Electronic Design®

By RAFAEL TAUBINGER,
Technical Marketing Specialist, IAR Systems

Techniques to Minimize Code Footprint in RISC-V-Based Apps

In this article, we'll look how developers can help the compiler make better decisions about what to do with their code to achieve optimizations in RISC-V-based applications.

The RV32I is the base instruction set that can get the standard extensions listed in *Figure 1*, such as:

- M (Integer Multiplication)
- A (Atomic Instructions)
- F (Single Floating Point)
- D (Double Floating Point)
- C (Compressed Instructions)
- B (Bit Manipulations)
- and so on….

Most extensions *(Fig. 1)* are ratified or frozen, but new ones are currently being worked on. An example of supported extension in various cores can be seen in *Figure 2*.

If we take the generic device RV32, we can recognize that it supports M, F, D, and C. C (Compressed Instructions) reduces static and dynamic code size by adding short 16-bit instructions for operations, resulting in average 25%-30% code-size reduction, and leading to lower power consumption and memory use. In addition, the RV32E base instruction set (embedded) is designed to provide an even smaller base core for embedded microcontrollers with 16 registers.

Designers are free to implement their own extensions for specific needs, e.g., machine learning, low-power application, or optimized SoC for metering and motor control. The purpose of the standard extensions or custom extension is to achieve faster response time from calculations and processing performed in hardware that require mostly one or just a few cycles.

Why Professional Tools for RISC-V?

With the growth of RISC-V, the need arises for professional tools that can take full advantage of the core features and extensions. A well-designed and optimized SoC also should run the best optimized code so that companies can innovate fast, have outstanding products, and derive the best cost benefit out of it.

When it comes to code density, every byte that can be saved counts. Professional tools help to optimize the application to best fit the required needs. By optimizing the application, customers will be able to save money by using devices with smaller memory or aggregate value by adding functionality to the existing platform *(Fig. 3)*.

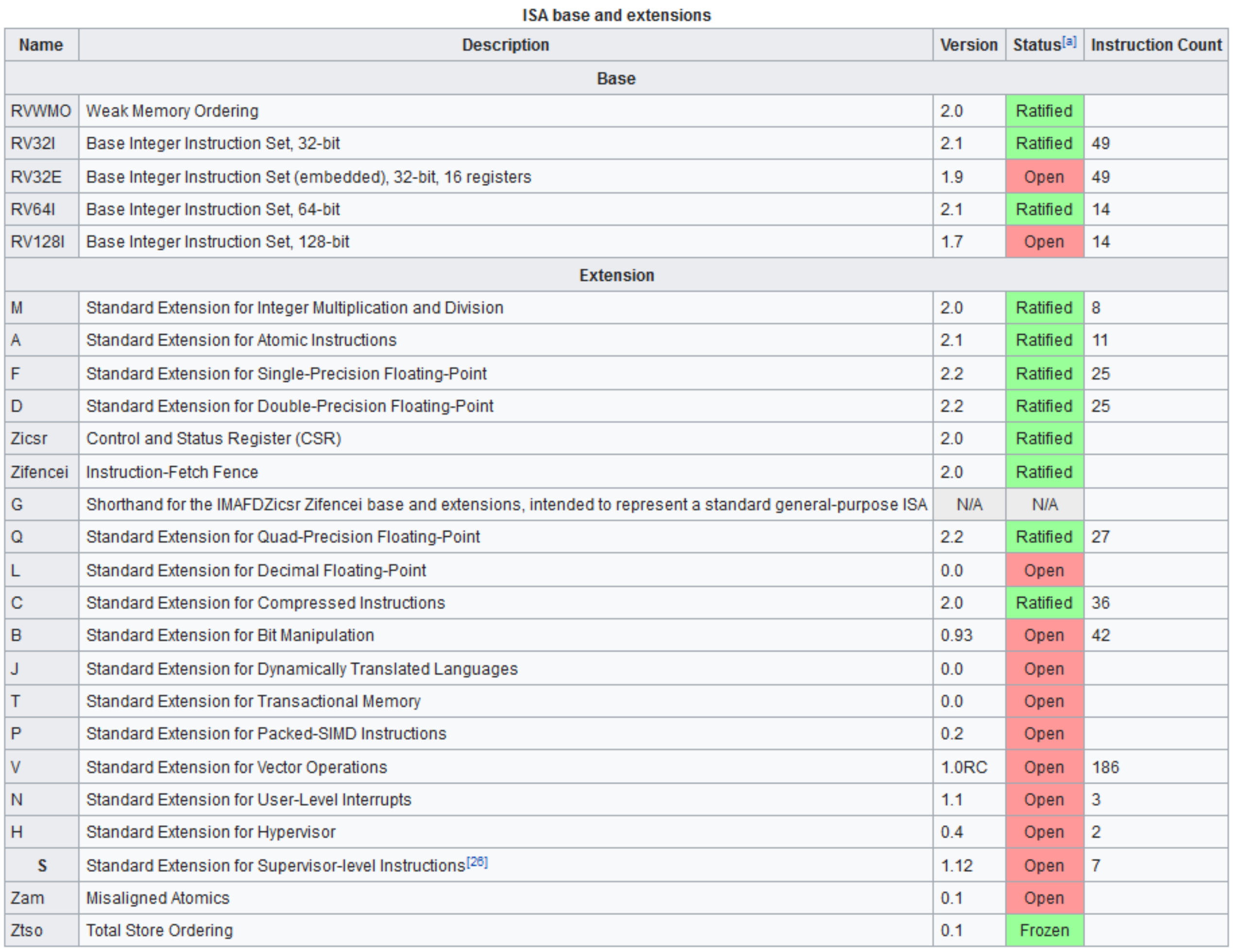
A professional compiler for RISC-V can generate, on average, 7%-10% smaller code when compared to other tools.

