**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CURSO DE ENGENHARIA DE COMPUTAÇÃO**

# PORTE DO COMPILADOR SDCC PARA O PROCESSADOR ZR16S08

**TRABALHO DE CONCLUSÃO DE CURSO**

**Rafael Billig Tonetto**

**Santa Maria, RS, Brasil**
**2014**

# PORTE DO COMPILADOR SDCC PARA O PROCESSADOR ZR16S08

**Rafael Billig Tonetto**

Monografia apresentada à Universidade Federal
de Santa Maria
- UFSM, como requisito parcial
para obtenção do título de
**Engenheiro de Computação**

**Orientador: Prof. Carlos Henrique Barriquello**

**Santa Maria, RS, Brasil**
**2014**

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Curso de Engenharia de Computação**

A Comissão Examinadora, abaixo assinada, aprova a Monografia

**PORTE DO COMPILADOR SDCC PARA O PROCESSADOR ZR16S08**

elaborada por
**Rafael Billig Tonetto**

como requisito parcial para obtenção de título de
**Engenheiro de Computação**

**COMISSÃO EXAMINADORA**

**Carlos Henrique Barriquello, Dr.**
(Orientador)

**José Eduardo Baggio, Dr** (UFSM)

**João Baptista Martins, Dr** (UFSM)

Santa Maria, 10 de julho de 2014.

# Resumo

Monografia
Curso de Engenharia de Computação
Universidade Federal de Santa Maria

**PORTE DO COMPILADOR SDCC PARA O PROCESSADOR ZR16S08**
AUTOR: RAFAEL BILLIG TONETTO
ORIENTADOR: CARLOS HENRIQUE BARRIQUELLO
Data e Local da Defesa: Santa Maria, 10 de julho de 2014.

Este trabalho tem como intuito o porte do compilador de código aberto SDCC (Small Device C Compiler) para o processador ZR16S08. A principal motivação deste esforço vem da observação de que comumente o compilador oficial do MCU (microcontrolador) ZR16S08 gera programas não muito eficientes, com muitas instruções redundantes e/ou desnecessárias que podem ser otimizadas. Programas otimizados são de crucial importância para sistemas embarcados, visto que geralmente não possuem muita capacidade de processamento e limitações de memória. Neste sentido, fez-se um esforço a fim de se adquirir um novo e mais eficiente compilador para o processador ZR16S08. Questões como tempo de execução e memória de programas são a base para a avaliação de desempenho em qualquer sistema computacional. Desta forma, procurou-se projetar um compilador com capacidade para reduzir tais fatores ao máximo possível. Comparações de desempenho do novo compilador com o compilador oficial foram essenciais a fim de avaliar o real aumento de performance. Assim sendo, *benchmarks* foram aplicados a fim de coletar dados através de simulações, tais como número de ciclos executados pelos programas e a quantidade de instruções geradas pelos compiladores. Os mesmos programas de teste foram utilizados com o compilador oficial do ZR16S08 e o SDCC, desta forma, uma medida confiável de melhoria do novo compilador foi confirmada para o *benchmark* utilizado no presente trabalho.



**Palavras-chave:** Microcontroladores. Compiladores. Sistemas embarcados. Desempenho.

## Abstract

This study aims to target the open source SDCC (Small Device C Compiler) compiler for the ZR16S08 processor. The main reason for this effort comes from the fact that the official ZR16S08 compiler usually generates programs that do not have a high level of performance, since the programs have many redundant and obsolete instructions that can be optimized. Optimized programs have a crucial importance for embedded systems, since they usually do not have much processing and memory capacity. In this sense, an effort was done in order to acquire a new and more efficient compiler for the ZR16S08 microcontroller. Issues such as time processing and program size are the basis for performance evaluation of any computational system. Therefore, an attempt to project a compiler able to reduce these factors as much as possible was made. Performance comparisons between the new targeted compiler and the official ZR16S08 compiler were essential in order to evaluate the real acquired improvement. Thus, benchmarks were applied in order to collect data through simulations, such as the total amount of cycles executed by both programs and their sizes. The same test programs were compiled with SDCC and ZR16 Compiler, hence, a reliable measure of improvement was confirmed for the benchmark used in this present study.



**Keywords:** Microcontrollers. Compilers. Embedded Systems. Performance.

## SUMÁRIO

# INTRODUÇÃO

O presente trabalho tem o intuito de portar o compilador Small Device C Compiler (SDCC) para o processador ZR16S08. Este microcontrolador trata-se do primeiro MCU desenvolvido genuinamente no Brasil. Foi projetado na SMDH (Santa Maria Design House) e será discutido no capítulo 1. Neste contexto, portar significa adaptar ou estender o número de microcontroladores atualmente suportado pelo SDCC para o MCU acima citado. A principal motivação deste porte é a obtenção de um compilador capaz de gerar programas eficientes, que possuem tempo de execução e número de instruções tão pequenos quanto possível.

É importante que compiladores para sistemas embarcados otimizem bem seus códigos, pois os microcontroladores geralmente possuem capacidade de processamento e tamanho de memória limitados. Visto que o SDCC é um ótimo compilador otimizador e pelo fato de ser de código aberto, o mesmo foi escolhido como compilador alternativo para o MCU ZR16S08, já que, o compilador oficial deste microcontrolador não é otimizador.

Uma breve apresentação das características gerais do ZR16S08 será feita no capítulo 1. Como será apresentado também neste capítulo, o compilador oficial do ZR16S08 (ZR16 Compiler) não gera códigos muito eficientes. Nota-se que tal compilador gera um elevado número de instruções desnecessárias e/ou redundantes.

Visto que este trabalho se trata sobre compiladores, o capítulo 2 faz uma breve discussão sobre os conceitos básicos que os concernem. Este capítulo não faz nenhuma análise aprofundada sobre o assunto, porém, os conceitos fundamentais serão apresentados, visto que são essenciais ao entendimento do capítulo subsequente.

No capítulo 3 faz-se uma discussão e análise do compilador SDCC. Serão apresentadas as suas técnicas de otimização e geração de código. Este capítulo é fundamental ao entendimento do capítulo seguinte.

No capítulo 4 tem-se a descrição de como foi feito o porte do SDCC para o processador ZR16S08. Nesta etapa, será ilustrado como foram implementados os algoritmos responsáveis pela geração de algumas instruções, como atribuições, AND bit a bit (&), complemento (~) e comparações (== e !=).

O objetivo principal deste trabalho, além de obter o porte do compilador, é

fazer uma avaliação de desempenho considerando o tempo de execução e tamanho dos programas compilados pelo SDCC. Para tal, foi aplicado um *benchmark* para fins de comparação de desempenho dos programas compilados com o SDCC e ZR16 Compiler.

No capítulo 5 estão expostos os resultados das comparações acima citadas. Foram tabelados e ilustrados com gráficos o número de ciclos tomados pelos programas compilados pelo SDCC e ZR16 Compiler bem como os tamanhos dos programas gerados por ambos os compiladores.

# 1 O MICROCONTROLADOR ZR16S08

## 1.1 Introdução

O conhecimento das características básicas do MCU ZR16S08 é necessário a fim fazer o porte para o compilador SDCC, bem como fazer a análise do aumento ou diminuição de desempenho obtido do compilador portado.

Nesta seção será feita uma breve apresentação descrevendo a arquitetura, organização, conjunto de instruções e modos de endereçamento do processador.

É essencial ter um conhecimento prévio do conjunto de instruções e modos de endereçamento do processador para gerar as instruções corretamente e, se possível, otimizá-las, durante a fase do porte do compilador.

## 1.2 Características gerais

### 1.2.1 Organização e conjunto de instruções

O ZR16S08 é um processador de 8 bits RISC *(Reduced Instruction Set Computer* ou *Computador com um Conjunto Reduzido de Instruções)* de arquitetura Harvard que opera em uma frequência de 4 MHz.

Possui 24 instruções, mostradas no quadro 1.1. Tem um banco de 16 registradores, sendo 13 de uso geral. Possui uma memória EEPROM (*Electrically Erasable Programmable Read-Only Memory* ou *Memória Eletricamente Apagável e Programável Somente Leitura*) de 1024 X 16 bits e uma memória RAM (*Random Access Memory* ou *Memória de Acesso Aleatório*) de 256 x 8 bits. Também possui uma pilha de 4 posições e tem suporte a interrupções. A figura 1.1 mostra o diagrama de blocos ilustrando as funcionalidades do processador, e.g., conversor A/D e Timer. A figura 1.2 ilustra a organização geral do ZR16S08.

As instruções do processador são gravadas na EEPROM (memória de programa), sendo possível gravar 1024 instruções de 16 bits cada. Por questões de projeto, esta memória é dividida em 8 partições (páginas) de 128 x 16 bits. Cada página pode ser acessada de acordo com os bits p0, p1 e p2 do registrador R14.

A memória RAM de 256 x 8 bits é responsável por guardar os dados do programa e só pode ser lida ou escrita pelo registrador R0, segundo os modos de endereçamento mostrados no quadro 1.2.

De acordo com o quadro 1.3, os bits 11 e 10 são responsáveis por definir o modo de endereçamento do valor à esquerda da instrução (e.g., 00 faz referência ao valor do registrador). Os bits 9 e 8 definem o modo de endereçamento do valor direito da instrução (e.g., 11 significa valor imediato).

Os bits 12-15 das instruções definem os *opcodes* das operações (e.g., a instrução MOV tem *opcodes* 1101). O quadro 1.6 mostra os *opcodes* para cada uma das instruções do MCU.

O quadro 1.2 mostra como o hardware decodifica a instrução MOV. Os bits que identificam o *opcode*, modo de endereçamento, origem e destino são os *mesmos* para as instruções AND, OR , XOR, ADD, SUB e CMP. Neste contexto, *mesmos*, não significa que os bits tem o mesmo valor, mas sim, que ocupam a mesma posição na instrução.

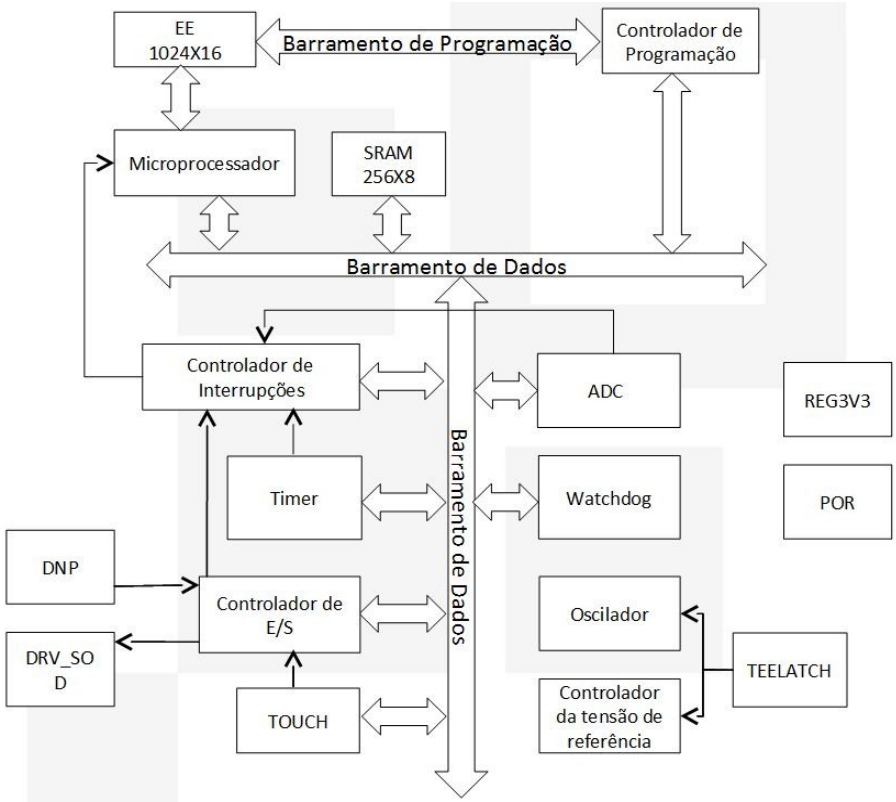


Figura 1.1 - Esquema geral do processador ZR16S08.
(Fonte: SMDH)

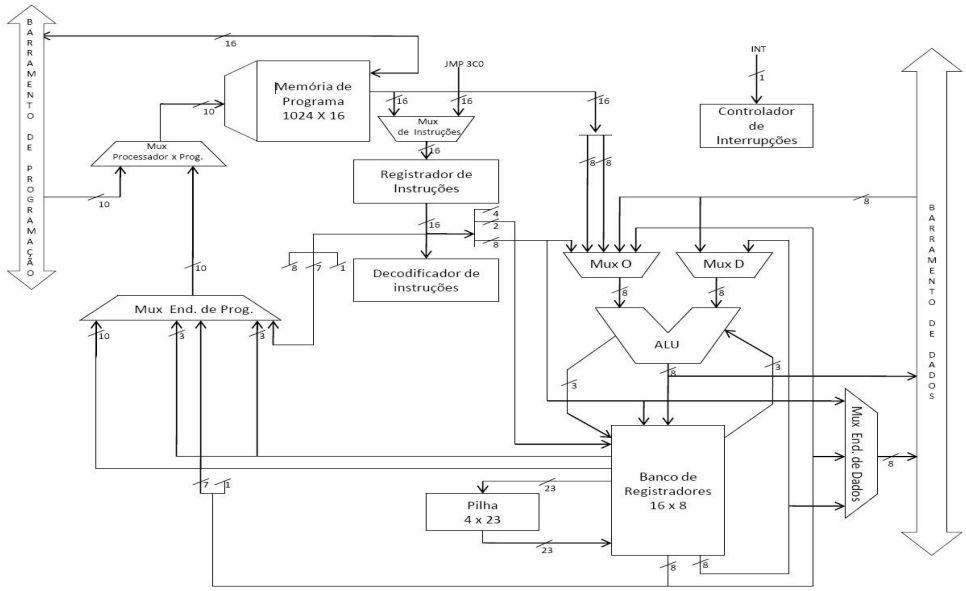


Figura 1.2 - Organização do processador ZR16S08.
(Fonte: SMDH)

Dos 16 registradores do processador, 13 são de uso geral (R0 a R12). Sabendo-se que o processador possui memória de programa de 1024 posições, precisa-se de um *program counter* (PC) de 10 bits, desta forma, o registrador R13 contém os 8 primeiros bits do PC sendo que os outros dois bits estão no R14, como ilustra o quadro 1.4.

| Instruction Set | |
|---|---|
| MOV | move |
| ADD | addition |
| SUB | subtraction |
| CMP | compare |
| MVS | move short |
| INC | incremental |
| DEC | decremental |
| AND | bitwise and |
| OR | bitwise or |
| XOR | bitwise xor |
| ROT | rotate |
| SHL | logical shift |
| SHA | arithmetic shift |
| RET | return |
| RETC | return and not hint |
| RETZ | return and hint=0 |
| RETS | return and hint=1 |
| JMP | jump |
| CALL | call |
| JZ | jump if Z |
| JNZ | jump if not Z |
| JC | jump if C |
| JVP | jump if V_P |
| DJNZ | dec reg and jump if not Z |

| Expression usage only (DV) | |
|---|---|
| = | attribution |
| + | addition |
| - | subtraction |
| / | division |
| * | multiply |
| & | logical and |
| \| | logical or |
| ^ | logical xor |
| ~ | logical not |
| sl | shift left (sl,sl2,..,sl15) |
| sr | shift right (sr,sr2,..,sr15) |
| ( | parenthesis |
| ) | parenthesis |
| **Directives** | |
| ORG | origin – set address |
| DC | define constant |
| DV | define variable |
| DA | define address |
| DR | define register |
| **Special features** | |
| Macro | macro - replace code |
| $ | current prog mem pos |
| warning_on | enable warning |
| warning_off | disable warning |

| Registers | |
|---|---|
| r0 | General use register |
| r1 | General use register |
| r2 | General use register |
| r3 | General use register |
| r4 | General use register |
| r5 | General use register |
| r6 | General use register |
| r7 | General use register |
| r8 | General use register |
| r9 | General use register |
| r10 | General use register |
| r11 | General use register |
| r12 | General use register |
| r13 | PC |
| r14 | PC & Control |
| r15 | Control & Flags |
| **Data Memory** | |
| 256 x 8 bits | |
| **Program Memory** | |
| 1024 x 16 bits | |

Quadro 1.1 - Conjunto de instruções do processador ZR16S08
(Fonte: SMDH)

| Opcode | | Destino/IO | Origem | | | | Assembler | |
|---|---|---|---|---|---|---|---|---|
| Bit | [15:12] | [11:10] | [9:8] | | [7:3] | [4:0] | | |
| MOV | 1101 | 00 | 0 | 0 | rd | ro | mov | rd,ro |
| | | | 0 | 1 | rd | (ro) | mov | rd,(ro) |
| | | | 1 | 0 | (end) | | mov | r0, (end) |
| | | | 1 | 1 | imediato | | mov | r0,imediato |
| | | 01 | 0 | 0 | (rd) | ro | mov | (rd),ro |
| | | | 0 | 1 | (rd) | (ro) | mov | (rd),(ro) |
| | | | 1 | 0 | (end) | | mov | (r0), (end) |
| | | | 1 | 1 | imediato | | mov | (r0),imediato |
| | | 10 | 0 | 0 | (end) | | mov | (end),r0 |
| | | | 0 | 1 | (end) | | mov | (end),(r0) |
| | | 11 | 0 | 0 | (end) | | mov io | (end), r0 |
| | | | 0 | 1 | (end) | | mov io | r0, (end) |
| | | | 1 | 0 | (rd) | ro | mov io | (rd), ro |
| | | | 1 | 1 | rd | (ro) | mov io | rd,(ro) |

Quadro 1.2 - Decodificação da instrução MOV.
(Fonte: SMDH)

O quadro 1.6 mostra os *opcodes* para cada instrução do processador.

| bit 11 | bit 10 | bit 9 | bit 8 | Endereçamento | bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | r_destino, r_origem | num reg_destino | | | | num_reg_origem | | | |
| 0 | 0 | 0 | 1 | r_destino, (r_origem) | num reg_destino | | | | num_reg_origem | | | |
| 0 | 0 | 1 | 0 | r0, (end) | endereço | | | | | | | |
| 0 | 0 | 1 | 1 | r0, imediato | imediato (*) | | | | | | | |
| 0 | 1 | 0 | 0 | (r_destino), r_origem | num reg_destino | | | | num_reg_origem | | | |
| 0 | 1 | 0 | 1 | (r_destino), (r_origem) | num reg_destino | | | | num_reg_origem | | | |
| 0 | 1 | 1 | 0 | (r0),(end) | endereço | | | | | | | |
| 0 | 1 | 1 | 1 | (r0),imediato | imediato (*) | | | | | | | |
| 1 | 0 | 0 | 0 | (end), r0 | endereço | | | | | | | |
| 1 | 0 | 0 | 1 | (end), (r0) | endereço | | | | | | | |

Quadro 1.3 - Modo de endereçamento do ZR16 (não válido para ROT, SHL e SHA).
(Fonte: SMDH)

| Special Registers | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| r15 | Z | C | V_P | uf1 | uf0 | uc | e/d | hint |
| r14 | mp | p2 | p1 | p0 | ppc9 | ppc8 | PC9 | PC8 |
| r13 | PC7 | PC6 | PC5 | PC4 | PC3 | PC2 | PC1 | PC0 |

Quadro 1.4 - Registradores de uso não geral.
(Fonte: SMDH)

O registrador R15 é condicional e guarda informações a respeito da última instrução executada, como valor nulo (Z = 1) ou *carry* (C = 1). O bit *hint* é utilizado para habilitar interrupções.

O quadro 1.5 ilustra a funcionalidade de  cada um dos bits dos registradores

R13, R14 e R15.

