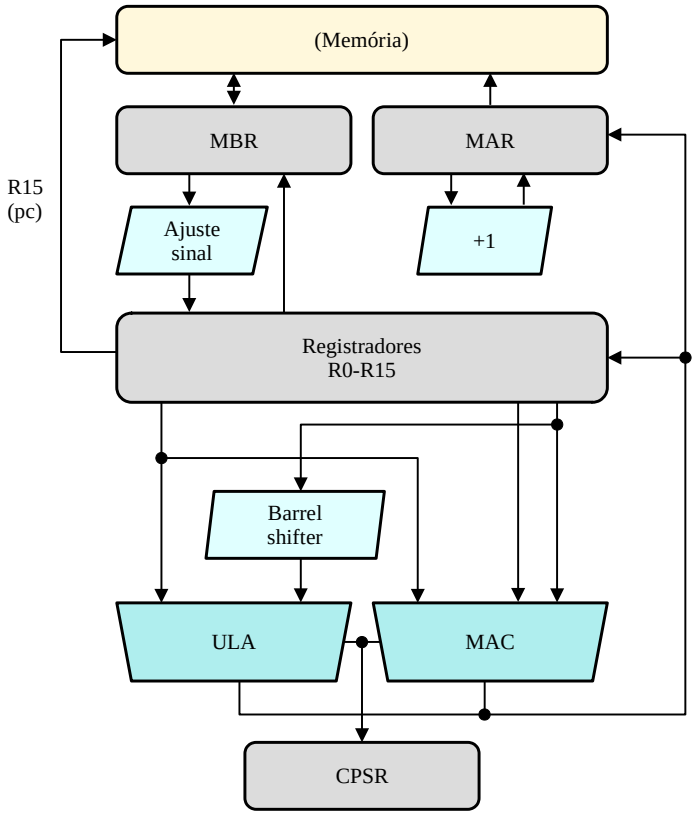
Índice

Arquitetura do processador ARM

O processador ARM (*Advanced Risc Machine*, originalmente *Acorn Risc Machine*) é um microprocessador RISC (*Reduced Instruction Set Computer*) de 32 bits muito popular. Sua primeira versão foi desenvolvida em 1985 e várias revisões foram feitas ao longo dos anos. As versões mais utilizadas de sua arquitetura hoje em dia são a versão 4 (em microcontroladores), a versão 7 (computadores e dispositivos portáteis de 32 bits) e a versão 8 (computadores e dispositivos portáteis de 64 bits). Como regra geral, as versões mais recentes implementam todas as instruções das versões mais antigas.

A arquitetura original foi implementada com um *pipeline* de três estágios, apresentado na figura seguinte:



Outras versões da arquitetura ARM contam com *pipelines* com maior número de estágios, *pipelines* superescalares e podem incluir processadores secundários, como processador de ponto flutuante escalar (armhf) e processador vetorial (NEON). Além disso, arquiteturas com múltiplos núcleos (SMP) são comuns. No entanto, o modelo lógico do fluxo de dados visto anteriormente pode ser utilizado para compreender o conjunto de instruções básico, com o qual todos os processadores de todas as versões da arquitetura são compatíveis.

Conjunto de registradores

O coração da CPU é um conjunto de dezesseis registradores, cujos nomes são R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14 e R15. Em conformidade com a filosofia dos processadores RISC, qualquer desses registradores pode ser utilizado por qualquer instrução de máquina, de forma simétrica; contudo, os registradores R14 e R15 têm funções especiais.

Contador de programa

O registrador R15 é o **contador de programa**, e seu conteúdo aponta o endereço da memória no qual a próxima instrução será buscada (estágio "*fetch*"

do *pipeline*). A execução de instruções de salto e a ocorrência de exceções ou interrupções vão alterar diretamente o conteúdo desse registrador.

Link register

O registrador R14 é chamado de **link register**: ele é utilizado para o armazenamento temporário do endereço de retorno de uma sub-rotina ou de um serviço de interrupção. Ao ser executada uma instrução *branch and link* (`bl`) ou ao ser reconhecida uma exceção ou interrupção, o valor do contador de programa (registrador R15) é copiado no registrador R14 antes de ser alterado para o endereço de destino do salto.

Unidade Lógica e Aritmética

O **estágio de execução** do *pipeline* inclui uma Unidade Lógica e Aritmética (ALU) bastante poderosa, um multiplicador com acúmulo e um deslocador tipo "barril" (*barrel shifter*) em uma das entradas da ALU. A Unidade Lógico e Aritmética do ARM é capaz de executar todas as operações lógicas, as operações de soma e subtração (neste último caso também permitindo inverter a ordem dos operandos) e pode alterar o valor dos *flags* do processador (N, V, C e Z, no registrador CPSR) conforme o resultado da operação. Um campo especial na instrução (bit "S" ou "*set flags*") permite habilitar ou inibir a atualização dos *flags* pela ALU como resultado da execução da instrução.

Multiplicador

O **multiplicador** permite calcular o produto entre um número de 32 bits e um número de oito bits (até a versão 4 da arquitetura) ou de dezesseis bits (versão 5) e acumular o resultado. Dessa forma, não é possível completar uma instrução de multiplicação de 32 bits por 32 bits em um único ciclo.

Barrel shifter

O ***barrel shifter*** é capaz de deslocar o valor de 32 bits em sua entrada de qualquer quantidade de bits à direita ou à esquerda, também podendo considerar o deslocamento do sinal (deslocamento *aritmético* à direita) e executar **rotações** à direita. O *barrel shifter* é um **circuito combinatório**, realizando esse deslocamento ou rotação em um único ciclo. Além da função óbvia no processamento de deslocamentos e rotações, o *barrel shifter* é comumente utilizado em conjunto com a operação realizada pela ALU e é muito útil nas instruções de acesso à memória com índice.

CPSR

Os *flags de estado* do processador são armazenados em um registrador especial denominado CPSR (*current program status register*), cujos bits possuem diferentes significados, conforme a tabela a seguir:

Bits	Nome	Significado
31	N	Resultado negativo
30	Z	Resultado nulo
29	C	Vai-um (*carry*) detectado
28	V	Estouro (*overflow*) detectado
...	...	(Função varia conforme a família)
5	T	Executando instrução de 16 bits (modo *Thumb*)
4-0	MODE	Modo atual de execução

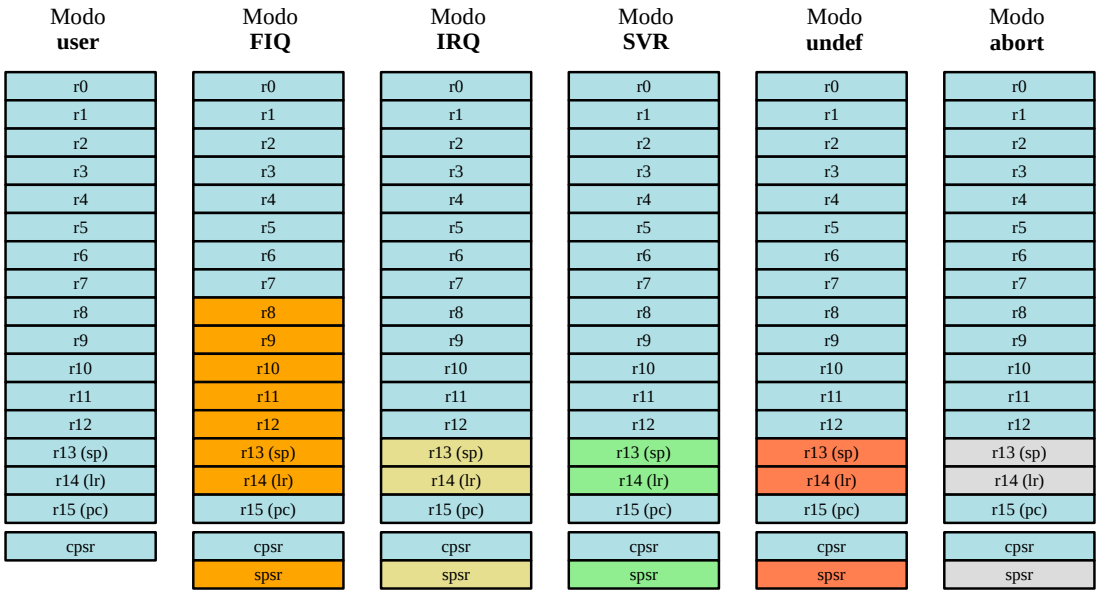
Valor	Modo	Valor	Modo
10000	User	10111	Abort
10001	FIQ	11011	Undef
10010	IRQ	11010	Hyper
10011	SVR	11111	System
10110	Monitor		

Modos de execução

O **modo de execução** do processador é utilizado para definir níveis de privilégio, permitindo o controle de características especiais da arquitetura, como o acesso a recursos da CPU e conjuntos de registradores alternativos. Normalmente, o processador executa no modo usuário ("*User*"), que possui o menor privilégio; ao ser atendida uma interrupção, o modo é alterado para "IRQ" ou "FIQ", dependendo da configuração das interrupções; a ocorrência de exceções pode levar o processador ao estado "Undef" (instrução não definida), "Abort" (falha de acesso à memória) ou "SVR" (supervisor: chamada ao Sistema Operacional com a instrução `swi`).

SPSR

Instâncias dedicadas de alguns registradores são exclusivas de cada modo, mantendo o seu valor inalterado enquanto o processador executar em outros modos. Veja no esquema da figura abaixo que os registradores R0 até R7 são comuns a todos os modos e que os registradores R13, R14 e SPSR têm instâncias específicas para cada um dos modos. O registrador SPSR (*saved program status register*) de cada modo é automaticamente atualizado com o valor do registrador CPSR no momento da mudança de modo.



Os registradores R8 até R12 têm instâncias especiais no modo "FIQ" (chamado de modo "*fast interrupt*" ou "interrupção rápida"), com o objetivo de agilizar o tratamento dessa interrupção, uma vez que a cópia e a restauração dos valores originais desses registradores podem ser dispensadas pelo serviço de interrupção correspondente.

Stack pointer

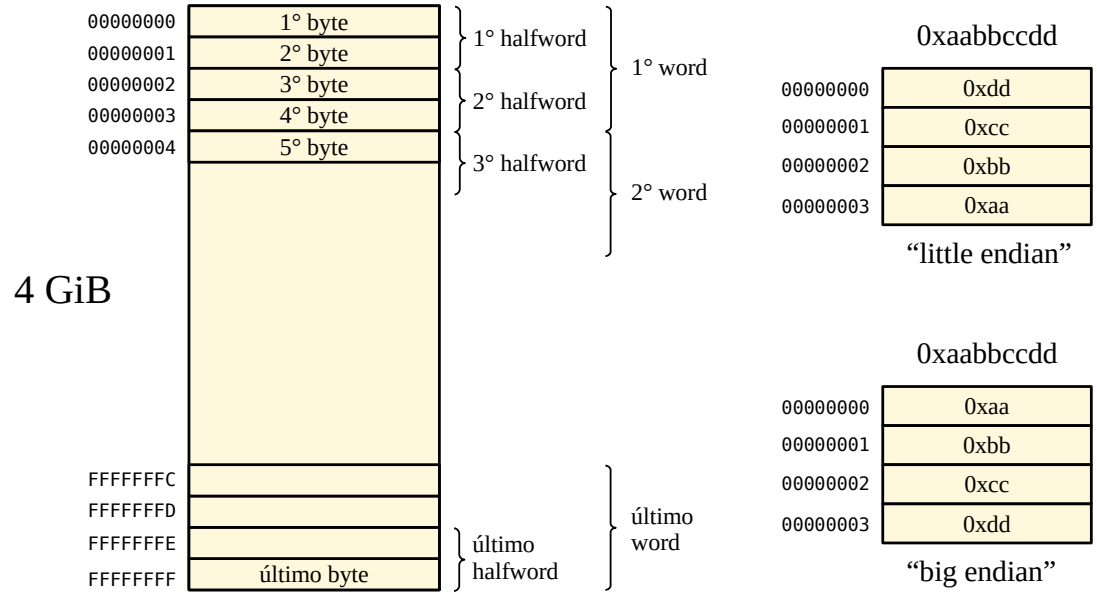
Observe que o registrador R13 também é preservado durante as mudanças de modo. Por convenção, esse registrador normalmente é utilizado como **ponteiro de pilha** (*stack pointer*). Na verdade, a arquitetura ARM não define uma "pilha" e qualquer registrador poderia ser utilizado com essa finalidade, mas o registrador R13 possui a vantagem de ter instâncias independentes, automatizando a definição de pilhas separadas para cada modo.

