A compact code 16-bit processor for embedded applications

Prabhas Chongstitvatana
Department of Computer Engineering, Chulalongkorn University
Bangkok 10330, Thailand
e-mail: prabhas@chula.ac.th

Abstract

This work proposed an instruction set that achieved small executable codes for embedded applications. The aim of the design is to reduce the size of the executable code while maintaining the execution speed. Rather than applying instruction compression which required complex additional circuits, the approach taken in this work is to design the instruction set for the purpose of compact code. The result from a small set of benchmark illustrated that the static code size can be half of a conventional instruction set while the execution speed is maintained.

Key-Words: Compact code, instruction compression, embedded processor.

1. Introduction

For small embedded applications, one chip solution has been widely used due to its cost advantage. The cost of this type of application includes processor area, code segment area and data segment area. The size of an executable code is a major part of the total cost so reducing the size is an important issue. There are many approaches to reduce the size of executable code [1-10]. The instruction compression applies data compression and compiler optimization to the executable code.

There are two major approaches: code compression and code compaction. The first approach, code compression, uses data compression algorithms on machine code. Decompression will slow down the system operation in code compression. On the other hand, code compaction reduces the program size by using compiler optimization in rearranging and eliminating superfluous code. This allows the compressed program to be executed immediately without needing decompression as in the code compression. However, using compression algorithms in code compression is more effective in reducing the program size more using code compaction.

This work takes a different approach. The design of the instruction set is aimed to reduce the code size. There are two possibilities to reduce the code size:

1) Designing the instruction set such that the total number of instruction in an application is minimised.
2) Designing each instruction to be small.

The next section exposes the main contribution of this work, the instruction set design.

2. Instruction set

For a 16-bit processor, a 16-bit fixed length instruction format will allow faster instruction fetch, plus it is large enough to contain three small arguments, so it is adopted as the choice of the instruction size. The instruction set is divided into three main groups: arithmetic/logic, load/store, and control flow. The arithmetic and logic group, add sub and or xor etc., has three arguments. The load and store group has one register and one 9-bit address. The control flow group, jmp, call, jt, jf, ret etc., has one displacement, the relative displacement is 6 bits, the absolute address is 14 bits. The instruction has 4 formats (Fig. 1):

xx abs:14	jump, call
xx op:2 r1:3 a:9	load, store direct
xx op:5 r1:3 r2:3 r3:3	arithmetic, logic
xx op:5 r1:3 d:6	control flow

Figure 1. The instruction set format

With these instruction formats, the processor has 14-bit code address (16 Kwords), 9-bit address that directly access data (512 words), 16-bit total data address (64 Kwords). The direct access data space is used to store the global data. The whole data space can be accessed via index addressing. If a larger code space is needed, a segment extension can be implemented to extend to 64 Kwords (4 segments) as the program counter is 16 bits. A part of instruction set is shown in Fig. 2.

Instruction	Meaning
jmp a	jump to ads
call a	call ads
ret s	return
retv r1 s	return r1
jt r1 d	if r1 != 0 pc+=d
jf r1 d	if r1 == 0 pc+=d
aop r1 r2 r3	ri = r2 aop r3
aop r1 r2 #n	r1 = r2 aop n
ld r1 a	r1 = M[a]
st r1 a	M[a] = r1
ldx r1 r2 r3	r1 = M[r2+r3]
stx r1 r2 r3	M[r2+r3] = r1
mov r1 r2	r1 = r2
mvx r1 r2	pass r2 to next r1

where aop is add, sub, and, or, xor, not, shl, shr etc.

Figure 2. A part of the instruction set

As the argument is limited to 3 bits, the number of direct-accessed variable is eight, they are denoted r0..r7. r1..r7 are local variables. r0 is special, it is global and it is used to return a value to the caller. The following code (Fig. 3) shows an example of the use of this instruction set. This assembly language fragment shows a routine to swap two elements of the array "data" and how the main function passed parameters; i, j and call "swap":

```
fun swap a b [ t tmp dp ]
  ld dp data
  ldx t dp a          ; t=data[a]
  ldx tmp dp b        ; tmp=data[b]
  stx tmp dp a        ; data[a]=tmp
  stx t dp b          ; data[b]=t
  ret 0

fun main [ i j ]
  ...
  mvx r1 i
  mvx r2 j
  call swap
  ...
```

Figure 3. A fragment of an assembly program

3. Register Window

A "frame pointer", FP, is a pointer to the base of a current activation record. Accessing a register is relative to FP.

$$rn = R[FP-n] \quad (1)$$

where R[] is the buffer.

