packing and unpacking operations can be greatly reduced by providing direct instruction set support for manipulating bit sections.

In this paper we present the *Bit Section eXtension* (BSX) to the ARM instruction set. We selected the ARM processor for this research because it is one of the most popular embedded processor which is also being used as the basis of building many commercial network processing architectures. We present the design of BSX instructions and their encoding into the ARM instruction set. We have incorporated the implementation of BSX into the Simplescalar ARM simulator from Michigan. Results of experiments with programs from various benchmark suites show that by using BSX instructions the total number of instructions executed at runtime by many transformed functions are reduced by 4.26% to 27.27% and their code sizes are reduced by 1.27% to 21.05%.

Categories and Subject Descriptors

C.1 [**Computer Systems Organization**]: Processor Architectures; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms

Algorithms, Measurement, Performance

Keywords

bit section operations, multimedia data, network processing

CASES 2002, October 8–11, 2002, Grenoble, France.

1. INTRODUCTION

Programs for embedded applications frequently manipulate data represented by bit sections within a single word. The need to operate upon bit sections arises because such applications often involve data which is smaller than a word, or even a byte. Moreover it is also the characteristic of many such applications that at some point the data has to be maintained in packed form, that is, multiple data items must be packed together into a single word of memory. In fact in most cases the input or the output of an application consists of packed data. If the input consists of packed data, the application typically unpacks it for further processing. If the output is required to be in packed form, the application computes the results and explicitly packs it before generating the output. Since packing and unpacking of data is a characteristic of the application domain, it is reflected in the source program itself. In this work we assume that the programs are written in the C language as it is a widely used language in the embedded domain. In C programs packing and unpacking of data involves performing many bitwise logical operations and shift operations.

Important applications that manipulate subword data include *media processing applications* that manipulate packed narrow width media data and *network processing applications* that manipulate packets. Typically such embedded applications receive media data or data packets over a transmission medium. Therefore, in order to make best use of the communication bandwidth, it is desirable that each individual subword data item be expressed in its natural size and not expanded into a 32 bit entity for convenience. However, when this data is deposited into memory, either upon its arrival as an input or prior to its transmission as an output, it clearly exists in packed form.

The processing of packed data that typically involves unpacking of data, or generation of packed data that typically involves packing of data, both require execution of additional instructions that carry out shift and logical bitwise operations. These instructions cost cycles and also increase the code size. The examples given below are taken from `adpcm` (audio) and `gsm` (speech) applications respectively. The first example is an illustration of an unpacking operation which extracts a 4 bit entity from `inputbuffer`. The second example illustrates the packing of a 5 bit entity taken from $LARc[2]$ with a 3 bit entity taken from $LARc[3]$.

Unpacking:
$$delta = (inputbuffer >> 4) \& 0xf$$

Packing:
$$*c++ = ((LARc[2] \& 0x1f) << 3) | \\ ((LARc[3] >> 2) \& 0x7)$$

In addition to the generation of extra instructions for packing and unpacking data, there are other consequences of packing and unpacking. Additional memory locations and registers are required to hold values in packed and unpacked form. Increase in register pressure results which can further increase memory requirements and cache activity. Finally all of the above factors influence the total energy comsumption which can be of vital concern.

In this paper we present the *Bit Section eXtension* (BSX) to the ARM processor's instruction set. Bit sections are the subword entities that are manipulated by the programs. We selected the ARM processor for this research because it is one of the most popular embedded processor which is also being used by many commercial network processing architectures being built today. We present the design of BSX instructions and their encoding into the ARM instruction set. The newly designed instructions allow us to specify register operands that are bit sections of 32 bit values contained within registers. As a result the data stored in packed form can be directly accessed and manipulated and thus the need for performing explicit unpacking operations is eliminated. Similarly computed results can be stored directly into packed form which eliminates the need for explicit packing operations.

We have incorporated the implementation of BSX in the Simplescalar ARM simulator from Michigan. Results of experiments with programs from various benchmark suites show that by using BSX instructions the number of instructions executed by these programs can be significantly reduced. For the functions in which BSX instructions were used we observed a reduction in dynamic instruction counts that ranges from 4.26% to 27.27%. The code sizes of the functions were reduced by 1.27% to 21.05%.

The remainder of the paper is organized as follows. In Section 2 we describe the design of bit section specification methods and their incorporation in various types of instructions. We also show how these new instructions are encoded using the unused encoding space of the ARM instruction set. Section 3 describes our approach to generating code that makes use of BSX instructions. Section 4 describes our experimental setup and results of experiments. Related work on instruction set design and compiler techniques for taking advantage of these instructions is discussed in Section 5. Concluding remarks are given in Section 5.

2. BIT SECTION EXTENSIONS (BSX)

2.1 Bit Section Descriptors

Subword level data entities are called *bit sections*. A bit section is a sequence of consecutive bits within a word. A bit section can vary from 1 bit long to 32 bits long. We specify bit sections through use of *bit section descriptors* (BSDs).

To specify a bit section within a word, we have two options. One way is to specify the starting bit position and the ending bit position within the word. Another way is to specify the starting bit position and bit section length. Either way it takes 10 bits to specify a single bit section: 5 bits for the starting position and 5 bits for the length or ending position.

We use the form which specifies length of the bit section. By analyzing the MediaBench and CommBench programs we found that many operations involve multiple bit section operands of the same size. Therefore when one instruction involves multiple bit section operands, they can share the same bit section *length* specification. While the lengths of multiple bit sections used by an instruction are often the same and can be specified once, the ending bit position of these bit sections can often be different and thus unlike the length the ending position specification cannot be shared.