Os registradores do ARM podem conter valores de 32 bits ("*words*"), cujo sinal pode ser considerado (*signed*, em complemento de dois) ou não (*unsigned*). As instruções `load` e `store` também podem carregar ou salvar valores de 16 bits

("*half words*") ou de 8 bits ("*bytes*") da memória: nesses casos, se o sinal for considerado, ele é **estendido** do bit 15 (no caso de *half words* com sinal) ou do bit 7 (no caso de *bytes* com sinal) até o bit 31 do registrador de destino.

Organização da memória

A memória é sempre organizada em *bytes* (cada *endereço* contém um *byte*) e, como qualquer registrador pode ser usado como um **ponteiro**, ela pode conter até 2^{32} endereços distintos (4 GiB)[1]. No entanto, tanto palavras quanto instruções de 32 bits devem ser **alinhadas** em endereços *múltiplos de quatro*; *half words* e instruções de 16 bits (modo *Thumb*) somente podem ser acessadas em endereços *pares*.



A arquitetura ARM pode salvar dados compostos por múltiplos *bytes* na memória empregando tanto a ordenação *little endian* (*bytes* menos significativos são salvos em endereços menores) quanto *big endian* (*bytes* mais significativos em endereços menores). Após o *reset*, a ordenação padrão é *little endian*.

Em contraste com as arquiteturas RISC tradicionais, com o objetivo de reduzir custos, a arquitetura original (versão 1) não incluía uma memória *cache* interna nem exigia algum tipo especial de barramento de memória ou a existência de uma memória *cache* externa. Nessa situação, é comum na prática que diversos acessos consecutivos à memória sejam necessários para a execução de uma instrução `load` ou `store`. Considerando esse fato, os projetistas do ARM introduziram também instruções capazes de ler e de escrever *diversos* registradores em *sequência* na memória (`load multiple` e `store multiple`). Todas essas operações normalmente levam vários ciclos de máquina para serem executadas.

Observe que apesar de existirem instruções especiais para a troca de dados com coprocessadores, qualquer acesso a dispositivos e periféricos deve ser realizado através das mesmas instruções `load` e `store`, usando o endereçamento de memória convencional (entrada e saída mapeada em memória).

1 A versão 8 da arquitetura ARM pode operar como uma máquina de 64 bits, aumentando muito o espaço de endereçamento. Na versão 7, alguns *chips* possuem um endereçamento externo (físico) significativamente maior (até 1 TiB) do que o endereçamento virtual, o que é chamado de LPAE (*Large Physical Address Extension*).

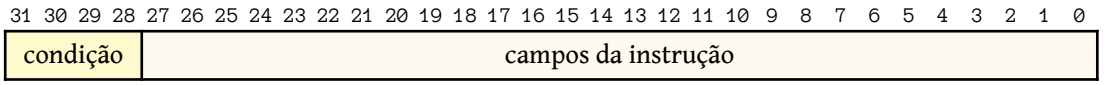
O conjunto de instruções do Processador ARM

A maioria das famílias de processadores ARM é capaz de reconhecer mais do que um conjunto de instruções. O conjunto de instruções padrão define instruções que possuem sempre 32 bits de tamanho, denominado A32; a maioria das famílias, a partir da versão 4 da arquitetura, reconhecem também uma versão compacta desse conjunto de instruções, na qual as instruções são formatadas em apenas 16 bits: esse conjunto de instruções é chamado "*Thumb*" ou T16. Os processadores de 64 bits (versão 8 da arquitetura) reconhecem também uma extensão do conjunto A32, denominado A64.

Formato geral das instruções A32

Instruções A32

Todas as instruções do conjunto A32 possuem 32 bits, não existindo instruções maiores (ocupando mais do que quatro bytes na memória de programa). Os quatro bits mais significativos de todas as instruções A32 sempre representam um *código de condição* para a execução dessa instrução, ou seja, **todas** as instruções do conjunto A32 são **condicionais**.



Códigos de condição

Dependendo do **código de condição** presente no início da instrução e da situação atual dos *flags* N, Z, C e V, a instrução pode ser executada (a condição está satisfeita) ou descartada (condição não satisfeita). Neste último caso, a instrução é ignorada pelo estágio de execução do *pipeline* ("nop" ou "*no operation*"). Os códigos de condição definidos no processador ARM são apresentados na tabela a seguir:

código	*flags* NZCV	Nome	código	*flags* NZCV	nome
0000	`-1--`	igual (eq)	1000	`-01-`	maior que (hi)[2]
0001	`-0--`	diferente (ne)	1001	`--0-` `-1--`	menor ou igual (ls)[2]
0010	`--1-`	*carry* (cs)	1010	`0--0` `1--1`	maior ou igual (ge)
0011	`--0-`	sem *carry* (cc)	1011	`0--1` `1--0`	menor que (lt)
0100	`1---`	negativo (mi)	1100	`00-0` `10-1`	maior que (gt)

0101	`0---`	não negativo (pl)	1101	`-1--`
				`1--0`
				`0--1`
					menor ou igual (le)

0110	`---1`	*overflow* (vs)	1110	`----`	incondicional
0111	`---0`	sem *overflow* (vc)	1111	`----`	inválido

As instruções cujo código de condição não seja "1110" são, portanto, **condicionais**, e ganham um *sufixo* em seu mnemônico para indicar esse fato. Por exemplo, uma instrução de adição "`add`" cuja execução seja condicionada ao caso do *flag carry* ser "um" será denominada "`addcs`"; neste caso, os bits mais significativos da instrução devem ser "0010".

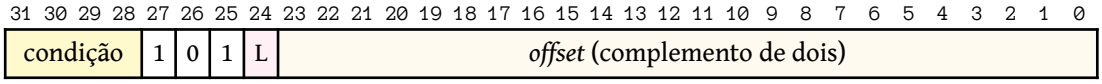
2 Nestes dois casos o sinal é desconsiderado.

Instruções de Salto

Saltos e sub-rotinas

As instruções de salto alteram o valor do contador de programa (R15), modificando, portanto, o fluxo de execução do programa. Os saltos podem ou não armazenar o endereço de retorno (instrução "*branch and link*" ou `bl`): neste caso, o valor do registrador R15 é copiado no registrador R14 (*link register*) antes de sua atualização. Dessa forma, a sub-rotina chamada pode posteriormente retornar ao programa chamador.

Normalmente, a execução de um salto provoca o esvaziamento do *pipeline*; a partir da versão 4, a arquitetura ARM inclui algoritmos para previsão de desvios. O formato das instruções de salto é apresentado a seguir:



Saltos condicionais e incondicionais

O campo de código de condição é o mesmo descrito na seção anterior. Saltos *incondicionais* devem ter o campo de condição igual a "1110". O valor do bit "L" (bit 24) define se o endereço de retorno deve ser salvo (instrução *branch and link*: `bl`) ou não (instrução *branch*: `b`).

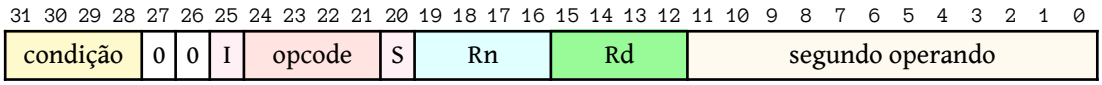
Saltos relativos

Os saltos executados pelas instruções `b` e `bl` são sempre **relativos**: o endereço de destino do salto é calculado a partir da *posição atual* do contador de programa. Para determinar o endereço de destino, o valor do *offset* contido na instrução é multiplicado por quatro e somado ao valor corrente do registrador R15. Como o campo de *offset* possui 24 bits com sinal, até 32 Mi posições de memória para frente (*offset* positivo) e para trás (*offset* negativo) do contador de programa podem ser alvo de um salto.

Instruções ALU

As instruções ALU são as instruções que podem alterar o valor de um registrador mediante uma operação lógica ou aritmética sobre outros dois valores (operandos), um dos quais está em um registrador (eventualmente o mesmo registrador de destino); o segundo operando da instrução pode ser um registrador ou uma constante numérica positiva, introduzida diretamente no corpo da instrução (valor imediato).

Assim, as instruções de ALU são *tuplas*: uma única instrução pode envolver **três** registradores diferentes (na verdade, graças ao *barrel shifter*, são **quatro** registradores, como veremos a seguir).



Código de condição e "*set flags*"

O código de condição tem o mesmo formato das demais instruções do ARM: caso a condição expressa na instrução não seja válida, a instrução é ignorada. No caso das instruções ALU, o bit "S" ou "*set flags*" (bit 20) especifica se essa instrução deve ou não deve alterar os valores dos *flags* do processador de acordo com o resultado da operação; o bit "S" é sempre usado em conjunto com os códigos de condição: com o bit "S" igual a "zero", os *flags* não são alterados,

por exemplo para não interferir com o estado de um teste anterior, que determinou a execução condicional desta instrução em particular e que deve continuar válido para a(s) próxima(s) instrução(ões). O mnemônico da instrução ganha o sufixo "s" (por exemplo, `adds`, `orrs`, etc.) quando o campo de *set-flags* está ativo.

