Programação com linguagem máquina

GNU Assembler – as

Evocação

Evocação na linha de comando do GNU *assembler*.
as [-a[=file]] [-defsym sym=val] [--gstabs] [-I dir] [-o objfile] srcfile

-a	Gerar listagem.
=file	Incluir a listagem no ficheiro indicado.
-defsym	Definir símbolo.
--gstabs	Incluir informação para debugging no ficheiro de saída.
-I	Definir directoria de pesquisa para ficheiros de include.
-o	Definir o nome do ficheiro de saída.
srcfile	O ficheiro com a fonte do programa.

Sintaxe

Um programa é composto por uma sequência de expressões na forma:

```
<label> <instruction> <comment>
```

label	Símbolo seguido do carcter **':'**. Define o endereço do elemento seguinte – instrução ou variável. Pode ser composto por letras, dígitos e os caracteres **'.'** (ponto) e **'_'** (sublinhado). Os símbolos locais são definidos por um digito na forma N: com N de 0 a 9 e referidos por Nb ou Nf.
instruction	Qualquer instrução ARM, pseudo-instruções ou directivas.
comment	Os caracteres **@** ou **;** indicam comentário até ao fim da linha. O carácter # indica comentário até ao fim da linha se for o primeiro da linha. Podem também ser usados comentários como em C com **/*** no início e ***/** no fim.

Expressões

Uma expressão produz um valor numérico, operando valores numéricos ou símbolos com valor numérico.

Uma expressão pode aparecer em qualquer instrução onde se espere uma constante. Na sintaxe ARM as constantes são prefixadas por #.

Os valores numéricos podem ser representados em decimal, hexadecimal (0xddd), octal (0ddd), binário (0bdddd), caracteres entre **' '** (plicas) . Exemplos: 34, 0x3f, 034, 0b0101, 'K'

Operadores unários

-	Negação em complemento para 2.
~	Negação bit a bit.

Operadores binários

*	Multiplicação	|	Disjunção bit a bit	+	adição
/	Divisão	&	Conjunção bit a bit	-	subtracção
%	Resto da divisão	^	Disjunção exclusiva bit a bit		
<<	Deslocar para a esquerda	>>	Deslocar para a direita		

Directivas

.align	Insere bytes a zero até um endereço múltiplo de 4.
.ascii "string"	Insere os caracteres que compõem a string.
.asciz "string"	Insere os caracteres que compõem a string com terminação a zero.
.byte expression	Insere o valor especificado representado a 8 bits (1 byte).
.2byte expression	Insere o valor especificado representado a 16 bits (2bytes).
.hword expression	
.4byte expression	Insere o valor especificado representado a 32 bits (2bytes).
.long expression	
.word expression	
.int expression	
.space size, fill	Insere size bytes com o valor fill
.skip size, fill	
.text	Passa a inserir na secção assinalada. A secção .text é para
.data	o código das instruções. A secção .data para as variáveis iniciadas.
.bss	A secção .bss para as variáveis não iniciadas e a secção .rodata
.rodata	para os dados constantes
.global symbol	Declara symbol visível para os outros módulos.
.extern symbol	Declara que symbol é declarado noutro módulo.
.include "file"	Insere o conteúdo de file na posição desta directiva.
.end	O compilador termina a compilação quando encontra esta directiva
.if expression	Para compilação condicional.
.else if expression	
.else	
.endif	
.equ symbol, expression	Define expression como o valor de symbol.
.set symbol, expression	
.err	Imprime uma mensagem de erro e termina a compilação.
.macro	Definição de macro.
.endm	

Referências

Using as, The gnu Assembler, Version 2.14

Módulos

A divisão do texto fonte dos programas por múltiplos ficheiros é uma prática necessária que facilita o desenvolvimento, a reutilização e a manutenção. A um ficheiro contendo uma parte do programa chama-se um módulo.

A interacção entre as partes do programa dispersas em módulos é feita através de variáveis e de chamadas a funções. As referências, para variáveis ou para funções, são feitas através de símbolos. O símbolo representa o endereço do elemento referenciado – da variável ou da função.