Segundo o *ZR16S08 User Guide* (2013, p.11) “mp=1 acessa a memória de programa quando usando as instruções MOV Rd, (Ro) ou MOV R0, (end)”. Este bit pode ser usado para acessar variáveis do tipo `const`, que são alocadas em memória de programa.

| Description | | |
|---|---|---|
| PC | PC9 - PC0 | PC (program counter) bits (10) |
| Control | mp | mp=1 indicates use of Program Memory. mp=0 indicates use of data memory. Inst: MOV rd,(ro) or MOV r0, (addr) |
| Control | p2 - p0 | if mp=1 indicates the partition to be used (4 partitions = 2 bits) |
| Control | ppc9 - ppc8 | indicates the next bits 9 and 8 for PC if instruction in use changes the content of PC |
| Flag | Z | system flag **Z**ERO |
| Flag | C | system flag **C**ARRY |
| Flag | V_P | system flag O**V**ERFLOW or **P**ARITY |
| Flag | uf1 - uf0 | user flags |
| Control | uc | uc=1 indicates Carry must be used for instructions ADD, SUB, CMP and ROT |
| Control | e/d | indicates the shift for bits on instructions ROT, SHL and SHA. (e/d=1) LEFT / (e/d=0) RIGHT |
| Control | hint | hint=1 interrupt enabled / hint=0 interruptions are disabled. |

Quadro 1.5 - Funcionalidade dos bits dos registradores R13, R14 e R15.
(Fonte: SMDH)

| Instrução | Bit 15 a 12 | Outros |
|---|---|---|
| JMP | 0000 | - |
| JZ, JNZ,JC,JVP | 0001 | - |
| CALL | 0010 | - |
| RET, RETC, RETS, RETZ | 0011 | Bit 7 =1 |
| MVS | 0011 | Bit 7 = 0 |
| AND | 0100 | - |
| OR | 0101 | - |
| XOR | 0110 | - |
| CMP | 0111 | - |
| ADD | 1000 | - |
| SUB | 1001 | - |
| ROT | 1010 | - |
| SHL | 1011 | - |
| SHA | 1100 | - |
| MOV | 1101 | - |
| DJNZ | 1110 | - |
| INC | 1111 | Bit 8 = 0 |
| DEC | 1111 | Bit 8 = 1 |

Quadro 1.6 - *Opcodes* das instruções do ZR16S08.
(Fonte: SMDH)

A memória de programa do ZR16 é dividida em 8 partições de 128 x 16 bits. Desta forma, os 3 bits p0, p1 e p2 (quadro 1.4) indicam qual partição deve ser acessada.

Os bits ppc9-ppc8 são utilizados para indicar o próximos bits 8 e 9 do PC (pc8 e pc9) caso a instrução sendo executada seja de desvio.

### 1.2.2. Funcionalidades

Além das memórias de programa e dados, o ZR16S08 possui um bloco de memória de 256 x 8 bits para entrada e saída de sinais digitais e analógios que fornecem recursos como:

a) Timer para disponibilizar funções de temporização;
b) Controlador de interrupções (ver Apêndice B);
c) Watchdog;
d) Conversor A/D;
e) Touch controller (sensor de toque).

## 1.3 Compiladores, códigos, memória e MCU's.

### 1.3.1 Introdução

Até então foram mostradas características básicas do processador ZR16S08. Nesta seção, será ilustrado o problema que o corrente trabalho tem o intuito de resolver: a geração de códigos mais eficientes para o MCU ZR16S08.

O ZR16S08 possui um compilador C, que como será ilustrado, não gera códigos muito eficientes. O processo de compilação não otimiza o código e acaba gerando muitas redundâncias e instruções não necessárias.

No decorrer do capítulo, serão ilustrados códigos escritos em C e os seus respectivos códigos compilados pelo processador oficial do ZR16S08. Observando-se os códigos compilados e simulando-os, pode-se discutir questões como o tamanho do programa gerado e o número de ciclos necessários para o microcontrolador executar o programa.

O processo de gerar códigos eficientes para microcontroladores é importante, pois geralmente possuem memória de programa e de dados pequena, bem como baixa capacidade de processamento. Por esta razão, é de suma importância que os códigos para sistemas embarcados sejam otimizados: quanto menor o tamanho do programa e menor o número de ciclos necessários para a sua execução, melhor será o desempenho do microcontrolador.

Aho (2006, p.327), afirma que “...algumas medidas de custo comuns são o tempo de compilação, o tamanho, o tempo de execução e o consumo de energia do código objeto”. Compiladores otimizadores para sistemas embarcados podem ser de grande utilidade neste aspecto. Por esta, dentre outras razões, o compilador SDCC foi escolhido como alternativa para a resolução do problema do compilador oficial do ZR16S08. O SDCC será discutido no capítulo 3.

### 1.3.2 O compilador ZR16 Compiler

É importante saber os tipos e tamanhos de dados suportados pelo ZR16 Compiler a fim de entender o modo que as instruções são emitidas e a maneira que o compilador escreve os dados na memória. O quadro 1.7 ilustra tais tamanhos, bem como os intervalos de valores.

| Type | Bits | Range | Bytes |
|---|---|---|---|
| unsigned char | 8 | 0 : 255 | 1 |
| signed char | 8 | -128 : 127 | 1 |
| unsigned byte | 8 | 0 : 255 | 1 |
| signed byte | 8 | -128 : 127 | 1 |
| unsigned int | 16 | 0 : 65535 | 2 |
| signed int | 16 | -32768 : 32767 | 2 |
| unsigned short | 16 | 0 : 65535 | 2 |
| signed short | 16 | -32768 : 32767 | 2 |

Quadro 1.7 - Tipos de dados suportados pelo compilador ZR16S08
(Fonte: SMDH)

Como exemplo inicial, tem-se o código da figura 1.3 escrito em C. O respectivo código compilado pelo ZR16 Compiler está exposto na figura 1.4.

No código compilado, ';' denota comentário. Deve-se lembrar que somente o registrador R0 pode ler e escrever na memória.

```
1  char x;
2
3  void main ()
4  {
5       x = 0;
6       x++;
7  }
```

Figura 1.3 - Exemplo de código a ser analisado.

É fácil perceber o grande número de instruções não necessárias emitidas pelo compilador. Um código abstrato, otimizado, que não é gerado pelo compilador, é mostrado na figura 1.5 para fins de comparação.

```
11 ; x foi alocado no endereço 2
12
13 MOV R0 , 0
14 MOV (2) , R0        ; x  = 0
15 MOV R0 , (2)        ; R0 = 0
16 MOV R2 , R0         ; R2 = 0
17 MOV R3 , R2         ; R3 = R2 = 0
18 MOV R0 , 1          ; R0 = 1
19 ADD R3 , R0         ; R3 = 1
20 MOV R0 , R3         ; R0 = 1
21 MOV (2) , R0        ; x  = 1
```

Figura 1.4 - Código da figura 1.3 gerado pelo ZR16 Compiler.

Analisando o código emitido pelo compilador, nota-se que as linhas 13 e 14 fazem a atribuição `x=0` da maneira mais otimizada possível. Porém, o comando de incremento não é otimizado, nota-se que o compilador faz a alocação de três registradores, porém pode-se incrementar um valor de memória utilizando somente um registrador. A linha 15 é desnecessária, visto que não está mudando o valor de R0. A figura 1.5 ilustra de maneira mais clara como o programa poderia ter sido compilado sem os problemas acima citados. Um código ainda mais otimizado (o mais otimizado possível) é mostrado na figura 1.6. Neste caso, o código foi gerado pelo compilador SDCC.

```
25 MOV R0 , 0
26 MOV (2) , R0
27 ADD R0 , 1
28 MOV (2) , R0
```

Figura 1.5 - Otimização do código da figura 1.4.

```
31 MOV R0 , 1
32 MOV (2) , R0
```

Figura 1.6 - Otimização do código da figura 1.5 (gerado pelo SDCC).

O compilador SDCC possui natureza *interpretativa.* No caso do exemplo atual, o SDCC interpretou, em tempo de compilação, que a única instrução útil do código é a atribuição x=1.

Sempre que possível, o SDCC calculará os resultados de expressões como a do exemplo acima em tempo de compilação. Isto acarreta em códigos mais eficientes. Além da parte interpretativa, o SDCC aplica várias técnicas de otimização, que serão discutidas no capítulo 3.

Códigos mais eficientes são códigos menores e/ou que executam em um menor número de ciclos, por esta razão, o número de ciclos de execução pode ser usado como referência na medida de desempenho dos compiladores. Para executar o código da figura 1.3, o compilador original do ZR16S08 requer 10 ciclos, já o código compilado pelo SDCC faz as mesmas operações em apenas 2 ciclos.

Se a medida de desempenho a ser tomada for o tamanho do programa, o SDCC geralmente será melhor. No exemplo acima, o código do ZR16 Compiler gerou um código de 18 Bytes (9 instruções), valor muito alto se comparado com os 2 Bytes gerados pelo SDCC (2 instruções).

A fim de ilustrar melhor o desempenho ruim do ZR16 Compiler, será mostrado mais um exemplo de como o código ilustrado na figura 1.7 é compilado. Neste caso, as variáveis são do tipo volátil, significando que o SDCC não fará qualquer operação em tempo de compilação. Os resultados só podem ser calculados em tempo de execução. Pois estão associados a portas de entrada do microcontrolador, que podem ter seus valores alterados a qualquer momento.

```
1 volatile int x , y , z;
2
3 void main ()
4 {
5       x = y&z;
6 }
```

Figura 1.7 - Operação AND bit a bit.

As figuras 1.8 e 1.9 mostram os códigos gerados pelo ZR16 Compiler e pelo SDCC, respectivamente. Deve-se ter em mente que, segundo o quadro 1.7, variáveis inteiras possuem 16 bits. Para fazer operações que requerem leitura e/ou escrita de tais variáveis deve-se ler e/ou escrever a variável em duas etapas: uma para a parte baixa, outra para a parte alta, pois os registradores do ZR16S08 são de 8 bits.

Novamente, nota-se a grande diferença no tamanho dos códigos gerados pelos dois compiladores. O ZR16 Compiler gerou um código de 24 Bytes (12 instruções) que são executadas em 16 ciclos. Já o SDCC gerou um código com 12 Bytes (6 instruções) que são executadas em 10 ciclos.

```
12 ; x_L: parte baixa de x: MEM(2)
13 ; x_H: parte alta de x:  MEM(3)
14 ; y_L: parte baixa de y: MEM(4)
15 ; y_H: parte alta de y:  MEM(5)
16 ; z_L: parte baixa de z: MEM(6)
17 ; z_H: parte alta de z:  MEM(7)
18
19 MOV R0 , (4)   ; R0 <- y_L
20 MOV R2 , R0    ; R2 <- R0
21 MOV R0 , (5)   ; R0 <- y_H
22 MOV R3 , R0    ; R3 <- R0
23 MOV R0 , (6)   ; R0 <- z_L
24 AND R2 , R0    ; R2 <- y_L & z_L
25 MOV R0 , (7)   ; R0 <- z_H
26 AND R3 , R0    ; R3 <- y_H & z_H
27 MOV R0 , R2    ; R0 <- y_L & z_L
28 MOV (2) , R0   ; x_L <- R0
29 MOV R0 , R3    ; R0 <- y_H & z_H
30 MOV (3) , R0   ; x_H <- R0
```

Figura 1.8 - Código AND compilado pelo ZR16 Compiler.

```
33 MOV  R0 , (4)  ; R0 <- y_L
34 AND R0 , (6)   ; R0 <- y_L & z_L
35 MOV  (2) , R0  ; x_L <- R0
36 MOV  R0 , (5)  ; R0 <- y_H
37 AND R0 , (7)   ; R0 <- y_H & z_H
38 MOV  (3) , R0  ; x_H <- R0
```

Figura 1.9 - Código AND compilado pelo SDCC.

Embora o SDCC tenha diminuído o número de instruções em 50%, o número

de ciclos necessários para executar o programa diminuiu 37.5%. Isto ocorre porque as instruções com leitura de memória (e.g., `MOV R0,(4)` e `AND R0,(6)`), requerem 2 ciclos, e não 1.

Para finalizar esta seção, foi escolhido convenientemente o exemplo da figura 1.10 por questões estatísticas de otimização.

```
1 volatile x , y;
2
3 void main ()
4 {
5       x = y == 0;
6 }
```

Figura 1.10 - Comparação com zero.

Comparações com zero são estatisticamente as mais comuns, portanto, pode-se notar significativo aumento de desempenho se esta instrução for bem elaborada. A figura 1.11 ilustra o código compilado pelo ZR16 Compiler. A figura 1.12 mostra a comparação com zero gerada pelo SDCC, que é a mais eficiente possível para valores de 2 Bytes.

```
10 MVS R2 , 0
11 MOV R0 , (4)
12 MOV R4 , R0
13 MOV R0 (5)
14 MOV R5 , R0
15 MOV R0 , R5
16 AND R0 , 0x80
17 JNZ EXIT_LBL
18 MOV R0 , 0
19 CMP R5 , R0
20 JNZ EXIT_LBL
21 MOV R0 , 0
22 CMP R4 , R0
23 JNZ EXIT
24 TRUE_LBL: MVS R2 , 1
25 EXIT_LBL: MOV R0 , R2
26 MOV (2) , R0
27 MOV R0 , R3
28 MOV (3) , R0
```

Figura 1.11 - Comparação com zero gerada pelo ZR16 Compiler.

Este resultado pode ser ainda melhor notado se esta instrução for usada em *loops*, que são predominantes no tempo de execução da maioria dos aplicativos.

É fácil notar que, para valores inteiros (2 Bytes), pode-se detectar o valor zero fazendo a operação `OR` com suas partes alta e baixa. Esta técnica otimizada requer sempre 11 ciclos para ser executada, já o código gerado pelo ZR16 Compiler requer,

```
31 MVS  R9 , 0x00
32 MOV  R0 , (y_L)
33 OR   R0 , (y_H)
34 JNZ  THERE
35 MVS  R9 , 0x01
36 THERE: MOV R0 , R9
37 MOV  (x_L) , R0
38 MOV  R0 , 0x00
39 MOV  (x_H) , R0
```

Figura 1.12 - Comparação com zero do SDCC.

em média, 18 ciclos. Este valor é médio porque depende dos desvios, que são determinados pelo valor de `y`.

A comparação com zero do ZR16 Compiler possui 19 instruções (38 Bytes), já o SDCC compilou o código com apenas 10 instruções (20 Bytes).

### 1.3.3 Conclusão

O presente capítulo tenta apresentar as características gerais do processador ZR16S08, tanto em hardware quanto em software, bem como a baixa eficiência seu compilador.

O propósito deste trabalho é obter o porte do compilador SDCC, que será estudado no capítulo 3, a fim de gerar códigos mais eficientes para o ZR16S08 e resolver os problemas mostrados nos exemplos da seção 1.3.2.

Para tal, é necessário que sejam apresentadas noções básicas do funcionamento dos compiladores antes do SDCC ser estudado. O capítulo 2 faz uma introdução aos conceitos essenciais que os concernem.

Nenhuma análise aprofundada sobre compiladores pode ser feita em um

único capítulo, no entanto, os conceitos necessários para obter o porte do compilador ZR16S08 serão apresentados. Conceitos como código intermediário, otimização e geração de código são essenciais ao entendimento do SDCC, portanto, o capítulo 2 foi dedicado a eles.

# 2 NOÇÕES BÁSICAS SOBRE COMPILADORES

## 2.1 Introdução

A função básica dos compiladores é receber como entrada um código fonte escrito em uma linguagem de alto nível, como C, e traduzir tal código para uma linguagem de baixo nível executável pelo processador para o qual o compilador foi especificamente criado.

O processo de compilação, em poucas palavras, consiste em gerar código executável pelo processador a partir de um código fonte de nível mais alto que a linguagem de máquina.

O desenvolvimento de compiladores tem sua base na grande dificuldade encontrada pelos programadores em desenvolver programas diretamente em linguagens de máquina.

## 2.2 Estrutura de um compilador

No processo de compilação existem duas partes distintas: análise (*front-end*) e a síntese (*back-end*).

Na parte de análise, o compilador recebe o código fonte de entrada e cria um código intermediário do mesmo, bem como uma *tabela de símbolos,* que contém informações sobre o programa. É na fase de análise que o compilador irá detectar possíveis erros de sintaxe ou semântica no código fonte. Caso um destes erros ocorra, o compilador deverá informar o erro.

A parte de síntese, como o nome já informa, constrói o código objeto para o processador a partir da representação intermediária e das informações armazenadas na tabela de símbolos.

O processo de compilação é dividido em várias fases sequenciais. A saída de cada fase é uma entrada para a fase seguinte e a tabela de símbolos é usada durante todo o processo de compilação.

Uma parte de otimização de código é opcional no compilador. Esta etapa pode ser implementada de várias formas diferentes, dentre elas, a técnica de

*peephole* pode ser utilizada. Esta técnica será discutida nas próximas seções. A figura 2.1 mostra o esquema genérico do processo de compilação.

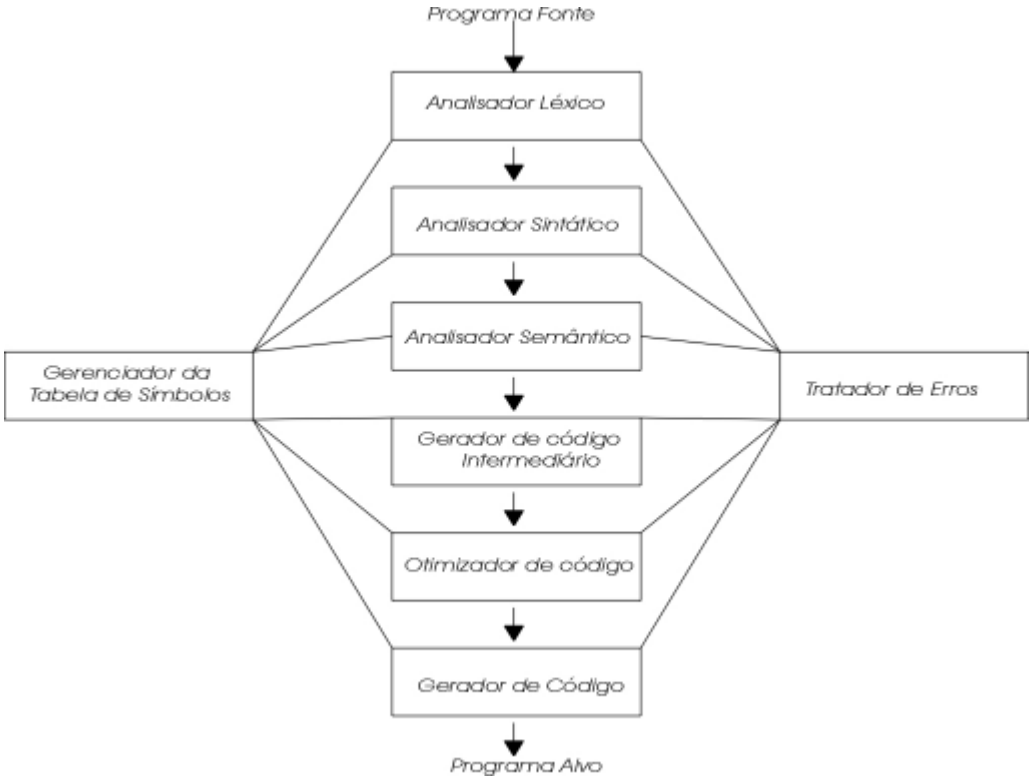


Figura 2.1 - Fases do processo de compilação.