Os campos referentes aos registradores de destino (Rd) e do primeiro operando (Rn) contém quatro bits, representando os índices correspondentes aos registradores R0 (valor "0000") até R15 (valor "1111"). Os bits no campo "*opcode*" selecionam a operação a ser realizada pela ALU, conforme descrito pela tabela a seguir:

Opcodes

opcode	instrução	opcode	instrução
0000	`and`	1000	`tst`
0001	`eor`	1001	`teq`
0010	`sub`	1010	`cmp`
0011	`rsb`	1011	`cmn`
0100	`add`	1100	`orr`
0101	`adc`	1101	`mov`
0110	`sbc`	1110	`bic`
0111	`rsc`	1111	`mvn`

Instruções lógicas

- Instruções *lógicas*: "e" (`and`), "ou" (`orr`), "ou exclusivo" (`eor`) e "e-não" ("*bit clear*" ou `bic`). Essa última instrução é um mecanismo eficiente para *zerar* bits no registrador de destino;

Instruções aritméticas

- Instruções *aritméticas*: soma (`add`), soma com vai-um (`adc`), subtração (`sub` e `rsb`) e subtração com empresta-um (`sbc` ou `rsc`). As instruções `sub` e `sbc` subtraem o segundo operando do primeiro, enquanto que as instruções `rsb` ("*reverse subtract*") e `rsc` ("*reverse subtract with carry*") subtraem o primeiro operando do segundo. Considerando que *somente o segundo operando pode ser uma constante imediata*, as instruções "reversas" tornam-se particularmente úteis;

Comparações

- Instruções de *comparação*: `cmp` e `cmn`. A instrução `cmp` é equivalente à instrução `subs`, sem, contudo, salvar o resultado da subtração no registrador de destino. O resultado da subtração de dois valores pode ser nulo (os valores são *iguais*), positivo (o primeiro valor é *maior* do que o segundo) ou negativo (o primeiro valor é *menor* do que o segundo). De forma análoga, a instrução `cmn` é equivalente à instrução `adds`: nesse caso, o segundo operando é considerado com o sinal invertido, o que é muito conveniente para a comparação com *valores constantes negativos*, uma vez que as constantes introduzidas nas instruções (segundo operando) são sempre positivas;

Testes lógicos

- Instruções de *teste lógico*: `tst` e `teq`. À semelhança das instruções de comparação, as instruções de teste lógico não alteram o registrador de destino, apenas servindo para alterar o valor dos *flags*. A instrução `tst` é equivalente à instrução `ands`, servindo para testar se determinados bits

do operando são iguais a "um". A instrução `teq` é equivalente à instrução `eors`, permitindo verificar se dois valores são iguais, somente afetando o *flag* Z, sem interferir com os *flags* C e V;

- Instruções de *movimentação de dados*: `mov` e `mvn`. Nessas instruções, a ALU somente *copia* o valor do operando ao registrador de destino (`mov`) ou copia o *inverso bit a bit* (operação lógica "não") do operando ao registrador de destino. Essa última operação é particularmente útil para carregar *valores constantes negativos* em um registrador, uma vez que as constantes introduzidas nas instruções (segundo operando) são sempre positivas. Observe que as instruções de movimentação de dados também podem modificar os valores dos *flags*, bastando que o bit "S" da instrução seja igual a "um".

Movimentação de dados

As instruções de comparação e testes lógicos somente fazem sentido quando o bit "S" (*set flags*) é igual a "um": o comportamento do processador ao executar uma instrução desse tipo com o bit "S" igual a "zero" não é especificado pela arquitetura e tal instrução deve ser considerada inválida.

Quando o registrador de destino é o *contador de programa* (`r15`), o resultado da operação é um *salto*. Neste caso especial, o significado do bit "S" é diferente: caso o valor de "S" seja "um", o valor do registrador `cpsr` é inteiramente *substituído* pelo valor do registrador `spsr` do modo atual; isso é necessário, por exemplo, para retornar de serviços de interrupção.

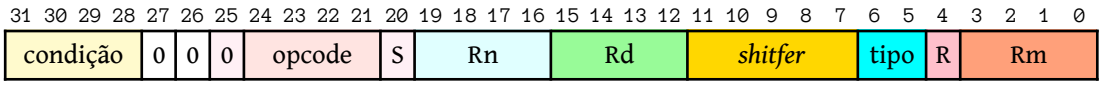
o registrador r15

Segundo operando

O segundo operando da instrução sempre pode ser alterado pelo *barrel shifter*. O significado dos bits referentes ao segundo operando depende do valor do bit "I" (bit 25): caso o bit "I" seja "um", o campo contém um valor **imediato**; caso seja "zero", o campo de segundo operando especifica um **terceiro registrador**.

O segundo operando

Na situação na qual o campo de segundo operando corresponde a um registrador (bit "I" igual a "zero"), o campo de quatro bits Rm conterá o índice desse registrador, no mesmo formato que os campos Rd e Rn. O valor contido no registrador Rm pode ser ainda manipulado pelo *barrel shifter* antes de ser utilizado como operando pela ALU.

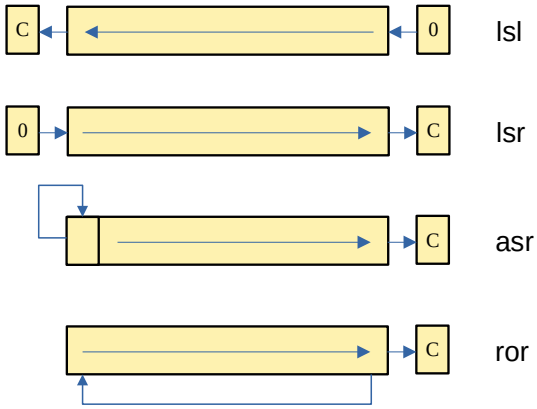


O comportamento do *barrel shifter* com relação ao valor de Rm é definido pelos campos "*shifter*" (bits 7-11) e "tipo" (bits 5 e 6) da instrução.

Tipo de operações do *barrel shifter*

"tipo"	operação *barrel shifter*
00	Deslocamento lógico à esquerda (lsl)
01	Deslocamento lógico à direita (lsr)
10	Deslocamento aritmético à direita (asr)
11	Rotação à direita (ror)

Veja na figura a seguir as operações que podem ser realizadas pelo *barrel shifter*, considerando uma movimentação (deslocamento ou rotação) de apenas um bit. O *flag carry* somente é afetado se o bit "S" estiver ativo; além disso, dependendo da instrução realizada, a ALU poderá modificar em seguida o estado desse *flag*.



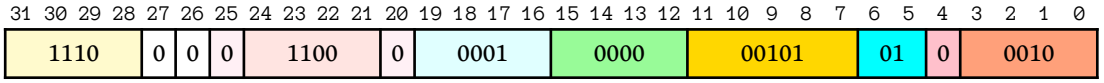
No caso em que o bit "R" é "zero" trata-se de uma movimentação de tamanho fixo, especificado diretamente pelo campo "*shifter*", de cinco bits, tomado como um valor imediato (0 até 31).

Exemplo

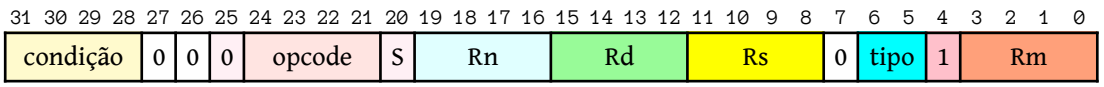
Exemplo: deslocamento fixo

A palavra a seguir (0xE18102A2) corresponde à instrução

```
orr R0, R1, R2, lsr #5
```



Por outro lado, na situação na qual o bit "R" é "um", o tamanho do deslocamento ou rotação é determinado pelo valor de um **quarto** registrador, especificado pelo campo Rs (bits 8-11), no mesmo formato que Rd, Rn e Rm:

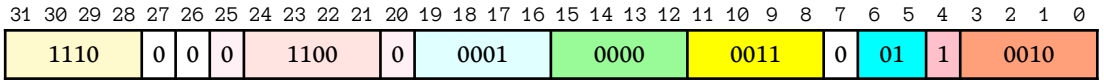


Exemplo

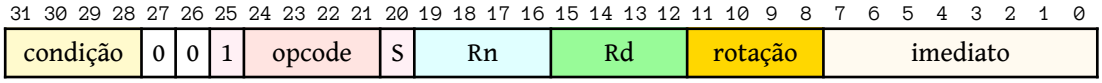
Exemplo: deslocamento definido por registrador

A palavra a seguir (0xE1810332) corresponde à instrução

```
orr R0, R1, R2, lsr R3
```



Finalmente, o último caso é aquele no qual o bit "I" é igual a "1" e o campo de segundo operando contém um valor **imediato positivo**. Também neste caso, o *barrel shifter* pode ser utilizado para aumentar o número de constantes que podem ser representadas, através do campo "rotação" (bits 8-11):

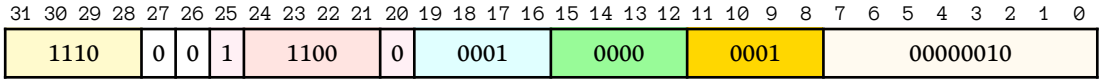


O valor imediato é representado nos bits 0-7 da instrução, sem sinal. O campo "rotação", de quatro bits, estabelece a quantidade de bits para uma **rotação à direita** (operação "ror") para reposicionar esse valor imediato em uma palavra de 32 bits, composta pelo *barrel shifter*. Como há somente quatro bits disponíveis para o campo "rotação", esse valor **é multiplicado por dois**, permitindo rotações de zero ("rotação" vale "0000") até 30 ("rotação" vale "1111") bits.

Exemplo

Exemplo: operando constante

A palavra a seguir (0xE3810102) corresponde à instrução
`orr R0, R1, #0x80000000`



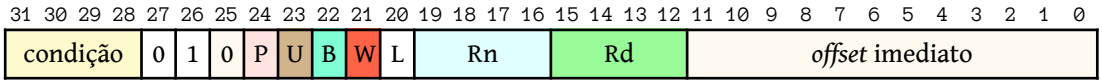
Instruções de acesso à memória

Como em qualquer processador RISC, as únicas instruções do conjunto de instruções ARM que podem fazer acesso à memória são as instruções `load` e `store`. As instruções `load` e `store` precisam que o endereço da memória (ponteiro) a ser acessado esteja armazenado previamente em um registrador, denominado **registrador base**: em outras palavras, somente é possível o *endereçamento indireto* (baseado em um ponteiro). Um valor de **índice** pode ser somado ou subtraído ao valor do registrador base para calcular o valor efetivo do endereço de acesso à memória; tal índice pode ser incluído na instrução como uma constante imediata (*offset*) ou ser proveniente de um registrador. Neste último caso, o valor do registrador de índice pode ainda ser manipulado pelo *barrel shifter*: útil para alinhar endereços na memória.

Registrador base

Índice

O formato das instruções `load` e `store` com *offset* imediato é apresentado na figura a seguir.



`load`/`store` com *offset* imediato

O código de condição tem o mesmo formato das demais instruções do ARM: caso a condição expressa na instrução não seja válida, a instrução é ignorada. O campo Rd especifica, no mesmo formato que as instruções ALU, o registrador de *destino* (no caso de uma instrução `load`) ou de *origem* (no caso de uma instrução `store`) da informação a ser lida ou escrita na memória. Rn identifica o registrador **base** (ponteiro), a partir do qual será calculado o endereço da memória a ser acessado. O bit "L" define a instrução como `load` (bit "L" igual a "um") ou `store` (bit "L" igual a "zero").

Os bits "P", "U", "B" e "W" da instrução são usados para especificar o *modo de endereçamento*, conforme a tabela:

P	pré-(1) ou pós-(0) indexar
U	somar com a (1) ou subtrair da (0) base
B	transferir byte (1) ou word (0)
W	alterar registrador base (1)

O modo **pré-indexado** (bit "P" da instrução igual a "um") calcula o endereço efetivo a partir do endereço base **antes** da operação e utiliza o endereço calculado para o acesso à memória. O valor contido no registrador base permanece inalterado. Alternativamente, o modo **pós-indexado** (bit "P" da instrução igual a "zero") realiza a operação de leitura ou escrita na memória usando o endereço base e, **depois** disso, **modifica** o valor do registrador base para o valor do endereço efetivo calculado. Caso o programador deseje que o valor do registrador base seja atualizado também no modo pré-indexado, pode usar o bit "W" (*write back*): após um acesso pré-indexado com o bit "W" igual a "um", o registrador base é atualizado para o valor do endereço efetivo utilizado na operação de memória.

modos pré-indexado e pós-indexado

modo pré-indexado com *write back*

direção do índice

A **direção** da indexação é definida pelo valor do bit "U": caso seja "1" (*up*), o valor de índice (em ambos os casos, *offset* imediato ou registrador) é **somado** ao registrador base; caso seja "zero", o endereço efetivo é calculado **subtraindo** o índice do registrador base. Observe-se que o valor do *offset* imediato (bits 0 até 11) é sempre positivo.