Para que um símbolo definido num módulo seja visível noutro é necessário que este seja globalmente visível. Na linguagem Assembly GNU, por omissão, um símbolo é visível apenas no módulo onde é declarado. Para o tornar globalmente visível é necessário explicitar com a directiva **.global** como no seguinte exemplo:

```
.global    main
main:
```

Um símbolo global precisa ser conhecido no módulo onde é referenciado. O **as** assume que um símbolo referenciado e não declarado no módulo actual é global e está definido noutro módulo.

Exemplo de programa dividido em dois módulos (ou ficheiros) **main.s** e **memcopy.s**.

```
.equ        SIZE, 10
.data
block1:     .space SIZE, 1
block2:     .space SIZE, 2

.text
.global _start
_start:
      ldr   r0, =block1
      ldr   r0, [r0]
      ldr   r1, =block2
      ldr   r1, [r1]
      mov   r2, =SIZE
      bl    memcopy
      b     .
```

main.s

```
.text
.global memcopy
memcopy:
      ldrb  r3, [r0], #1
      strb  r3, [r1], #1
      subs  r2, r2, #1
      bne   memcopy
      mov   pc, lr
```

memcopy.s

Compilação dos módulos em separado:

```
$ arm-elf-as main.s -o main.o
$ arm-elf-as memcopy.s -o memcopy.o
```

Observação do resultado da compilação para o módulo **main**:

```
$ arm-elf-objdump -D main.o
```

```
main.o:     file format elf32-littlearm
Disassembly of section .text:

00000000 <_start>:
   0: e59f0014    ldr   r0, [pc, #20]     ; 1c <.text+0x1c>
```

```
   4: e5900000    ldr   r0, [r0]
   8: e59f1010    ldr   r1, [pc, #16]     ; 20 <.text+0x20>
   c: e5911000    ldr   r1, [r1]
  10: e3a02010    mov   r2, #16     ; 0x10
  14: ebfffffe    bl    0 <memcopy>
  18: eafffffe    b     18 <_start+0x18>
  1c: 00000000    andeq r0, r0, r0
  20: 00000010    andeq r0, r0, r0, lsl r0
Disassembly of section .data:

00000000 <block1>:
   0: 01010101    tsteq r1, r1, lsl #2
   4: 01010101    tsteq r1, r1, lsl #2
   8: 01010101    tsteq r1, r1, lsl #2
   c: 01010101    tsteq r1, r1, lsl #2

00000010 <block2>:
  10: 02020202    andeq r0, r2, #536870912      ; 0x20000000
  14: 02020202    andeq r0, r2, #536870912      ; 0x20000000
  18: 02020202    andeq r0, r2, #536870912      ; 0x20000000
  1c: 02020202    andeq r0, r2, #536870912      ; 0x20000000
```

Os simbolos definidos neste módulo são **_start**, **block1** e **block2**. Dentro da respectiva secção estes simbolos são numerados a partir de zero. Ao simbolo externo **memcopy** é atribuido o valor zero.

Na fase de localização são atribuidos endereços absolutos às secções. O endereço final de cada simbolo é calculado somando, o endereço base da secção a que pertence, mais a sua posição relativa ao início da secção. O endereço base da secção depende da dimensão ocupada por outros módulos nessa secção.

A ligação consiste em resolver as referências para simbolos definidos noutro módulo ou noutra secção. Estes simbolos são indicados na tabela de relocalizações do módulo.

```
$arm-elf-objdump -r main.o
```

```
main.o:     file format elf32-littlearm

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000014 R_ARM_CALL        memcopy
0000001c R_ARM_ABS32       .data
00000020 R_ARM_ABS32       .data
```

No caso de **main** há três referencias por resolver: **memcopy**, **block1** e **block2**. Na fase de compilação só é possível resolver referências para simbolos na mesma secção com referência relativa.

A referência a **memcopy** na intrução **bl** é relativa à posição do *program counter*, não pode ser resolvida porque o simbolo é definido noutro módulo e não se conhece, neste altura, a sua posição. Já a referência ao simbolo **memcopy,** feita no próprio módulo **memcopy**, na instrução **bne,** é resolvida na fase de compilação.

