FEATURE ARTICLE

Sandeep Dutta

Anatomy of a Compiler

A Retargetable ANSI-C Compiler

Wouldn't it be great if there was an affordable C compiler that was specifically designed to the needs of 8-bit micros? Not to worry, Sandeep explains the inner workings of just such a compiler, which happens to be quite affordable—it's free!

Small device C compiler (SDCC) is an open-source optimizing C compiler developed primarily for 8-bit MCUs. Although there are several free C compilers available to address the general-purpose processors (GCC, LCC), there are no free C compilers available except SDCC that address specific needs of 8-bit MCUs. In this article, I'll explore SDCC and some of the special considerations when designing a compiler for 8-bit MCUs.

I'll discuss the Intel 8051 because it's a widely used MCU. The concepts have been implemented in SDCC and the source code is available under GPL in the hope that other people will find it useful and contribute (either by providing feedback or making enhancements). Several commercial compilers implement the concepts demonstrated here.

ADDRESS SPACES

Unlike their 32-bit brethren, most of the 8-bits use Harvard architecture, which means the code and data reside in different address spaces and are usually accessed using different instructions. For example, the 8051 family of controllers has three address spaces (four if you consider the upper 128 bytes of internal RAM as a different address space). C language allows for storage classes, however they are restricted to const, volatile, static, auto, and register. Although these are adequate for Von Neuman architecture, they're not sufficient for architectures with many address spaces.

The problem is more complex when you consider pointers. Where does the pointer reside? Which address space is it pointing to? Also consider library routines, which take pointers as parameters. Do you need to write library routines for all the combinations of address spaces?

SDCC handles this problem by adding keywords for new storage classes. Using the 8051 as an example, SDCC has storage class specifiers for each MCU address space. Listing 1 shows examples of declaring variables in different address spaces.

Frequently, a variable must be allocated at a specific/absolute address (i.e., a memory-mapped I/O device). Again, standard C doesn't provide for this, you would have to provide a special assembler routine (or inline assembly code). SDCC, however, provides a special keyword "at" to specify an absolute address for a variable. The memory-mapped I/O device can then be accessed in an expression using standard C syntax.

POINTERS

The same concept of storage class extension can be used to solve the pointer problem. Listing 2 shows different ways to specify pointers. This leaves the problem of library routines, not knowing which storage class the pointer points to at compile time. The SDCC solution is generic 3-byte pointers; the third (highest order) byte contains information about the pointed at object's storage class.

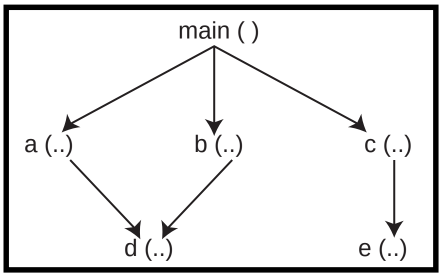

Figure 1—*The parameter locals of functions a, b, and c can be overlaid.*

At runtime, the compiler calls a routine that determines the storage class from the third byte and uses the appropriate instruction to fetch or store data. This technique increases code and data size, but is a compromise to allow coding general-purpose library routines, like `strcmp`.

STACK

The most difficult obstacle in programming small devices (such as the 8051) in high-level language like C is the limited stack space available for local variables and parameter passing. Using registers for parameter passing lessens the problem, but you still must allocate local variables. SDCC solves this problem by treating parameters and local variables as static (at the expense of re-entrancy). It goes a step further by overlaying parameters and local variables of leaf functions (i.e., functions that call no other functions) to the same memory region.

What if you need the re-entrancy? You could either compile the entire source file with the stack-auto compiler option (all functions in the source file will be treated as re-entrant), or you can choose only specific functions to be reentrant by using the reentrant keyword in the function declaration. SDCC allocates parameters and local variables of a reentrant function on the stack.

The current version of the compiler only overlays local variables and parameters of a leaf function, but this isn't enough in some cases. Development is underway to do function call tree analysis, which would allow the compiler to overlay parameters and local variables of functions that don't belong to the same call sub-tree.