`load/store` *byte*

Finalmente, o bit "B" da instrução permite especificar o tamanho da transferência: um *word* (quatro *bytes*, bit "B" igual a "zero") ou um *byte* (bit "B" igual a "um"). Observe que, no caso de uma transferência de 32 bits, o endereço efetivo precisa ser múltiplo de quatro para que a execução da instrução obtenha o resultado esperado! A ordenação (*endianess*) dos dados de 32 bits lidos da memória pode ser definida pelo programador, sendo que a ordenação padrão é *little-endian*.

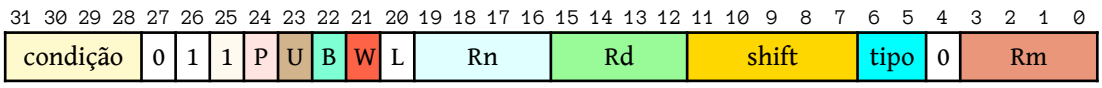
As transferências de um *byte* não têm restrição quanto ao valor do endereço efetivo. As instruções de `load` e `store` no nível de *bytes* têm os mnemônicos `ldb` e `stb`, respectivamente. Atenção ao fato de que, na instrução `ldb`, o sinal do *byte* lido **não é estendido**: os bits 8-31 do registrador de destino são zerados; no caso da instrução `stb`, somente os bits 0-7 do registrador de origem são transferidos à memória.

Convencionam-se os formatos das instruções `load` e `store` com *offset* imediato para os programas *assembler* da seguinte forma:

Instrução	Modo	P	U	W
`ldr r0, [r1]`	(Indefinido: *offset* zero)	-	-	-
`ldr r0, [r1], #4`	Pós-indexado, *offset* positivo	0	1	-
`ldr r0, [r1], #-4`	Pós-indexado, *offset* negativo	0	0	-
`ldr r0, [r1, #4]`	Pré-indexado, *offset* positivo	1	1	0
`ldr r0, [r1, #-4]`	Pré-indexado, *offset* negativo	1	0	0
`ldr r0, [r1, #4]!`	Pré-indexado, *offset* positivo, atualiza base	1	1	1

`load/store` com registrador de índice

O formato das instruções `load` e `store` com registrador de índice é semelhante:



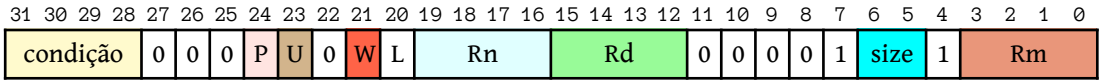
A especificação do *modo de endereçamento* é a mesma do formato anterior (bits "P", "U", "B" e "W"). O índice a ser somado ao (bit "U" igual a "um") ou subtraído do (bit "U" igual a "zero") registrador base para o cálculo do endereço efetivo é proveniente do registrador Rm (registrador de índice). Esse valor também pode ser manipulado pelo *barrel shifter*, de forma equivalente ao que acontece com as instruções ALU. A operação do *barrel shifter* é definida pelos campos "tipo" e "shift": o campo "*shift*" permite especificar um número constante de bits (valor imediato de zero a 31) para deslocamento lógico à esquerda ("tipo" igual a "00"), lógico à direita ("tipo" igual a "01"), aritmético à direita ("tipo" igual a "10") ou rotação à direita ("tipo" igual a "11").

Convencionam-se os formatos das instruções `load` e `store` com registrador de índice para os programas *assembler* da seguinte forma:

Instrução	Modo	P	U	W
`ldr r0, [r1], r2`	Pós-indexado	0	1	-
`ldr r0, [r1], r2, lsl #4`	Pós-indexado, com deslocamento	0	1	-
`ldr r0, [r1], -r2`	Pós-indexado, inverte sinal do índice	0	0	-
`ldr r0, [r1, r2]`	Pré-indexado	1	1	0
`ldr r0, [r1, r2, lsl #4]`	Pré-indexado, com deslocamento	1	1	0
`ldr r0, [r1, -r2]`	Pré-indexado, inverte sinal do índice	1	0	0
`ldr r0, [r1, r2]!`	Pré-indexado, atualiza base	1	1	1

`load/store` *half*

Outro grupo de instruções `load/store` permite a manipulação de ***half words*** na memória, com ou sem sinal, bem como *bytes* com sinal (complementando a limitação das instruções de `load/store` *word*). Quando é empregado um registrador de índice, o formato dessas instruções é o seguinte:

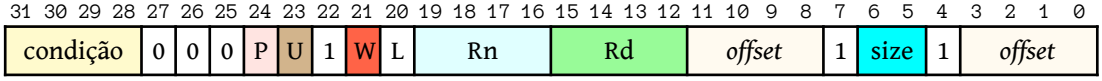


Os campos da instrução são equivalentes aos vistos anteriormente. Um detalhe importante é que não é possível alterar o valor do registrador de índice por intermédio do *barrel shifter*, como podia acontecer na instrução de `load/store` *word*. O campo "*size*" especifica o tamanho da transferência:

size	Transferência
00	Reservado (instrução `swp`)
01	*half word* sem sinal (`ldrh/strh`)
10	*byte* com sinal (`ldrsb/strsb`)
11	*half word* com sinal (`ldrsh/strsh`)

No caso das instruções *half word*, o endereço efetivo (calculado usando o registrador base e o registrador índice) deve ser um **número par**. O bit 6 determina se o sinal do valor lido da memória deve ser estendido no registrador de destino (instruções `ldrsh` e `ldrsb`); o estado do bit 6 é indiferente para instruções `store`.

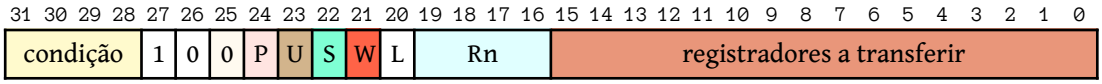
O formato das instruções *half word* usando um *offset* imediato é diferenciado pelo estado do bit 22, conforme ilustrado na figura a seguir:



O comportamento dos diversos campos é idêntico aos das instruções vistas anteriormente. Observe-se que o *offset* imediato possui 8 bits no total, sem sinal, divididos em duas metades: bits mais significativos nos bits 8-11 da instrução e bits menos significativos nos bits 0-3 da instrução.

`load/store` *multiple*

O último tipo de instruções de acesso à memória são as instruções de `load/store` ***multiple***. Essas instruções permitem a escrita ou leitura de diversas posições de memória em sequência, podendo ser utilizadas para a transferência dos valores de diversos registradores.



De forma semelhante às instruções anteriores, os bits "P", "U", "S" e "W" da instrução são usados para especificar o *modo de endereçamento*, conforme a tabela:

P	pré-(1) ou pós-(0) indexar
U	somar com a (1) ou subtrair da (0) base
W	alterar registrador base (1)
S	salvar `spsr` junto com R15 (privilegiado)

Cada um dos bits 0-15 é associado a um dos registradores R0 a R15 (bit 0 a R0, bit 1 a R1, etc.). Quando o valor de um desses bits é "um", o conteúdo do registrador correspondente será transferido da ou para a memória. O bit especial "S" é utilizado para forçar o salvamento ou restauração do registrador `spsr` sempre que o contador de programa (R15) estiver envolvido na transferência, o que é importante para, por exemplo, retornar de um serviço de interrupção – situação na qual o modo de execução anterior do processador deve ser restaurado.

Convencionam-se os formatos das instruções `load` e `store` *multiple* para os programas *assembler* da seguinte forma:

Instrução	Modo	P	U	W
`ldmda r13, {r2-r5, r7}`	Ler e decrementar (*decrement after*)	0	0	-
`ldmia r13, {r2-r5, r7}`	Ler e incrementar (increment after)	0	1	-
`ldmdb r13, {r2-r5, r7}`	Decrementar e ler (*decrement before*)	1	0	0
`ldmdb r13!, {r2-r5, r7}`	Decrementar, ler e atualizar base	1	0	1
`ldmib r13, {r2-r5, r7}`	Incrementar e ler (in*crement before*)	1	1	0
`ldmib r13!, {r2-r5, r7}`	Incrementar, ler e atualizar base	1	1	1
`ldmda r13, {r2-r5, r15}^`	Ler e decrementar (ler `spsr`, S=1)	0	0	-

As instruções `load/store` *multiple* podem ser usadas em pilhas, filas, *buffers*, etc., bem como no processo de cópia de áreas de memória. Embora instruções de acesso múltiplo demandem diversos ciclos de máquina para ser executadas, há ganho significativo de velocidade em relação à transferência do valor de cada registrador individualmente, com uma instrução separada.

Reinício, Exceções e Interrupções

reset

Ao ser ligada a energia ou ao ser detectado o sinal de *reset*, o processador é iniciado no modo *supervisor* (SVR ou o modo mais privilegiado disponível na arquitetura) e o contador de programa (`r15`) é carregado com o valor **zero**. A instrução que se encontra nesse endereço será portanto a primeira instrução a ser executada, seguindo-se a trajetória das instruções a partir dessa primeira instrução.

exceções e interrupções

A sequência das instruções executadas pelo processador pode no entanto ser quebrada por determinados eventos *síncronos* ("exceções") ou *assíncronos* ("interrupções"). Eventos síncronos, causados pela própria execução de uma instrução, ocasionam o abandono da execução dessa instrução (o que pode acontecer em diversos estágios da sua execução) e o *desvio* para um endereço específico da memória; processo semelhante acontece com eventos assíncronos, causados por sinais provenientes de dispositivos *externos* à CPU (por exemplo, periféricos, outros processadores, etc.), denominados **interrupções**. Neste último caso, porém, a instrução corrente é executada até o seu final e somente então o desvio é realizado. Tanto as exceções quanto as interrupções alteram o modo de execução do processador.

vetor de interrupções

Assim como o ponto de entrada do reinício (*reset*), os demais endereços alvo das exceções e interrupções são definidos na área ocupando as 32 primeiras posições da memória física, denominada *vetor de interrupções*:

Endereço	Vetor	Prioridade	Modo	Retorno
0x00000000	*Reset*	1	*SVR*	–
0x00000004	*Undefined Instruction*	6	*UNDEF*	*r14*
0x00000008	*Software Interrupt*	6	*SVR*	*r14*
0x0000000C	*Prefetch abort*	5	*ABORT*	*r14 - 4*
0x00000010	*Data abort*	2	*ABORT*	*r14 - 8*
0x00000014	(não usado)			
0x00000018	*Interrupt*	4	*IRQ*	*r14 - 4*
0x0000001C	*Fast interrupt*	3	*FIQ*	*r14 - 4*

Observe que existem somente quatro posições de memória reservadas para cada vetor, o suficiente para *apenas uma* instrução de máquina do ARM. Sendo assim, na quase totalidade dos casos, essas posições de memória vão conter uma instrução de salto (instrução *branch*) ou de atualização do registrador `r15` (instrução *load*).

prioridades

Essa tabela mostra também relações de *prioridade* (quanto menor o número, mais prioritário): o processador executando no modo usuário pode ser interrompido por qualquer evento, digamos, uma interrupção (prioridade 4), mudando para o modo IRQ. Ainda executando no modo IRQ, o processador poderia ser interrompido por uma interrupção rápida (prioridade 3), e assim por diante. O sinal de *reset* tem a maior prioridade e não pode ser mascarado.