2.2 Bit Section Addressing Modes

There are two different *addressing modes* through which bit section descriptors can be specified. While the position of many bit sections within the word boundary can be determined at compile time, the position of some bit sections can only be determined at run time. Therefore we need two addressing modes for specifying bit sections: bit section operand can be specified as an *immediate value* encoded within the instruction; or bit section can be specified in a *register* as it cannot be expressed as an immediate constant. The number of bit section operands that are used by various instructions can vary from one to three.

2.2.1 Immediate Bit Section Descriptors

An immediate bit section descriptor is encoded as part of the instruction. Let us assume that R is a register operand of the instruction which is specified using 4 bits as ARM contains 16 registers ($R0...R15$). If the operand is a bit section within R whose position within R is known to be fixed, then an *immediate bit section descriptor* is associated with the register as follows:

$R[\#start, \#len]$ refers to:
bits $[\#start..\#start + \#len - 1]$ of R.

The constant $\#start$ is 5 bits as the starting position of the bit section may vary from bit 0 to bit 31 and $\#len$ is also 5 bits as the number of bits in the bit section can at most include all the bits (0..31) of the register. Note that for valid bit section descriptors $\#start + \#len - 1$ is never greater than 31.

Immediate bit section descriptors are used if either the instruction has only one such bit section operand or two bit section operands. When two bit section descriptors need to be specified, the $\#len$ specification is the same and hence shared by the two descriptors as shown below.

$R1[\#start1], R2[\#start2], \#len$ refers to:
bits $[\#start1..\#start1 + \#len - 1]$ of $R1$; and
bits $[\#start2..\#start2 + \#len - 1]$ of $R2$.

2.2.2 Register Bit Section Descriptors

When both the operands of an instruction as well as its result are bit sections, then three bit section descriptors need to be specified. Even though all three bit sections share the same length, it is not possible to specify all three bit sections as immediates because not enough bits are available in an instruction to carry out this task. Therefore in such cases the specification of the bit section descriptors is stored in a register rather than as an immediate value in the instruction itself.

There is another reason for specifying bit section descriptors in registers. In some situations the positions and lengths of the bit sections within a register are not fixed but rather determined at runtime by the program. In this case the bit section descriptor is not an immediate value specified as part of the instruction but rather the descriptor is computed into the register which is then specified as part of the instruction. The register which specifies the bit section descriptor may specify one, two or three bit sections in one, two, and three (possibly different) registers as shown below:

$$R1[R]$$
$$R1, R2[R]$$
$$R1, R2, R3[R]$$

where register R contains the bit section descriptors for the appropriate operand registers $R1$, $R2$ and $R3$. The contents of R are organized as shown in Figures 1, 2 and 3.

31–10	9–5	4–0
X X X X X X X X X X X X X X X X X X X X X X	start	len

Figure 1: Bit Section Descriptor for 1 Bit Section.

31–15	14–10	9–5	4–0
X X X X X X X X X X X X X X X X X	start1	start2	len

Figure 2: Bit Section Descriptor for 2 Bit Sections.

31–20	19–15	14–10	9–5	4–0
X X X X X X X X X X X X	start1	start2	start3	len

Figure 3: Bit Section Descriptor for 3 Bit Sections.

2.3 Bit Section Instructions & their Encoding

Next we describe the ARM instructions that are allowed the use of bit section operands. While in principle it is possible to allow any existing ARM instruction with register operands to access bit sections within the register as operands, we cannot allow all instructions this flexibility as there would be too many new variations of instructions and there is not enough space in the 32 bit encoding of ARM instructions to accommodate these new instructions. Therefore we choose a selected subset of instructions which are most likely to be involved in bit section operations and developed variations for them. In the benchmarks we studied most of the possible operations are related to data processing. Therefore eight data processing instructions are chosen from version 5T of the ARM instruction set which include six ALU instructions (ADD, SUB, AND, EOR, ORR, and RSB) as well as compare and move (CMP and MOV) instructions. The selection of these instructions was based on studying a number of multimedia benchmarks and determining the type of instructions that are most commonly needed. Figure 4 shows the percantage of total executed instructions that fall in the category of above instruction types selected for supporting bit section operands. As we can see, the selected instructions account for a significant percentage of dynamic instruction counts.

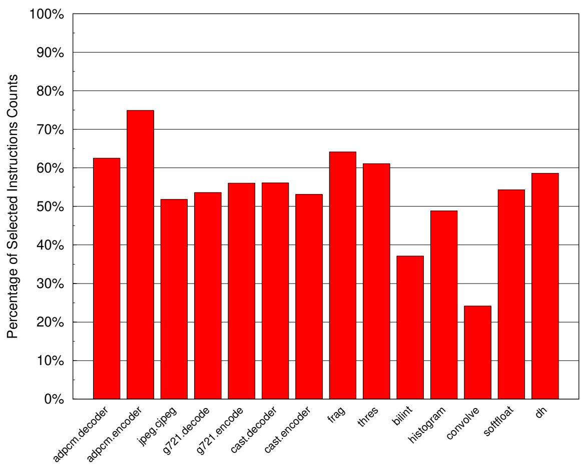

Figure 4: Dynamic frequency of selection instructions.

2.3.1 Instructions with Immediate BSDs

For each of the above instructions we provide three variations when immediate bit section operands are used. In version 5T of the ARM instruction set the encoding space with prefix 11110 is undefined. We use the remaining 27 bits of space of this undefined instruction to deploy the new instructions. Of these 27 bits three bits are used to distinguish between the eight operations that are involved.