Writing Compiler-Friendly Code for Better Optimizations

An optimizing compiler tries to generate code that's both small and fast by selecting the right instructions in the best order for execution. It does so by repeatedly applying a number of transformations to the source program. Most optimizations follow mathematical or logical rules based on a sound theoretical foundation. Other transformations are based on heuristics, where experience has shown that some transformations often result in good code or open up opportunities for further optimization.

So, the way you write your source code can determine whether an optimization can be applied to your program or not. Sometimes small changes in the source code could significantly impact the efficiency of the code generated by the compiler.

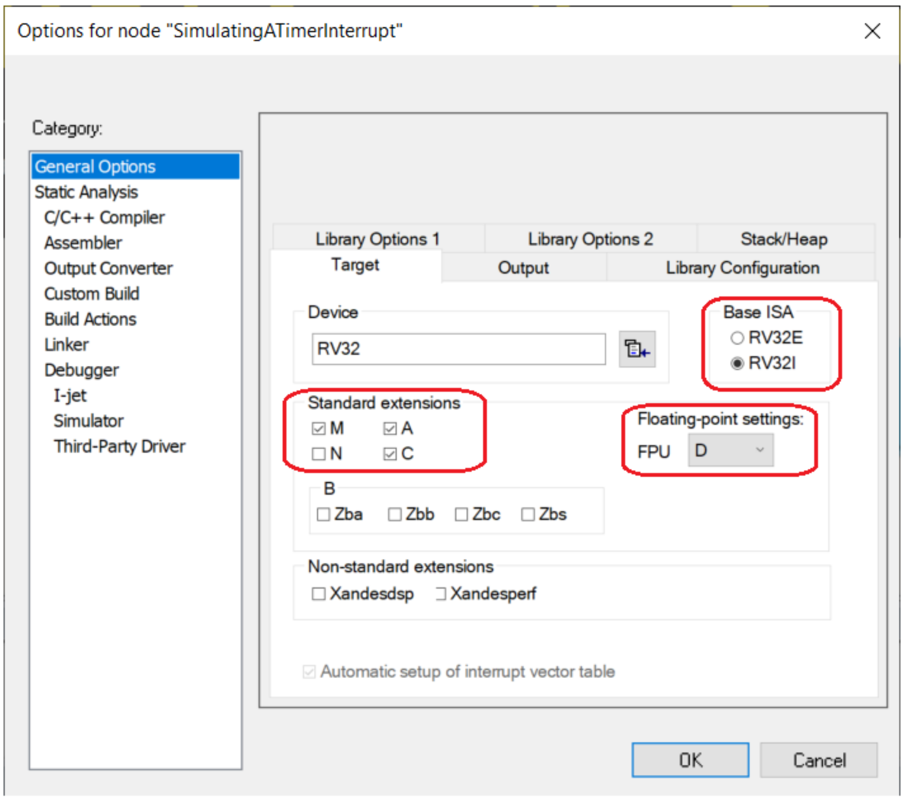
Trying to write your code on as few lines as possible, using ?:-expressions, postincrements, and comma expressions to squeeze in a lot of side effects in a single expression, will not make the compiler generate more efficient code. The best hint is to write your code in a style that's easy to read.