A referência absoluta a **block1** e **block2** resulta da utilização das pseudo-instruções **ldr r0,=block1** e **ldr r1, block2** que usam duas *words* para guardarem, respectivamente, os endereços **block1** e **block2** (veja-se as posições 0x1c e 0x20 do programa). A referência a estas *words* é relativa e feita pelas instruções **ldr r0, [pc, #20]** e **ldr r1, [pc, #16]**.

A localização e ligação é feita pelo programa **ld**.

```
$ arm-elf-ld -T ldscript main.o memcopy.o -o prog.elf
```

A indicação dos endereços a atribuir a cada secção é feita através do ficheiro ldscript que se apresenta de seguida:

```
ENTRY(_start)
MEMORY
{
    ram : ORIGIN = 0x40000000, LENGTH = 0x10000
}
 SECTIONS
{
    .text : {
      __text_start__ = ABSOLUTE(.);
      *(.text*);
      __text_end__ = ABSOLUTE(.);
     } > ram

    .data ALIGN(4) : {
      __data_start = ABSOLUTE(.);
      *(.data*);
      __data_end__ = ABSOLUTE(.) ;
      } > ram
}
```

Neste caso foi definida uma zona de memória, a que se chamou **ram**, com endereço inicial em **0x40000000** e 64 Kbyte de dimensão. Se se pretendesse gravar o programa em FLASH então ter-se-ía que definir também a zona de memória correspondente à FLASH.

Nesta fase existem dois conceitos de secção: a secção de entrada e a secção de saída. O **ld** cria secções de saída e concatena nessas secções, as secções de entrada provenientes de vários módulos.

Por exemplo: a secção de saída **.text**

```
.text : {
```

é formada pelos contributos de todos os módulos com as seguintes secções de entrada:

```
      *(.text*);
```

A secção de saída resultante é alojada na zona de memória **ram**.

```
} > ram
```

O seu endereço final depende das secções que foram atribuidas anteriormente a esta zona de memória.

Em cada secção de saída são criadas duas labels que definem o seu endereço de início e o seu endereço de fim, respectivamente. Por xemplo:

```
      __data_start__ = ABSOLUTE(.);
      __data_end__ = ABSOLUTE(.);
```

Observe-se agora o resultado da ligação e localização.

```
$ arm-elf-objdump -D main.elf
```

```
main.elf:     file format elf32-littlearm

Disassembly of section .text:

40000000 <_start>:
40000000:   e59f0014    ldr   r0, [pc, #20]     ; 4000001c <.text+0x1c>
40000004:   e5900000    ldr   r0, [r0]
40000008:   e59f1010    ldr   r1, [pc, #16]     ; 40000020 <.text+0x20>
```

```
4000000c:   e5911000    ldr   r1, [r1]
40000010:   e3a02010    mov   r2, #16     ; 0x10
40000014:   eb000002    bl    40000024 <memcopy>
40000018:   eafffffe    b     40000018 <__text_start__+0x18>
4000001c:   40000038    andmi r0, r0, r8, lsr r0
40000020:   40000048    andmi r0, r0, r8, asr #32

40000024 <memcopy>:
40000024:   e4d03001    ldrb  r3, [r0], #1
40000028:   e4c13001    strb  r3, [r1], #1
4000002c:   e2522001    subs  r2, r2, #1  ; 0x1
40000030:   1afffffb    bne   40000024 <memcopy>
40000034:   e1a0f00e    mov   pc, lr
Disassembly of section .data:

40000038 <block1>:
40000038:   01010101    tsteq r1, r1, lsl #2
4000003c:   01010101    tsteq r1, r1, lsl #2
40000040:   01010101    tsteq r1, r1, lsl #2
40000044:   01010101    tsteq r1, r1, lsl #2

40000048 <block2>:
40000048:   02020202    andeq r0, r2, #536870912      ; 0x20000000
4000004c:   02020202    andeq r0, r2, #536870912      ; 0x20000000
40000050:   02020202    andeq r0, r2, #536870912      ; 0x20000000
40000054:   02020202    andeq r0, r2, #536870912      ; 0x20000000
```

A secção **.text** engloba a contribuição do módulo main e do módulo memcopy. A sua dimensão é a soma da secção de entrada **.text** do módulo **main** e da secção de entrada **.text** do módulo **memcopy**. O seu endereço é **0x40000000** porque coincide com o início da memória.