Let us discuss the three variations of each of the ALU instructions. In the *first variation* (FV) of the above ALU instructions, the corresponding instructions have two bit section operands. Therefore one of the operands acts both as the source operand and the destination. The variants of CMP and MOV instructions are slightly different as they require only two operands, unlike the ALU instructions which require three operands. For CMP the two bit section operands are both source operands and for MOV one operand is the source and the other is the destination. We cannot allow three operands to be all bit section operands at the same time because three bit section operands will need at least 32 bits to specify.

The encoding of these instructions is shown below. The prefix 11110 in bits 31 to 27 indicates the presence of BSX instruction. Three bits that encode the eight operations are bits 24 to 26. Bit 23 is 0, which indicates this is the first variation of the instruction. The remaining bits encode the two bit section descriptors: $Rd[Rds, len]$ and $Rm[Rms, len]$.

31	30	29	28	27	26–24	23	22–19	18–14	13–10	9–5	4–0
1	1	1	1	0	opcode	0	Rd	Rds	Rm	Rms	len

Figure 5: First Variation: ALU Instructions.

The *second variation* (SV) of instructions has three operands. One is a destination register (not a bit section), one is a source register (not a bit secion), and the third operand is a bit section operand. In this variation the operation is done as if the bit section is zero extended. To specify this variation bit 23 must be 1 and bit 14 must be 0. The instruction format and encoding is shown below.

31	30	29	28	27	26–24	23	22–19	18–15	14	13–10	9–5	4–0
1	1	1	1	0	opcode	1	Rd	Rn	0	Rm	Rms	len

Figure 6: Second Variation: ALU Instructions.

CMP and MOV are again slightly different as they need only two operands. Bit 15 is a flag to indicate whether the bit section is to be treated as an unsigned or signed entity. If it is 0, then it is unsigned and then zero extended before the operation. If it is 1, the bit section is signed, and therefore the first bit in the bit section is extended before the operation.

31	30	29	28	27	26–24	23	22–19	18–16	15	14	13–10	9–5	4–0
1	1	1	1	0	opcode	1	Rd	X X X	S	0	Rm	Rms	len

Figure 7: CMP and MOV Instructions.

31	30	29	28	27	26–24	23	22–15	14	13–10	9–5	4–0
1	1	1	1	0	opcode	1	imm8	1	Rm	Rms	len

Figure 8: Third Variation: ALU Instructions.

The *third variation* (TV) has one 8 bit immediate value which is one of the operands and one bit section descriptor which represents the second operand. The latter bit section also serves as the destination operand. To specify this variation, bit 23 must be 1 and bit 14 must be 1. The instruction format and encoding is shown above.

2.3.2 *Instructions with Register BSDs*

For each of the above instructions we have three variations when register bit section operands are used. These variations differ in the number of bit section operands. We found another undefined instruction space with prefix 11111111 to encode these instructions into version 5T of the ARM instruction set. The encoding of the instructions is as follows. Bits 19 to 21 contain the opcode while bits 17 and 18 stand for the number of the bit section operands in the instructon. Therefore 01, 10, and 11 correspond to presence of 1, 2, and 3 bit section operands. The S bit specifies whether the bit section contains unsigned or signed integer. The format and encoding of the instructions is given below.

31							24	23	22	21 19	18 17	16	15 12	11 8	7 4	3 0
1	1	1	1	1	1	1	1	X	X	opcode	num	S	Rd	Rn	Rm	Rb

Figure 9: ALU Instructions: Register BSDs.

31							24	23	22	21 19	18 17	16	15 12	11 8	7 4	3 0
1	1	1	1	1	1	1	1	X	X	opcode	num	S	Rd	X X X X	Rm	Rb

Figure 10: CMP and MOV Instructions: Register BSDs.

31						25	24	23	22 18	17 12	11 6	5 0
1	1	1	1	1	0	0	X	X	Rd	Rns	Rms	len

Figure 11: Setup BSD Instruction.

5	4 0
1	imm5

5	4	3 0
0	0	R

5	4	3 0
0	1	X X X X

Figure 12: Setup Specifier.

Instructions CMP and MOV are a little bit different, they can have at most two bit section operands. Therefore bits 17 and 18 can only be 01 or 10 and bits 8 to 11 are not specified.

The bit section descriptor in itself contains several bit sections. Therefore setup costs of a bit section descriptor in a register can be high. Therefore we introduce new instructions with opcode `setup` to set up the bit section descriptors efficiently. These instructions can set multiple values in bit section descriptor simultaneously. The format and encoding of these instructions are given in Figures 11 and 12.

The instruction `setup Rd, Rns, Rms, len` can set up the value of Rns, Rms and len fields in bit section descriptor held in Rd simultaneously. A 6-bit setup specifer describes how a field is set up. In each setup specifier, if bit 5 is 1, then bits 0 to 4 represent an immediate value. The field is setup by copying this immediate value. If bit 5 is 0, and bit 4 is 0, then bits 0 to 3 are used to specify a register. The field is setup by copying the last five digits in the register. For Rns specifier, if bit 5 is 0 and bit 4 is 1, then Rns is not a valid bit section specifier and must be ignored. In general, since all three values (Rns, Rms, and len) can be in registers, we need to read these registers to implement the instruction in one cycle. However, in practice we never encountered a situation where there was a need to read three registers.

2.4 BSX Implementation

To implement the BSX instructions two approaches are possible. One approach involves redesign of the register file. The bit section can be directly supplied to the register file during a read or write operation and logic inside the register file ensures that only the appropriate bits of a register are read or written.