### 2.2.1 Análise Léxica

A análise léxica é a primeira fase do compilador. Basicamente, é um algoritmo (analisador léxico) que lê o fluxo de caracteres do programa fonte e os armazena em unidades chamadas lexemas. Cada lexema é um par <*chave , valor*> chamado comumente de *token.* Os lexemas são utilizados como entrada na próxima fase da compilação, a análise sintática.

O atributo *chave*, do *token*, é um símbolo utilizado na análise sintática. O atributo *valor* é utilizado como entrada para a tabela de símbolos que retorna um valor como saída.

ara facilitar o entendimento do analisador léxico, suponha que um código fonte contenha a expressão

```
valor = valorInicial + taxa*10
```
(2.1)

Os caracteres deste comando são lidos pelo analisador léxico, que ao mesmo tempo cria uma tabela de símbolos e os lexemas, na sequência:

a) `valor` gera o token <*id , 1*>, sendo que *id* é um símbolo identificador e 1 aponta para a entrada da tabela de símbolos onde se encontra `valor`. A tabela manterá informações a respeito de `valor`, como nome e tipo;
b) O símbolo = é mapeado para o *token* padrão <=>. Como o símbolo = não possui valor, nome ou tipo, não precisa ser armazenado na tabela de símbolos, logo não precisa do campo de entrada para a tabela;
c) `valorInicial` gera o *token <id , 2*>, onde 2 é uma entrada para a tabela de símbolos que contem informações sobre `valorInicial;`
d) + gera o token padrão <+>;
e) `taxa` gera o *token <id , 3*>, onde 3 é a entrada para a tabela de símbolos para `taxa;`
f) `* gera o token padrão <*>;`
g) `10` gera o *token* padrão <*10*>.

Após ter sido feita a análise léxica da expressão 1.1, a seguinte sequência de *tokens é gerada:*

*<id , 1*> <=> <*id , 2* > <+> <*id , 3*> <*> <*10*>

A sequência de *tokens* é então passada para a próxima fase da compilação, a análise sintática.

### 2.2.2 Análise Sintática

Esta é a segunda fase da compilação. Nesta etapa, o analisador sintático utiliza os lexemas criados pelo analisador léxico.

O analisador sintático representa as expressões através de *árvores sintáticas*, em que cada nó interior da árvore representa um operador e os filhos do nó representam os operandos.

A árvore sintática deve ser construída de forma que a prioridade das operações matemáticas seja mantida. Por exemplo, como mostra a figura 2.2, o valor 10 é multiplicado a *taxa* antes da soma ser feita.

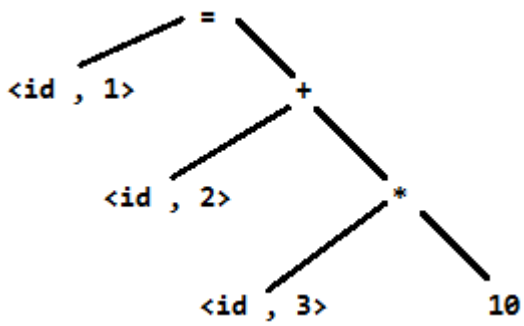


Figura 2.2 - Árvore de  sintaxe para a expressão 2.1.

### 2.2.3 Análise Semântica

Nesta fase, o analisador semântico utiliza a árvore de sintaxe e a tabela de símbolos para verificar a consistência semântica das expressões. Por exemplo, a verificação de tipos de dados em expressões é feita nesta etapa da compilação, qualquer inconsistência nas expressões, como a atribuição de uma variável de ponto flutuante a uma variável do tipo inteira será detectada nesta fase.

O analisador coleta informações sobre os tipos e os salva na tabela de símbolos ou na própria árvore para usos subsequentes na geração de código intermediário.

### 2.2.4 Geração de Código Intermediário

O processo de gerar códigos de máquina exige que o programa fonte de alto nível seja convertido em um código intermediário antes da geração do programa executável.

As árvores sintáticas já são em si, uma forma de código intermediário, mas normalmente só são utilizadas nas duas primeiras fases da compilação.

Tradicionalmente, após as duas primeiras fases da compilação, o compilador gera um código intermediário “textual” que pode ser a linguagem de máquina do processador, ou um código de uma máquina abstrato, que deve ser convertido em código de máquina em uma próxima etapa. Um exemplo de código abstrato são os *iCodes*, do compilador SDCC, que será discutido no capítulo 3.

Mais especificamente, os códigos intermediários são *instruções de três endereços,* que constituem instruções de máquina, abstratas ou não, com três operandos cada uma. Por código de três endereços define-se toda expressão do tipo A = B **op** C*.* Onde A*,* B e C representam três operandos e **op** representa uma operação. Cada operando das instruções pode ser associado a um registrador ou a endereços de memória do computador e a cada operação é associada uma variável temporária, que torna possível a execução do cálculo.

Como exemplo, a expressão 2.1 pode gerar o código intermediário

```
temp1 <= id3*10
temp2 <= id2 + temp1
id1 <= temp2
```

Onde `temp1` e `temp2` são variáveis temporárias criadas pelo compilador e `id1`, `id2` e `id3` representam `valor`, `valorInicial` e `taxa`, respectivamente.

Tendo-se a tabela de símbolos e o código intermediário, como mostra o exemplo acima, pode-se gerar as instruções *assembly* específicas para um processador alvo.

Nota-se que as instruções do código intermediário não são necessariamente de três endereços, como na atribuição `id1 <= temp2`. Mas isto não acarretará em nenhum problema no processo de geração de código, visto que é possível gerar instruções *assembly* com um único operando.

### 2.2.5 Otimização de código

Nesta etapa, o código intermediário é tomado como entrada num otimizador. Visto que o otimizador recebe o código intermediário, e não as instruções *assembly*, este tipo de otimização é independente de máquina (independente da arquitetura, organização e conjunto de instruções).

Por otimização, define-se a geração de um código menor (que utiliza menos memória) ou mais rápido que o código não otimizado (menor tempo de execução).

Um exemplo muito simplista de otimização nesta fase pode ser visto da seguinte maneira: Suponha que um compilador gere o seguinte código intermediário

```
temp1 <= id1
temp1 <= id1 + 3
```

Com a otimização, este código pode ser convertido para

```
temp1 <= id1 + 3
```

visto que  a primeira instrução é desnecessária.

A etapa de otimização de código durante esta fase é opcional e varia muito de compilador a compilador. Porém, como será visto no decorrer do trabalho, a otimização de códigos representa uma grande melhoria de desempenho no programa objeto.

## 2.3 Geração de Código

Esta é a fase final da compilação. Esta etapa recebe o código intermediário (otimizado ou não) ou instruções *assembly*  como entrada e gera código objeto específico para o processador. As instruções são, em seguida,  interpretadas por um *montador*, que cria o arquivo objeto final. O processo de *montagem* não será abordado neste trabalho. Porém, deve-se ter em mente que as instruções de máquina geradas pelo compilador estarão diretamente relacionadas ao código

montado. Quanto mais otimizado é o conjunto de instruções *assembly* gerado pelo compilador, melhor será o código objeto.

Outro aspecto importante desta fase é a parte de associação de variáveis do programa com registradores (alocação de registradores) ou endereços de memórias. Por exemplo, o código intermediário

```
temp1 <= 3
id1 <= temp1 + 2
```

poderá ser convertido no código de máquina

```
MOV R0 , 3
ADD R0 , 2
MOV (0x00) , R0
```

onde `R0` é associado a variável temporária *temp1* , somado a 2 e, em seguida, guardado no endereço de memória 0, que representa a variável id1.

## 2.4 Conclusão

Até então foram apresentados conceitos básicos sobre a estrutura e função dos compiladores. A apresentação destes conceitos, embora simplificada, é necessária para o entendimento do capítulo subsequente.

No capítulo 3 será apresentado o compilador de código aberto SDCC. Embora este compilador seja um caso específico, todas as características do funcionamento dos compiladores apresentadas no presente capítulo, como as fases de compilação (análise e síntese), geração de código intermediário e otimização são partes do funcionamento do SDCC.

# 3 SDCC – SMALL DEVICE C COMPILER

## 3.1 Introdução

SDCC é um compilador C de código aberto e redirecionável para processadores de 8 bits. O propósito geral deste compilador é gerar códigos executáveis para diferentes processadores a partir do mesmo código intermediário.

A versão atual do SDCC suporta os processadores Intel MCS51 (8031, 8032, 8051, 8052, etc.), Dallas DS80C390, Freescale HC08 (HC08 e S08), Zilog Z80 e STMicroelectronics STM8.

Basicamente, pode-se redirecionar este compilador para um processador específico a partir do código intermediário (*iCode,* ou *intermediate code*) criado pelo *front-end* do compilador.

No processo de geração de código executável, deve-se tomar como entrada uma sequência de *iCodes* que definem a sequência de instruções a serem emitidas para um processador específico e gerar uma saída com instruções para a máquina alvo.

O propósito deste trabalho é estender o número de alvos portados pelo SDCC, definindo-se um novo porte para o processador ZR16S08.

A escolha deste compilador foi feita com base na sua natureza de porte livre e ótimas técnicas de otimização de código. Como será mostrado no capítulo 4, este compilador pode gerar códigos de alta performance.

A figura 3.1 ilustra a ideia básica do funcionamento do SDCC. Após tomar-se um código fonte escrito em C como entrada e gerar-se o código intermediário, uma função específica é chamada, por meio da seleção do processador alvo, a fim de emitir as instruções necessárias e gerar o seu respectivo código *assembly*.

Após a geração do código, o mesmo pode ser otimizado pela técnica de *peephole*, que será discutida na seção 3.2.2.

A seção 3.2.1 ilustra as otimizações gerais do SDCC. Estas otimizações são gerais porque podem ser aplicadas em todos os processadores suportados pelo compilador.

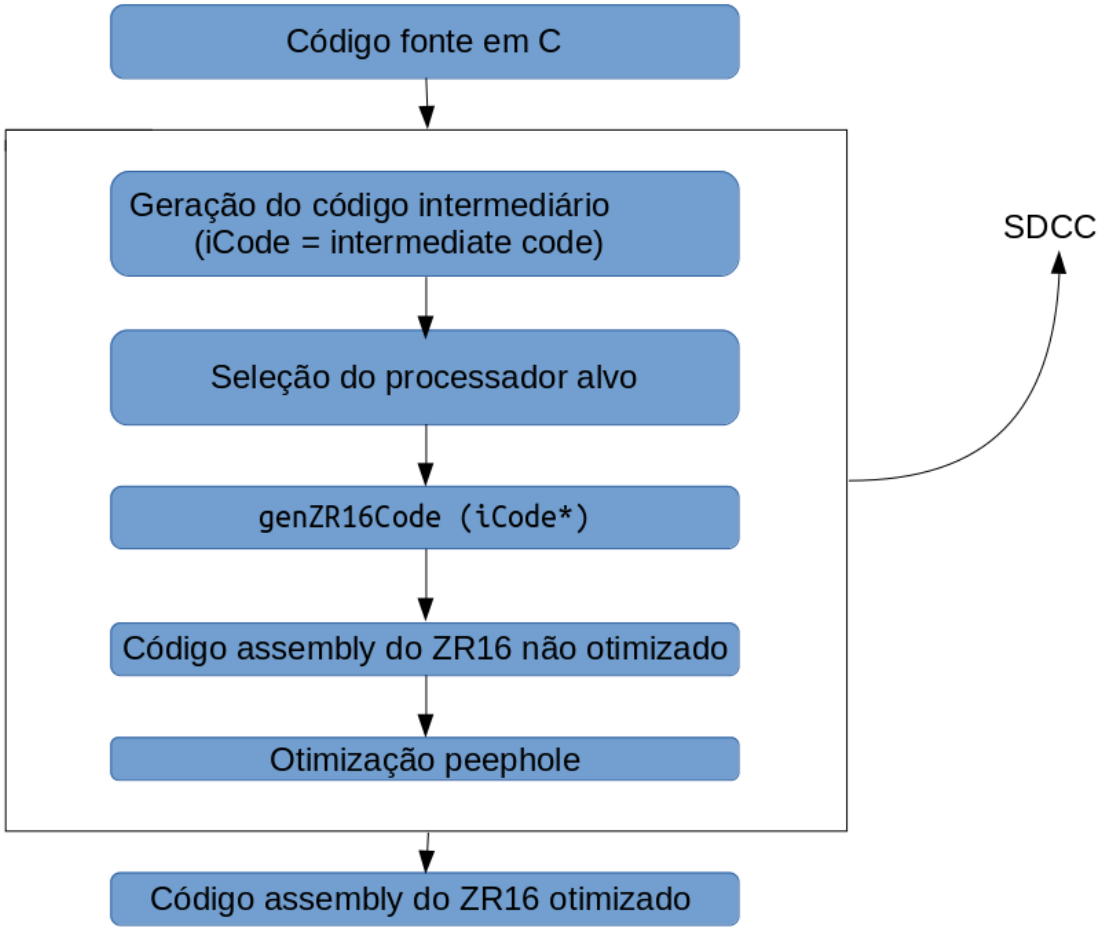


Figura 3.1: Esquema geral do funcionamento do SDCC.

## 3.2 Otimização de código do SDCC

O SDCC é um compilador otimizador, pois possui várias técnicas de otimização de código, algumas otimizações são gerais e válidas para todos os portes. Estas otimizações serão apresentadas na seção 3.2.1.

A técnica de otimização mais eficiente deste compilador é o *peephole,* que não é geral e deve ser definida para cada alvo durante a criação do porte do compilador, pois esta técnica é dependente do conjunto instruções de cada processador.

Algumas técnicas de otimização gerais do SDCC são: eliminação de códigos não utilizados (*Dead-Code Elimination*), propagação de cópia (*Copy-Propagation*), otimizações de *loop (Loop Invariant e Loop Reversing)* e simplificações algébricas.

A seguir, será apresentado um exemplo do funcionamento de cada uma das técnicas citadas acima.

### 3.2.1 Otimizações gerais

a) *Dead-Code Elimination:*

Esta técnica evita que códigos não utilizados sejam compilados. O código da figura 3.2 mostra um exemplo de programa com código morto, a figura 3.3 mostra como o SDCC otimiza o código suprimindo todos os comandos não necessários no código.

```
1  int global;
2
3  void f ()
4  {
5       int i;
6       i = 1;              /* Dead code */
7       global = 2;         /* Dead code */
8       global = 3;
9       return;
10      global = 4;         /* Código inatingível */
11 }
```

Figura 3.2 - Programa com código morto.

Linhas de comando, como as 6, 7 e 10 da figura 3.2, são frequentemente escritas por descuido do programador. Sempre que possível, o SDCC evitará que tais códigos sejam compilados interpretando o código da figura 3.2 da maneira mostrada na figura 3.3.

```
1 int global;
2
3 void f ()
4 {
5       global = 3;
6 }
```

Figura 3.3 - Programa com código morto eliminado pelo SDCC.

b) *Copy-Propagation:*

O código mostrado na figura 3.4 mostra um exemplo de *copy- propagation* no que diz respeito a atribuições.

É necessário saber que fazer referência a variáveis significa ler um endereço de memória, acarretando em um maior número de ciclos. Para o processador ZR16S08, por exemplo, atribuir um valor direto a um endereço de memória requer, no mínimo, um ciclo, já atribuir um endereço de memória a outro requer pelo menos três ciclos.

```
1 int f ()
2 {
3       int i , j;
4       i = 10;
5       j = i;
6       return j;
7 }
```

Figura 3.4 – Copy-propagation.

As figura 3.5 mostra como o SDCC converte o código (altera a linha 5) da figura 3.4 para fins de otimização. O SDCC aplica esta modificação visto que a atribuição `j=10` é mais rápida que `j=i`, pois desta forma não há necessidade de acessar a memória e ler a variável `i`.

```
10 int f ()
11 {
12       int i , j;
13       i = 10;
14       j = 10;
15       return j;
16 }
```

Figura 3.5 - Código otimizado.

c) *Loop Invariant:*

Pode-se ganhar um significativo aumento de performance se esta técnica de

otimização for aplicada no código mostrado na figura 3.6.

```
1 for (i = 0 ; i < 100 ; i++)
2 {
3       f += k + l;
4 }
```

Figura 3.6 - Loop Invariant.

```
7 temp = k + l;
8
9 for (i = 0 ; i < 100 ; i++)
10 {
11      f += temp;
12 }
```

Figura 3.7 - Código transformado pelo SDCC.

Na figura 3.6, é evidente que a soma `k+l` é constante, desta forma, o SDCC faz a soma k+l fora do *loop* uma única vez, eliminando a necessidade de fazer tal operação em cada iteração do *loop*.

O código da figura 3.6 é modificado de acordo com a figura 3.7. Esta técnica representa um ganho muito significativo de eficiência, visto que, estatisticamente, as operações com *loop* são predominantes no tempo de execução da maioria dos programas.

d) *Loop Reversing:*

Esta técnica é aplicada somente em *loops reversíveis* a fim de simplificar as verificações de comparação para término do *loop*.

*Loops* reversíveis são *loops* que podem ter seus campos trocados de maneira que o resultado dos cálculos de seu bloco não sejam alterados. Por exemplo, os *loops* das figuras 3.8 e 3.9 tem a mesma funcionalidade, o valor final de `count` é o mesmo para ambos os casos.

```
1 for (i = 0 ; i < 100 ; i++)
2 {
3       count++;
4 }
```

Figura 3.8 - *Loop* reversível.

```
7 for (i = 100 ; i > 0 ; i--)
8 {
9       count++;
10 }
```

Figura 3.9 - *Loop* da figura 3.8 invertido.

O *loop* mostrado na figura 3.10 não é reversível, pois a sua inversão acarretaria em valores incorretos para o vetor `element`.

Caso o *loop* fosse invertido, a variável `i` seria iterada de 100 a 1, e não de 0 a 99, que é a maneira desejada ilustrada na figura 3.10. Isto acarretaria em um *overflow* no vetor `element` e impediria que o valor `element[0]` fosse inicializado.

```
1 for (i = 0 ; i < 100 ; i++)
2 {
3       element[i] = i;
4 }
```

Figura 3.10 - *Loop* não reversível.

Pode-se obter otimização significativa se for feita a inversão de *loops* como o mostrado na figura 3.8 visto que para cada iteração do *loop* a verificação da expressão `i > 0` é mais eficiente do que a verificação de `i < 100`, pois comparações com zero são sempre mais rápidas, pois pode-se usar instruções como *DJNZ* (*decrement and jump if not zero*) ou *JZ* (*jump if zero*) .

No capítulo 1 foi mostrada a grande eficiência que o SDCC possui em detectar o valor zero.

```
6 for (i = 100 ; i > 0 ; i--)
7 {
8       element[i] = i;
9 }
```

Figura 3.11 - *Loop* indevidamente invertido.

O SDCC tem um conjunto específico de critérios para determinar se o *loop* é reversível ou não, são eles:

1) O *loop* `for` é da forma

```
for(<symbol>=<exp>;<sym>[<|<=]<exp>;[<sym>++|<sym>+= 1])
<for body> ;
```

2) `<for body>` não pode conter `continue` ou `break;`

3) Não pode existir chamadas de procedimentos no *loop;*

4) O valor `<sym>` não pode ser referenciado no corpo do *loop;*

5) Não pode haver `switch` no loop.

e) Simplificações algébricas:

O SDCC aplica várias simplificações algébricas, alguns exemplos são:

| | | |
|---|---|---|
| 1) `i = j + 0` | é mudado para | `i = j;` |
| 2) `i = j*2` | é mudado para | `i = j << 1;` |
| 3)`i = j/2` | é mudado para | `i = j >> 1;` |
| 4) `i = j - j` | é mudado para | `i = 0;` |
| 5) `i = 10 + 20` | é mudado para | `i = 30.` |

### 3.2.2 Otimizações específicas – *peephole*

Visto que se torna muito difícil gerar códigos otimizados diretamente no momento em que as instruções são emitidas pelo compilador, muitas melhorias podem ser feitas no código após o mesmo ter sido gerado.