Consider the call tree illustrated in Figure 1. In this case, local variables and parameters (auto variables) of functions *d()* and *e()* can be overlaid (and are by the compiler). In addition, auto variables of functions *a()*, *b()*, and *c()* can be overlaid with each other, because they don't call each other and are not present in the call trees of any of their children. This kind of overlaying needs to be done by the linker because the compiler has only a partial view of the call tree.

Listing 1—*Here's the pointer declarations with extended storage classes.*

```
/* the following array will be declared in program memory
    MOVC instruction will be used to access this array */
 code  short array_in_code[3] = {0x01,0x02,0x03};

/* The integer will be allocated in internal ram space
MOV instruction will be used access this data item */
data unsigned char in_internal_ram ;

/* this will allocated in the external RAM and MOVX will
    be used to access this data item */
xdata char array_in_external_ram[9];

/* This variable will be allocated at address 0x8000 of
the external RAM */
xdata at 0x8000 ADC_PORTA;

/* pointer in data space points to object in xdata */
xdata char * p;

/* pointer in xdata space points to object in code space */
code char * xdata p;

/* pointer in code space points to object in data space */
data char * code p;
```

Listing 2—*This sample code illustrates iCode generation and optimizations.*

```
xdata int * p;
int gint;
                    /* This function does nothing useful. It is
                      used for the purpose of explaining iCode */
short function (data int *x)
{
short i=10;         /* dead initialization eliminated */
short sum=10;       /* dead initialization eliminated */
short mul;
int j ;
while (*x) *x++ = *p++;
sum =0 ;
mul =0;
                    /* compiler detects i,j to be induction vari-
                      ables */
for (i = 0, j = 10 ; i < 10 ; i++, j—) {
    sum += i;
    mul += i * 3;   /* this multiplication remains */
    gint += j * 3;  /* this multiplication changed to addition */
}
return sum+mul;
}
```

INTERNAL DETAILS

The current version of SDCC can generate code for Intel 8051 and Z80 MCUs. It's easy to retarget for other 8-bit MCUs. Let's take a look at some of the internals of the compiler.

Parsing involves reading the input source file and creating an Annotated Syntax Tree (AST). This phase also involves propagating types (annotating each node of the parse tree with type information) and semantic analysis. There are some MCU-specific parsing rules. For example, the extended storage classes are MCU specific: while there may be an xdata storage class for 8051, there's no such storage class for the Z80 or Atmel AVR. SDCC allows MCU-specific storage class extensions to be treated as a storage class specifier when parsing 8051 C code, but to be treated as a C identifier when parsing Z80 or Atmel AVR C code.

In the intermediate code generation phase, the AST is broken into three operand forms (iCode). These forms are represented as doubly linked lists. iCode is the term given to the intermediate form generated by the compiler. Listing 3 shows examples of iCode generated for simple C source functions.

The bulk of target-independent optimizations is performed during optimization. Optimizations include constant propagation, common sub-expression elimination, loop-invariant code movement, strength reduction of loop induction variables, and dead-code elimination.

During the intermediate code generation phase, the compiler assumes the target machine has an infinite number of registers and generates many temporary variables. The live range computation determines the lifetime of each of these compiler-generated temporaries. iCode example sections in Listing 3 show the live range annotations for each operand. Note that each iCode is assigned a number in the order of its execution, which compute the live ranges. The `from` is the iCode number that first defines the operand and `to` signifies the iCode that uses this operand last.