An alternative approach which does not require any modification to the register file reads or writes an entire register. During a read, entire register is read, and then logic is provided so that the relevant bit section can be selected to generate the bit section operand for an instruction. Similarly during a write to update only some of the bits in a register, in the cycle immediately before the cycle in which the write back operation is to occur, the contents of the register to be partially overwritten are read. The value read is made available to the instruction during the write back stage where the relevant bit section is first updated and then written to the register file. An extra dedicated read port should be provided to perform the extra read associated with each write operation.

The advantage of the first approach is that it is more energy efficient. Even though it requires the redesign of the register file, it is also quite simple. The second approach is not as energy efficient, it requires greater number of register reads, and is also somewhat more complex to implement.

3. GENERATING BSX ARM CODE

Our approach to generating code that uses the BSX instructions is to take existing ARM code generated for programs using the unmodified compiler and then, in a postpass, selectively replace the use of ARM instructions by BSX instructions to generate the optimized code. The optimizations are aimed at packing and unpacking operations in context of bit sections with *compile time fixed* and *dynamically varying* positions.

3.1 Fixed Unpacking

An unpacking operation involves merely extracting a bit section from a register that contains packed data and placing the bit section by itself in the lower order bits of another register. The example below illustrates unpacking which extracts bit section 4..7 from `inputbuffer` and places it in lower order bits of `delta` (the higher order bits of `delta` are 0). As shown below, the ARM code requires two instructions, a `shift` and an `and` instruction. However, a single BSX instruction which takes bits 4 to 7, zero extends them, and places them in a register is sufficient to perform unpacking.

C code
$delta = (inputbuffer >> 4) \& 0xf;$
ARM code
mov r3, r8, asr #4 and r12, r3, #15 ; 0xf
BSX ARM code
mov r12, r8[#4,#4]

The general transformation that optimizes the unpacking operation takes the following form. In the ARM code an `and` instruction extracts bits from register `ri` and places them in register `rj`. Then the extracted bit section placed in `rj` is used possibly multiple times. In the transformed code, the `and` instruction is eliminated and each use of `rj` is replaced by a direct use of bit section in ri. This transformation also eliminates the temporary use of register rj. Therefore, for this transformation to be legal, the compiler must ensure that register rj is indeed temporarily used, that is, the value in register rj is not referenced following the code fragment.

Before Transformation
and rj, ri, #mask(#s,#l)
inst1 use rj
...
instn use rj
Precondition
the bit section in ri remains unchanged until instn and rj is dead after instn.
After Transformation
inst1 use ri[#s,#l]
...
instn use ri[#s,#l]

3.2 Fixed Packing

In ARM code when a bit section is extracted from a data word we must perform `shift` and `and` operations. Such operations can be eliminated as a BSX instruction can be used to directly reference the bit section. This situation is illustrated by the example given below. The C code takes bits 0..4 of LARc[2] and concatenates them with bits 2..4 of LARc[3]. The first two instructions of the ARM code extract the relevant bits from LARc[3], the third instruction extracts relevant bits from LARc[2], and the last instructions concatenates the bits from LARc[2] and LARc[3]. As we can see, the BSX ARM code only has two instructions. The first instruction extracts bits from LARc[3], zero extends them, and stores them in register r0. The second instruction moves the relevant bits of LARc[2] from register r1 and places them in proper position in register r0.

C code
$*c++ = ((LARc[2]\&0x1F) << 3)$
$\|((LARc[3] >> 2)\&0x7);$
ARM code
; r0 ← LARc[3]
; $(LARc[3] >> 2)\&0x7$
mov r0, r0, lsr #2
and r0, r0, #7
; r1 ← LARc[2]
; $(LARc[2]\&0x1F) << 3$
and r2, r1, #31
orr r0, r0, r2, asl #3
BSX ARM code
; r0 ← LARc[3]
; r1 ← LARc[2]
mov r0, r0[#2,#3]
mov r0[#3,#5], r1[#0,#5]

In general the transformation for eliminating packing operations can be characterized as follows. An instruction defines a bit section and places it into a temporary register ri. The need to place the bit section by itself into a temporary register ri arises because the bit section is possibly used multiple times. Eventually the bit section is packed into another register rj using an `orr` instruction. In the optimized code, when the bit section is defined, it can be directly computed into the position it is placed by the packing operation, that is, into rj. All uses of the bit section can directly reference the bit section from rj. Therefore the need for temporary register ri is eliminated and the packing `orr` instruction is eliminated. For this transformation to be legal, the compiler must ensure that register ri is indeed temporarily used, that is, the value in ri is not referenced after the code fragment.

Before Transformation
inst0 def ri ;bit section definition in a whole register
...
inst1 use ri ;use register
...
instn use ri ;use register
...
orr rj, rj, ri ;pack bit section
Precondition
the bit sections in ri and rj remain unchanged until orr and ri is dead after orr.
After Transformation
inst0 def rj ;define and pack
...
inst1 use rj ;use bit section
...
instn use rj ; use bit section

3.3 Dynamic Unpacking

There are situations in which, while extraction of bit sections is to be carried out, the position of the bit section is determined at runtime. In the example below, a number of lower order bits, where the number equals the value of variable $size$, are extracted from put_buffer, zero extended, and placed back into put_buffer. Since the value of $size$ is not known at compile time, an immediate value cannot be used to specify the bit section descriptor. Instead the first three ARM instructions shown below are used to dynamically construct the `mask` which is then used by the `and` instruction to extract the required value from put_buffer. In the optimized code the bit section descriptor is setup in register r3 and then used by the `mov` instruction to extract the require bits and place them by themselves in r7.