Como afirma o *SDCC Compiler User Guide* (2014), a técnica *peephole (*otimização de janela*)* é uma técnica de transformação baseada em um conjunto de regras definidas pelo programador que são aplicadas pelo otimizador através de um reconhecimento de padrões. Tais regras são regras de substituição de instruções,

ou sequências de instruções, desta forma, este método é dependente da máquina alvo.

Esta técnica é feita através de uma *janela deslizante* que “varre” o conjunto de instruções não otimizado gerado pelo compilador sequencialmente. Sempre que possível, o conjunto de instruções da janela é substituído por um conjunto de instruções menor ou mais rápido.

As regras de substituição devem ser definidas pelo programador e, como já citado, são específicas para cada máquina.

Visto que o intuito deste trabalho é o porte do compilador SDCC para o processador ZR16S08, o conjunto de instruções do mesmo será utilizado para ilustrar alguns exemplos de otimização pela técnica *peephole.*

No SDCC, as regras de substituição por janela são definidas da seguinte forma:

```
replace { <alguma sequência de instruções> }
by      { <outra sequência de instruções>  }
```

A fim de ilustrar o funcionamento desta técnica, suponha que o SDCC gere o código ilustrado na figura 3.12 para o processador ZR16S08:

```
69 MOV R0 , R1
70 MOV R1 , R0
```

Figura 3.12 - Exemplo de código redundante.

Após a geração do código, um otimizador pode aplicar a técnica de otimização de janela e eliminar a instrução da linha 70, mostrada na figura 3.12.

Como exemplo, a regra para eliminar a instrução redundante do exemplo acima é ilustrada na figura 3.13.

A eliminação da instrução, na verdade, é uma substituição de todo um conjunto de instruções por outro mais eficiente.

Na figura 3.13, %n (n = 1, 2, …) representa qualquer variável, registrador, valor numérico ou *label*.

```
59 replace
60 {
61       MOV %1 , %2
62       MOV %2 , %1
63 }
64 by
65 {
66       MOV %1 , %2
67 }
```

Figura 3.13 - Regra *peephole* para eliminar instrução redundante.

Outro exemplo de código que pode ser otimizado é mostrado na figura 3.14.

```
33 JNZ label_A
34 JMP label_B
35 label_A: INC R0
36 label_B: DEC R0
```

Figura 3.14 - Código pode ser melhorado com *peephole*.

Um código otimizado para este exemplo é mostrado na figura 3.15, que pode ser obtido com a regra *peephole* da figura 3.16.

```
39 JZ label_B
40 label_A: INC R0
41 label_B: DEC R0
```

Figura 3.15 - Código otimizado.

Na regra mostrada na figura 3.16, %1 e %2 representam os *labels* `label_A` e `label_B`, respectivamente. Esta regra aparenta não ter muito efeito no aumento de desempenho, pois elimina uma única instrução. Porém, as instruções de desvio mostradas acima são geralmente criadas em *loops,* desta forma pode-se observar um significativo aumento de performance em algoritmos com laços, visto que para cada iteração do *loop* tem-se uma instrução a menos a ser executada.

O SDCC aplica as regras de otimização uma por uma em sequência a partir do inicio da sequência de regras. Este processo de otimização terminará quando

```
43 replace
44 {
45       JNZ %1
46       JMP %2
47       %1:
48       %2:
49 }
50 by
51 {
52       JZ %2
53 }
```

Figura 3.16 - Transforma o código da fig. 3.15 no código da fig. 3.16.

todas as regras forem verificadas. Segundo Aho (2006, p.348) “É característica da otimização por janela que cada melhoria possa gerar oportunidades para melhorias adicionais”. Para tal, se a opção *restart* for especificada nas regras o otimizador reiniciará o processo verificação de substituição desde o início do código.

Para ilustrar o uso da opção *restart*, observe como a regra de substituição da figura 3.17 afeta as instruções da figura 3.18, não conseguiríamos obter o código mais eficiente possível, visto que ainda sobrariam redundâncias.

```
11 replace
12 {
13       ADD %1 , %2
14       SUB %1 , %2
15 }
16 by {}     ; Substitui por nada.
```

Figura 3.17 - Regra para eliminar instruções.

```
 7 ADD R3 , R4
 8 ADD R1 , R2
 9 SUB R1 , R2
10 SUB R3 , R4
```

Figura 3.18 - Código a ser otimizado.

Aplicando-se a regra da figura 3.17 no código da figura 3.18, o SDCC acabaria gerando o código mostrado na figura 3.19.

```
7 ADD R3 , R4
8 SUB R3 , R4
```

Figura 3.19 - Código parcialmente otimizado.

O código ainda pode ser eliminado se a opção *restart* for especificada, como mostra a figura 3.20.

```
11 replace restart
12 {
13      ADD %1 , %2
14      SUB %1 , %2
15 }
16 by {}     ; Substitui por nada.
```

Figura 3.20 - Regra com *restart*.

Os resultados desta técnica de otimização são muito significativos, como será mostrado posteriormente. Será apresentada uma comparação do tempo de execução do algoritmo *heapsort* com otimização *peephole* e sem a mesma para fins de comparação. Além do tamanho do programa, o número de ciclos de execução diminui significativamente se esta técnica for aplicada.

## 3.3 iCode – *Intermediate Code*

### 3.3.1 Introdução

No processo de compilação de códigos, o SDCC quebra as instruções do código fonte a ser compilado em um formato de instruções de três endereços chamados *iCodes.* Este formato de código intermediário guarda informações sobre a operação a ser aplicada entre seus operandos. Informações sobre os operandos também são agrupadas em cada *iCode*, como o nome (endereço de memória), tipo

e tamanho, por exemplo.

### 3.3.2 Geração de código do SDCC

Para cada programa compilado, uma lista duplamente encadeada de *iCodes* é gerada de forma que o fluxo no qual as instruções são emitidas possa ser gerenciado corretamente. Cada nodo da lista representa uma instrução de três endereços que inclui uma operação que deve ser usada no processo de geração de códigos de máquina. Como é afirmado no *SDCC Compiler User Guide* (2014), a geração de código envolve traduzir estas operações em *assembly* correspondente do processador.

Para ilustrar o funcionamento dos *iCodes*, suponha o comando exemplo abaixo.

```
result = left + right;
```

Neste exemplo, o SDCC geraria um *iCode* para a geração de soma e atribuição do tipo IC_RESULT = IC_LEFT + IC_RIGHT. Neste *iCode* tem-se todas as informações necessárias para a geração do código de máquina. Sabendo-se a operação a ser aplicada (+) e tendo-se dados a respeito dos três operandos é possível emitir as instruções corretas para um processador alvo.

No exemplo acima, o SDCC analisaria o *iCode* da atribuição com soma e chamaria a rotina `genPlus`. Nesta rotina, deve ser implementada a lógica necessária pra a geração de soma para o processador alvo, por exemplo, a rotina `genPlus` para o processador ZR16S08 poderia emitir a sequência de instruções mostrada na figura 3.21.

```
1 MOV R0 , (left)
2 ADD R0 , (right)
3 MOV (result) , R0
```

Figura 3.21 - Código gerado pela rotina `genPlus` do ZR16S08.

A ideia básica na geração do código de máquina é percorrer toda a lista de *iCodes*, analisar a operação a ser feita por cada nó e emitir as instruções necessárias envolvendo os operandos.

O código mostrado na figura 3.22 ilustra o processo básico da geração de algumas instruções para o processador ZR16S08. Esta é a fase de *back-end* do compilador. Nela, recebe-se o código intermediário como parâmetro, gerado na fase anterior do processo de compilação. Esta é a essência do SDCC: receber a lista de *iCodes* e traduzi-la para o código de máquina correspondente

Por questões de espaço, o código da figura 3.22 não está completo. Somente foram ilustradas algumas das funcionalidades do SDCC para propósitos de explicação do funcionamento do compilador. O código completo se encontra-se no Apêndice C.

```
1  void genZr16Code(iCode * lic)
2  {
3      iCode *ic;
4
5      /* Percorre toda a lista de iCodes e analisa cada nó. */
6      for (ic = lic ; ic ; ic = ic->next)
7      {
8
9      /* Verifica a operação a ser realizada. */
10         switch (ic->op)
11         {
12             case '=': genAssign(ic);                              /* Ex.: x = y;     (IC_RESULT = IC_RIGHT) */
13                 break;
14             case '+': genPlus(ic);                                /* Ex.: x = y + z; (IC_RESULT = IC_LEFT + IC_RIGHT) */
15                 break;
16             case '>': genCmpGt(ic, ifxForOp(IC_RESULT(ic), ic));  /* Ex.: x = y > z;  (IC_RESULT = IC_LEFT > IC_RIGHT) */
17                 break;
18             case '<': genCmpLt(ic, ifxForOp(IC_RESULT(ic), ic));  /* Ex.: x = y < z;  (IC_RESULT = IC_LEFT < IC_RIGHT) */
19                 break;
20             case EQ_OP: genCmpEq(ic, ifxForOp(IC_RESULT(ic), ic)); /* Ex.: x = y == z; (IC_RESULT = IC_LEFT == IC_RIGHT) */
21                 break;
22             case AND_OP: genAndOp(ic);                            /* Ex.: x = y && z; (IC_RESULT = IC_LEFT && IC_RIGHT) */
23                 break;
24             case '!': genNot(ic);                                 /* Ex.: x = !y;     (IC_RESULT = !IC_LEFT) */
25                 break;
26             case '|': genOr(ic, ifxForOp(IC_RESULT(ic), ic));    /* Ex.: x = y | z;   (IC_RESULT = IC_LEFT | IC_RIGHT)*/
27                 break;
28             case IFX: genIfx(ic, NULL);                           /* if (IC_COND) goto IC_TRUE  ou if (!IC_COND) goto IC_FALSE */
29                 break;
30             case SEND: genSend(ic);               /* Passa parâmetros por registrador (Move IC_LEFT para algum registrador alocado). */
31                 break;
32             default:
33         }
34     }
35
36     /* Após a geração do código, pode-se aplicar o PeepHole para otimização. */
37
38     if (!options.nopeep)
39     {
40         peepHole(&ic);
41     }
42 }
```

Figura 3.22 - Geração de instruções para cada *iCode*.

Como mostra a linha 6 do código, um *loop* percorre todos os nodos da lista de *iCodes,* ao encontrar um *iCode* com a operação '+', por exemplo, a rotina `genPlus` é chamada (linha 14). Esta rotina deve prover o algoritmo necessário para emitir as

instruções de soma para o processador alvo.

Após todo o código fonte ter sido gerado, com o fim do *loop,* uma opção de otimização pode definir se o código deve ser otimizado ou não. Se a rotina `peepHole` for chamada (linha 40), a técnica de otimização por janela é aplicada no código gerado.

# 4 PORTE SDCC PARA O ZR16S08

## 4.1 Introdução

Nesta seção será demonstrado como gerar códigos para o processador ZR16S08 a partir dos *iCodes*. A ideia básica deste capítulo é ilustrar o funcionamento do processo de como as instruções são emitidas. Por questões de espaço, não será será feita uma discussão detalhada de como cada uma das funções foi implementada.

Será ilustrado como instruções comuns, como atribuição (=), AND bit a bit (&), complemento (~) e comparação podem ser implementadas para o processador ZR16S08.

Como discutido anteriormente, cada *iCode* é um código de três operandos. No processo de geração de código, deve-se extrair a operação a ser realizada do *iCode*, verificar qual função deve ser chamada (como mostra a figura 3.22) e em seguida, chamar o procedimento necessário para a emissão de código.

## 4.2 Emissão de instruções

### 4.2.1 Geração de atribuições

A figura 4.1 mostra um exemplo simplificado de como emitir instruções de atribuição para o processador ZR16S08.

Este exemplo tem somente um propósito ilustrativo, na prática, o método `genAssign` não foi implementado da forma mostrada na figura 4.1. Vários detalhes de implementação estão ocultos e não é de interesse mostrá-los nesta sessão. Porém, é mostrado essencialmente como o método `genAssign` foi construído. O exemplo agora citado ilustra uma possível implementação para atribuição de variáveis globais. Porém, deve-se ter em mente que existem várias possibilidades de atribuição, como de valores numéricos, parâmetros, variáveis temporárias, ponteiros

e atribuições de valores com tamanhos diferentes. O método `genAssign` deve prover técnicas de geração de código para cada um destes casos.

Logo que o compilador chama a função `genZr16Code`, um *loop* começa a "percorrer" a lista de *iCodes,* ao encontrar um nodo da lista com a operação '=', o método `genAssign` é chamado. O primeiro passo deste método é extrair os operandos do resultado da atribuição, e o valor a ser atribuído. Neste caso, isto nada mais é do que gerar um ponteiro para os endereços de memória do valor a ser atribuído e para o resultado.

```
1  /*
2       genAssign(): Código responsável por gerar atribuições.
3
4       Exemplo:
5
6            int x , y;
7            x = y;          (IC_RESULT = IC_RIGHT)
8  */
9
10 static void genAssign (iCode * ic)
11 {
12      /* Extrai resultado da operação (x). */
13      operand *result = IC_RESULT(ic);
14
15      /* Extrai o valor a ser atribuído ao resuldado da operação (y).*/
16      operand *right = IC_RIGHT(ic);
17
18      /* Pega o endereço de memória do resultado: Parte baixa. */
19      char *result_Addr_LOW = getOperandAddr (result , LOW);
20
21      /* Pega o endereço de memória do resultado: Parte alta. */
22      char *result_Addr_HIGH = getOperandAddr (result , HIGH);
23
24      /* Pega o endereço de memória do valor direito: Parte baixa. */
25      char *right_Addr_LOW  = getOperandAddr (right , LOW);
26
27      /* Pega o endereço de memória do valor direito: Parte alta. */
28      char *right_Addr_HIGH  = getOperandAddr (right , HIGH);
29
30      /* Emite instruções para o ZR16 em um arquivo de saída. */
31
32      emitcode ("MOV" , "R0 , (%s)", right_Addr_LOW);        // R0 <- (right_Addr_LOW)
33      emitcode ("MOV" , "(%s) , R0", result_Addr_LOW);       // (result_Addr_LOW) <- R0
34
35      if (AOP_SIZE (result) == 1) return;
36
37      emitcode ("MOV" , "R0 , (%s)", right_Addr_HIGH);       // R0 <- (right_Addr_HIGH)
38      emitcode ("MOV" , "(%s) , R0", result_Addr_HIGH);      // (result_Addr_HIGH) <- R0
39 }
```

Figura 4.1 - Exemplo simplificado da geração de atribuições para o ZR16S08.

Os comandos `IC_RESULT (iCode*)`, `IC_RIGHT(iCode*)` (linhas 13 e 16) retornam ponteiros para os operandos `result` (x) e `right` (y) da atribuição. Em

seguida, a função `getOperandAddr` retorna ponteiros para os seus respectivos endereços de memória.

Se os operandos ocuparem 2 Bytes, serão necessários dois endereços de memória para serem alocados, por este motivo, pode-se definir parâmetros `LOW` e `HIGH` que retornam os endereços dos Bytes inferior e superior, respectivamente.

Após os ponteiros para os respectivos valores do resultado e do valor atribuído terem sido alocados, só resta emitir as instruções necessárias para escrever os valores na memória do processador (linhas 32 a 38).

Note que na linha 35 do código da figura 4.1 o tamanho do resultado é verificado. Se for de 1 Byte, a rotina retorna, pois o valor do resultado ocupa somente um endereço de memória (1 Byte). Caso o tamanho do resultado seja de 2 Bytes, as linhas 37 e 38 emitem as instruções para escrever a outra metade do seu valor em outro endereço.

### 4.2.2 Geração de lógica AND bit a bit

A figura 4.2 ilustra uma possível implementação da lógica AND (`genAnd`) bit a bit para o ZR16S08. Novamente, detalhes de implementação estão ocultos.

A lógica de implementação da função `genAnd` é similar à lógica da função `genAssign`. Primeiramente são extraídos os operandos, que neste caso são três, atribuídos os ponteiros para os seus respectivos endereços de memória e, em seguida, as instruções são emitidas.

A lógica AND das partes alta e baixa é emitida nas linhas 30 e 38, respectivamente, ficando guardadas em R0. O resultado das partes alta e baixa é atribuído nas linhas 31 e 39, respectivamente.

```
1  /*
2       genAnd(): Código responsável por gerar AND  bit a bit.
3
4       Exemplo:
5
6            int x , y , z;
7            x = y & z;          (IC_RESULT = IC_LEFT & IC_RIGHT)
8  */
9
10 static void genAnd (iCode * ic)
11 {
12      operand *result = IC_RESULT(ic);    // Extrai resultado da operação (x).
13      operand *left   = IC_LEFT(ic);      // Extrai o valor esquerdo (y).
14      operand *right  = IC_RIGHT(ic);     // Extrai o valor direito(z).
15
16      char *result_Addr_LOW  = getOperandAddr (result , LOW);     // Pega o endereço de memória do resultado: Parte baixa.
17      char *result_Addr_HIGH = getOperandAddr (result , HIGH);   //  Pega o endereço de memória do resultado: Parte alta.
18
19      char *left_Addr_LOW   = getOperandAddr (left , LOW);        // Pega o endereço de memória do valor esquerdo: Parte baixa.
20      char *left_Addr_HIGH  = getOperandAddr (left , HIGH);       // Pega o endereço do valor do valor esquerdo: Parte alta.
21
22      char *right_Addr_LOW   = getOperandAddr (right , LOW);      // Pega o endereço de memória do valor direito: Parte baixa.
23      char *right_Addr_HIGH  = getOperandAddr (right , HIGH);     // Pega o endereço de memória do valor direito: Parte alta.
24
25      /* Emite instruções para o ZR16 em um arquivo de saida. */
26
27      /* AND na parte baixa. */
28
29      emitcode ("MOV" , "R0 , (%s)", left_Addr_LOW);         // R0 <- (left_Addr_LOW)
30      emitcode ("AND" , "R0 , (%s)", right_Addr_LOW);        // R0 <- R0 & (right_Addr_LOW)
31      emitcode ("MOV" , "(%s) , R0", result_Addr_LOW);       // (result_Addr_LOW) <- R0
32
33      if (AOP_SIZE (result) == 1) return;
34
35      /* AND na parte alta. */
36
37      emitcode ("MOV" , "R0 , (%s)", left_Addr_HIGH);        // R0 <- (left_Addr_HIGH)
38      emitcode ("AND" , "R0 , (%s)", right_Addr_HIGH);       // R0 <- R0 & (right_Addr_HIGH)
39      emitcode ("MOV" , "(%s) , R0", result_Addr_HIGH);      // (result_Addr_HIGH) <- R0
40 }
```

Figura 4.2 – Implementação da geração de lógica AND bit a bit para o ZR16S08.

### 4.2.3 Geração de lógica complemento

A figura 4.3 mostra uma implementação da lógica de complemento para o ZR16S08. Novamente, o funcionamento desta função é similar ao das outras. Extrai-se os operandos, calcula-se os ponteiros para os seus respectivos endereços de memória e emite-se as instruções.