ISA base and extensions

Name	Description	Version	Status[a]	Instruction Count
Base				
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified	49
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open	49
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified	14
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open	14
Extension				
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified	8
A	Standard Extension for Atomic Instructions	2.1	Ratified	11
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified	25
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified	25
Zicsr	Control and Status Register (CSR)	2.0	Ratified	
Zifencei	Instruction-Fetch Fence	2.0	Ratified	
G	Shorthand for the IMAFDZicsr Zifencei base and extensions, intended to represent a standard general-purpose ISA	N/A	N/A	
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified	27
L	Standard Extension for Decimal Floating-Point	0.0	Open	
C	Standard Extension for Compressed Instructions	2.0	Ratified	36
B	Standard Extension for Bit Manipulation	0.93	Open	42
J	Standard Extension for Dynamically Translated Languages	0.0	Open	
T	Standard Extension for Transactional Memory	0.0	Open	
P	Standard Extension for Packed-SIMD Instructions	0.2	Open	
V	Standard Extension for Vector Operations	1.0RC	Open	186
N	Standard Extension for User-Level Interrupts	1.1	Open	3
H	Standard Extension for Hypervisor	0.4	Open	2
S	Standard Extension for Supervisor-level Instructions[26]	1.12	Open	7
Zam	Misaligned Atomics	0.1	Open	
Ztso	Total Store Ordering	0.1	Frozen	

1. These are the standard extensions for the RISC-V ISA. (Courtesy of Wikipedia - https://en.wikipedia.org/wiki/RISC-V)



Device: RV32

Base ISA: RV32E [✓ RV32I]

Standard extensions: [✓ M] [✓ A] N [✓ C]

Floating-point settings: FPU D

B: Zba Zbb Zbc Zbs

Non-standard extensions: Xandesdsp Xandesperf

[✓ Automatic setup of interrupt vector table]

2. Standard extension support in the compiler is specified by these selections.

Developers can help the compiler make better decisions by paying attention to the following hints in the source code:

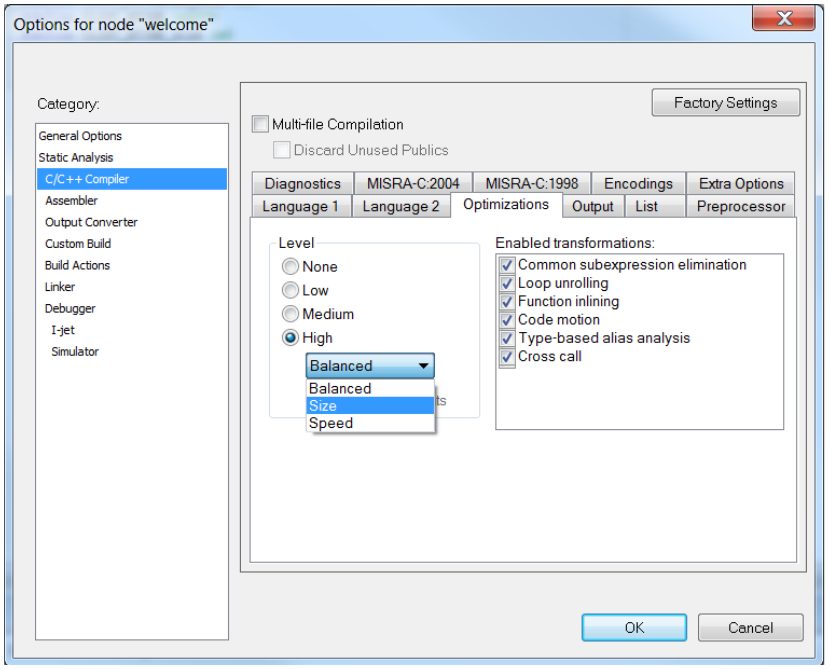
1. Make a function call only once. A compiler generally has difficulty looking into common subexpressions because the subexpressions can have side effects that the compiler may not know a priori if they're necessary. Hence, the compiler will make multiple calls to the same function when instructed to do so, which wastes code space and execution overhead. It's better to assign the function to a variable (which will most likely be stored in a register) and perform operations while it's in an easily accessed register *(Fig. 4)*.

2. Pass by reference rather than by copy. When you pass a pointer to a primitive rather than the primitive itself, you save the compiler the overhead of copying that primitive somewhere in RAM or in a register. For a large array, this can save quite a bit of execution time. Passing by copy will force the compiler to insert code to copy the contents of the primitive.

3. Use the correct data size. Some MCUs like an 8051 or AVR are 8-bit micros; some like the MSP430 are 16-bit; and some like Arm and RISC-V are 32-bit. When using an "unnatural" size for your core, then the compiler must create extra overhead to interpret the data contained therein, e.g., a 32-bit MCU will need to do shift, mask, and sign-extend operations to get to the value needed to perform its operation. It's therefore best to use the natural size of the MCU for your data types unless there's a compelling reason not to, i.e.. doing I/O. You also need a precise number of bits, or bigger types (such as a character array) would take up too much memory.

4. Using signedness appropriately. The signedness of a variable can affect the code that's generated by the compiler. For example, division by a negative number is treated differently (by the rules of the C language) than that for a positive number. Ergo, if you use a signed number that will never be negative in your application, you can incur an extra test-and-jump condition in your code that wastes both code space and execution time. In addition, if the purpose of a variable is to do bit-manipulation, it should be unsigned or you could have unintended consequences when doing shifting and masking.

5. Avoid becoming a castaway. C will often perform implicit casts (e.g., be-


Options for node "welcome"

Category: General Options, Static Analysis, **C/C++ Compiler**, Assembler, Output Converter, Custom Build, Build Actions, Linker, Debugger, I-jet, Simulator

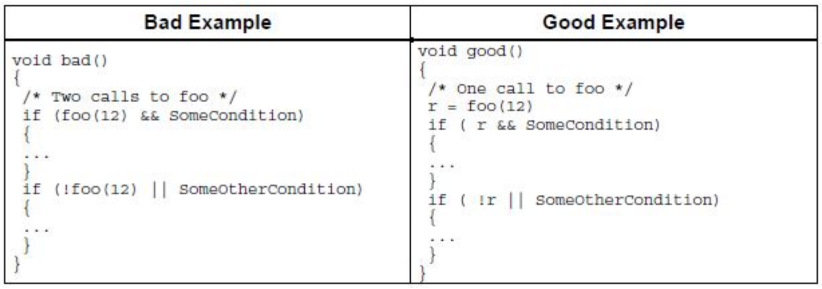
☐ Multi-file Compilation
☐ Discard Unused Publics

Tabs: Diagnostics, MISRA-C:2004, MISRA-C:1998, Encodings, Extra Options, Language 1, Language 2, **Optimizations**, Output, List, Preprocessor