C code
$mask = (1 << size) - 1$
$put\_buffer = put\_buffer\&mask$
ARM code
; r5 ← size
; r7 ← put_buffer
mov r3, #1
mov r3, r3, lsl r5
sub r3, r3, #1
and r7, r7, r3
BSX ARM code
; start = 0
; len = r5
setup r3, _, #0, r5
mov r7, r7[r3]

The general form of this transformation is shown below. The ARM instructions that construct the mask are replaced by a single setup instruction. The and instruction can be replaced by a mov of a bit section whose descriptor can be found in the register set up by the setup instruction.

ARM code

mov ri, #1
mov ri, ri, lsl rj
sub ri, ri, #1
and rd, rn, ri

Precondition
value in ri should be dead after and instruction.
BSX ARM code
setup ri, rj, rj mov rd, rn[ri]

3.4 Dynamic Packing

Packing of bit sections together, whose sizes are not known till runtime, can cost several instructions. The C code given below extracts lower order p bits from m and higher order $16 - p$ bits from n and packs them together into o. The ARM code for this operation involves many instructions because first the required masks for m and n are generated. Next the relevant bits are extracted using the masks and finally they are packed together using the orr instruction. In contrast the BSX ARM code uses far fewer instructions. Since p's value is not known at compile time, we must use register bit section descriptors for m and n.

C code
$o = (m\&((1 << p) - 1)\|$ $(n\&(((1 << (16 - p)) - 1) << p))$
ARM code

; r3 $\leftarrow p$
mov r12, #1
rsb r2, r3, #16 ; $16 - p$
mov r2, r12, lsl r2 ; $1 << (16 - p)$
sub r2, r2, #1 ; $(1 << (16 - p)) - 1$
; $r1 \leftarrow n$
and r1, r1, r2, lsl r3 ; $n\&((1 << (16 - p)) - 1)$
mov r12, r12, lsl r3 ; $1 << p$
sub r12, r12, #1 ; $(1 << p) - 1$
; $r0 \leftarrow m$
and r0, r0, r12 ; $m\&((1 << p) - 1)$
orr r0, r0, r1 ; $o \leftarrow r0$

BSX ARM code

; $r3 \leftarrow p$
setup r12, _, #0, r3 ; descriptor for m's bit section
rsb r2, r3, #16
setup r2, _, r3, r2 ; descriptor for n's bit section
; $r0 \leftarrow m$
mov r0, r0[r12] ; put m's relevant bits in r0
mov r0, r1[r2] ; put n's relevant bits in r0

In general the transformation for optimizing dynamic packing operations can be described as follows. Two or more bit sections, whose positions and lengths are unknown at compile time, are extracted from registers where they currently reside and put into separate registers respectively. A mask is constructed and an and instruction is used to perform the extraction. Finally they are packed togehter into one register using orr instruction. In the optimized code, for each bit section, we setup a register bit section descriptor first, and then move the bit section into the final register with the bit section descriptor directly. As a result, orr instruction is removed. By using the setup instruction to simultaneously setup several fields in the bit section descriptor, we reduce the number of instructions in comparison to the instruction sequence used to create the masks in the original code. Different types of instruction sequences can be used to create a mask and thus it is not always possible to identify such sequences. Our current implementation can only handle some commonly encountered sequences.

ARM code
instruction sequence to create mask1 and ra, rb, mask1 instruction sequence to create mask2 and rc, rd, mask2 orr rm, ra, rc
BSX ARM code
setup register bit section descriptor 1 move bit section 1 to rm using bit section descriptor 1 setup register bit section descriptor 2 move bit section 2 to rm using bit section descriptor 2

4. EXPERIMENTAL EVALUATION

4.1 Experimental Setup

Before we present the results of the experiments, we describe our experimental setup which includes a simulator for ARM, an optimizing compiler, and a set of relevant benchmarks.

Processor Simulator

We started out with a port of the cycle level simulator Simplescalar [1] to ARM available from the University of Michigan. This version simulates the five stage pipeline described in the preceding section which is the Intel's SA-1 StrongARM pipeline [8] found, for example, in the SA-110. The I-Cache configuration for this processor are: 16Kb cache size, 32b line size, and 32-way associativity, and miss penalty of 64 cycles (a miss requires going off-chip). The timing of the model has been validated against a Rebel NetWinder Developer workstation [16] by the developers of the system at Michigan.

We have extended the above simulator in a number of important ways for this research. First we modified Simplescalar to use the system call conventions followed by the Newlib C library instead of glibc which it currently uses. We made this modification because Newlib has been developed for use by embedded systems [10]. Second we incorporated the implementation of the BSX instructions for the purpose of their evaluation. In addition, we have also incorporated the Thumb instruction set into Simplescalar. However, this feature is not relevant for this paper.

Optimizing Compiler

The compiler we used in this work is the gcc compiler which was built to create a version that supports generation of ARM, Thumb as well as mixed ARM and Thumb code. Specifically we use the xscale-elf-gcc compiler version 2.9-xscale. All programs were compiled at -O2 level of optimization. We did not use -O3 because at that level of optimization function inlining and loop unrolling is enabled. Clearly since the code size is an important concern for embedded systems, we did not want to enable function inlining and loop unrolling.

The translation of ARM code into optimized BSX ARM code was carried out by an optimization postpass. Only the frequently executed functions in each program that involve packing, unpacking, and use of bit section data were translated into BSX ARM code. The remainder of the program was not modified. As we have seen from the transformations of the preceding section, temporary registers are freed by the optimizations. While it may be possible to improve the code quality by making use of these registers, we do not do so at this time due to the limitations of our implementation.