Neste caso, a lógica de complemento é implementada fazendo-se a operação XOR das partes baixa (linha 33) e alta (linha 39) com 0xFF, o que resulta num valor complementado.

```
/*
     genCpl(): Código responsável por gerar complemento.

     Exemplo:

          int x , y;
          x = ~y;        (IC_RESULT = ~IC_LEFT)
*/

static void genCpl (iCode * ic)
{
     /* Extrai resultado da operação (x). */
     operand *result = IC_RESULT(ic);

     /* Extrai o valor a ser atribuído ao resuldado da operação (y).*/
     operand *left = IC_LEFT(ic);

     /* Pega o endereço de memória do resultado: Parte baixa. */
     char *result_Addr_LOW = getOperandAddr (result , LOW);

     /* Pega o endereço de memória do resultado: Parte alta. */
     char *result_Addr_HIGH = getOperandAddr (result , HIGH);

     /* Pega o endereço de memória do valor esquerdo: Parte baixa. */
     char *left_Addr_LOW  = getOperandAddr (left , LOW);

     /* Pega o endereço de memória do valor esquerdo: Parte alta. */
     char *left_Addr_HIGH  = getOperandAddr (left , HIGH);

     /* Emite instruções para o ZR16 em um arquivo de saída. */

     emitcode("MOV", "R0 , (%s)", left_Addr_LOW);          // R0 <- (left_Addr_LOW);
     emitcode("XOR", "R0 , 0xFF");                         // R0 <- R0 ^ 0xFF
     emitcode("MOV", "(%s) , R0", result_Addr_LOW));       // (result_Addr_LOW) <- R0

     if (AOP_SIZE (result) == 1) return;

     emitcode("MOV", "R0 , (%s)", left_Addr_HIGH);         // R0 <- (left_Addr_HIGH);
     emitcode("XOR", "R0 , 0xFF");                         // R0 <- R0 ^ 0xFF
     emitcode("MOV", "(%s) , R0", result_Addr_HIGH);       // (result_Addr_HIGH) <- R0
}
```

Figura 4.3 – Lógica de complemento (~) para o ZR16S08.

### 4.2.4 Geração de comparações

Para finalizar este capítulo, será ilustrado mais um exemplo de implementação: a função `genCmpEq`, que compara dois valores e retora `TRUE`, se forem iguais, ou `FALSE`, se forem diferentes. Para o entendimento desta função deve-se entender também a função `genCjne` (*compare and jump if not equal*) que compara dois valores e salta para um determinado *label*, se forem diferentes. A função `genCjne` e `genCmpEq` são mostradas nas figuras 4.4 e 4.5, respectivamente.

```
1  /*
2       Compara dois valores. Desvia para labelValoresDiferentes se os valores não forem iguais.
3
4       Obs.: A instrução CMP left , right seta flag Z = 1 se left == right.
5  */
6
7
8  static void genCjne(operand * left, operand * right , symbol *labelValoresDiferentes)
9  {
10      char *left_Addr_LOW   = getOperandAddr (left , LOW);        // Pega o endereço de memória do valor esquerdo: Parte baixa.
11      char *left_Addr_HIGH  = getOperandAddr (left , HIGH);       // Pega o endereço do valor do valor esquerdo: Parte alta.
12
13      char *right_Addr_LOW   = getOperandAddr (right , LOW);      // Pega o endereço de memória do valor direito: Parte baixa.
14      char *right_Addr_HIGH  = getOperandAddr (right , HIGH);     // Pega o endereço de memória do valor direito: Parte alta.
15
16      /* Compara as partes baixas de left e right. */
17
18      emitcode ("MOV" , "R0 , (%s)", left_Addr_LOW);              // R0 <- (left_Addr_LOW)
19      emitcode ("CMP", "R0 , (%s)",right_Addr_LOW);               // CMP R0 , (right_Addr_LOW)
20      emitcode ("JNZ", "!tlabel", labelValoresDiferentes);        // JNZ labelValoresDiferentes
21
22      if (AOP_SIZE (left) == 1 && AOP_SIZE (right) == 1)
23           return;
24
25      /* Compara as partes altas de left e right. */
26
27      emitcode ("MOV" , "R0 , (%s)", left_Addr_HIGH);             // R0 <- (left_Addr_HIGH)
28      emitcode ("CMP", "R0 , (%s)",right_Addr_HIGH);              // CMP R0 , (right_Addr_HIGH)
29      emitcode ("JNZ", "!tlabel", labelValoresDiferentes);        // JNZ labelValoresDiferentes
30 }
```

Figura 4.4 - Função `genCjne`: compara e desvia se dois valores são diferentes.

O funcionamento da função `genCjne` é simples: compara primeiramente as partes baixas dos valores, se forem diferentes o programa desvia para o *label* `LabelValoresDiferentes`. Caso contrário, a comparação é feita com as partes baixas e, novamente, se os valores não forem iguais, o programa desvia para `labelValoresDiferentes`.

No SDCC, *labels* são definidos e emitidos por um ponteiro do tipo `symbol`. O formato de saída "`!tlabel`" (linhas 20 e 29) é padrão do SDCC. Através deste formato, a função `genLabel` emitirá um *label* temporário quando chamada.

O primeiro passo da função `genCmpEq` é verificar se os valores são iguais ou diferentes, a verificação ocorre com a chamada do procedimento `genCjne` (linha 39). Caso a função `genCjne` não desvie o programa para o *label* `labelValoresDiferentes`, um desvio para `labelValoresIguais` é feito (linha 41).

No próximo passo, a emissão de código dependerá se a operação de comparação está sendo feita dentro de um `if` ou não. Isto é feito pelo comando da linha 43.

Caso a operação de comparação não seja parte de um comando `if` (e.g., `x = y == z`), o corpo da instrução `else` (linha 62 a 69) é executado. Neste caso, se os dois valores forem iguais, o valor do resultado é assinado com 1 (linha 64), caso contrário, o resultado será 0 (linha 67).

Se a operação de comparação for parte de um comando `if` (eg., `if (left == right)`), o corpo do comando `if` (linha 43 é executado). Os comentários na imagem 4.5 tentam explicar o funcionamento do código. Nesta etapa, a condição `if(IC_TRUE (ifx))` avisará o compilador que se os valores forem diferentes o corpo do `if` deve ser executado. Esta condição serve para códigos do tipo `if (left != right) {…}`, embora o nome da função seja `genCmpEq`.

```
34 static void genCmpEq(iCode * ic, iCode * ifx)
35 {
36      symbol *labelValoresDiferentes = newiTempLabel (NULL);
37      symbol *labelValoresIguais     = newiTempLabel (NULL);
38      symbol *labelExit              = newiTempLabel (NULL);
39      genCjne (IC_LEFT (ic) , IC_RIGHT (ic) , labelValoresDiferentes); // Desvia para labelValoresDiferentes se left != right
40       /* Se a função genCjne não desviar para labelValoresDiferentes, desvia para labelValoresIguais. */
41      emitcode ("JMP" , labelValoresIguais);
42
43      if (ifx)  // Comparação dentro de if.
44      {
45           if (IC_TRUE(ifx)) //if (left != right) {  /* ... */ }
46           {
47                emitLabel(labelValoresIguais);
48                emitcode("JMP", "!tlabel",labelExit);   // Se os valores são IGUAIS, não executa o corpo do if.
49
50                emitLabel(labelValoresDiferentes);
51                // Daqui pra diante, o corpo do if é gerado.
52           }
53           else if (IC_FALSE(ifx)) // if (left == right) { /* ... */ }
54           {
55                emitLabel(labelValoresDiferentes);
56                emitcode("JMP", "!tlabel", labelExit));      // Se os valores são DIFERENTES, não executa o corpo do if.
57                emitLabel(labelValoresIguais);
58                // Daqui pra diante, o corpo do if é gerado.
59           }
60      }
61      else   // Comparação fora de if (Ex.: x = y == z).
62      {
63                emitLabel (labelValoresIguais);
64                assign (IC_RESULT(ic) , 1);             // Resultado <- 1.
65                emitcode("JMP", "!tlabel",labelExit);
66                emitLabel (labelValoresDiferentes);
67                assign (IC_RESULT(ic) , 0);             // Resultado <- 0.
68                emitcode("JMP", "!tlabel",labelExit);
69      }
70
71      emitLabel(labelExit);
72 }
```

Figura 4.5 – Função de comparação entre dois valores: `genCmpEq`.

Se a condição `if(IC_FALSE (ifx))` for verdadeira, o corpo do comando `if` será executado se os operadores forem iguais.

### 4.2.5. Outras funções

No porte atual do processador ZR16S08 foram implementadas as seguintes funções:

a) Soma '+' (`genPlus`);

b) Subtração '-' (`genMinus`);

c) AND lógica (&&) (`genAndOp`);

d) OR lógica (||) (`genOrOp`);

e) AND bit a bit (&) (`genAnd`);

f) OR bit a bit (|) (`genOr`);

g) XOR bit a bit (^) (`genXor`);

h) NOT (!) (`genNot`);

i) Operações de comparação == , !=, <, <=, > e >=;

j) Deslocamentos `<<` e `>>` (`genShiftLeft` e `genRightLeft`);

k) Passagem e recebimento de parâmetros (`genSend` e `genReceive`);

l) Complemento (~) (`genCpl`);

m) `goto` (`genGoto`);

n) Multiplicação (*) (`genMul`);

o) Divisão (/) (`genDiv`).

É impraticável tentar explicar aqui como cada uma das funções acima citadas foi implementada para o processador ZR16S08. Ao invés disto, tentou-se passar uma ideia básica de como isto foi feito, ou pode ser feito para outras funções.

Nem todas as funcionalidades da linguagem C podem ser compiladas pelo porte atual do SDCC, a função módulo (%) (`genMod`), por exemplo, não está atualmente implementada.

Ponteiros ainda não podem ser passados como parâmetros em funções, mas podem ser atribuídos.

A versão atual do porte também não suporta operações com números em arimética de ponto flutuante. Em todos os testes aplicados foram utilizados os tipos de dados `int` e `char`.

O compilador também ainda não suporta variáveis com o atributo `const`.

## 4.3 Conclusão

O presente capítulo tenta ilustrar como pode ser feito o porte para instruções básicas do ZR16S08. Praticamente todas as instruções podem ser implementadas da mesma forma padrão: extrair os ponteiros para os operandos, calcular os seus respectivos endereços de memória e emitir as instruções na ordem necessária a fim de implementar a lógica da função a ser gerada.

Como citado na seção 4.2.5, o porte do compilador não está completo atualmente, no entanto, as funcionalidades até então suportadas são suficientes para a aplicação de *benchmarks* e fazer uma avaliação de desempenho do compilador.

# 5 Resultados

## 5.1 Introdução

Com o porte do compilador feito, é necessário que o mesmo seja testado e a sua performance seja avaliada.

A fim de avaliar as instruções emitidas pelo compilador, foi escolhido um algoritmo CRC de verificação de erros com uma alta variedade de instruções. O propósito deste teste não é somente avaliar o desempenho do compilador, mas observar se as instruções são emitidas corretamente.

As avaliações de desempenho aqui consideradas dizem respeito ao número de ciclos necessários que o processador necessita para executar uma determinada tarefa e/ou o tamanho do código gerado. Desta forma, é mais interessante utilizar algoritmos com *loops*, que executam durante vários ciclos, assim pode-se fazer uma boa análise dos tempos de resposta. Neste contexto, os algoritmos *bubblesort* e *heapsort* foram utilizados para testes de performance. O fato dos dois algoritmos possuírem complexidades algorítmicas diferentes também é interessante, pois *benchmarks* com uma maior variedade de testes fornecem diferentes respostas do compilador.

Segundo PATTERSON (1998), é importante utilizar diferentes métricas e aplicações para avaliar o desempenho de sistemas embarcados. Desta forma, as métricas de avaliação de performance serão baseadas no número de ciclos necessários para a execução e tamanho de programa gerado pelos compiladores SDCC e ZR16 Compiler.

## 5.2 Avaliação de desempenho

### 5.2.1 Teste com o *bubblesort*

A fim de avaliar o desempenho do compilador, foi executado o algoritmo *bubblesort* variando-se o tamanho de um vetor e observando-se a taxa de crescimento do número de ciclos necessários para o vetor ser ordenado.

O mesmo algoritmo foi compilado com SDCC e com o ZR16 Compiler a fim de coletar os dados necessários para realizar a comparação e avaliação.

A Tabela 5.1 mostra o número de ciclos tomados para o vetor ser ordenado para cada tamanho, para ambos os compiladores. A quarta coluna da tabela ilustra a melhora de performance do SDCC.

Tabela 5.1 – Comparação do número de ciclos para o *bubblesort.*

| Tamanho do Vetor | SDCC | ZR16 Compiler | Melhora (%) |
|---|---|---|---|
| 5 | 930 | 1360 | 31.61 |
| 10 | 3940 | 5730 | 31.23 |
| 15 | 9025 | 13100 | 31.10 |
| 20 | 15240 | 20995 | 27.41 |
| 25 | 22567 | 29366 | 23.15 |
| 30 | 31829 | 40366 | 21.14 |
| 35 | 42712 | 53172 | 19.67 |
| 50 | 98118 | 136620 | 28.18 |
| 60 | 139793 | 192480 | 27.37 |
| 70 | 188715 | 257590 | 26.73 |

Para verificar os resultados graficamente, foi feita uma interpolação para os dados gerados pelos programas compilados com o SDCC e ZR16 Compiler.

Visto que a complexidade do algoritmo *bubblesort* é $O(n^2)$ no pior caso, a interpolação aplicada foi quadrática. Desta maneira, a equação gerada se aproxima melhor dos valores medidos para os ciclos.

As equações aproximadas resultantes das interpolações quadráticas para o

SDCC (1) e ZR16 Compiler (2), são, respectivamente,

$$Ciclos1 = 40x^2 - 157x + 1131 \quad (1)$$

$$Ciclos2 = 58x^2 - 384x + 2971 \quad (2)$$

onde x denota o tamanho do vetor, Ciclos1 e Ciclos2 denotam o número de ciclos tomados pelos programas compilados pelo SDCC e pelo ZR16 Compiler, respectivamente.

Os gráficos da figura 5.1 ilustram o número de ciclos em função do tamanho do vetor sendo aproximados pelas interpolações (1) e (2), respectivamente. No eixo vertical do gráfico, tem-se o número de ciclos executados, no eixo horizontal, tem-se o tamanho do vetor, que foi variado de 5 até 70.

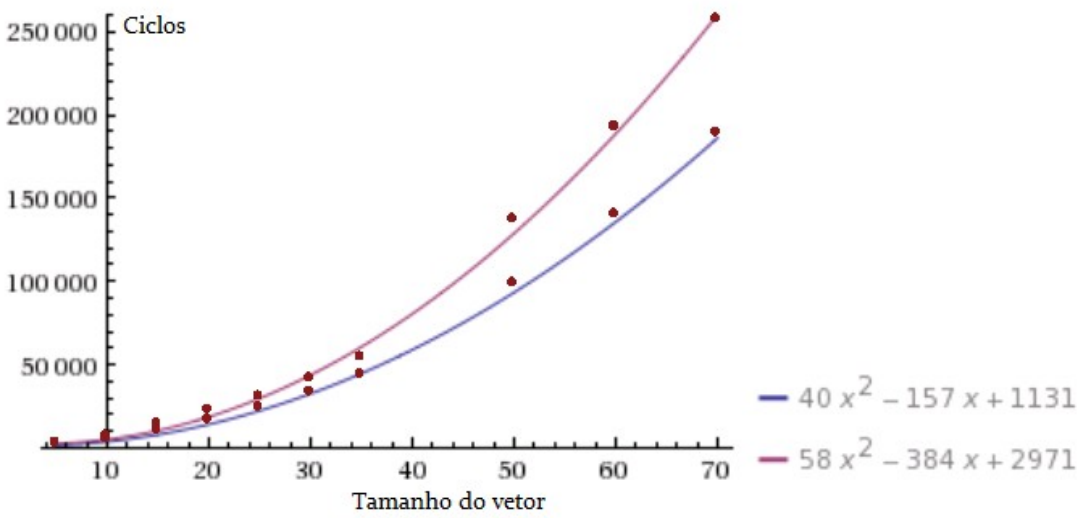


Figura 5.1 - Número de ciclos para o *bubblesort*.

De acordo com a figura 5.1, é evidente como o SDCC (gráfico azul) tem um desempenho melhor que o ZR16 Compiler (gráfico vinho) na medida em que o tamanho do vetor aumenta.

Para avaliar a performance dos dois programas, pode-se tomar a derivada das equações em um determinado ponto e avaliar a diferença de inclinação entre ambas. Sendo assim, as derivadas para as interpolações do SDCC (3) e ZR16 Compiler (4) são, respectivamente,

$$d\frac{(Ciclos1)}{dx}=80x-157 \quad (3)$$

$$d\frac{(Ciclos2)}{dx}=116x-384 \quad (4)$$

Para analisar o quanto o número de ciclos está crescendo, num ponto específico, foi escolhido o ponto em x = 50 elementos. Neste ponto, as inclinações para o SDCC (5) e ZR16 Compiler (6) são, respectivamente

$$d\frac{(Ciclos1)}{dx}(x=50)=3843 \quad (5)$$

$$\frac{d(Ciclos2)}{dx}(x=50)=5416 \quad (6)$$

Nota-se que, para um vetor com 50 elementos, o SDCC cresce 5416/3843 = 1.4 vezes menos que o ZR16 Compiler.

Para ilustrar esta diferença graficamente, foram traçados os gráficos com as inclinações (5) e (6) que cruzam pelos pontos (50,Ciclos1(50)) e (50,Ciclos2(50)), que representam o número de ciclos aproximado executados pelos programas compilados com o SDCC e com ZR16 Compiler para um vetor de 50 elementos.

Os gráficos das figuras 5.2 e 5.3 ilustram as inclinações, em x = 50, para o SDCC e ZR16 Compiler, respectivamente.

As retas tangentes no ponto x = 50, em ambos os casos, tem o propósito de ilustrar graficamente a diferença de inclinação das parábolas no ponto evidenciando a melhora de performance atingida pelo SDCC.

Observando-se a diferença entre as inclinações pode-se concluir que o número de ciclos do programa compilado pelo SDCC cresce menos que o número de ciclos do programa compilado pelo ZR16 Compiler conforme o tamanho do vetor é aumentado.

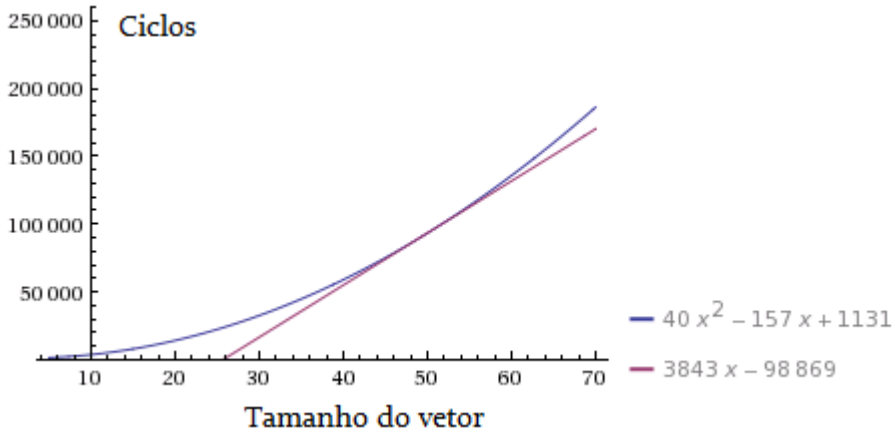


Figura 5.2 – Taxa de variação com 50 elementos para o SDCC.

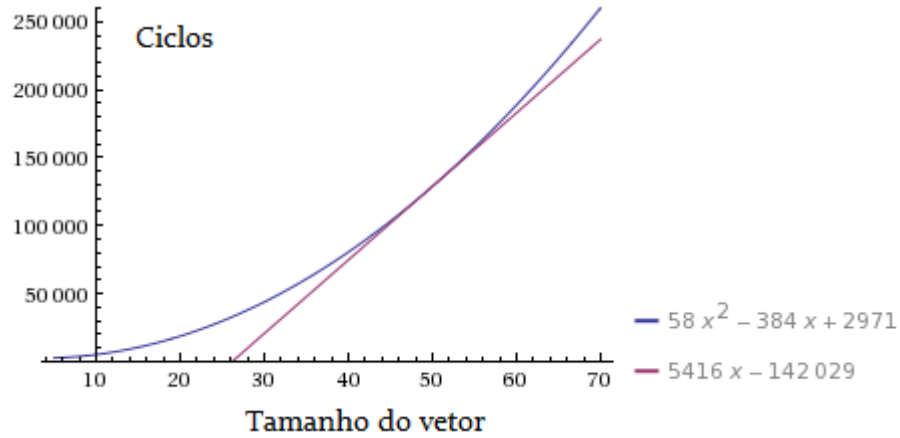


Figura 5.3 – Taxa de variação com 50 elementos para o ZR16 Compiler.