The structure of an activation record is as follows:

```
hi

old pc  <- FP
r1
r2
...
r7

low
```

Figure 4. The structure of an activation record

The size of the current activation record is specified in the "call" and "ret" instructions, so it is not necessary to store that information in the activation record.

To directly support the creation and deletion of an activation record, a part of stack segment is cached into the register set. The registers become a buffer storing the most recent activation record. This buffer is implemented as a circular buffer (Fig. 5).

When the buffer becomes overflow, such as the creation of a new activation record, the oldest elements will be "spill" to the memory. On returning from a function call and deleting the current activation record, an underflow may occurs. When this happens, the buffer is restored by "pull" old elements from the memory. The larger buffer will reduce the number of spill/pull.

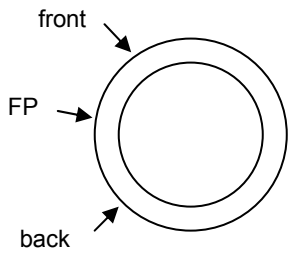

Figure 5. The circular register buffer

The maximum size of an activation record is eight. The size of buffer should be at least twice the maximum size of an activation record to prevent "thrashing". Two pointers: back, front, are used to keep track of the register window. FP is between back and front. This constraint is always true:

$$front - back + 1 \leq W \quad (2)$$

where W is the size of buffer.

Please note that, the arithmetic operations on these pointers: front, back, FP, must be modulo W as the buffer is circular of size W.

The spill/pull conditions can be described as follows: when accessing a register x, if x is outside the window and overflow/underflow is (front - back + 1 > W)

1) x > front move front up,

```
f' = front,
front = x,
if overflow then
  move back up
  b' = back,
  back = back + (front - f'),
  spill registers (b'.. back) to memory
```

2) x < back move back down,

```
b' = back,
back = x,
pull registers (b'.. back) from memory
if underflow  then
  move front down
  front = front - (b' - back)
```

When underflow occurs, it is not necessary to pull registers because it is not the current activation record. In fact, the "forwarding" register, (registers between FP and front), will be used only to pass parameters. They will never be overflown into the current activation record as the size of register window is at least twice the size of maximum activation record. There is no need to "spill" there registers when the front is moved down.

4. Experimental results

The following benchmark programs are used:

bubble	sort 20 items
hanoi	move 6 disks
matmul	multiply 8x8 matrices
perm	permuting 4 digits
quick	sort 20 items
sieve	find prime ≤ 500

Figure 6. The benchmark programs

To measure the effectiveness of the proposed scheme, this instruction set is compared to its predecessor, sm3 [11]. Sm3 is a stack-based 16-bit processor, it is used as a reference. This processor has byte-coded instructions, with the size one, two or three bytes depending on the size of its argument. This reference instruction set is typical for a byte-coded instruction set. Table 1 compare the static code size in byte. Table 2 compare the dynamic instruction count (no. of instruction). Sizing the buffer, Table 3 shows the number of spill/pull of all tested programs.

Table 1. The static code size (byte), the proposed ISA xs1, the reference sm3, average xs1/sm3 is 0.50

	sm3	xs1	xs1/sm3
bubble	282	142	0.50
hanoi	200	104	0.52
matmul	575	288	0.50
perm	249	102	0.41
quick	353	186	0.53
sieve	381	194	0.51

Table 2. The dynamic instruction count (no. of instuctions), the proposed ISA xs1, the reference sm3, average xs1/sm3 is 0.41

	sm3	xs1	xs1/sm3
bubble	11925	4379	0.37
hanoi	2317	1198	0.52
matmul	13886	6566	0.47
perm	5469	2083	0.38
quick	3972	1853	0.47
sieve	17151	4376	0.26

Table 3 Number of spill+pull of varying buffer size

no.of reg	16	24	32
bubble	0	0	0
hanoi	165	71	29
matmul	66	0	0
perm	548	212	52
quick	354	246	230
sieve	2	0	0

It is not possible to compare the number of clock cycle as the implementation of xs1 has not been completed. The cost of spill/pull depends on the access time of the memory. Because of the register buffer, accessing local variables, which is the most frequent, is fast. The register buffer is the internal fast register. From Table 1, the executable code size is half of the size of the reference so the goal of achieving a small executable is satisfied. Table 2 indicates the performance comparsion. The performance of xs1 chip should be good as it executed around 41% the number of instruction count of the reference. The effect, together with the fast instruction fetch, as xs1 has 16-bit fixed length instruction format, will result in very good performance compared to the reference. The average

speedup will be at least 60% (more than 2 times faster), not taking into account the cost of spill/pull. The number of spill/pull in Table 3 is reasonably small compared to the number of executed instruction. It should not have a huge impact on the cost.