Representative Benchmarks

The benchmarks we use are taken from the `Mediabench` [12], `Commbench` [21], `Netbench` [14], and `Bitwise` [18] suites as they are representative of a class of applications important for the embedded domain. We also added an image processing application `thres`. The following programs are used:

Mediabench:
`adpcm` - `encoder` and `decoder`; `g721` - `decode` and `encode`; and `jpeg` - `cjpeg`.

Commbench:
`frag` and `cast` - `decoder` and `encoder`.

Image Processing: `thres`.

Bitwise:
`bilint`, `histogram`, `convolve`, and `softfloat`.

Netbench: `dh`.

4.2 Results

Next we present the results of experiments that measure the improvements in code quality due to the use of BSX instructions. We measured the reductions in both the instruction counts and cycle counts for BSX ARM code in comparison to pure ARM code. The results are given in Tables 1 and 2. In these results we provide the percentage improvements for each of the functions that were modified as well as the improvements in the total counts for the entire program. The reduction in *instruction counts* for the modified functions varies between 4.26% and 27.27%. The net instruction count reductions for the entire programs are lower and range from 0.45% to 8.79%. This is to be expected because only a subset of functions in the programs can make significant use of the BSX instruction. The reductions in *cycle counts* for the modified functions varies between 0.66% and 27.27%. The net cycle count reductions for the entire programs range from 0.39% to 8.67%. In Table 5 the reductions in code size of functions that were transformed to make use of BSX instructions are given. The code size reductions range from 1.27% to 21.05%.

Finally we also studied the usage of BSX instructions and transformations used by the benchmarks. In Table 3 we show the types of BSX instructions that were used by each of the benchmarks. In particular, we indicate whether *fixed BSDs* were used in the instructions or *dynamic BSDs* were used. For fixed BSDs we also indicate which of the three variations of bit section referencing instructions (FV, SV, and TV) were used by the benchmark. For dynamic BSDs we also indicate the use of *setup* instruction. As we can see, fixed BSDs are more commonly used and situations involving the three variations of bit section operands arise. In Table 4 we show the kind of transformations that were found to be applicable to each of the benchmarks - packing and unpacking involving fixed or dynamic BSDs. As we can see, each optimization and every BSX instruction was used in some program. The results of Tables 3 and 4 indicate that the *fixed BSD instructions* that we have included in BSX are appropriate and useful. The results for *register BSDs* are negative. While we found instances where the positions of the BSDs vary at runtime, we are not able to develop the appropriate compiler transformations to effectively take advantage of these situations using our instructions.

One of the benefits of using BSX instructions is that often the number of registers required is reduced. This is because multiple subword data items can now simultaneously reside in a single register and it is no longer to separate them in hold them in different registers. The performance data presented above is based upon BSX ARM code that does not take advantage of the additional registers that may become available. Once the registers are used one can expect additional performance gains. While the problem of global register allocation for subword data is beyond the scope of this paper, in a related paper [19] we have shown that register requirements can be reduced by 12% to 50% for functions that can take advantage of BSX instructions.

5. RELATED WORK

A wide variety of instruction set support has been developed to support multimedia and network processing applications. Most of these extensions have to do with exploiting subword [5] and superword [11] parallelism. The instruction set extensions proposed by Yang and Lee [22] focus on permuting subword data that is packed together in registers. The network processor described [15] also supports bit section referencing. In this paper we carefully designed an extension consisting of a small subset of flexible bit-section referencing instructions and showed how they can be easily incorporated in a popular embedded ARM processor.

Compiler research on subword data can be divided into two categories. First work is being done to automatically identify narrow width data. Second techniques to automatically pack narrow width data and perform register allocation, instruction selection, and generation of SIMD parallel instructions is being carried out.

There are several complementary techniques for identifying subword data. Stephenson et al. [18] proposed *bitwidth analysis* to discover narrow width data by performing value range analysis. Budiu et al. [2] propose an analysis for inferring individual *bit values* which can be used to narrow the width of data. Tallam and Gupta [19] propose a new type of *dead bits analysis* for narrowing the width of data. The analysis by Zhang et al. [7] is aimed at automatic discovery of multiple data items packed into program variables.

The works on packing narrow width data after its discovery include the following. Davidson and Jinturkar [3] were first to propose a compiler optimization that exploits narrow width data. They proposed *memory coalescing* for improving the cache performance of a program. Zhang and Gupta [23] have proposed techniques for *compressing* narrow width and pointer data for improving cache performance. Both of these techniques were explored in context of general purpose processors and both change the data layout in memory through packing. Aggressive packing of scalar variables into registers is studied in [19]. As mentioned earlier, this register allocation technique, when combined with the work in this paper, can further improve performance. Another work on register allocation in presence of bit section referencing is by Wagner and Leupers

Table 1: Reduction in Dynamic Instruction Counts.