### 5.2.2 Teste com o *heapsort*

Uma análise semelhante a da seção 5.2.1 foi feita com o algoritmo *heapsort.*

A Tabela 5.2 ilustra o número de ciclos tomados pelo algoritmo compilado pelo SDCC e pelo ZR16 Compiler. A quarta coluna da tabela ilustra a melhora de performance do SDCC.

Após os dados terem sido amostrados, foi novamente aplicada uma técnica de interpolação para os dois programas.

Tabela 5.2 – Comparação do número de ciclos para o *heapsort.*

| Tamanho do Vetor | SDCC | ZR16 Compiler | Melhora (%) |
|---|---|---|---|
| 5 | 688 | 1742 | 60.50 |
| 10 | 1811 | 4278 | 57.66 |
| 15 | 3068 | 7403 | 58.55 |
| 20 | 4619 | 11146 | 58.55 |
| 25 | 5960 | 14442 | 58.73 |
| 30 | 7826 | 18569 | 57.85 |
| 35 | 9342 | 22167 | 57.85 |
| 50 | 14836 | 35072 | 57.69 |
| 60 | 18613 | 33065 | 43.70 |
| 70 | 22964 | 44976 | 48.94 |

Visto que o *heapsort* tem complexidade O(nlog(n)) no pior caso, e que o tamanho do vetor foi no máximo 70, uma interpolação linear fica bem aproximada dos ciclos tabelados.

As equações de interpolação linear para o programa compilado com o SDCC (7) e ZR16 Compiler (8) são, respectivamente,

$$Ciclos1 = 345\,x - 2035 \tag{7}$$

$$Ciclos2 = 655\,x - 1700 \tag{8}$$

Os gráficos da figura 5.4 ilustram o número de ciclos em função do tamanho do vetor sendo aproximados pelas interpolações (7) e (8), respectivamente.

Novamente, de acordo com a figura 5.4, nota-se uma significativa melhora no desempenho do SDCC (gráfico azul) com relação ao ZR16 Compiler (gráfico vinho).

A relação (9) mostra que o SDCC executará o *heapsort* aproximadamente na metade do tempo tomado pelo ZR16 Compiler para vetores grandes.

Segundo as equações de interpolação, o algoritmo compilado pelo SDCC sempre será executado pelo menos 50% mais rápido para vetores com tamanhos pequenos (menores que 68).

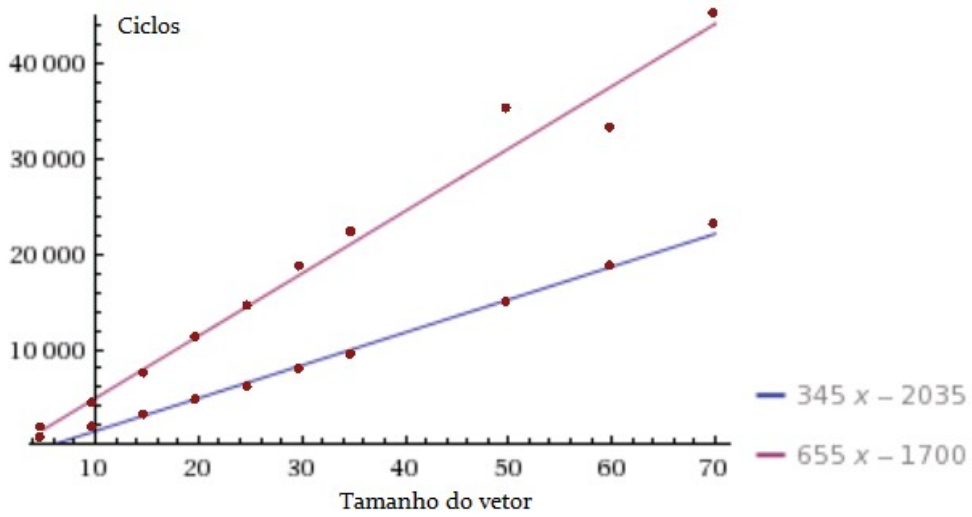


Figura 5.4 – Taxa de crescimento do *heapsort.*

Conforme o tamanho do vetor aumenta, a relação entre as inclinações converge para 1.89, segundo a equação (9).

$$\lim_{x \to \infty} (655\,x - 1700)/(345\,x - 2035) = 1.89 \tag{9}$$

### 5.2.3 Avaliação do *peephole* com *heapsort*

Para o propósito de verificar a influência da técnica de otimização por janela, foi executado o algoritmo *heapsort* com e sem a otimização peephole.

O número de ciclos tomados por ambos os algoritmos são mostrados na Tabela 5.3. A quarta coluna da tabela ilustra a melhora de performance atingida pelo *peephole*.

A Tabela 5.3 ilustra o quão influente a otimização por janela pode ser. Para ilustrar graficamente, os valores da tabela foram novamente interpolados.

A equação de interpolação linear para o *heapsort* não otimizado (10) é:

$$Ciclos\,2 = 442\,x - 2679 \tag{10}$$

Tabela 5.3 – O *heapsort* com e sem otimização *peephole.*

| Tamanho do Vetor | Com *peephole* | Sem *peephole* | Melhora (%) |
|---|---|---|---|
| 5 | 688 | 855 | 19.53 |
| 10 | 1811 | 2291 | 20.95 |
| 15 | 3068 | 3903 | 21.39 |
| 20 | 4619 | 5892 | 21.60 |
| 25 | 5960 | 7611 | 21.69 |
| 30 | 7826 | 10013 | 21.84 |
| 35 | 9342 | 11975 | 21.98 |
| 50 | 14836 | 19036 | 22.06 |
| 60 | 18613 | 23928 | 22.21 |
| 70 | 22964 | 29524 | 22.21 |

onde Ciclos2 é o número de ciclos tomados pelo *heapsort* não otimizado.

Os gráficos da figura 5.5 ilustram o número de ciclos em função do tamanho do vetor sendo aproximados pelas interpolações (7) e (10), respectivamente.

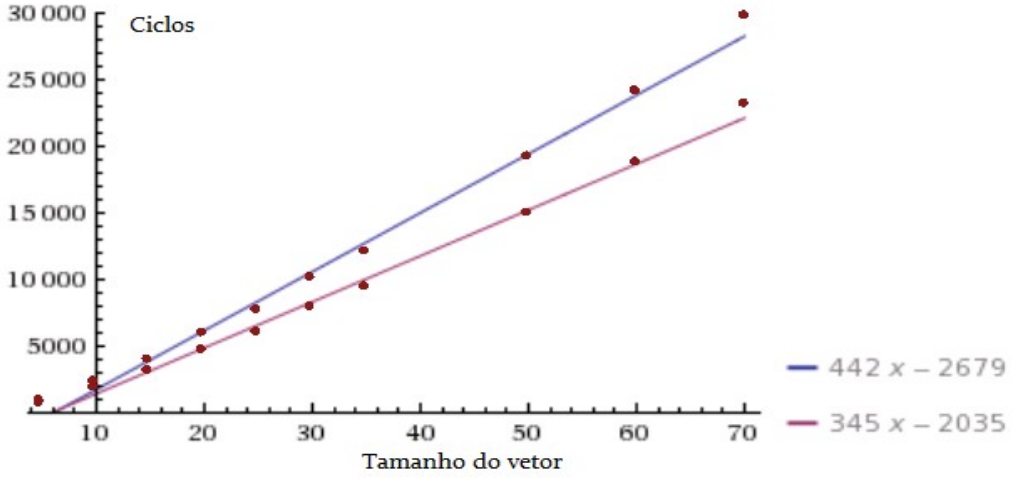


Figura 5.5 - Análise de influência do *peephole*.

Fazendo-se uma análise similar à da seção anterior, observa-se que a otimização *peephole* tem uma influência de reduzir aproximadamente 442/345=1.28 vezes o número de ciclos.

Ainda pode-se observar que, mesmo sem a otimização peephole (gráfico azul), o programa compilado pelo SDCC tem um desempenho melhor que compilado pelo ZR16 Compiler.

## 5.3 Avaliação de memória

O tamanho de código dos programas para sistemas embarcados é de suma importância. Desta forma, também é interessante que as técnicas de otimização reduzam ao máximo o número de instruções geradas pelo compilador.

A Tabela 5.4 ilustra o número de instruções dos programas CRC, *bubblesort* e *heapsort* compilados pelo SDCC com e sem otimização *peephole* e pelo ZR16 Compiler.

Tabela 5.4 – Tamanho dos programas (número de instruções).

| Programa | SDCC com *peephole* | SDCC sem *peephole* | ZR16 Compiler |
|---|---|---|---|
| CRC | 33 | 37 | 118 |
| *heapsort* | 40 | 105 | 220 |
| *bubblesort* | 87 | 100 | 122 |

Nota-se claramente a significativa melhora do SDCC com relação ao ZR16 Compiler, mesmo que não se aplique a técnica de *peephole*.

O número médio[1] aproximado de instruções geradas dos três programas mostrados na Tabela 5.4, compilados com o SDCC otimizado, não otimizado e ZR16 Compiler é, respectivamente, 53, 80 e 153 instruções. Isto mostra que o SDCC otimizado ocupa, em média, 153/53 = 2.88 vezes menos memória que o ZR16 Compiler.

[1]Embora o algoritmo *heapsort* seja mais complexo que o *bubblesort*, o número de instruções gerado pelo mesmo foi menor, pois nos seus testes, foi utilizado o tipo de dados unsigned char. Para o bubblesort, o tipo usado foi signed int. Esta diferença de tamanho ocorre devido ao fato de que comparações para unsigned char são bem mais simples do que comparações para signed int.

# CONCLUSÕES

Otimização deve sempre ser uma palavra-chave quando se fala em compiladores para sistemas embarcados. No decorrer do trabalho, fez-se um esforço a fim de demonstrar o quanto as técnicas de melhora de desempenho podem influenciar na performance de microcontroladores.

No que diz respeito a geração de código com o SDCC, tentou-se explicar da maneira mais concisa possível as técnicas utilizadas para portá-lo para o processador ZR16S08, embora nem todos os aspectos do porte tenham sido aqui esclarecidos. No entanto, espera-se ser possível entender a ideia básica deste processo através dos exemplos ilustrativos.

Através de dados levantados por meio de simulações foi possível verificar a melhora de desempenho do MCU ZR16S08 através de tabelas e gráficos.

Deve ser enfatizado que a melhora de desempenho mostrada neste trabalho é válida somente para o *benchmark* aqui utilizado. A performance do SDCC foi definitivamente melhor com os algoritmos CRC, *bubblesort* e *heapsort.* No entanto, este aumento de performance não pode ser generalizado. Provavelmente existem casos em que o ZR16 Compiler é mais eficiente que o SDCC. Segundo Stallings (2009), o desempenho do processador com um dado programa não necessariamente determina a sua performance com um outro tipo de aplicação.

É preciso lembrar que o SDCC tem porte livre e está sendo atualizado frequentemente, várias técnicas de otimização e regras do *peephole* são modificadas, ou adicionadas o tempo todo. Desta forma, os dados de desempenho mostrados neste trabalho podem mudar, para melhor ou pior, com frequência.

O capítulo 1 não deve ser considerado um manual de referência sobre o ZR16S08, é apenas uma presentação básica sobre o MCU, cujo entendimento é necessário a fim de entender a maneira pela qual as instruções devem ser emitidas pelo SDCC no processo de porte que foi verificado no capítulo 4.

# REFERÊNCIAS


PATTERSON, DAVID A. **Computer Organization and Design: The Hardware/Software Interface.** 2.ed. San Francisco, CA, USA: Morgan Kaufmann. 1998. 965 p.

SDCC Compiler User Guide. EUA: Sourceforge, 2014. Disponível em: <http://sdcc.sourceforge.net/doc/sdccman.pdf>. Acesso em 1 jul. 2014.

STALLINGS, WILLIAM. **Computer Organization and Architecture: Designing for Performance. 8**.ed. Upper Saddle River, NJ, USA: Prentice Hall. 2009. 792 p.

V. AHO, ALFRED. **COMPILERS: Principles, Techniques and Tools.** 2.ed. Boston, MA, USA: Addison Wesley. 2006. 634 p.

ZR16S08 User Guide. Santa Maria: SMDH, 2013. Disponível em: <http://w3.ufsm.br/smdh/downloads.php>. Acesso em 1 jul. 2014.

# APÊNDICES

## Apêndice A – Passagem de parâmetros

Por padrão, o primeiro parâmetro das chamadas de funções no SDCC é passado por registradores. O restante dos parâmetros são passados por endereços de memória, que são alocados em tempo de compilação.

A figura A.1 ilustra um código fonte com a chamada de dois procedimentos.

```
 1 void f1 (int , int);
 2 void f2 (int , int);
 3
 4 int x , y , w , z;
 5
 6 void main ()
 7 {
 8      f1 (0x1122 , 0x3344);
 9      f2 (0x5566 , 0x7788);
10 }
11
12 void f1 (int arg1 , int arg2)
13 {
14      x = arg1;
15      y = arg2;
16 }
17
18 void f2 (int arg1 , int arg2)
19 {
20      w = arg1;
21      z = arg2;
22 }
```

Figura A.1 – Passagem de parâmetros.

Considerando-se que no programa não haverá chamadas de procedimentos por outros procedimentos, por exemplo, `f1` chamar `f2`, com exceção da função `main`, é mais conveniente alocar os mesmos endereços de memória para os parâmetros de diferentes funções, desta forma, utiliza-se a menor quantidade de memória possível.

Como pode ser observado na figura A.2, os segundos parâmetros das funções `f1` e `f2` foram alocados no mesmo endereço de memória. Obviamente, se a função `f1` chamar a função `f2`, ou vice-versa, antes dos valores `x , y , w` ou `z` serem atribuídos, os valores resultantes serão errados.

```
1 DA 0 {x_L}     DA 1 {x_H}              ; Aloca endereços de memórias para variáveis globais (L: parte baixa  H: parte alta)
2 DA 2 {y_L}     DA 3 {y_H}
3 DA 4 {w_L}     DA 5 {w_H}
4 DA 6 {z_L}     DA 7 {z_H}
5 DA 8 {_f1_PARM_2_L}                    ; Aloca endereços de memória para o segundo parâmetro de f1 (parte baixa)
6 DA 8 {_f2_PARM_2_L}                    ; Aloca endereços de memória para o segundo parâmetro de f2 (parte baixa)
7 DA 9 {_f1_PARM_2_H}                    ; Aloca endereços de memória para o segundo parâmetro de f1 (parte alta)
8 DA 9 {_f2_PARM_2_H}                    ; Aloca endereços de memória para o segundo parâmetro de f2 (parte alta)
9
10 main:MOV  R0 , 0x44
11      MOV  (_f1_PARM_2_L) , R0         ; Passa parte baixa do segundo parâmetro de f1
12      MOV  R0 , 0x33
13      MOV  (_f1_PARM_2_H) , R0         ; Passa parte alta do segundo parâmetro de f1
14      MVS  R8 , 0x22                   ; Primeiro parâmetro é passado por registradores
15      MVS  R9 , 0x11
16      CALL f1
17      MOV  R0 , 0x88
18      MOV  (_f2_PARM_2_L) , R0
19      MOV  R0 , 0x77
20      MOV  (_f2_PARM_2_H) , R0
21      MVS  R8 , 0x66
22      MVS  R9 , 0x55
23      CALL f2
24      RET
25 f1:  MOV  R0 , R8
26      MOV  (x_L) , R0
27      MOV  R0 , R9
28      MOV  (x_H) , R0
29      MOV  R0 , (_f1_PARM_2_L)
30      MOV  (y_L) , R0
31      MOV  R0 , (_f1_PARM_2_H)
32      MOV  (y_H) , R0
33      RET
34 f2:  MOV  R0 , R8
35      MOV  (w_L) , R0
36      MOV  R0 , R9
37      MOV  (w_H) , R0
38      MOV  R0 , (_f2_PARM_2_L)
39      MOV  (z_L) , R0
40      MOV  R0 , (_f2_PARM_2_H)
41      MOV  (z_H) , R0
42      RET
```

A figura A.2 - Passagem de parâmetros. Código compilado.

Para evitar tal problema, deve-se alocar os parâmetros da função chamadora (função reentrante) em uma pilha. Para tal, o SDCC dispõe da palavra-chave `__reentrant`. A figura A.3 ilustra um exemplo.

Ao declarar uma função com a palavra-chave `__reentrant,` o SDCC passará os parâmetros do método por pilha, com exceção do primeiro. Desta forma, evita-se que um valor errado seja assinado ao resultado caso a função chame alguma outra.

```
1  void f1 (int , int) __reentrant;
2  void f2 (int , int);
3
4  int x , y , w , z;
5
6  void main ()
7  {
8       f1 (0x1122 , 0x3344);
9  }
10
11 void f1 (int arg1 , int arg2) __reentrant
12 {
13      f2 (0x5566 , 0x7788);
14      x = arg1;
15      y = arg2;
16 }
17
18 void f2 (int arg1 , int arg2)
19 {
20      w = arg1;
21      z = arg2;
22 }
```

Figura A.3 – Função reentrante.

As figuras A.4 e A.5 mostram as funções `main` e `f1` compiladas, respectivamente.

```
6  main: MVS sp , 200                    ; Define inicio de sp no endereço 200
7       MOV  R0 , 0x44                   ; Parametros da função reentrante são passados na pilha
8       MOV  (sp) , R0
9       INC  sp
10      MOV  R0 , 0x33
11      MOV  (sp) , R0
12      INC  sp
13      MVS  R8 , 0x22
14      MVS  R9 , 0x11
15      CALL f1
16      RET
```

Figura A.4 – Passagem de parâmetros por pilha.

Caso uma função não seja reentrante, é mais conveniente não utilizar o atributo `__reentrant`, pois alocar os parâmetros na pilha em tempo de execução resulta em *overhead* gerado pelas operações de incremento, decremento e ajustes do *stack pointer*.

```
17 f1:  MOV R3 , sp                                    ; Salva sp
18      MOV  R0 , 0x88
19      MOV  (_f2_PARM_2_L) , R0
20      MOV  R0 , 0x77
21      MOV  (_f2_PARM_2_H) , R0
22      MOV  (sp) , R9                                 ; Salva registradores do primeiro parametro na pilha
23      INC  sp
24      MOV  (sp) , R8
25      INC  sp
26      DEC sp                                         ; Ajusta sp para a função f2
27      DEC sp
28      MVS  R8 , 0x66
29      MVS  R9 , 0x55
30      CALL f2
31      INC sp                                         ; Ajusta sp para recuperar R8 e R9
32      MOV  R8 , (sp)
33      DEC  sp
34      MOV  R9 , (sp)
35      MOV  R0 , R8
36      MOV  (x_L) , R0
37      MOV  R0 , R9
38      MOV  (x_H) , R0
39      MOV  sp , R3                                   ; Recupera valor de sp para a função f1
40      MOV  R0 , 2
41      SUB  sp , R0
42      MOV  R0 , (sp)
43      MOV  (y_L) , R0
44      INC  sp
45      MOV  R0 , (sp)
46      MOV  (y_H) , R0
47      RET
```

Figura A.5 – Recebimento de parâmetros por pilha.