Listing 3—*This is the iCode generated for the sample code in Listing 2.*

```
Sample.c (5:1:0:0)   _entry($9) :
Sample.c(5:2:1:0)   proc _function [lr0:0]{function short}
Sample.c(11:3:2:0)    iTemp0 [lr3:5]{_near * int}[r2] = recv
Sample.c(11:4:53:0) preHeaderLbl0($11) :
Sample.c(11:5:55:0)    iTemp6 [lr5:16]{_near * int}[r0] := iTemp0
   [lr3:5]{_near * int}[r2]
Sample.c(11:6:5:1)  _whilecontinue_0($1) :
Sample.c(11:7:7:1)    iTemp4 [lr7:8]{int}[r2 r3] = @[iTemp6
   [lr5:16]{_near * int}[r0]]
Sample.c(11:8:8:1)    if iTemp4 [lr7:8]{int}[r2 r3] == 0 goto
   _whilebreak_0($3)
Sample.c(11:9:14:1)   iTemp7 [lr9:13]{_far * int}[DPTR] := _p
   [lr0:0]{_far * int}
Sample.c(11:10:15:1)   _p [lr0:0]{_far * int} = _p [lr0:0]{_far *
   int} + 0x2 {short}
Sample.c(11:13:18:1)   iTemp10 [lr13:14]{int}[r2 r3] = @[iTemp7
   [lr9:13]{_far * int}[DPTR]]
Sample.c(11:14:19:1)   *(iTemp6 [lr5:16]{_near * int}[r0]) :=
   iTemp10 [lr13:14]{int}[r2 r3]
Sample.c(11:15:12:1)   iTemp6 [lr5:16]{_near * int}[r0] = iTemp6
   [lr5:16]{_near * int}[r0] +
                                                       0x2 {short}
Sample.c(11:16:20:1)   goto _whilecontinue_0($1)
Sample.c(11:17:21:0)_whilebreak_0($3) :
Sample.c(12:18:22:0)   iTemp2 [lr18:40]{short}[r2] := 0x0 {short}
Sample.c(13:19:23:0)   iTemp11 [lr19:40]{short}[r3] := 0x0 {short}
Sample.c(15:20:54:0)preHeaderLbl1($13) :
Sample.c(15:21:56:0)   iTemp21 [lr21:38]{short}[r4] := 0x0 {short}
Sample.c(15:22:57:0)   iTemp23 [lr22:38]{int}[r5 r6] := 0xa {int}
Sample.c(15:23:58:0)   iTemp17 [lr23:38]{int}[r7 r0] := 0x1e {int}
Sample.c(15:24:26:1)_forcond_0($4) :
```

(continued)

The register allocation determines the type and number of registers needed by each operand. In most MCUs, only a few registers can be used for indirect addressing. The compiler tries to allocate the appropriate register to pointer variables.

Listing 3 shows the operands annotated with the registers assigned to them. The compiler tries to keep operands in registers. The compiler uses several schemes to achieve this. When the compiler runs out of registers, it checks if there are any live operands that are not used or defined in the current basic block being processed. If found, it will push that operand and use the registers in this block. Then, the operand will be popped at the end of the basic block.

There are other MCU-specific considerations in this phase. Some MCUs have an accumulator so short-lived operands may be assigned to the accumulator instead of a general-purpose register.

A complete table that defines the iCode operations supported by the compiler is available on the *Circuit*

Listing 3—*continued*

```
Sample.c(15:25:27:1)   iTemp13 [lr25:26]{char}[CC] = iTemp21
   [lr21:38]{short}[r4] < 0xa {short}
Sample.c(15:26:28:1)   if iTemp13 [lr25:26]{char}[CC] == 0 goto
   _forbreak_0($7)
Sample.c(16:27:31:1)   iTemp2 [lr18:40]{short}[r2] = iTemp2
   [lr18:40]{short}[r2] +
                                                  ITemp21
   [lr21:38]{short}[r4]
Sample.c(17:29:33:1)   iTemp15 [lr29:30]{short}[r1] = iTemp21
   [lr21:38]{short}[r4] * 0x3 {short}
Sample.c(17:30:34:1)   iTemp11 [lr19:40]{short}[r3] = iTemp11
   [lr19:40]{short}[r3] +
                                                   iTemp15
   [lr29:30]{short}[r1]
Sample.c(18:32:36:1:1) iTemp17 [lr23:38]{int}[r7 r0]= iTemp17
   [lr23:38]{int}[r7 r0]- 0x3 {short}
Sample.c(18:33:37:1)   _gint [lr0:0]{int} = _gint [lr0:0]{int} +
   iTemp17 [lr23:38]{int}[r7 r0]
Sample.c(15:36:42:1)   iTemp21 [lr21:38]{short}[r4] = iTemp21
   [lr21:38]{short}[r4] + 0x1 {short}
Sample.c(15:37:45:1)   iTemp23 [lr22:38]{int}[r5 r6]= iTemp23
   [lr22:38]{int}[r5 r6]- 0x1 {short}
Sample.c(19:38:47:1)   goto _forcond_0($4)
Sample.c(19:39:48:0)_forbreak_0($7) :
Sample.c(20:40:49:0)   iTemp24 [lr40:41]{short}[DPTR] = iTemp2
   [lr18:40]{short}[r2] +
                                                     ITemp11
   [lr19:40]{short}[r3]
sample.c(20:41:50:0)   ret iTemp24 [lr40:41]{short}
sample.c(20:42:51:0)_return($8) :
sample.c(20:43:52:0)   eproc _function [lr0:0]{ ia0 re0
   rm0}{function short}
```