A ocorrência de uma exceção ou interrupção faz com que o processador execute as seguintes operações:

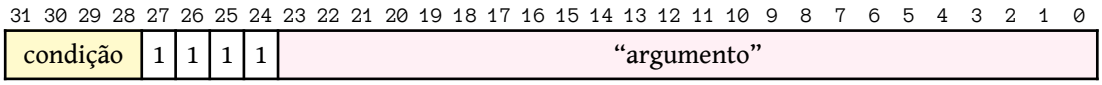
1. No caso de interrupções, a instrução atual é executada até o final, enquanto são inseridos "*nops*" no *pipeline*;
2. O *pipeline* é esvaziado;
3. O estado atual dos *flags* do processador (`cpsr`), incluindo os *bits* especificando o modo atual é salvo no registrador `spsr` e o contador de programa (`r15`) é salvo no registrador `r14`, nas instâncias correspondentes ao modo de destino;
4. O modo do processador é modificado para o modo correspondente ao evento;
5. O contador de programa (`r15`) é carregado com o endereço do vetor correspondente ao evento.

retorno de exceção

Observe-se que o endereço para um eventual retorno ao fluxo original da execução, usando o valor armazenado no registrador `r14` (*link register*), **depende do estágio** no qual o *pipeline* foi interrompido e, como mostra a tabela anterior, o endereço correto para ser escrito no contador de programa varia conforme o **tipo** de evento. O modo de execução original (antes da exceção) é recuperado automaticamente ao ser restaurado o valor do registrador `cpsr`.

software interrupt

A exceção "*software interrupt*" é ocasionada pela execução da instrução `swi`, cujo uso principal é a elevação do nível de privilégio do processador e a execução de algum serviço prioritário, geralmente oferecido pelo Sistema Operacional aos programas executando em modo usuário.



Os bits do argumento (bits `0` a `23`) são ignorados pela arquitetura, mas podem ser analisados pelo serviço de interrupção (lendo a memória com `r14` como registrador base).

aborts

As exceções "*abort*" estão relacionadas com o acesso à memória, tanto a leitura da próxima instrução (*prefetch abort*) quanto a leitura ou escrita de dados (*data abort*). Algumas situações são *erros*, tais como *violações* a proteções de memória previamente definidas, uso de endereços não alinhados corretamente, etc. Podem, contudo, ocorrer em situações corriqueiras, como o acesso a páginas de memória virtual não disponíveis na memória física (ainda não carregadas ou trocadas com a memória secundária): nesses casos, o Sistema Operacional deve corrigir a situação e a instrução que provocou a exceção será executada novamente (e desta vez, supostamente, será bem sucedida).

Instrução indefinida

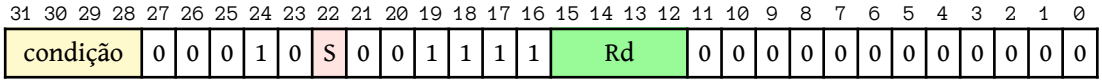
Algumas situações podem ocasionar a exceção de instrução indefinida: caso a decodificação da instrução falhe, com um *opcode* inválido ou com um conjunto de argumentos inconsistente, e, no caso de uma instrução de coprocessador, caso o coprocessador endereçado não esteja presente no sistema. É comum que o tratamento dessa exceção seja utilizado para tentar identificar a instrução desejada e emular a sua execução em software ("máquina virtual"), por

exemplo, a execução de operações de ponto flutuante nas CPUs que não possuam o coprocessador correspondente (coprocessador 10).

Algumas instruções permitem a modificação explícita do modo de execução do processador, por exemplo para ter acesso aos registradores de suas respectivas instâncias e para retornar à execução no modo usuário. O modo de execução do processador é definido pelos bits menos significativos do registrador `cpsr`, que pode ser alterado por instruções privilegiadas.

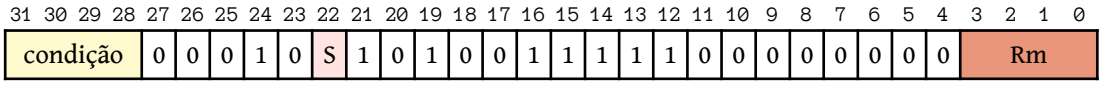
instrução `mrs`

A instrução `mrs` (*move register from psr*) pode ser usada para **ler** o valor atual do registrador `cpsr` ou `spsr` (conforme o valor do *flag* "S" na instrução). Um registrador deve ser especificado como o destino da informação ("Rd"):



instrução `msr`

A instrução `msr` (*move psr from register*) é privilegiada e pode ser usada para modificar o valor atual do registrador `cpsr` ou `spsr` (conforme o valor do *flag* "S" na instrução). O registrador contendo o valor a ser atribuído deve ser especificado pelo campo "Rm":



Coprocessadores

Como é usual entre os processadores RISC, cada unidade de processamento (ou "núcleo") ARM é *modular*, sendo que funções adicionais podem ser incluídas no sistema mediante o uso de *coprocessadores*. Até dezesseis coprocessadores podem existir dentro de um núcleo ARM, servido a finalidades diversas. Alguns desses coprocessadores são padronizados:

coprocessador 15

- Coprocessador 15 ou *coprocessador de sistema* – sempre presente em todas as arquiteturas ARM, é utilizado para configuração geral do núcleo (*clock*, operação dos *caches*, controle de energia, temperatura, temporização principal, controle de interrupções, etc.); nos núcleos que possuem *gerenciamento de memória* (MMU) ou *proteção de memória* (MPU), esses sistemas também são controlados pelo coprocessador 15;

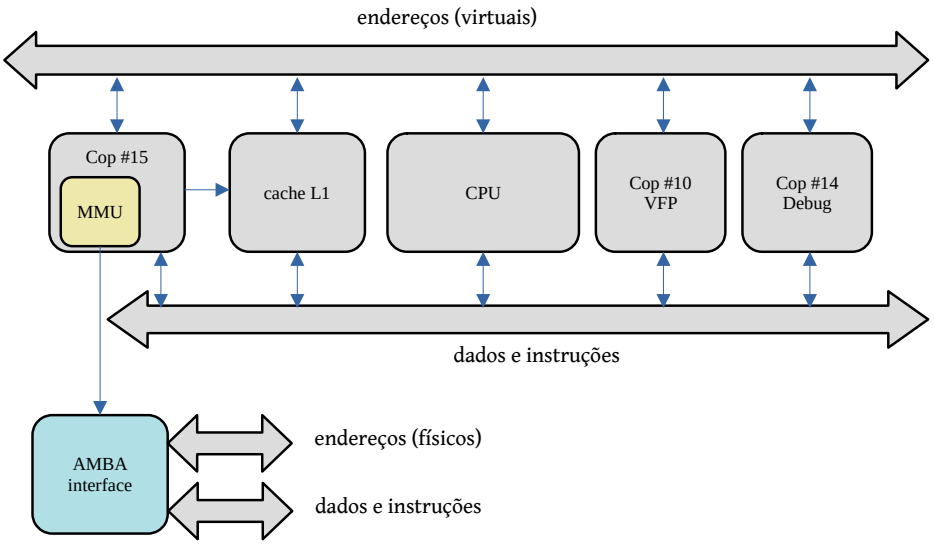
coprocessador 14

- Coprocessador 14 ou "*debug*" – presente na grande maioria dos *chips*, é responsável pelas funções de depuração (*breakpoints*, *watchpoints*, protocolos para troca de informações com depuradores, etc.);

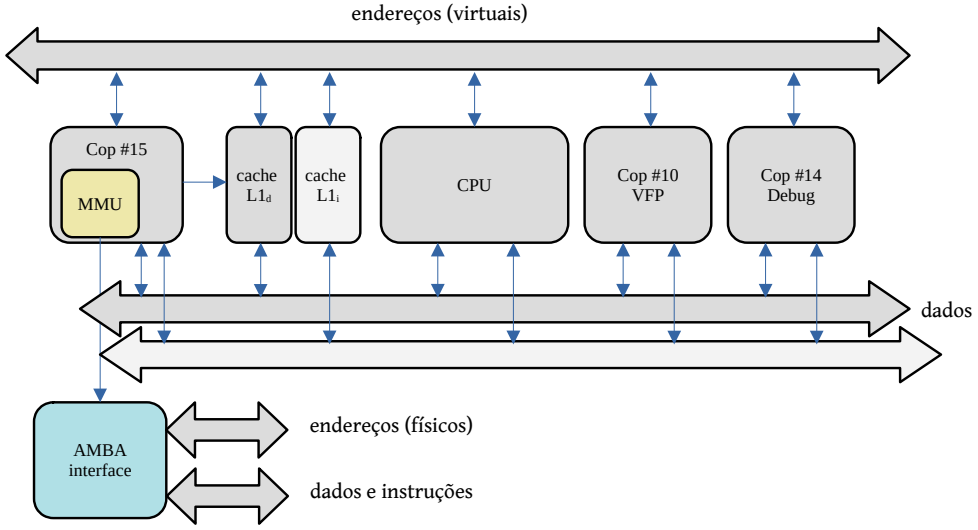
coprocessador 10

- Coprocessador 10 – execução de operações de ponto flutuante em *hardware* (processador "vfpv4"). Trata-se de uma unidade de processamento de ponto flutuante completa, com seu próprio conjunto de registradores: 32 registradores de 32 bits (*single precision*), que podem ser agrupados dois a dois para formar 16 registradores de 64 bits (*double precision*). Os registradores podem ser lidos e escritos de e para os registradores da CPU (`r0` a `r15`), de e para a memória (instruções de `load` e `store`). Instruções específicas (instruções de ponto flutuante) são decodificadas e executadas pelo coprocessador 10, alterando os valores de seus registradores. Operações que afetam múltiplos (ou eventualmente todos) registradores são permitidas (operações vetoriais).

Todos os coprocessadores compartilham os barramentos da CPU principal, podendo ter acesso aos endereços (virtuais), aos dados e instruções que circulam internamente.



organização interna de um núcleo (von Neumann)



organização interna de um núcleo (Harvard)

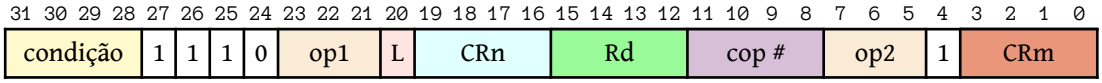
As instruções que envolvem coprocessadores dependem da execução coordenada e da sincronização entre o processador principal e o coprocessador especificado pela instrução. O campo de execução condicional está presente em todas as instruções, inclusive nas instruções de coprocessador: a execução de uma instrução de coprocessador é sempre iniciada pela CPU, após verificada a validade da condição presente na instrução, de acordo com os valores atuais dos *flags* no registrador `cpsr`.

sincronização

Caso a instrução deva ser executada, a CPU usa um sinal específico ($\overline{\text{CPI}}$) para notificar os coprocessadores. Caso o coprocessador, endereçado pelo campo especificado na instrução, não exista no núcleo, o sinal CPA é ativado (coprocessador ausente) – a instrução é, então, considerada *inválida* pela CPU. Caso o sinal CPA indique processador presente, o próprio coprocessador, através do sinal CPB (*coprocessor busy*), informa à CPU se pode executar a instrução: nesta situação, a CPU segue para a busca da próxima instrução na memória, enquanto o coprocessador executa a instrução atual; do contrário, "*nops*" são introduzidos no *pipeline* da CPU até que o sinal CPB seja desligado.

instruções `mrc` e `mcr`

As instruções `mrc` (*move register from coprocessor*) e `mcr` (*move register to coprocessor*) permitem trocar dados entre os registradores da CPU e os registradores de um coprocessador:



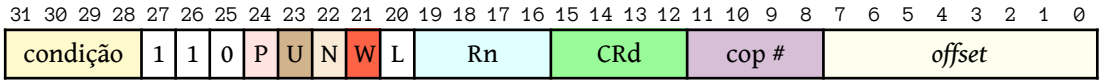
Os campos "condição" e "Rd" são tratados pela CPU, enquanto os demais campos somente têm significado para o coprocessador endereçado pelo campo "cop #" (bits 8-11). Caso a condição para a execução da instrução esteja satisfeita, o valor do registrador especificado pelo campo "Rd" será enviado ao coprocessador (bit "L" igual a "zero": instrução `mcr`) ou será recebido do coprocessador (bit "L" igual a "um": instrução `mrc`).