Benchmark Function	Instruction Count ARM	Instruction Count BSX ARM	Savings [%]
adpcm.decoder			
adpcm_decoder	6124744	5755944	6.02%
Total	6156561	5787760	5.99%
adpcm.encoder			
adpcm_encoder	7097316	6654756	6.24%
Total	7129778	6687534	6.20%
jpeg.cjpeg			
emit_bits	634233	586291	7.56%
Total	15765616	15694887	0.45%
g721.decode			
fmult	47162982	43282495	8.23%
predictor_zero	9293760	8408640	9.52%
step_size	1468377	1320857	10.05%
reconstruct	2628342	2480822	5.61%
Total	258536428	253180667	2.07%
g721.encode			
fmult	48750464	44367638	8.99%
predictor_zero	9293760	8408640	9.52%
step_size	2372877	2225357	6.22%
reconstruct	2645593	2498073	5.58%
Total	264021499	258163419	2.22%
cast.decoder			
CAST_encrypt	41942016	37850112	9.76%
CAST_ofb64_encrypt	26980992	25190784	6.64%
Total	109091228	103209100	5.40%
cast.encoder			
CAST_encrypt	41942016	37850112	9.76%
CAST_ofb64_encrypt	26980992	25190784	6.64%
Total	105378485	99496358	5.58%
frag			
in_cksum	26991150	25494952	5.54%
Total	37506531	36010318	3.99%
threshold			
coalesce	3012608	2602208	13.62%
memo	3223563	2814963	12.68%
blocked_memo	2941542	2531826	13.93%
Total	20959630	19730898	5.86%
bilint			
main	87	79	9.20%
Total	496	488	1.61%
histogram			
main	317466	301082	5.16%
Total	327311	310857	5.03%
convolve			
main	30496	30240	0.84%
Total	30799	30542	0.83%
softfloat			
float32_signals_nan	132	96	27.27%
addFloat32Sigs	29	23	20.70%
subFloat32Sigs	29	23	20.70%
float32_add	8	7	12.50%
float32_sub	8	7	12.50%
float32_mul	30	23	23.33%
float32_div	30	24	20.00%
float32_rem	28	23	17.86%
float32_sqrt	23	20	13.04%
float32_eq	23	17	26.09%
float32_lt	17	11	35.29%
Total	898	819	8.79%
dh			
NN_DigitMult	153713163	141768387	7.77%
NN_DigitDiv	19319249	18495393	4.26%
Total	432372762	419604191	2.95%

Table 2: Reduction in Dynamic Cycle Counts.

Benchmark Function	Instruction Count ARM	Instruction Count BSX ARM	Savings [%]
adpcm.decoder			
adpcm_decoder	6424241	6202961	3.44%
Total	6499880	6278786	3.40%
adpcm.encoder			
adpcm_encoder	7958088	7515456	5.56%
Total	8035001	7592761	5.50%
jpeg.cjpeg			
emit_bits	1047235	999163	4.59%
Total	19611965	19535002	0.39%
g721.decode			
fmult	63914793	60034237	6.07%
predictor_zero	12834446	11949382	6.90%
step_size	1564728	1269752	18.85%
reconstruct	2601534	2454014	5.67%
Total	347037906	341531879	1.59%
g721.encode			
fmult	65798336	61415518	6.66%
predictor_zero	12834447	11949327	6.90%
step_size	2630082	2335106	11.22%
reconstruct	2636030	2488439	5.60%
Total	353610636	347605462	1.70%
cast.decoder			
CAST_encrypt	46557053	40674664	12.63%
CAST_ofb64_encrypt	32224708	30434422	5.56%
Total	141113081	133440304	5.44%
cast.encoder			
CAST_encrypt	46557174	40674817	12.63%
CAST_ofb64_encrypt	32224703	30434428	5.56%
Total	135572465	127900147	5.66%
frag			
in_cksum	32698919	31205099	4.57%
Total	57745393	56207197	2.66%
threshold			
coalesce	4355796	3937458	9.60%
memo	4725060	4307735	8.83%
blocked_memo	22092904	21683166	1.85%
Total	181425566	180186381	0.68%
bilint			
main	887	808	8.91%
Total	5957	5878	1.32%
histogram			
main	481531	462532	3.95%
Total	496650	477807	3.79%
convolve			
main	40215	39949	0.66%
Total	44945	44803	0.32%
softfloat			
float32_signals_nan	132	96	27.27%
addFloat32Sigs	324	247	23.77%
subFloat32Sigs	675	595	11.85%
float32_add	81	80	1.23%
float32_sub	217	153	29.49%
float32_mul	577	513	11.09%
float32_div	397	380	4.28%
float32_rem	453	314	30.68%
float32_sqrt	383	295	22.98%
float32_eq	242	227	6.20%
float32_lt	305	229	24.92%
Total	10255	9366	8.67%
dh			
NN_DigitMult	236874768	224929644	5.04%
NN_DigitDiv	26223468	25400239	3.14%
Total	578187905	565434086	2.21%

Table 3: BSX Instruction Usage.

Benchmark	Fixed BSDs			Dynamic BSD	Setup BSD
	FV	SV	TV		
adpcm.decoder		yes	yes		
adpcm.encoder	yes		yes		
jpeg.cjpeg	yes			yes	yes
g721.decode		yes	yes		
g721.encode		yes	yes		
cast.decoder	yes		yes		
cast.encoder	yes		yes		
frag		yes			
thres			yes		
bilint		yes			
histogram	yes		yes		
convolve		yes			
softfloat	yes	yes	yes		
dh	yes	yes			

Table 4: Transformations Applied.

Benchmark	Fixed BSDs		Dynamic BSDs	
	Pack	Unpack	Pack	Unpack
adpcm.decoder		yes		
adpcm.encoder	yes	yes		
jpeg.cjpeg				yes
g721.decode		yes		
g721.encode		yes		
cast.decoder	yes	yes		
cast.encoder	yes	yes		
frag		yes		
thres		yes		
bilint		yes		
histogram	yes			
convolve		yes		
softfloat	yes	yes		
dh	yes			

Table 5: Reduction in Code Size.