5. Related work

A stack-based byte-coded instruction set is well-known to be small, for example JVM [12]. However, to reduce the number of instruction, a three-argument register-based instruction set is more compact. A proposal to combine the best of these two ideas is introduced in [13], where the instruction set is three-argument register-based but includes an automatic register windowing to manage the activation record during call/return.

One disadvantage of three-argument register-based instruction set is the size of each instruction. There are many fields for each instruction hence the size is not small. The popular approach to reduce the size of each instruction is to limit the range of argument, such as the number of register, the size of literal contained in the instruction. This approach is used in two well-known products widely used in mobile devices, ARM/Thumb [14] and MIPS/MIPS16 [15] where the compact form of their instruction sets are available.

The use of register windowing to manage parameter passing was invented at the period of RISC concept in RISC1[16] and still use in present in SPARC [17]. The caching of stack segment into an on-chip buffer has been done in Picojava chip [18].

6. Conclusion

This work proposed a design of a compact code instruction set for a processor suitable for embedded applications. The main goal is to achieve a small executable code. Using a small set of benchmark, the static of size is half of the reference. The number of executed instruction is also greatly reduced to less than half of the reference. To evaluate the performance, the detailed design of microarchitecture must be completed so that the cost of spill/pull of register buffer can be evaluated.

7. References

[1] M. Kozuch, and A. Wolfe, "Compression of embedded system programs", Proc. Int. Conf. on Computer Design: VLSI in Computers & Processors. IEEE Computer Society Press, Los Alamitos, Calif. 1994.

[2] C. Lefurgy, P. Bird, I. Chen, and T. Mudge, "Improving code density using compression techniques", Int. Symposium on Microarchitecture 30 (1997).

[3] I. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression", Proc. of the ACM SIGPLAN'97 Conf. on Programming Languages Design and Implementation 32 (15 – 18 June 1997) : 358 - 365

[4] M. Game, and A. Booker, "CodePack: Code Compression for PowerPC Processors", MicroNews 5 (1) , IBM, 1999.

[5] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a high performance code compression method", Proc. Annual Int. Symposium on Microarchitecture 32nd (1999) : 93-102

[6] K. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors", Proc. ACM SIGPLAN '99 Conf. on Programming language design and implementation, May 1-4, 1999, Atlanta, GA, pp.139-149.

[7] H. Lekatsas, J. Henkel, W. Wolf, "Code Compression for Low Power Embedded System Design", Proc. of the 37th Conf. on Design automation, 2000.

[8] W. Evans and C. Fraser, "Bytecode Compression via Profiled Grammar Rewriting", in ACM Sigplan Conference on Programming Language Design and Implementation, 2001, pp.148-155.

[9] V. Kotrajaras, P. Chongstitvatana, "Nibbling Java byte code for resource-critical devices", National Conf. of Computer Science and Engineering, Thailand, 2003.

[10] P. Nanthanavoot, P. Chongstitvatana, "Code-size reduction for embedded systems using bytecode translation unit", ECTI 2004.

[11] A. Burutarchanai, P. Nanthanavoot, C. Aporntewan, P. Chongstitvatana, "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON 2004, 21-24 November 2004, Thailand.

[12] B. Joy (Ed), G. Steele, J. Gosling, G. Bracha, Java(TM) Language Specification (2nd Ed), Addison Wesley Pub., 2000.

[13] P. Chongstitvatana, "The art of instruction set design", invited paper in Conf. of Electrical Engineering, Thailand, 2003.

[14] Advanced RISC Machines Ltd. An Introduction to Thumb. March 1995.

[15] K. D. Kissell. MIPS16: High-density MIPS for the Embedded Market. In Proc. of Real Time Systems '97 (RTS97), 1997.

[16] D. Patterson, "Reduced instruction set computers", Communications of the ACM, vol.28, no.1, 1985, pp.8-21.

[17] http://www.sun.com/processors/

[18] H. McGhan and M. O'Conner, "PicoJava : a direct execution engine for Java bytecode", IEEE Computer, Vol.31 No. 10, 1998.