Geralmente os campos "CRn" e "CRm" especificam registradores internos (operandos) do coprocessador, mas o seu significado é arbitrário. Os campos

"op1" e "op2" são usados para formar diferentes operações a ser realizadas com os operandos e seu formato também é ignorado pela CPU.

instruções `ldc` e `stc`

Os registradores do coprocessador podem também ser lidos e escritos da memória, usando as instruções `ldc` e `stc`, que também permitem utilizar os modos de endereçamento do ARM, à semelhança das instruções `ldrh` e `strh`:



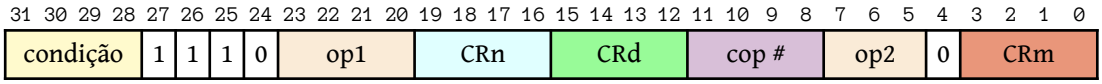
Os bits "P", "U", "N", "B" e "W" da instrução são usados para especificar o *modo de endereçamento*, conforme a tabela:

P	pré-(1) ou pós-(0) indexar
U	somar com a (1) ou subtrair da (0) base
N	Tamanho da transferência ("l" de *long*)
B	transferir byte (1) ou word (0)
W	alterar registrador base (1)

O significado desses bits é equivalente às instruções vistas anteriormente, permitindo o emprego dos modos de endereçamento pré e pós-indexados, atualização de índice e a definição da direção do *offset*. O valor do registrador "Rn" é utilizado como base para o cálculo do endereço efetivo da memória, que será enviado ao coprocessador para a realização da leitura ou escrita, conforme definido pelo bit "L": "zero" – escrita (instrução `stc`) ou "um" – leitura (instrução `ldc`). O campo "CRd" especifica o registrador de origem ou de destino, interno ao coprocessador endereçado pelo campo "cop #".

instrução `cdp`

Finalmente, a instrução genérica de coprocessador (`cdp`) somente tem o campo de execução condicional conferido pela CPU e seu efeito é totalmente dependente do coprocessador responsável por sua execução:



Como regra geral, os campos "op1" e "op2" são utilizados pelo coprocessador para determinar a operação (*opcode*) a ser realizada e os campos "CRd", "CRn" e "CRm" especificam os registradores de destino, do primeiro operando e do segundo operando para a operação, respectivamente, pertencentes ao conjunto de registradores do coprocessador. No entanto, o significado desses campos é arbitrário, e podem ser codificados da forma mais conveniente para o coprocessador. Por exemplo, no caso do coprocessador 10, existem 32 registradores de precisão simples e alguns dos bits do campos de *opcode* são utilizados para complementar os campos "CRd", "CRn" e "CRm".

Conjunto de instruções Thumb

A partir da versão 4 da arquitetura (ARM v.4), os processadores passaram a reconhecer o conjunto de instruções chamado "*thumb*", no qual as instruções são codificadas em 16 bits, com o objetivo de reduzir a quantidade de memória necessária para armazenar um programa ("*footprint*") e aumentar a taxa de execução de instruções ("*throughput*"), usando barramentos mais simples e baratos.

Nos processadores ARM convencionais, cada instrução *thumb* é traduzida em uma instrução ARM equivalente e executada, de forma transparente: muito poucas instruções *thumb* não possuem uma tradução direta em instruções de 32 bits. O processador identifica o conjunto de instruções através de um "modo", que é selecionado através do bit "T", presente no registrador `cpsr`.

seleção do conjunto de instruções

A seleção do conjunto de instruções obedece as seguintes regras:

- Após um *reset* ou qualquer exceção ou interrupção, o modo de execução é sempre ARM (bit "T" igual a "zero");
- Ao *retornar* de uma exceção ou interrupção, o valor **anterior** do bit "T", que é salvo automaticamente em `spsr`, é restaurado e o processador assume novamente o modo de instruções anterior à exceção;
- Normalmente não se deve alterar o valor do bit "T" (por exemplo, usando a instrução `msr`). Para alternar entre os modos ARM e *thumb*, as instruções `bx` (*branch and exchange*) e `blx` (*branch, link and exchange*), que existem em ambos os conjuntos de instrução, podem ser usadas;
- As instruções `bx` e `blx` usam o *bit menos significativo* do endereço efetivo de destino para identificar o conjunto de instruções de destino: caso seja "zero" (endereço **alinhado** em *half-words*), o conjunto será ARM (bit "T" é zerado); caso seja "um" (endereço **desalinhado**), o endereço é corrigido e o conjunto de instruções será o *thumb* (bit "T" ligado).

registradores "*lo*" e "*hi*"

A implementação do conjunto de instruções *thumb* divide o conjunto de registradores em dois grupos: "*lo*", de r0 até r7 e "*hi*", de r8 a r15. A maioria das instruções *thumb* somente permite o acesso aos registradores do primeiro grupo. Algumas instruções especiais permitem manipular registradores do grupo "*hi*", em particular envolvendo o contador de programa (r15) e o *stack pointer* (r13).

microcontroladores ARM

Nos processadores da família Cortex-M (microcontroladores), a arquitetura implementada suporta *somente* o conjunto de instruções *thumb*, geralmente com algumas outras diferenças significativas, com relação aos estados do processador e o acesso a coprocessadores.

A tabela a seguir ilustra as principais diferenças entre os conjuntos de instrução "arm" e "*thumb*".

Característica	ARM	Thumb
Tamanho da instrução	32 bits	16 bits
Formato da instrução	Uniforme	Bastante irregular
Códigos de condição	Todas as instruções	Somente saltos
Atualização dos *flags*	Opcional (bit "S")	Obrigatório[3]
Registradores uso geral	r0-r12	r0-r7[4]
Campos de registrador	Quatro bits	Três bits
Operandos	Endereçamento triplo	Endereçamento duplo
Barrel shifter	Segundo operando ALU	Só instruções de *shift*
Instruções de *shift*	Pseudo-instruções	Instruções
Instruções push/pop	Pseudo-instruções	Instruções
Modos de endereçamento	Pré e pós indexado	Somente pré-indexado
Stack pointer	Qualquer registrador	r13
Entrada em exceção	Sim, sempre	Não
Bit "T" em `cpsr`	zero	um
Instruções `msr/mrs`	Sim	Não
Coprocessadores	Sim	Não
Tamanho de programa	100%	70%
Trajetória de programa	100%	140%
Taxa de execução (32bits)	100%	140%
Taxa de execução (16bits)	100%	145%

3 Os *flags* não são alterados quando a instrução tiver como destino algum dos registradores "*hi*" (r8-r15).

4 Algumas instruções Thumb podem acessar os registradores r8-r15, chamados de registradores altos ("*hi*"). Existem instruções thumb especiais para tratar o contador de programa (r15) e o *stack pointer* (r13).

Instruções de salto

Existem seis tipos de instruções *thumb* para a realização de saltos, que também implicam distâncias máximas diferentes até o endereço de destino:

- Saltos *condicionais* de curto alcance (no máximo 256 *bytes* para frente ou para trás): instruções `beq`, `bne`, `bgt`, etc.;
- Saltos *incondicionais* de "médio" alcance (no máximo 2 kiB para frente ou para trás): instrução `b`;
- Chamadas de sub-rotina de "longo" alcance (no máximo 4 MiB para frente ou para trás). Para que esse alcance "longo" seja conseguido, é utilizada uma sequência de duas instruções *thumb*, ou seja, ocupando 32 bits no total: instrução `bl`;
- Saltos com eventual mudança de conjunto de instrução, baseados em registrador (qualquer posição da memória): `bx` e `blx`;
- Interrupção de software (`swi`) para endereço fixo (0x08), mudando o conjunto de instruções para "arm";
- Saltos baseados na instrução `mov`, a partir do valor de outro registrador (qualquer posição da memória).

A tabela a seguir mostra os formatos das instruções de salto:

Instrução	Operação	Exemplos
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 0 1 | cond | *offset* (com sinal)	se condição pc←pc+2**offset*	`beq #10` `bmi #-4`
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 1 0 0 | *offset* (com sinal)	pc←pc+2**offset*	`b #512` `b #-8`
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 1 1 H | *offset* (com sinal)	pc←pc+2**offset* *composto* lr←$pc_{anterior}$+3	`bl #0x10000`
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 1 0 0 0 1 1 1 0 H | Rm | 0 0 0	pc←Rm, T←Rm[0] H = 1 para r8-r15	`bx r0` `bx r14`
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 0 1 1 1 1 1 | i	pc←0x08	`swi #50`

offsets

O *offset* para as instruções de salto pode ter oito bits (saltos condicionais), onze bits (saltos incondicionais) ou vinte e dois bits (sub-rotinas), sempre com sinal; esse valor de *offset* é multiplicado por dois (as instruções *thumb* têm 16 bits) para o cálculo do endereço efetivo do salto.

instrução `bl`

A instrução `bl` (*branch and link*) precisa de duas instruções *thumb* consecutivas, a primeira com o bit "H" igual a "zero" e a segunda com o bit "H" igual a "um". O registrador r14 (*link register*) é utilizado como rascunho para montar o *offset* para o contador de programa: na primeira instrução (H=0), o registrador r14 armazena os doze bits *mais significativos* do destino; a segunda instrução (H=1) atualiza r14 com os doze bits *menos significativos* e esse valor é então somado a r15.

instrução `bx`

A instrução `bx` (*branch and exchange*) carrega o contador do programa com o valor de qualquer outro registrador. Como o campo "Rm" possui apenas três bits, o bit "H" é usado para definir o bit mais significativo do índice do

registrador: “zero” para os registradores “*lo*” (r0-r7) e “um” para os registradores “*hi*” (r8-r15). O conjunto de instruções, após a execução do salto, é definido pelo bit menos significativo do endereço de destino: se for “um” (endereço desalinhado), o modo continua sendo *thumb*; se for “zero” (endereço alinhado), o conjunto de instruções muda para ARM.

saltos condicionais

Os saltos condicionais são as únicas instruções *thumb* que verificam o estado dos *flags* “N”, “V”, “C” e “Z” do registrador `cpsr`. O campo de condição na instrução segue o mesmo padrão das condições das instruções do conjunto “arm”:

Condição	Instrução	*Flags*	Condição	Instrução	*Flags*
0000	`beq`	Z = 1	0111	`bvc`	V = 0
0001	`bne`	Z = 0	1000	`bhi`	C=1, Z=0
0010	`bcs`	C = 1	1001	`bls`	C=0, Z=1
0011	`bcc`	C = 0	1010	`bge`	N = V
0100	`bmi`	N = 1	1011	`blt`	N ≠ V
0101	`bpl`	N = 0	1100	`bgt`	Z=0, N=V
0110	`bvs`	V = 1	1101	`ble`	Z=1, N≠V