Level: None, Low, Medium, [✓ High] — Balanced (Balanced, Size, Speed)

Enabled transformations:
☑ Common subexpression elimination
☑ Loop unrolling
☑ Function inlining
☑ Code motion
☑ Type-based alias analysis
☑ Cross call

4. The compiler may be able to optimize redundant calls, but it's better to write the code properly.

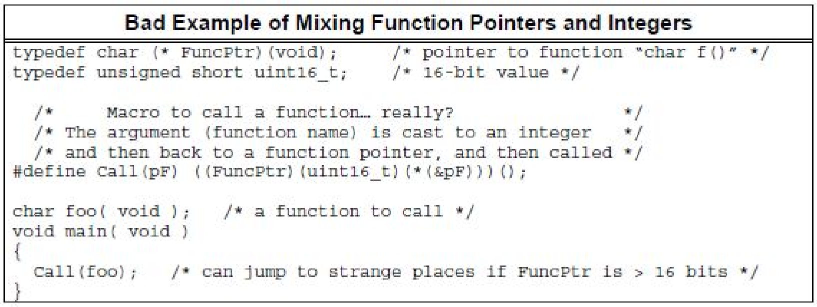

Bad Example

```
void bad()
{
 /* Two calls to foo */
 if (foo(12) && SomeCondition)
 {
 ...
 }
 if (!foo(12) || SomeOtherCondition)
 {
 ...
 }
}
```

Good Example

```
void good()
{
 /* One call to foo */
 r = foo(12)
 if ( r && SomeCondition)
 {
 ...
 }
 if ( !r || SomeOtherCondition)
 {
 ...
 }
}
```

3. Compiler optimization transformations can be selectively enabled.


Bad Example of Mixing Function Pointers and Integers

```
typedef char (* FuncPtr)(void);      /* pointer to function "char f()" */
typedef unsigned short uint16_t;     /* 16-bit value */

  /*     Macro to call a function… really?              */
  /* The argument (function name) is cast to an integer   */
  /* and then back to a function pointer, and then called */
#define Call(pF)  ((FuncPtr)(uint16_t)(*(&pF)))();

char foo( void );   /* a function to call */
void main( void )
{
  Call(foo);   /* can jump to strange places if FuncPtr is > 16 bits */
}
```

5. Avoid becoming a castaway by using the correct cast.

Example

```
unsigned char gGlobal;  /* global variable */

void foo(int x)
{
  unsigned char  ctemp;
  ctemp = gGlobal;  /* should go into register */
  ...
  /* Calculations involving ctemp, i.e. gGlobal */
  bar(z);  /* does not read or write gGlobal, otherwise error */
  /* More calculations on ctemp */
  ...
  gGlobal = ctemp;   /* make sure to remember the result */
}
```

6. This is the right way to handle global variables and registers.

"Clever" solution

```
unsigned long int a;
unsigned char     b;

/* Move bits 0..20 to positions 11..31
 * If non-zero, first ! gives 0      */
b |= !!(a << 11);
```

Straightforward solution

```
unsigned long int a;
unsigned char     b;

/* Straight-forward if statement */
if( (a & 0x1FFFFF) != 0)
  b |= 0x01;
```

7. Don't Write Clever Code 1: Comments are good but not an excuse for bad, clever code.

"Clever" solution

```
int bar(char *str)
{
  /* Calculating with result of */
  /* comparison.                */
  return foo(str+(*str=='+'));
}
```

Straightforward solution

```
int bar(char *str)
{
  if(*str=='+')
    str++;
  return foo(str);
}
```

8. Don't Write Clever Code 2: You're not making something more efficient by writing bad code.

tween floats and integers and between ints and long longs) and these are not free. Casting from a smaller type to a bigger type will use sign extend operations, casting to and from a float will introduce the need for the floating-point library (which can dramatically increase the size of your code). Naturally, you should avoid making explicit casts as much as possible to sidestep this extra overhead. This problem can be easily seen when a desktop programmer who is accustomed to using ints and function pointers interchangeably makes the jump to embedded programming *(Fig. 5)*.