## Apêndice B – Interrupções

Utiliza-se o atributo `__interrupt` para definir funções de interrupção. Se tal atributo for habilitado a uma função, como mostra a figura B.1, o SDCC criará uma instrução de desvio para tal rotina no endereço 0x3C0, que é o endereço do início do vetor de interrupções do ZR16S08.

Para habilitar ou desabilitar interrupções no ZR16S08 deve-se setar o ou resetar o LSB (*Least Significant Bit)* do registrador R15. Isto só pode ser feito emitindo instruções diretamente em *assembly*, o que é feito pelas definições `CLI` e `SEI` (linhas 1 e 2) mostradas na figura B.1.

O código tem somente propósito ilustrativo e não faz nada de útil, pois não foi configurada nenhum tipo de interrupção.

As interrupções devem ser configuradas setando-se os bits necessários no bloco IO. O ZR16S08 suporta interrupções de Timer, ADC e outras duas interrupções externas.

```
#define CLI __asm__ ("\n MOV R0 , 1 \n OR R15 , R0\n");
#define SEI __asm__ ("\n MOV R0 , 0xFE \n AND R15 , R0\n");


void ALARME () __interrupt;

int x , y , z;

void main ()
{
     CLI  // Habilita interrupções

     /*
          Configurar as interrupções pelos pino IO
     */

     while (1);

}

void ALARME () __interrupt
{
     x = (y&z)^0xFFFF;
}
```

Figura B.1 – Tratamento de interrupções.

A figura B.2 mostra o código de ISR (*interrupt service routine*) compilado. Logo que a rotina de interrupção é chamada, todos os valores alterados pela ISR são salvos na pilha. Após a rotina terminar, os valores são restaurados, como mostra a figura B.2.

```
DR sp = R10

DA 0 {x_L}
DA 1 {x_H}
DA 2 {y_L}
DA 3 {y_H}
DA 4 {z_L}
DA 5 {z_H}


main: MVS sp , 200
     MOV R0 , 1
     OR R15 , R0                  ; Habilita interruções
L00102:   JMP  L00102             ; Loop infinito
     RET
ALARME:   MOV  (sp) , R9          ; Salva registradores R9 e R8, que são alterados durante interrupção.
     INC  sp
     MOV  (sp) , R8
     INC  sp
     MOV  R0 , (y_L)              ; Inicio da operação ISR: x = (y&z)^0xFFFF;
     AND R0 , (z_L)
     MOV  R8 , R0
     MOV  R0 , (y_H)
     AND R0 , (z_H)
     MOV  R9 , R0
     MOV R0 , R8
     XOR R0 , 0xFF
     MOV  (x_L) , R0
     MOV  R0 , R9
     XOR R0 , 0xFF
     MOV  (x_H) , R0              ; Fim da operação da ISR
     DEC  sp
     MOV  R8 , (sp)               ; Recupera valores de R8 e R9
     DEC  sp
     MOV  R9 , (sp)
     RETS                         ; Retorna da ISR
     ORG  0x3C0                   ; Inicio do vetor de interrupções
     JMP  ALARME
```

Figura B.2 – Código da figura B.1 compilado.

## Apêndice C – `genZr16Code` completo

```
void genZr16Code(iCode * lic)
{
	iCode *ic;
	for (ic = lic; ic; ic = ic->next){
		initGenLineElement();
		genLine.lineElement.ic = ic;
		switch (ic->op){
			case '!':
				genNot(ic);
				break;
			case '~':
				genCpl(ic);
				break;
			case UNARYMINUS:
				genUminus(ic);
				break;
			case IPUSH:
				genIpush(ic);
				break;
			case IPOP:{
				iCode *ifxIc, *popIc;
				for (ifxIc = ic->next; ifxIc && ifxIc->op == IPOP; ifxIc =
ifxIc->next);
				for (popIc = ic; popIc && popIc->op == IPOP; popIc = popIc-
>next)
					CommonRegs |= (ifxIc && ifxIc->op == IFX && !ifxIc-
>generated && regsInCommon(IC_LEFT(popIc), IC_COND(ifxIc)));
				if (CommonRegs)
					genIfx(ifxIc, ic);
				else
					genIpop(ic);
			}
				break;
			case CALL:
				genCall(ic);
				break;
			case PCALL:
				genPcall(ic);
				break;
			case FUNCTION:
				genFunction(ic);
				break;
			case ENDFUNCTION:
				genEndFunction(ic);
```

```
                        break;
                case RETURN:
                        genRet(ic);
                        break;
                case LABEL:
                        genLabel(ic);
                        break;
                case GOTO:
                        genGoto(ic);
                        break;
                case '+':
                        genPlus(ic);
                        break;
                case '-':
                        if (!genDjnz(ic, ifxForOp(IC_RESULT(ic), ic)))
                                genMinus(ic);
                        break;
                case '*':
                genMult(ic);
                        break;
                case '/':
                        genDiv(ic);
                        break;
                case '%':
                        genMod(ic);
                        break;
                case '>':
                        genCmpGt(ic, ifxForOp(IC_RESULT(ic), ic));
                        break;
                case '<':
                        genCmpLt(ic, ifxForOp(IC_RESULT(ic), ic));
                        break;
                case LE_OP:
                case GE_OP:
                case NE_OP:
                        werror(E_INTERNAL_ERROR, __FILE__, __LINE__, "got '>=' or '<='
shouldn't have come here");
                        break;
                case EQ_OP:
                        genCmpEq(ic, ifxForOp(IC_RESULT(ic), ic));
                        break;
                case AND_OP:
                        genAndOp(ic);
                        break;
                case OR_OP:
                        genOrOp(ic);
                        break;
```

```
                        case '^':
                               genXor(ic, ifxForOp(IC_RESULT(ic), ic));
                               break;
                        case '|':
                               genOr(ic, ifxForOp(IC_RESULT(ic), ic));
                               break;
                        case BITWISEAND:
                               genAnd(ic, ifxForOp(IC_RESULT(ic), ic));
                               break;
                        case INLINEASM:
                               genInline(ic);
                               break;
                        case RRC:

                        genRRC(ic);
                               break;
                        case RLC:
                               genRLC(ic);
                               break;
                        case GETHBIT:
                               assert(0);
                               break;
                        case GETABIT:
                               genGetAbit(ic);
                               break;
                        case GETBYTE:
                               genGetByte(ic);
                               break;
                        case GETWORD:
                               genGetWord(ic);
                               break;
                        case LEFT_OP:
                               genLeftShift(ic);
                               break;
                        case RIGHT_OP:
                               genRightShift(ic);
                               break;
                        case GET_VALUE_AT_ADDRESS:
                               genPointerGet(ic, hasInc(IC_LEFT(ic), ic,
getSize(operandType(IC_RESULT(ic)))),
                                                ifxForOp(IC_RESULT(ic), ic));
                               break;
                        case '=':
                               if (POINTER_SET(ic)) genPointerSet(ic, hasInc(IC_RESULT(ic),
ic, getSize(operandType(IC_RIGHT(ic)))));
                               else genAssign(ic);
                               break;
```

```
                case IFX:
                        genIfx(ic, NULL);
                        break;
                case ADDRESS_OF:
                        genAddrOf(ic);
                        break;
                case JUMPTABLE:
                        genJumpTab(ic);
                        break;
                case CAST:
                        genCast(ic);
                        break;


                case RECEIVE:
                        genReceive(ic);
                        break;
                case SEND:
                        addSet(&_G.sendSet, ic);
                        break;
                case DUMMY_READ_VOLATILE:
                        genDummyRead(ic);
                        break;
                case CRITICAL:
                        genCritical(ic);
                        break;
                case ENDCRITICAL:
                        genEndCritical(ic);
                        break;
                case SWAP:
                        genSwap(ic);
                        break;
                default:
                        ic = ic;
        }
}

if (mulWasGenerated){
        mulWasGenerated = false;
        genZr16Mult();
}

if (divWasGenerated){
        divWasGenerated = false;
        genZr16Div();
}

if (ISRWasGenerated){
```

```
                ISRWasGenerated = false;
                emitcode ("ORG" , "0x3C0");
                emitcode("JMP" , "%s", ISRname);
        }
        genLine.lineElement.ic = NULL;
        if (!options.nopeep){
                peepHole(&genLine.lineHead);
        }
        printLine(genLine.lineHead, codeOutBuf);
        destroy_line_list();
}
```

## Apêndice D – Algoritmo *bubblesort*

```
int A[] = /* preencha o vetor */;
void main () {
        int tmp;
        unsigned char i , j , TAM = /* tamanho do vetor */;

        for(i = TAM-1 ; i >= 1 ; i--) {
        for(j = 0 ; j < i ; j++) {
                        if (A[j] > A[j+1]) {
                                tmp = A[j]; A[j] = A[j+1]; A[j+1] = tmp;
                        }}}}
```

## Apêndice E – Algoritmo *heapsort*

```
void heapsort(unsigned char n);


unsigned char a[] = /* preencha o vetor */;
unsigned char   t , pai , filho ;
void main () {
        heapsort(/*tamanho do vetor*/);
}
void heapsort(unsigned char n){
        unsigned char i =  n/2;
        for (;;) {
        pai = i; filho = 2*i;
        if (i > 0) {
                 i--; t = a[i]; pai = i; filho = 2*i;
        }
        eles {
                n--;
                if (n == 0) break;
                 t = a[n]; a[n] = a[0];
        }
        while (filho < n){
                     if ((unsigned char)(filho + 1) < n  &&  (a[(unsigned char)(filho + 1)]
        > a[filho])) filho++;
```

```
        if (a[filho] > t) a[pai] = a[filho]; pai = filho; filho = 2*pai + 1;
       else break;
       }
        a[pai] = t; }}
```

## Apêndice F – Algoritmo CRC

```
#define POLYNOMIAL 0xD8
unsigned char res;
unsigned char crcNaive(unsigned char message);


void main () {
       res = crcNaive(40);
}
unsigned char crcNaive(unsigned char message) {
    unsigned char  remainder;
    unsigned char bit;
    remainder = message;
    for (bit = 8; bit > 0; --bit){

        if (remainder & 0x80) remainder ^= POLYNOMIAL;
        remainder = (remainder << 1); }
    return (remainder>>4);
}
```

## Apêndice G – *Bubblesort* compilado pelo SDCC com *peephole*

```
main:  MVS    R9 , 0x04
       MOV    R0 , 0x04
       MOV    R9 , R0
L00109:MOV    R0 , R9
       CMP    R0 , 0x01
       JC     L00130
       MVS    R2 , 0x00
L00130:JZ     L00132
       JMP    L00111
L00132:MVS    R8 , 0x00
       MOV    R0 , 0x00
       MOV    R8 , R0
L00106:CMP    R8 , R9
       JC     L00133
       MVS    R2 , 0x00
L00133:JZ L00110
       MOV    R0 , R8
       ADD    R0 , R0
       ADD    R0 , A
       MOV    R7 , R0
       MOV    R5 , R7
       MOV    R0 , R8
       ADD    R0 , 0x01
       ADD     R0 , R0
       ADD     R0 , A
       MOV     R4 , R0
```

```
        MOV    R3 , R4
        MVS    R2 , 0x00
        INC    R3
        INC    R5
        MOV    R0 , (R3)
        AND    R0 , 0x80
        MOV    R1 , R0
        MOV    R0 , (R5)
        AND    R0 , 0x80
        CMP    R1 , R0
        MOV    R0 , R15
        DEC    R3
        DEC    R5
        MOV    R15 , R0
        JC     L00136
        JNZ    L00137
        INC    R3
        INC    R5
        MOV    R0 , (R3)
        XOR    R0 , 0xFF
        ADD    R0 , 0x01
        ADD    R0 , (R5)
        MOV    R1 , R0
        AND    R0 , 0x80
        MOV    R0 , R15
        DEC    R3
        DEC    R5
        MOV    R15 , R0
        JNZ    L00136
        MOV    R0 , 0
        CMP    R1 , R0
        JC     L00136
        JNZ    L00137
        CMP    (R5) , (R3)
        JC     L00136
        JZ     L00136
L00137:MVS    R2 , 0x01
L00136:MOV    R0 , R2
        JZ      L00107
        MOV    R0 , (R5)
        INC    R5
        MOV    R6 , (R5)
        MOV    R5 , R0
        MOV    R0 , R8
        ADD     R0 , 0x01
        ADD     R0 , R0
        ADD     R0 , A
        MOV     R4 , R0
        MOV    R2 , R4
        MOV    (R7) , (R2)
        INC    R7
        INC    R2
        MOV    (R7) , (R2)
        MOV    (R4) , R5
        INC    R4
        MOV    (R4) , R6
L00107:INC    R8
        JMP    L00106
L00110:DEC    R9
        JMP    L00109
L00111:RET
```

## Apêndice H – *Bubblesort* compilado pelo SDCC sem *peephole*

```
main: MVS     R9 , 0x04
       MOV    R0 , 0x04
       MOV    R9 , R0
L00109:MOV    R0 , R9
       CMP    R0 , 0x01
       JC     L00130
L00131:MVS    R2 , 0x00
L00130:JZ     L00132
       JMP    L00111
L00132:MVS    R8 , 0x00
       MOV    R0 , 0x00
       MOV    R8 , R0
L00106:CMP    R8 , R9
       JC     L00133
L00134:MVS    R2 , 0x00
L00133:JNZ    L00135
       JMP    L00110
L00135:MOV    R0 , R8
       ADD    R0 , R0
       MOV    R7 , R0
       MOV    R0 , R7
       ADD    R0 , A
       MOV    R7 , R0
       MOV    R5 , R7
       MOV    R0 , R8
       ADD     R0 , 0x01
       MOV    R4 , R0
       MOV    R0 , R4
       ADD    R0 , R0
       MOV    R4 , R0
       MOV    R0 , R4
       ADD    R0 , A
       MOV    R4 , R0
       MOV    R3 , R4
       MVS    R2 , 0x00
       INC    R3
       INC    R5
       MOV    R0 , (R3)
       AND    R0 , 0x80
       MOV    R1 , R0
       MOV    R0 , (R5)
       AND    R0 , 0x80
       CMP    R1 , R0
       MOV    R0 , R15
       DEC    R3
       DEC    R5
       MOV    R15 , R0
       JC     L00136
       JNZ    L00137
       INC    R3
       INC    R5
       MOV    R0 , (R3)
       XOR    R0 , 0xFF
       ADD    R0 , 0x01
       ADD    R0 , (R5)
       MOV    R1 , R0
       AND    R0 , 0x80
       MOV    R0 , R15
       DEC    R3
       DEC    R5
       MOV    R15 , R0
       JNZ    L00136
       MOV    R0 , 0
       CMP    R1 , R0
       JC     L00136
       JNZ    L00137
       CMP    (R5) , (R3)
```

```
       JC     L00136
       JZ     L00136
L00137:MVS    R2 , 0x01
L00136:MOV    R0 , R2
       JNZ    L00138
       JMP    L00107
L00138:MOV    R0 , (R5)
       INC    R5
       MOV    R6 , (R5)
       MOV    R5 , R0
       MOV    R0 , R8
       ADD    R0 , 0x01
       MOV    R4 , R0
       MOV    R0 , R4
       ADD    R0 , R0
       MOV    R4 , R0
       MOV    R0 , R4
       ADD    R0 , A
       MOV    R4 , R0
       MOV    R2 , R4
       MOV    (R7) , (R2)
       INC    R7
       INC    R2
       MOV    (R7) , (R2)
       MOV    (R4) , R5
       INC    R4
       MOV    (R4) , R6
L00107:INC    R8
JMP    L00106
L00110:DEC    R9
       JMP    L00109
       L00111:RET
```

## Apêndice I – *Bubblesort* compilado pelo ZR16 Compiler

```
main:  mov R0,0x5
                     mov (0x3),R0
for_l10c3:           mov R0,(0x3)
                     mov R2,R0
                     mov R0,0x1
                     sub R2,R0
                     mov R0,R2
                     mov (0x2),R0
loopfor_l10c3:mvs R2,0x0
                     mov R0,(0x2)
                     mov R3,R0
                     mov R0,0x1
                     cmp R3,R0
                     jc 0x24 ;label bool_l10c21_exit
                     jnz 0x23 ;label bool_l10c21_true
bool_l10c21_true:    mvs R2,0x1
bool_l10c21_exit:    djnz R2,0x8D ;label end_for_l10c3
for_l12c7:    mov R0,0x0
                     mov (0x4),R0
loopfor_l12c7:mvs R3,0x0
                     mov R0,(0x4)
                     mov R4,R0
                     mov R0,R4
                     cmp (0x2),R0
                     jc 0x30 ;label bool_l12c21_exit
                     jnz 0x2F ;label bool_l12c21_true
                     jz 0x30 ;label bool_l12c21_exit
bool_l12c21_true:    mvs R3,0x1
bool_l12c21_exit:    djnz R3,0x8B ;label end_for_l12c7
                     mvs R4,0x0
```

```
                    mov R0,(0x4)
                    mov R7,R0
                    mov R0,R7
                    add R0,R7
                    add R0,0x5
                    mov R5,(R0)
                    inc R0
                    mov R6,(R0)
                    mov R0,(0x4)
                    mov R9,R0
                    mov R0,0x1
                    add R9,R0
                    mov R0,R9
                    add R0,R9
                    add R0,0x5
                    mov R7,(R0)
                    inc R0
                    mov R8,(R0)
                    mov R0,R8
                    and R0,0x80
                    mov R1,R0
                    mov R0,R6
                    and R0,0x80
                    cmp R1,R0
                    jc 0x5C ;label bool_l14c13_exit
                    jnz 0x5B ;label bool_l14c13_true
                    mov R0,R8
                    xor R0,0xFF
                    add R0,0x1
                    add R0,R6
                    mov R6,R0
                    and R0,0x80
                    jnz 0x5C ;label bool_l14c13_exit
                    mov R0,0x0
                    cmp R6,R0
                    jc 0x5C ;label bool_l14c13_exit
                    jnz 0x5B ;label bool_l14c13_true
                    cmp R5,R7
                    jc 0x5C ;label bool_l14c13_exit
                    jnz 0x5B ;label bool_l14c13_true
                    jz 0x5C ;label bool_l14c13_exit
bool_l14c13_true:   mvs R4,0x1
bool_l14c13_exit:   djnz R4,0x89 ;label end_if_l14c4
                    mov R0,(0x4)
                    mov R7,R0
                    mov R0,R7
                    add R0,R7
                    add R0,0x5
                    mov R5,(R0)
                    inc R0
                    mov R6,(R0)
                    mov R0,R5
                    mov (0xF),R0
                    mov R0,R6
                    mov (0x10),R0
                    mov R0,(0x4)
                    mov R7,R0
                    mov R0,0x1
                    add R7,R0
                    mov R0,R7
                    add R0,R7
                    add R0,0x5
                    mov R5,(R0)
                    inc R0
                    mov R6,(R0)
                    mov R0,(0x4)
                    mov R7,R0
                    mov R0,R7
                    add R0,R7
                    add R0,0x5
```

```
                    mov (R0),R5
                    inc R0
                    mov (R0),R6
                    mov R0,(0xF)
                    mov R5,R0
                    mov R0,(0x10)
                    mov R6,R0
                    mov R0,(0x4)
                    mov R7,R0
                    mov R0,0x1
                    add R7,R0
                    mov R0,R7
                    add R0,R7
                    add R0,0x5
                    mov (R0),R5
                    inc R0
                    mov (R0),R6
end_if_l14c4: inc (0x4)
                    jmp 0x27 ;label loopfor_l12c7
end_for_l12c7:dec (0x2)
                    jmp 0x1C ;label loopfor_l10c3
end_for_l10c3:jmp 0x0
```