Benchmark Function	Code Size		Reduction [%]
	ARM	BSX ARM	
adpcm.decoder			
adpcm_decoder	260	248	4.62%
adpcm.encoder			
adpcm_encoder	300	284	5.33%
jpeg.cjpeg			
emit_bits	228	216	5.26%
g721.decode and g721.encode			
fmult	196	176	10.2%
predictor_zero	92	84	8.7%
step_size	76	72	5.26%
reconstruct	96	92	4.17%
cast.decoder and cast.encoder			
CAST_encrypt	1328	1200	9.64%
CAST_ofb64_encrypt	428	400	6.54%
frag			
in_cksum	108	88	18.52%
threshold			
coalesce	148	136	8.11%
memo	296	284	4.05%
blocked_memo	212	200	5.66%
bilint			
main	352	320	9.09%
histogram			
main	316	312	1.27%
convolve			
main	652	648	0.61%
softfloat			
float32_signals_nan	52	40	20.0%
addFloat32Sigs	348	324	6.90%
subFloat32Sigs	396	372	6.06%
float32_add	40	36	10.0%
float32_sub	40	36	10.0%
float32_mul	428	400	6.54%
float32_div	544	520	4.41%
float32_rem	648	628	3.09%
float32_sqrt	484	464	4.13%
float32_eq	152	120	21.05%
float32_lt	176	144	18.18%
dh			
NN_DigitMult	112	104	7.14%
NN_DigitDiv	420	404	3.81%

[20]. Their work exploits bit section referencing in context of variables that already contain packed data. They do not carry out any additional variable packing. Compiler techniques for carrying out SIMD operations on narrow width data packed in registers can be found in [4, 11].

6. CONCLUSIONS

We presented the design of the *Bit Section eXtension* (BSX) to the ARM processor which can be easily encoded into the free encoding space of the ARM instruction set. We found that bit sections are frequently manipulated by multimedia and network data processing codes. Therefore BSX instructions can be used quite effectively to improve the performance of these benchmarks. In addition, reductions in code size and register requirements also result when BSX instructions are used. We have incorporated the implementation of BSX in the Simplescalar ARM simulator from Michigan. Results of experiments with programs from the various benchmark suites show that by using BSX instructions the number of instructions executed by these programs can be significantly reduced. Our future work will focus on integrating the use of BSX instructions with register allocation techniques that aggressively pack subword variables into a single registers.

Acknowledgements

This work is supported by DARPA award F29601-00-1-0183 and National Science Foundation grants CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the University of Arizona.

7. REFERENCES

[1] D. Burger and T.M. Austin, "The Simplescalar Tool Set, Version 2.0," *Computer Architecture News*, pages 13–25, June 1997.

[2] M. Budiu, M. Sakr, K. Walker, and S.C. Goldstein, "BitValue Inference: Detecting and Exploiting Narrow Width Computations," *6th European Conference on Parallel Computing* (Euro-Par), August 2000.

[3] J. Davidson and S. Jinturkar, "Memory Access Coalescing : A Technique for Eliminating Redundant Memory Accesses," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 186–195, 1994.

[4] R.J. Fisher and H.G. Dietz, "Compiling for SIMD within Register," *11th International Workshop on Languages and Compilers for Parallel Computing* (LCPC), *LNCS, Springer Verlag*, Chapel Hill, NC, August 1998.

[5] J. Fridman, "Data Alignment for Sub-Word Parallelism in DSP," *IEEE Workshop on Signal Processing Systems* (SiPS), pages 251-260, 1999.

[6] S. Furber, "ARM system Architecture," Publisher: Addison Wesley Longman, 1996.

[7] R. Gupta, E. Mehofer, and Y. Zhang, "A Representation for Bit Section Based Analysis and Optimization," *International Conference on Compiler Construction* (CC), LNCS 2304, Springer Verlag, pages 62-77, Grenoble, France, April 2002.

[8] Intel Corporation, "SA-110 Microprocessor Technical Reference Manual," ftp://download.intel.com/design/strong/applnots/27819401.pdf.

[9] Intel Corporation, "The Intel XScale Microarchitecture Technical Summary," ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf.

[10] J. Johnston, Maintainer, Newlib - http://sources.redhat.com/newlib/.

[11] S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 145–156, Vancouver B.C., Canada, June 2000.

[12] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), Research Triangle Park, North Carolina, December 1997.

[13] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 11, November 1996.

[14] Memik, Mangione-Smith, and Hu, "NetBench: A Benchmarking Suite for Network Processors," *IEEE International Conference Computer-Aided Design* (ICCAD), Nov. 2001.

[15] X. Nie, L. Gazsi, F. Engel, and G. Fettweis, "A New Network Processor Architecture for High Speed Communications," *IEEE Workshop on Signal Processing Systems* (SiPS), pages 548-557, 1999.

[16] rebel.com, Netwinder Family, http://www.rebel.com/netwinder.

[17] D. Seal, Editor, "ARM Architecture Reference Manual," Second Edition, Addison-Wesley.

[18] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 108–120, Vancouver B.C., Canada, June 2000.

[19] S. Tallam and R. Gupta, "Bitwidth Aware Global Register Allocation," *Technical Report*, Dept. of Computer Science, University of Arizona, July 2002.

[20] J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor," *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES), pages 155-164, June 2001.

[21] T. Wolf and M. Franklin, "Commbench - A Telecommunications Benchmark for Network Processors," *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000.

[22] X. Yang and R.B. Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages," *International Conference on Computer Design* (ICCD), pages 15-22, Austin, Texas, September 2000.

[23] Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction* (CC), LNCS 2304, Springer Verlag, pages 14-28, Grenoble, France, April 2002.