A secção **.data** começa no endereço disponível a seguir à secção **.text** - endereço **0x40000038**.

A instrução **bl memcopy** tem codificado um deslocamento de duas words, relativo à posição do *program counter,* o que dá o endereço **0x40000024** (0x40000014 + 8 + 2 * 4) que corresponde ao endereço atribuido ao simbolo **memcopy**.

As words auxiliares, endereços **0x4000001c** e **0x40000020,** criadas pelas pseudo-instruções **ldr r0, =block1** e **ldr r1,=block2,** contêm os valores **0x40000038** e **0x40000048** que correspondem, respectivamente, aos endereços dos simbolos **block1** e **block2**.

Programação com linguagem C

O compilador GNU não produz, por omissão, modificações nos nomes dos simbolos. Assim as referências cruzadas entre módulos escritos em *assembly* ou em C usam o mesmo identificador.

Um programa em C é repartido por um conjunto de secções. O objectivo é poder manipular as secções em separado de acordo com a natureza dos elementos do programa que a integram. As secções normalmente produzidas são as seguintes:

.text para o código;

.data para as variáveis com valor inicial definido;

.bss para as variáveis não iniciadas;

.rodata para as constantes;

.stack para a pilha do programa.

Para que o programa se possa executar é necessário que estas secções sejam devidamente alojadas em memória e sejam devidamente iniciadas. Esta preparação prévia para a execução é muito variável. Por exemplo: a execução em RAM, com a ajuda de um *debugger*; a execução em RAM, estando o programa alojado em memória não volátil (ROM ou disco); execução directamente em ROM. Para cada um dos casos apresentados, e outros, é necessário criar um ambiente de execução compatível com a linguagem C: as variáveis são alojadas em RAM, há variáveis iniciadas e variáveis não iniciadas, existe um *stack*, o código não é modificável, etc.

É apresentado de seguida a preparação para a execução em RAM com a ajuda de um *debugger*. Nesta situação uma parte da preparação é efectuada pelo *debugger,* ao carregar o conteúdo das secções **.text**, **.dat**a e **.rodata** na memória RAM, e a outra parte é efectuada pelo próprio programa no módulo de arranque designado por **cstart** e escrito em *assembly*. Neste caso o **cstart** tem que providenciar a iniciação do *stack* e colocar a zero a secção **.bss**, correspondente às variáveis não iniciadas.

```
                            .text
   2                        .global     _start
   3                    _start:
   4 0000 30109FE5              ldr         r1, =__bss_start__
   5 0004 30209FE5              ldr         r2, =__bss_end__
   6 0008 0000A0E3              mov         r0, #0
   7 000c 020051E1              cmp         r1, r2
   8 0010 0600002A              bhs         2f
   9                    1:
  10 0014 040081E4              str         r0, [r1], #4
  11 0018 020051E1              cmp         r1, r2
  12 001c 0300003A              blo         1b
  13                    2:
  14 0020 18D09FE5              ldr         sp, =__stack_end__
  15 0024 00B0A0E3              mov         fp, #0
  16 0028 0000A0E3              mov         r0, #0            ;     argc = 0
  17 002c 0010A0E3              mov         r1, #0            ;     argv = 0
  18 0030 FEFFFFEB              bl          main
  19 0034 0B0000EB              bl          .
```

Para a localização e ligação de módulos pode ser utilizado o seguinte **ldscript**:

```
ENTRY(_start)

MEMORY
{
      ram : ORIGIN = 0x40000000, LENGTH = 0x10000
}

SECTIONS
{
      .text : {
            __text_start__ = ABSOLUTE(.);
            *(.text*) *(.gnu.warning) *(.gnu.linkonce*)
            *(.init) *(.glue_7) *(.glue_7t);
            __text_end__ = ABSOLUTE(.);
      } > ram

      .rodata ALIGN(4) : {
            __rodata_start__ = ABSOLUTE(.);
            *(.rodata*);
            . = ALIGN(4);
```

```
            __rodata_end__ = ABSOLUTE(.);
      } > ram

      .data ALIGN(4) : {
            __data_start = ABSOLUTE(.);
            *(.data*);
            *(.eh_frame);
            __data_end__ = ABSOLUTE(.) ;
      } > ram

      .bss ALIGN(4) : {
            __bss_start__ = ABSOLUTE(.);
            *(.bss*) *(COMMON);
            __bss_end__ = ABSOLUTE(.);
      } > ram

      .stack ALIGN(4) : {
            __stack_start__ = ABSOLUTE(.);
            *(.stack);
            . = . + 0x1000;
            __stack_end__ = ABSOLUTE(.);
      } > ram
}
```