6. Use function prototypes. If a prototype doesn't exist, then the C language rules dictate that all arguments must be promoted to an integer and—as previously discussed—this can link in unnecessary overhead from a runtime library.

7. Read global variables into temporary variables. If you're accessing a global variable several times within a function, you might want to read it into a local temporary variable. Otherwise, every time you access this variable, it will need to be read from memory. By putting it into a local temporary variable, the compiler will probably allocate a register to the value so that it can more efficiently perform operations on it *(Fig. 6)*.

8. Refrain from inlining assembly. Using inline assembly has a very deleterious impact on the optimizer. Since the optimizer knows nothing about the code block, it can't optimize it. Moreover, it's unable to do instruction scheduling of the handwritten block since it doesn't know what the code is doing (this can be especially damaging to DSPs). On top of that, the developer must inspect the handwritten code each time to make sure that it's correctly interspersed in the optimized C-code so that it doesn't produce unintended side effects. The portability of inline assembler is very poor, so it will need to be rewritten (and its ramifications understood) if you ever decide to move it to a new architecture. If you must inline assembler, you should split it into its own assembler file and keep it separated from source.

9. Don't write clever code. Some developers erroneously believe that writing fewer source lines and making clever use of C constructions will make the code smaller or faster (i.e., they're doing the compiler's job for it). The result is code that's difficult to read, impossible to understand for anyone but the person who originally wrote it and harder to compile. Writing it in a clear and straightforward manner improves the readability of your code and helps the compiler to make more informed decisions about how best to optimize your code.

For example, assume that we want to set the lowest bit of a variable b if the lowest 21 bits of another variable are set. The clever code uses the ! operator in C, which returns zero if the argument is non-zero ("true" in C is any value except zero), and one if the argument is zero. The straightforward solution is easy to compile into a conditional followed by a set bit instruction, since the bit-setting operation is obvious and the masking is likely to be more efficient than the shift. Ideally, the two solutions should generate the same code. The clever code, however, may result in more code since it performs two ! operations, each of which may be compiled into a conditional *(Fig. 7)*.

Another example involves the use of conditional values in calculations. The "clever" code will result in larger machine code since the generated code will contain the same test as the straightforward code and adds a temporary variable to hold the one or zero to add to str. The straightforward code can use a simple increment rather than a full addition and doesn't require the generation of intermediate results *(Fig. 8)*.

10. Access structures in order. If you order your structure

whereby you step from one element of the structure to the next instead of jumping around in the structure, the compiler can take advantage of increment operations to access the next element of the structure instead of trying to calculate its offset from the structure pointer. In a statically allocated structure, doing this will not save code since the addresses are computed a priori. However, in most applications, these are done dynamically.

Conclusion

Embedded compilers have evolved greatly over the last 30 years, especially as it pertains to their optimization capabilities. Modern compilers employ many different techniques to produce very tight and efficient code so that you can focus on writing your source in a clear, logical, and concise manner. Every developer strives to achieve the optimum efficiency in their software. Compilers are amazingly complex pieces of software that are capable of great levels of optimization, but by following these simple hints, you can help it achieve even greater levels of efficiency.

Rafael Taubinger is the Technical Marketing Specialist at IAR Systems, based at the headquarters in Sweden. Prior to his current role, he served as Global FAE Manager and Senior FAE at IAR Systems. He has over 15 years of experience in the embedded industry and holds a B.S. degree in Electrical Engineering with emphasis in Electronics and a Master of Business Administration (M.B.A) degree in Business Management.

Rafael is highly skilled in C/C++ programming languages for 8-, 16- and 32-bit microcontrollers, and has 2000+ hours of technical training in US, Latin America and Europe about embedded applications, C/C++ programming, safety coding, system reliability, efficiency, modeling, code analysis and best development practices.