Cellar web site. Code generation involves translating these operations into corresponding assembly code for the processor. This seems simple, but that's the essence of code generation. Some operations are generated in an MCU-specific manner. For example, the Z80 port doesn't use registers to pass parameters, so the Send and Recv operations won't be generated, and it doesn't support jumptables.

ICODE EXAMPLE

This section shows some details of iCode. The example C code isn't useful, but it illustrates the intermediate code generated by the compiler. Sample.c generates the iCode sequence in Listing 3.

In addition to the operands, each iCode contains information about the file name and line it corresponds to in the source file. The first field in the listing should be interpreted as follows:

File name (line number: iCode Execution sequence number: ICode hash table key: loop depth of the iCode).

The readable form of the iCode operation is found next. Each operand of this triplet form can be of three basic types—compiler generated temporary, user-defined variable, or a constant value. Note that local variables and parameters are replaced by compiler-generated temporaries. Live ranges are computed only for temporaries. Registers are allocated for temporaries only. Operands are formatted in the following manner:

Operand name [lr live-from: live-to] {
  type information} [registers allocated]

As mentioned, live ranges are computed in terms of the execution sequence of the iCodes. For example, the iTemp0 is a live `from` (i.e., first defined with execution sequence number 3) and is used last with number 5. For induction variables such as iTemp21, the live range computation extends the life from loopstart to end.

The register allocator used the live range information to allocate registers, the same registers may be used for different temporaries if their live ranges don't overlap. In addition, the allocator takes into consideration the type and usage of a temporary.

Some short-lived temporaries are allocated to special registers that have meaning to the code generator. The code generation makes use of this information to optimize a compare-and-jump iCode.

Several loop optimizations are performed by the compiler. It detects induction variables iTemp21(i) and iTemp23(j). And, the compiler does selective strength reduction (i.e., the multiplication of an induction variable in line 18 [gint = j × 3] is changed to addition, temporary iTemp17 is allocated and assigned an initial value, constant 3 is added for each loop iteration). The compiler does not change the multiplication in line 17, however, because the processor supports an 8 × 8 bit multiplication.

Note the dead code elimination optimization eliminated the dead assignments in line 7 and 8 to I and sum respectively.

READY, SET, COMPILE

You can download the compiler at sdcc.sourceforge.net. SDCC is an active project and, as with all GPL software, many people contributed. Recently, it was retargeted for Nintendo Gameboy.

The compiler is distributed as GPL software with the hope that you'll find it useful. The SDCC team believes in continuous improvement of the software so if you have any suggestions, feel free to send me an e-mail. ■

Sandeep Dutta is a compiler engineer working for WindRiver Systems Inc. She works on the DIAB optimizing C, C++, and Java compiler for 32-bit processors. You can reach her at sandeep.dutta@windriver.com.

SOFTWARE

A complete table of the iCode operations that are supported by the compiler as well as an additional sample code listing are available on the *Circuit Cellar* web site.