Exemplo de um programa com uma parte em C - ficheiro **main.c** e uma parte em assembly – ficheiros **division.s**.

Os restantes ficheiros envolvidos são o módulo **csstart.s**, que também faz parte do programa, sendo este efectivamente constituido por três módulos, o **ldscript**, e o **makefile**.

Os ficheiros específicos deste programa são o main.c, o division.s e o makefile, os restantes cstart.s e ldscript podem ser reutilizados para outros programas.

```
      .text
      .global division
division:
      cmp   r1, #0            Evitar divisão por zero
      beq   .

      mov   r2, #0            r2 - resto
      mov   r3, #0            r3 - quociente
      mov   r12, #32          r12 - 32 iterações
1:
      movs  r0, r0, lsl #1    dividendo = dividendo << 1
      adc   r2, r2, r2        resto = resto << 1 + bit de maior peso dividendo

      cmp   r2, r1            carry = 1 se resto(r2) >= divisor(r1)
      subcs r2, r2, r1        se resto >= divisor então resto = resto - divisor
      adc   r3, r3, r3        quociente = quociente << 1 + carry
      subs  r12, r12, #1
      bne   1b
      mov   r0, r3
      mov   pc, lr
```

division.c

```
int div(int, int);
int a;
int main() {
      a = division(200, 20);
      return 0;
}
```

main.c

```
main.elf: cstart.o division.o main.o
      arm-elf-gcc cstart.o division.o main.o -o main.elf -Tldscript \
            -nostdlib -lc -lgcc

main.o: main.c
      arm-elf-gcc -g -c  -o main.o main.c

cstart.o: cstart.s
      arm-elf-as --gstabs cstart.s -o cstart.o

division.o: division.s
      arm-elf-as --gstabs division.s -o division.o

clean:
      -rm -f *.elf *.o *.lst *.bak *.map *.ii *.i *.d
```

makefile

Um makefile contém uma sucessão de regras encadeadas.

Uma regra é composta por três campos: objectivo, dependências e comandos, dispostos da seguinte forma:

```
objectivo: dependências
      comandos
```

Uma regra interpreta-se da seguinte forma: para actualizar **objectivo**, executa **comando** se alguma **dependência** for mais recente.

As **dependencias** são ficheiros originais (ficheiros com fontes de programa) ou, por sua vez, são gerados por outras regras.

O ficheiro **makefile** é processado pelo programa **make**. Por omissão, este tenta actualizar o primeiro objectivo, neste caso o ficheiro com o código executável do programa **main.elf**.

```
$ make
```

Bibliotecas

Criação de bibliotecas

O objectivo ao constituir uma biblioteca é colocar disponíveis funções previamente compiladas e testadas.

No exemplo que se segue são utilizados para a criação da biblioteca os ficheiros **division.s** e **mutiplication.s**, com o código fonte, e o ficheiro **makelib.**

```
.global multiply
multiply:
      mul r2, r0, r1
      mov r0, r2
      mov pc, lr
```

multiplication.s

```
libmath.a: division.o multiplication.o
      arm-elf-ar -r libmath.a division.o multiplication.o
```

```
division.o: division.s
      arm-elf-as --gstabs -o division.o division.s

multiplication.o: multiplication.s
      arm-elf-as --gstabs -o multiplication.o multiplication.s
```

makelib

O processamento do ficheiro **makelib,** por parte do programa **make**, origina o ficheiro **libmath.a** com o código de ambas as funções, como se pode verificar:

$ arm-elf-objdump -d mathlib.a

```
In archive libmath.a:

division.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <divide>:
   0: e3510000    cmp   r1, #0      ; 0x0
   4: 0afffffe    beq   4 <divide+0x4>
   8: e3a02000    mov   r2, #0      ; 0x0
   c: e3a03000    mov   r3, #0      ; 0x0
  10: e3a0c020    mov   ip, #32     ; 0x20
  14: e1b00080    movs  r0, r0, lsl #1
  18: e0a22002    adc   r2, r2, r2
  1c: e1520001    cmp   r2, r1
  20: 20422001    subcs r2, r2, r1
  24: e0a33003    adc   r3, r3, r3
  28: e25cc001    subs  ip, ip, #1  ; 0x1
  2c: 1a000003    bne   14 <divide+0x14>
  30: e1a00003    mov   r0, r3
  34: e1a01002    mov   r1, r2
  38: e1a0f00e    mov   pc, lr

multiplication.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <multiply>:
   0: e0020190    mul   r2, r0, r1
   4: e1a00002    mov   r0, r2
   8: e1a0f00e    mov   pc, lr
```

Utilização de bibliotecas

Os módulos de utilizador (por exemplo **main.c**) podem evocar as funções incluídas na biblioteca. A insersão do códiga das funções em biblioteca no programa em produção ocorre na fase de ligação de módulos, durante o processamento de **ld**.

O **ld** precisa da indicação da directoria onde se encontra a biblioteca (opção **–L**) e da indicação da biblioteca (opção **–lmath**). Por convenção o nome dos ficheiros-biblioteca têm a seguinte forma: **libxxx.a** sendo **xxx** o identificador da biblioteca, neste caso **math**. Veja-se o exemplo de um makefile para utilizar as funções **divide** ou **multiply** na biblioteca **libmath.a**.

```
main.exe: cstart.o main.o
      arm-elf-ld cstart.o main.o -o main.exe –L. –lmath
```

```
cstart.o: cstart.s
      arm-elf-as --gstabs -o cstart.o cstart.s

main.o: main.c
      arm-elf-gcc -c -g -o main.o main.cpp
```

makefile

Biblioteca normalizada

A biblioteca normalizada contém o código das funções especificadas na norma adaptadas à plataforma de execução. No caso dos sistemas embebidos a plataforma pode ter limitações e nem todas as funções estarem disponíveis.

No mundo *open source* existe a biblioteca *newlib*, que realiza as funções independentes da plataforma, como é o caso das funções de processamento de strings, e partes genéricas de outras funções deixando bem definida uma interface de mais baixo nível que permite adaptar a biblioteca com um mínimo de esforço é o caso das funções de acesso a ficheiros.

Aos produzirem-se e instalarem-se as ferramentas de compilação e produção de programas pode incluir-se a biblioteca normalizada. Para a utilizar basta usar a opção **-lc** (o ficheiro chama-se libc.a) no programa **ld**.

```
arm-elf-ld cstart.o main.o -o main.elf -Tldscript -lc
```

Biblioteca do compilador

Em algumas situações o compilador apela a funções que admite estarem disponíveis em biblioteca própria (**libgcc.a**). Por exemplo, quando necessita de copiar um bloco de memória ou executar uma operação aritmética complexa, para as quais não existe suporte directo no processador, o compilador evoca funções em biblioteca.

As ferramentas de compilação e geração de programas GNU são instaladas numa sub-árvore de directorias bem definida. Quando se trata de as usar para gerar código para sistemas externos (*cross-compiling*) o local de insersão desta sub-árvore é muito variável.

Alguns locais possíveis são:/opt/gnutools; /usr/local/gnutools; /usr/cross. Admitindo que o local de instalação é **/usr/cross**, para ser encontrada devem ser incluídos na linha do programa **ld** as seguintes opções:

```
-L/usr/cross/arm-elf/lib/gcc/arm-elf/4.1.1 -lgcc
```

Opções do compilador

--save-temp Preserva os ficheiros intermédios da compilação. O ficheiro com a extenção **i** contém o código C depois de aplicado o pré-compilador. Nele pode observar-se o código original com a substituição do texto das macros. O ficheiro com a extenção **s** contém o código assembly gerado.