Operações aritméticas

A grande maioria das instruções *thumb* envolvendo a ALU somente têm acesso aos registradores do grupo "*lo*" (registradores r0 até r7), uma vez que os campos para a especificação dos índices dos registradores de destino ou de operando possuem apenas três bits. Além disso, a maioria das instruções suporta o *duplo endereçamento*: um registrador é ao mesmo tempo operando e destino do resultado da operação. O conjunto de instruções *thumb* não suporta operações envolvendo o processamento de um operando pelo *barrel shifter*: as operações de deslocamento são efetuadas por instruções dedicadas (`lsl`, `lsr` e `asr`).

duplo endereçamento

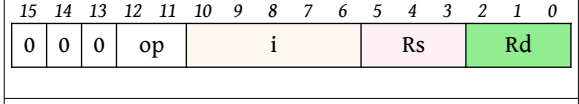
As instruções possíveis para cada combinação de operandos são:

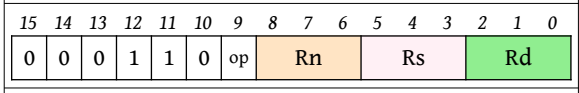
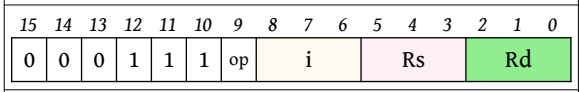
instruções válidas

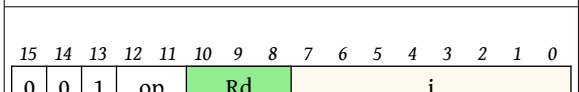
- Triplo endereçamento com três registradores "*lo*": `add` e `sub`;
- Triplo endereçamento com dois registradores "*lo*" e um valor imediato de cinco bits: `lsl`, `lsr` e `asr`;
- Triplo endereçamento com dois registradores "*lo*" e um valor imediato de três bits: `add` e `sub`;
- Duplo endereçamento com um registrador "*lo*" e um valor imediato de oito bits: `mov`, `cmp`, `add` e `sub`;
- Acesso ao contador de programa (r15) e *stack pointer* (r14): `add`;
- Acesso aos registradores "*hi*", com duplo endereçamento: `mov`, `cmp` e `add`;
- Duplo endereçamento com dois registradores "*lo*": todas as instruções.

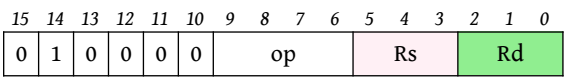
A tabela a seguir mostra os formatos das instruções lógicas e aritméticas do conjunto de instruções *thumb*:

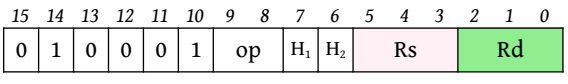
Instrução	Operação	Exemplos

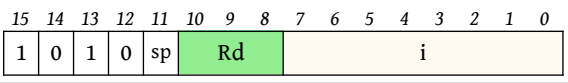
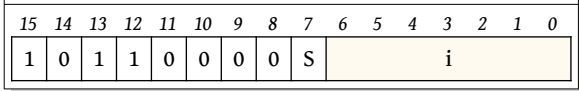
	rd←rs<<i (op=00)
	rd←rs>>i (op=01)
	rd←rs>>i (op=10) (arit.)
		`lsl r0,r1,#3`
		`lsr r2,r2,#1`
		`asr r4,r0,#16`

	rd←rs+rn (op=0) rd←rs-rn (op=1)	`add r0,r1,r2` `sub r1,r3,r4`
	rd←rs+i (op=0) rd←rs-i (op=1)	`add r0,r1,#1` `sub r1,r3,#7`

	rd←i (op=00)
	compara rd e i (op=01)
	rd←rd+i (op=10)
	rd←rd-i (op=11)
		`mov r0, #0`
		`cmp r3, #100`
		`add r0, #1`
		`sub r1, #1`

	rd←rd <op> rs	`lsl r0,r1`
		`mul r1,r2`
		`bic r3, r7`

	rd←rd+rs (00)
	compara rd e rs (01)
	rd←rs (10)
	rd(rs)→r0-r7 (H=0) ou
	rd(rs)→r8-r15 (H=1)
		`add r0, r10`
		`cmp r8, r9`
		`mov pc, lr`

	rd←pc+4*i (sp=0) rd←sp+4*i (sp=1)	`add r0,pc,#4` `add r0,sp,#8`
	sp←sp+4*i (s=0) sp←sp-4*i (s=1)	`add sp,#4` `add sp,#-8`

valores imediatos

Os valores imediatos podem ter três bits (`add` e `sub`), cinco bits (`lsl`, `lsr` e `asr`) ou oito bits (`mov`, `cmp`, `add` e `sub`), sendo sempre sem sinal. A instrução "`add sp`" possui um bit extra para definir o sinal (não é complemento de dois!). As instruções `add`, `cmp` e `mov` com dois registradores são as únicas que podem acessar os registradores "*hi*": para isso, elas dispõem de dois bits extra para complementar o índice dos registradores.

A maioria das instruções permite operar sobre dois registradores (duplo endereçamento), salvando o resultado em um deles (`Rd`):

instruções lógicas e aritméticas

opcode	Instrução	Operação	opcode	Instrução	Operação
0000	`and`	rd←rd & rs	1000	`tst`	rd&rs, seta *flags*
0001	`eor`	rd←rd ^ rs	1001	`neg`	rd←-rs
0010	`lsl`	rd←rd << rs	1010	`cmp`	rd-rs, seta *flags*
0011	`lsr`	rd←rd >> rs	1011	`cmn`	rd+rs, seta *flags*
0100	`asr`	rd←rd >> rs	1100	`orr`	rd←rd | rs
0101	`adc`	rd←rd+rs+C	1101	`mul`	rd←rd * rs
0110	`sbc`	rd←rd-rs-C	1110	`bic`	rd←rd & (~rs)
0111	`ror`	rd←rd<<@rs	1111	`mvn`	rd←~rs

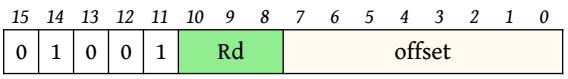
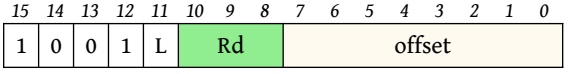
Instruções de acesso à memória (Load/Store)

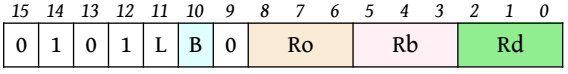
As instruções `load` e `store` do conjunto *thumb* usam um registrador como **base** e o modo de endereçamento pré-indexado, que pode envolver um segundo registrador ou um valor constante (*offset*). O contador de programa (r15) e o *stack pointer* (r13) são os únicos registradores "*hi*" que podem ser usados como registrador base.

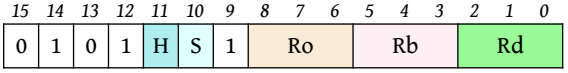
As instruções `load` e `store` no modo *thumb* são de seis tipos:

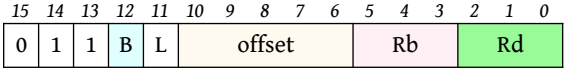
instruções válidas

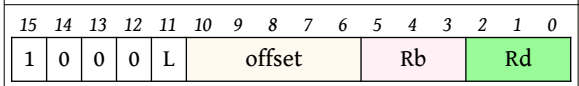
- Carga de uma palavra (32 bits) de uma posição de memória relativa ao contador de programa (*offset* imediato de oito bits)- somente `ldr`;
- Carga ou salvamento de palavra (32 bits) em uma posição de memória relativa ao *stack pointer* (*offset* imediato de oito bits)- `ldr` e `str`;
- Carga ou salvamento de dados (32, 16 ou 8 bits) em um posição de memória relativa a um registrador base (*offset* imediato de cinco bits)– `ldr`,`str`,`ldrb`,`strb`,`ldrh`,`strh`;
- Carga ou salvamento de dados (32, 16 ou 8 bits) em um posição de memória calculada com o valor de dois registradores (base e índice)- todas as instruções: `ldr`,`str`,`ldrb`,`strb`,`ldrsb`,`strsb`,`ldrh` e `strh`;
- Carga ou salvamento de *múltiplos* registradores na **pilha**- `push` e `pop`;
- Carga ou salvamento de *múltiplos* registradores "*lo*" a partir de um endereço base: `ldmia` e `stmia`.

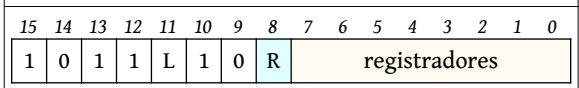
Instrução	Operação	Exemplos
	rd←[pc+4*offset]	`ldr r0,[pc,#4]`
	rd←[sp+4*offset] [sp+4*offset]←rd	`ldr r0,[sp,#4]` `str r0,[sp,#8]`

	rd←[rb+ro] (L=1)
	[rb+ro]←rd (L=0)
		`ldr r0,[r1,r2]`
		`ldrb r0,[r1,r2]`
		`str r0,[r1,r2]`

	rd←[rb+ro]
	[rb+ro]←rd
	(byte/word com ou sem sinal)
		`strh r0,[r1,r2]`
		`ldrh r0,[r1,r2]`
		`ldsb r0,[r1,r2]`
		`ldsh r0,[r1,r2]`

	rd←[rb+n*offset]
	[rb+n*offset]←rd
		`str r0,[r1,#10]`
		`ldr r0,[r1,#10]`
		`ldrb r0,[r1,#10]`

	rd←[rb+2*offset] [rb+2*offset]←rd	`strh r0,[r1,#10]` `ldrh r0,[r1,#10]`

	pilha←{regs} (L=0)
	{regs}←pilha (L=1)
	R = inclui pc/lr
		`push {r0-r3}`
		`push {r0-r3, lr}`
		`pop {r0-r3, pc}`

	[rb]←{regs} (L=0) {regs}←[rb] (L=1) rb é alterado	`stmia r5!,{r0-r3}` `ldmia r5!.{r0-r3}`

offsets imediatos

Os *offsets* imediatos podem ter cinco bits (somando a um registrador base entre r0 e r7) ou oito bits (somando ao registrador r15 ou ou registrador r13), sendo sempre sem sinal. Os *offsets* são multiplicados pelo tamanho dos dados a acessar: quatro vezes para palavras (32 bits) e duas vezes para *half-words* (16 bits). As instruções que têm r15 ou r13 como base somente podem acessar palavras (32 bits) na memória. As demais instruções possuem versões para leitura de *bytes* (bit "B") e *half-words* (bit "H"), com ou sem extensão de sinal (bit "S").

load/store multiple

As instruções `push`, `pop`, `ldmia` e `stmia` permitem transferir vários registradores em sequência e sempre atualizam o valor do registrador base. Os únicos registradores que podem ser acessados são os registradores "lo" (r0 a r7), com a exceção dos registradores pc (r15) e lr (r14) no caso das instruções de pilha, quando o bit "R" vale "um": `push` empilha r14 e `pop` desempilha seu valor em r15.