## Apêndice J – *Heapsort* compilado pelo SDCC com *peephole*

```
       main:  MVS    R8 , 0x05
       CALL   heapsort
       RET
heapsort:     MOV    R0 , R8
       MOV    R9 , R0
       MOV    R0 , 0b11111101
       AND    R15 , R0
       MOV    R0 , R9
       SHL    R0 , R0
       MOV    R8 , R0
       MOV    R0 , (R9)
L00116:MOV    R0 , R8
       MOV    (pai_L) , R0
       ADD    R0 , R0
       MOV    (filho_L) , R0
       MOV    R0 , R8
       JZ L00104
       DEC    R8
       MOV    R0 , R8
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R0 , (R7)
       MOV    (t_L) , R0
       MOV    R0 , R8
       MOV    (pai_L) , R0
       ADD    R0 , R0
       MOV    (filho_L) , R0
       JMP    L00112
L00104:DEC    R9
       MOV    R0 , R9
       JZ L00118
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R0 , (R7)
       MOV    (t_L) , R0
       MOV    R0 , (a0)
       MOV (R7) , R0
L00112:MOV    R0 , (filho_L)
       CMP    R0 , R9
       JC     L00147
```

```
       MVS    R2 , 0x00
L00147:JZ L00114
       MOV    R0 , (filho_L)
       ADD R0 , 0x01
       MOV    R7 , R0
       CMP    R7 , R9
       JC     L00150
       MVS    R2 , 0x00
L00150:JZ L00107
       MOV    R0 , (filho_L)
       ADD    R0 , a
       MOV    R6 , R0
       ADD  R0 , 0x01
       MOV    R7 , R0
       CMP    (R6) , (R7)
       JC     L00153
       MVS    R2 , 0x00
L00153:JZ     L00107
       INC    (filho_L)
L00107:MOV    R0 , (filho_L)
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R0 , (t_L)
       CMP    R0 , (R7)
       JC     L00156
       MVS    R2 , 0x00
L00156:JZ      L00114
       MOV    R0 , (pai_L)
       ADD    R0 , a
       MOV    (R0) , (R7)
       MOV    R0 , (filho_L)
       MOV    (pai_L) , R0
       ADD    R0 , R0
       MOV    R7 , R0
       ADD    R0 , 0x01
       MOV    (filho_L) , R0
       JMP    L00112
L00114:MOV    R0 , (pai_L)
       ADD    R0 , a
       MOV    R2 , R0
       MOV    R0 , (t_L)
       MOV    (R2) , R0
       JMP    L00116
L00118:RET
```

## Apêndice K – *Heapsort* compilado pelo SDCC sem *peephole*

```
main:  MVS    R8 , 0x05
       CALL   heapsort
L00101:       RET
heapsort:     MOV    R0 , R8
       MOV    R9 , R0
       MOV    R0 , 0b11111101
       AND    R15 , R0
       MOV    R0 , R9
       SHL    R0 , R0
       MOV    R8 , R0
       MOV    R0 , (R9)
L00116:MOV    R0 , R8
       MOV    (pai_L) , R0
       MOV    R0 , R8
       ADD    R0 , R0
       MOV    (filho_L) , R0
       MOV    R0 , R8
       JNZ    L00145
```

```
       JMP    L00104
L00145:DEC    R8
       MOV    R0 , R8
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R0 , (R7)
       MOV    (t_L) , R0
       MOV    R0 , R8
       MOV    (pai_L) , R0
       MOV    R0 , R8
       ADD    R0 , R0
       MOV    (filho_L) , R0
       JMP    L00112
L00104:DEC    R9
       MOV    R0 , R9
       JNZ    L00146
       JMP    L00118
L00146:MOV    R0 , R9
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R0 , (R7)
       MOV    (t_L) , R0
       MOV    R0 , (a0)
       MOV    R6 , R0
       MOV    (R7) , R6
L00112:MOV    R0 , (filho_L)
       CMP    R0 , R9
       JC     L00147
L00148:MVS    R2 , 0x00
L00147:JNZ    L00149
       JMP    L00114
L00149:MOV    R0 , (filho_L)
       ADD    R0 , 0x01
       MOV    R7 , R0
       CMP    R7 , R9
       JC     L00150
L00151:MVS    R2 , 0x00
L00150:JNZ    L00152
       JMP    L00107
L00152:MOV    R0 , (filho_L)
       ADD     R0 , 0x01
       MOV    R0 , R0
       MOV    R0 , R0
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R7 , R7
       MOV    R0 , (filho_L)
       ADD    R0 , a
       MOV    R6 , R0
       MOV    R6 , R6
       CMP    (R6) , (R7)
       JC     L00153
L00154:MVS    R2 , 0x00
L00153:JNZ    L00155
       JMP    L00107
L00155:INC    (filho_L)
L00107:MOV    R0 , (filho_L)
       ADD    R0 , a
       MOV    R7 , R0
       MOV    R7 , R7
       MOV    R0 , (t_L)
       CMP    R0 , (R7)
       JC     L00156
L00157:MVS    R2 , 0x00
L00156:JNZ    L00158
       JMP    L00114
L00158:MOV    R0 , (pai_L)
       ADD    R0 , a
       MOV    R0 , R0
       MOV    (R0) , (R7)
```

```
       MOV    R0 , (filho_L)
       MOV    (pai_L) , R0
       MOV    R0 , (pai_L)
       ADD    R0 , R0
       MOV    R7 , R0
       MOV    R0 , R7
       ADD    R0 , 0x01
       MOV    (filho_L) , R0
       JMP    L00112
L00114:MOV    R0 , (pai_L)
       ADD    R0 , a
       MOV    R0 , R0
       MOV    R2 , R0
       MOV    R0 , (t_L)
       MOV    (R2) , R0
       JMP    L00116
L00118:RET
```

## Apêndice L – *Heapsort* compilado pelo ZR16 Compiler

```
                    main:  mov R0,0x19
                    mov (0x3),R0
                    call 0x36 ;label funcl18c6heapsort
                    jmp 0x0
funcl18c6heapsort:  mov R0,(0x3)
                    mov R2,R0
                    mov R0,R2
                    mov (0x20),R0
                    mov R0,0x0
                    mov (0x21),R0
                    mvs R0,0x0
                    mov (0x23),R0
                    call 0xF2 ;label __m8_unsigned
                    mov R0,(0x22)
                    mov R2,R0
                    mov R0,R2
                    mov (0x2),R0
for_l23c3:          mov R0,(0x2)
                    mov R2,R0
                    mov R0,R2
                    mov (0x4),R0
                    mov R0,0x2
                    mov R2,R0
                    mov R0,R2
                    mov (0x20),R0
                    mov R0,(0x2)
                    mov (0x21),R0
                    mvs R0,0x0
                    mov (0x23),R0
                    call 0xF2 ;label __m8_unsigned
                    mov R0,(0x22)
                    mov R2,R0
                    mov R0,R2
                    mov (0x5),R0
                    mvs R2,0x0
                    mov R0,(0x2)
                    mov R3,R0
                    mov R0,0x0
                    cmp R3,R0
                    jc 0x5D ;label bool_l29c9_exit
                    jnz 0x5C ;label bool_l29c9_true
                    jz 0x5D ;label bool_l29c9_exit
bool_l29c9_true:    mvs R2,0x1
bool_l29c9_exit:    djnz R2,0x78 ;label else_l37c8
```

```
                    dec (0x2)
                    mov R0,(0x2)
                    mov R4,R0
                    mov R0,R4
                    add R0,0x6
                    mov R3,(R0)
                    mov R0,R3
                    mov (0x1F),R0
                    mov R0,(0x2)
                    mov R3,R0
                    mov R0,R3
                    mov (0x4),R0
                    mov R0,0x2
                    mov R3,R0
                    mov R0,R3
                    mov (0x20),R0
                    mov R0,(0x2)
                    mov (0x21),R0
                    mvs R0,0x0
                    mov (0x23),R0
                    call 0xF2 ;label __m8_unsigned
                    mov R0,(0x22)
                    mov R3,R0
                    mov R0,R3
                    mov (0x5),R0
                    jmp 0x90 ;label end_if_l29c3
else_l37c8:         dec (0x3)
                    mvs R2,0x0
                    mov R0,(0x3)
                    mov R3,R0
                    mov R0,0x0
                    cmp R3,R0
                    jnz 0x80 ;label bool_l40c10_exit
bool_l40c10_true:   mvs R2,0x1
bool_l40c10_exit:   djnz R2,0x82 ;label end_if_l40c4
                    jmp 0xF1 ;label end_for_l23c3
end_if_l40c4: mov R0,(0x3)
                    mov R3,R0
                    mov R0,R3
                    add R0,0x6
                    mov R2,(R0)
                    mov R0,R2
                    mov (0x1F),R0
                    mov R0,(0x6)
                    mov R2,R0
                    mov R0,(0x3)
                    mov R3,R0
                    mov R0,R3
                    add R0,0x6
                    mov (R0),R2
end_if_l29c3:       mvs R2,0x0
                    mov R0,(0x5)
                    mov R3,R0
                    mov R0,R3
                    cmp (0x3),R0
                    jc 0x99 ;label bool_l47c21_exit
                    jnz 0x98 ;label bool_l47c21_true
                    jz 0x99 ;label bool_l47c21_exit
bool_l47c21_true:   mvs R2,0x1
bool_l47c21_exit:   djnz R2,0xE9 ;label end_while_l47c8
                    mvs R3,0x0
                    mov R0,(0x5)
                    mov R5,R0
                    mov R0,0x1
                    add R5,R0
                    mov R4,R5
                    mov R0,R4
                    cmp (0x3),R0
                    jc 0xA6 ;label bool_l49c39_exit
                    jnz 0xA5 ;label bool_l49c39_true
```

```
                    jz 0xA6 ;label bool_l49c39_exit
bool_l49c39_true:   mvs R3,0x1
bool_l49c39_exit:   mvs R4,0x0
                    mov R0,(0x5)
                    mov R8,R0
                    mov R0,0x1
                    add R8,R0
                    mov R7,R8
                    mov R0,R7
                    add R0,0x6
                    mov R6,(R0)
                    mov R0,(0x5)
                    mov R8,R0
                    mov R0,R8
                    add R0,0x6
                    mov R7,(R0)
                    cmp R6,R7
                    jc 0xB9 ;label bool_l49c79_exit
                    jnz 0xB8 ;label bool_l49c79_true
                    jz 0xB9 ;label bool_l49c79_exit
bool_l49c79_true:   mvs R4,0x1
bool_l49c79_exit:   and R3,R4
                    djnz R3,0xBC ;label end_if_l49c8
                    inc (0x5)
end_if_l49c8:       mvs R3,0x0
                    mov R0,(0x5)
                    mov R6,R0
                    mov R0,R6
                    add R0,0x6
                    mov R4,(R0)
                    mov R0,(0x1F)
                    cmp R4,R0
                    jc 0xC8 ;label bool_l54c21_exit
                    jnz 0xC7 ;label bool_l54c21_true
                    jz 0xC8 ;label bool_l54c21_exit
bool_l54c21_true:   mvs R3,0x1
bool_l54c21_exit:   djnz R3,0xE7 ;label else_l61c4
                    mov R0,(0x5)
                    mov R6,R0
                    mov R0,R6
                    add R0,0x6
                    mov R4,(R0)
                    mov R0,(0x4)
                    mov R6,R0
                    mov R0,R6
                    add R0,0x6
                    mov (R0),R4
                    mov R0,(0x5)
                    mov R4,R0
                    mov R0,R4
                    mov (0x4),R0
                    mov R0,0x2
                    mov R4,R0
                    mov R0,R4
                    mov (0x20),R0
                    mov R0,(0x4)
                    mov (0x21),R0
                    mvs R0,0x0
                    mov (0x23),R0
                    call 0xF2 ;label __m8_unsigned
                    mov R0,(0x22)
                    mov R4,R0
                    mov R0,0x1
                    add R4,R0
                    mov R0,R4
                    mov (0x5),R0
                    jmp 0xE8 ;label end_if_l54c8
else_l61c4:         jmp 0xE9 ;label end_while_l47c8
end_if_l54c8:       jmp 0x90 ;label end_if_l29c3
end_while_l47c8:    mov R0,(0x1F)
```

```
                    mov R2,R0
                    mov R0,(0x4)
                    mov R3,R0
                    mov R0,R3
                    add R0,0x6
                    mov (R0),R2
                    jmp 0x43 ;label for_l23c3
end_for_l23c3:      ret
__m8_unsigned:      mvs R0,0x0
                    mov (0x24),R0
__m8:               mvs R0,0x0
                    cmp (0x20),R0
                    jz 0x10D ;label __m8_end
                    mov R0,(0x20)
                    mov R1,R0
                    mvs R0,0x0
__m8_loop:          jc 0x10E ;label overflow
                    add R0,(0x21)
                    djnz R1,0xFA ;label __m8_loop
                    mov (0x22),R0
                    mvs R0,0x0
                    cmp (0x24),R0
                    jz 0x105 ;label __m8_checkresneg
                    mov R0,0x80
                    and R0,(0x22)
                    cmp R0,0x0
                    jnz 0x10E ;label overflow
__m8_checkresneg:   mvs R0,0x0
                    cmp (0x23),R0
                    jz 0x10D ;label __m8_end
                    cmp (0x22),R0
                    jz 0x10D ;label __m8_end
                    mov R0,0xFF
                    xor (0x22),R0
                    inc (0x22)
__m8_end:     ret
```

## Apêndice M – CRC compilado pelo SDCC com *peephole*

```
       main:  MVS    R8 , 0x28
       CALL   crcNaive
       MOV    R0 , R8
       MOV    (res_L) , R0
       RET
crcNaive:MOV  R0 , R8
       MOV    R9 , R0
       MVS    R8 , 0x08
       MOV    R0 , 0x08
       MOV    R8 , R0
L00104:MOV    R0 , 0x80
       MOV    R1 , R9
       AND R1 , R0
       JZ L00102
       MOV    R0 , R9
       XOR R0 , 0xD8
       MOV    R9 , R0
L00102:MOV    R0 , R9
       ADD    R0 , R0
       MOV    R9 , R0
       DEC    R8
       JZ     L00122
       JMP    L00104
L00122:MOV    R0 , 0b11111101
       AND    R15 , R0
       MOV    R0 , R9
```

```
       SHL    R0 , R0
       SHL    R0 , R0
       SHL    R0 , R0
       SHL    R0 , R0
       MOV    R8 , R0
       MVS    R9 , 0x00
       RET
```

## Apêndice N – CRC compilado pelo SDCC sem *peephole*

```
main:  MVS    R8 , 0x28
       CALL   crcNaive
       MOV    R0 , R8
       MOV    (res_L) , R0
L00101:RET
crcNaive:MOV  R0 , R8
       MOV    R9 , R0
       MVS    R8 , 0x08
       MOV    R0 , 0x08
       MOV    R8 , R0
L00104:MOV    R0 , 0x80
       MOV    R1 , R9
       AND R1 , R0
       JNZ    L00120
       JMP    L00102
L00120:MOV    R0 , R9
       XOR R0 , 0xD8
       MOV    R9 , R0
L00102:MOV    R0 , R9
       ADD    R0 , R0
       MOV    R9 , R0
       DEC    R8
       JZ     L00122
L00121:JMP    L00104
L00122:MOV    R0 , 0b11111101
       AND    R15 , R0
       MOV    R0 , R9
       SHL    R0 , R0
       SHL    R0 , R0
       SHL    R0 , R0
       SHL    R0 , R0
       MOV    R8 , R0
       MOV    R8 , R8
       MVS    R9 , 0x00
       RET
L00106:RET
```

## Apêndice O – CRC compilado pelo ZR16 Compiler

```
main:    mov R0,0x28

                        mov (0x4),R0
                        call 0x8 ;label funcl13c15crcnaive
                        mov R0,(0x0)
                        mov R2,R0
                        mov R0,R2
                        mov (0x2),R0
                        jmp 0x0
funcl13c15crcnaive:     mov R0,(0x4)
```

```
                        mov R3,R0
                        mov R0,R3
                        mov (0x3),R0
for_l21c5:              mov R0,0x8
                        mov (0x5),R0
loopfor_l21c5:          mvs R3,0x0
                        mov R0,(0x5)
                        mov R4,R0
                        mov R0,0x0
                        cmp R4,R0
                        jc 0x17 ;label bool_l21c23_exit
                        jnz 0x16 ;label bool_l21c23_true
                        jz 0x17 ;label bool_l21c23_exit
bool_l21c23_true:       mvs R3,0x1
bool_l21c23_exit:       djnz R3,0x38 ;label end_for_l21c5
                        mov R0,(0x3)
                        mov R4,R0
                        mov R0,0x80
                        and R4,R0
                        mvs R5,0x1
                        mov R6,R4
                        mov R0,0x0
                        cmp R6,R0
                        jnz 0x22 ;label bool_l24c12_exit
bool_l24c12_true:       mvs R5,0x0
bool_l24c12_exit:       mov R0,R5
                        cmp R0,0x0
                        jz 0x2B ;label end_if_l24c9
                        mov R0,(0x3)
                        mov R6,R0
                        mov R0,0xD8
                        xor R6,R0
                        mov R0,R6
                        mov (0x3),R0
end_if_l24c9:    mov R0,(0x3)
                        mov R4,R0
                        mov R0,R4
                        mov (0x6),R0
                        mov R0,0x1
                        mov (0x7),R0
                        call 0x44 ;label __shl8
                        mov R0,(0x8)
                        mov R4,R0
                        mov R0,R4
                        mov (0x3),R0
                        dec (0x5)
                        jmp 0xE ;label loopfor_l21c5
end_for_l21c5:  mov R0,(0x3)
                        mov R3,R0
                        mov R0,R3
                        mov (0x6),R0
                        mov R0,0x4
                        mov (0x7),R0
                        call 0x58 ;label __shr8_unsigned
                        mov R0,(0x8)
                        mov R3,R0
                        mov R0,R3
                        mov (0x0),R0
                        ret
__shl8:                 mvs R0,0x0
                        mov (0x8),R0
```

```
                        cmp (0x6),R0
                        jz 0x53 ;label __shl8_end
                        mov R0,(0x7)
                        cmp R0,0x0
                        jz 0x52 ;label __shl8_keepvalue
                        mov R1,R0
                        mov R0,0x2
                        or R15,R0
                        mov R0,(0x6)
__shl8_loop:    shl R0,R0
                        djnz R1,0x4F ;label __shl8_loop
                        jmp 0x53 ;label __shl8_end
__shl8_keepvalue:       mov R0,(0x6)
__shl8_end:             mov (0x8),R0
                        ret
__shr8_signed:          mvs R0,0x1
                        mov (0xA),R0
                        jmp 0x5A ;label __shr8_init
__shr8_unsigned:        mvs R0,0x0
                        mov (0xA),R0
__shr8_init:            mvs R0,0x0
                        mov (0x8),R0
                        cmp (0x6),R0
                        jz 0x74 ;label __shr8_end
                        mov R0,(0x7)
                        cmp R0,0x0
                        jz 0x73 ;label __shr8_keepvalue
                        mov R1,R0
                        mov R0,0xFD
                        and R15,R0
                        mov R0,(0xA)
                        cmp R0,0x0
                        jz 0x6F ;label __shr8_startunsigned
                        mov R0,0x80
                        and R0,(0x6)
                        cmp R0,0x0
                        jz 0x6F ;label __shr8_startunsigned
                        mov R0,(0x6)
__shr8_loop_signed:     sha R0,R0
                        djnz R1,0x6C ;label __shr8_loop_signed
                        jmp 0x74 ;label __shr8_end
__shr8_startunsigned:   mov R0,(0x6)
__shr8_loop_unsigned: shl R0,R0
                        djnz R1,0x70 ;label __shr8_loop_unsigned
                        jmp 0x74 ;label __shr8_end
__shr8_keepvalue:       mov R0,(0x6)
__shr8_end:     mov (0x8),R0
                        ret
```