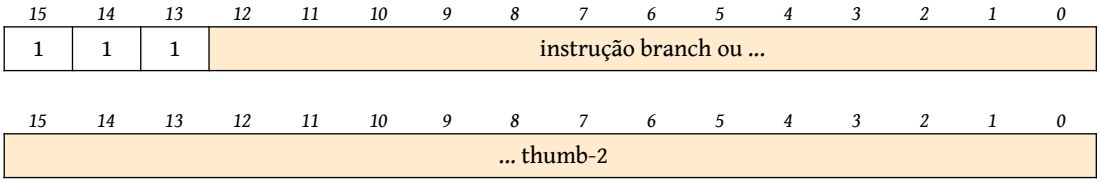
Conjunto de instruções Thumb-2

O conjunto de instruções *thumb* possui muitas vantagens em termos de otimização de uso de memória e velocidade de execução, porém, comparado ao conjunto de instruções original dos processadores ARM, é bastante limitado em recursos importantes, como o uso da *barrel shifter* em operações aritméticas, o triplo endereçamento de registradores e a limitação no tamanho das constantes imediatas e *offsets*.

Uma revisão do conjunto de instruções *thumb* foi introduzida na versão 6 da arquitetura (ARMv.6), chamada de "*thumb-2*". Esse conjunto de instruções possui tanto instruções de 16 bits (compatíveis com o *thumb* original) quanto de 32 bits, construídas a partir de **duas instruções** de 16 bits sucessivas, reutilizando a ideia empregada na instrução `bl`.

instruções de 32 bits

A primeira parte da instrução permite diferenciar entre *thumb* e *thumb-2* através dos bits mais significativos:



As instruções codificadas nos 29 bits do *thumb-2* permitem complementar o *thumb* com a maioria dos recursos existentes na arquitetura ARM:

- Campos imediatos maiores;
- Endereçamento do conjunto completo de registradores (campos de quatro bits);
- Triplo endereçamento (três campos para registradores);
- Uso do *barrel shifter* sobre operandos de instruções aritméticas;
- Instruções para tratamento do `cpsr` e de coprocessadores;

- Outras instruções do ARMv.6 que não eram suportadas pelo *thumb* original.

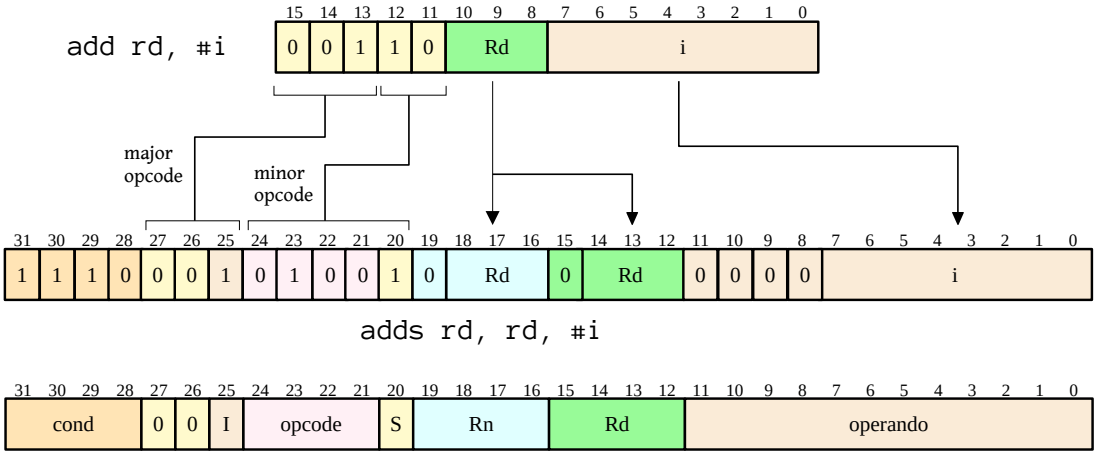
Para que o ganho de desempenho seja mantido, é necessário que a maioria das instruções geradas pelo compilador possa ser montada usando o conjunto de instruções *thumb* (16 bits), incluindo as instruções maiores somente em casos particulares.

Uma das características mais marcantes do conjunto de instruções ARM, a existência de campos condicionais em **todas** as instruções, é perdida no conjunto *thumb*. Para compensar essa deficiência, o conjunto *thumb-2* introduz uma instrução especial ("ifelse") que permite definir trechos de programa com execução condicional sem a necessidade de saltos.

instrução ifelse

Execução do conjunto de instruções Thumb

Os processadores ARM normalmente *interpretam* as instruções *thumb* ("máquina virtual *thumb*"), realizando buscas ("*fetch*") de 16 bits no primeiro estágio do *pipeline*. Essa interpretação é relativamente direta, considerando execução incondicional para as instruções e mapeando os campos *thumb* nos campos equivalentes de uma instrução ARM regular, eventualmente complementando com zeros.



Por outro lado, em arquiteturas especiais, como o Cortex-M (microcontroladores), o processador é projetado e otimizado para executar diretamente as instruções *thumb* (sem um processo de "tradução"), que passa a ser o único conjunto de instruções disponível.

Instalação do ambiente de desenvolvimento

Os pacotes de software que vamos utilizar são:

- `binutils-arm-none-eabi`
- `gcc-arm-none-eabi`
- `gdb-multiarch`
- `qemu-system-arm`
- `make`

Os softwares estão previamente instalados nos computadores do laboratório. Para instalá-los em seu computador, normalmente basta usar a ferramenta de instalação (`apt` no caso do Debian e do Ubuntu, `pacman` no ArchLinux, etc.), uma vez que a maioria dos repositórios já inclui essas ferramentas previamente compiladas para instalação. Esses são os nomes dos pacotes no repositório Debian, pode ser que em outras distribuições eles tenham outros nomes.

```
sudo apt install binutils-arm-none-eabi gcc-arm-none-eabi
sudo apt install gdb-multiarch qemu-system-arm make
arm-none-eabi-as -v  # versão do assembler
```

O ambiente de desenvolvimento GNU

O projeto "GNU" – que é a abreviação recursiva de "*Gnu is Not Unix*" – começou na década de 80 com o trabalho de Richard Stallman: seu objetivo era a criação de um Sistema Operacional completo, totalmente gratuito e sem nenhuma restrição de uso (*free-software* e *open-source*), baseado no Unix. Já que o próprio nome "Unix" era disputado como marca registrada, o sistema operacional foi chamado sarcasticamente de "*not* Unix".

Um dos primeiros impasses para o novo sistema viria a ser o seu próprio ambiente de desenvolvimento: não fazia sentido usar montadores, compiladores e vinculadores (*linkers*) comerciais – cuja licença era *fechada* – para o desenvolvimento de um software aberto. Qualquer usuário que pretendesse reconstruir o sistema a partir de seu código-fonte precisaria adquirir legalmente uma licença desses softwares, o que violaria diretamente a filosofia original.

Sendo assim, Stallman e sua equipe começaram a desenvolver *suas próprias ferramentas*, entre as quais um montador (`as`), vinculador (`ld`), arquivador (`ar`), compilador `C` e suas respectivas bibliotecas (`gcc`), depurador (`gdb`) e até um editor de texto (`emacs`). Hoje em dia essas ferramentas já foram adaptadas para praticamente todas as arquiteturas e sistemas operacionais, permitem compilar diversas linguagens de alto nível (de fato, "gcc" passou a ser a abreviatura de "*gnu compiler collection*") e foram utilizadas para o desenvolvimento de vários sistemas operacionais, entre eles o FreeBSD, Darwin (MacOS), Linux além, é claro, do próprio Hurd (que é o núcleo do sistema operacional Gnu).

De um modo geral, esses softwares produzem e analisam código para a *mesma* arquitetura que os estão executando: em um computador AMD-64 vão gerar e analisar instruções características do conjunto de instruções dessa arquitetura. No entanto, a linguagem de máquina de destino é arbitrária e poderia, eventualmente, corresponder a uma arquitetura *diferente* da arquitetura da máquina que executa esses programas: vamos chamar a máquina que executa o ambiente de desenvolvimento de *hospedeira* ("*host*") e a máquina que executará as instruções produzidas de *alvo* ("*target*"). No caso em que as arquiteturas hospedeira e alvo são a mesma, esse processo é chamado de *desenvolvimento nativo*; quando são arquiteturas diferentes, chamaremos de *desenvolvimento cruzado*.

Arquiteturas hospedeira e alvo

No caso particular do **`gcc`** e programas associados, desenvolvimentos cruzados são sempre tornados explícitos por uma nomenclatura especial dos aplicativos: o seu *prefixo*. Assim, o programa chamado "**`as`**" sempre produz código para a mesma arquitetura da máquina hospedeira (*assembly* nativo); um programa (diferente), cujo nome será algo como "**`mips-linux-gnueabi-as`**", vai produzir código para ser executado por uma arquitetura MIPS. A parte

Prefixo

"mips-linux-gnueabi-" é chamada de *prefixo*[5] e sempre vai identificar a produção de código para uma arquitetura diferente da arquitetura da máquina hospedeira.

O Emulador de Arquiteturas (QEMU)

O software *Quick Emulator* ou QEMU foi desenvolvido como software livre por Fabrice Bellard[6]. Sua principal função é imitar ou emular a execução de instruções de diferentes processadores, independentemente da máquina que o executa (denominada *máquina hospedeira* ou *host*), através de *tradução binária*: processo no qual uma ou mais instruções de máquina do computador hospedeiro são usadas para produzir um efeito *equivalente* ao da execução de uma instrução de uma arquitetura diferente (ou eventualmente da *mesma* arquitetura, mas em uma *máquina virtual* independente). O *Quick Emulator* também é utilizado para a criação de *máquinas virtuais*, emulando o funcionamento de um *hardware* eventualmente diferente do equipamento real.

Sufixo

De forma equivalente ao objetivo dos *prefixos* do Gnu, são os *sufixos* que identificam a arquitetura emulada por QEMU. Por exemplo, o programa de nome "**`qemu-arm`**" vai emular a execução de código que deve conter instruções do processador ARM, mesmo que a máquina local não tenha um processador compatível. Entre as arquiteturas que podem ser emuladas pelo `qemu` estão Intel x86, AMD-64, ARM, ARM-64, MIPS, MIPS-64, PowerPC, Sparc, e várias outras.

Integração com gdb

O modelo geral de desenvolvimento que vamos utilizar inicialmente envolve compilar ou montar programas contendo instruções do processador ARM usando o ambiente de desenvolvimento cruzado (**`arm-none-gnueabi-gcc`**) e executá-los nos computadores de arquitetura Intel com o emulador `qemu` (**`qemu-system-arm`**); vamos observar o comportamento da execução emulada através do depurador Gnu, **`gdb`** (**`gdb-multiarch`**). Para isso, os processos do emulador `qemu` e do depurador vão se conectar por um *socket*: comandos podem ser enviados do depurador para o emulador, que serão respondidos, eventualmente interrompendo a execução do programa emulado para a análise dos valores dos registradores, memória, etc.

O emulador `qemu-system-arm` cria uma máquina virtual com recursos de hardware que podem ser definidos na sua linha de comando. No nosso caso, precisamos apenas de um mínimo de recursos.

5 Preste atenção que o *prefixo* traz outras informações, além da arquitetura da máquina alvo, a saber: o sistema operacional alvo e o formato de interface binária (ABI), que informa, entre outras coisas, como diferentes tipos de dados são representados na memória e como parâmetros são passados para as funções nessa arquitetura.

6 O QEMU é o principal software de virtualização no Linux, com suporte ao *driver* KVM (*kernel-based virtual machine*). Você provavelmente já conhece vários outros programas que foram desenvolvidos por Bellard, tais como o FFmpeg, QuickJS, TinyC *compiler* e vários softwares para compactação de dados.

No exemplo a seguir, criamos uma máquina virtual genérica (tipo "virt" com a opção `-M`) e carregamos o código objeto do arquivo `kernel.elf` (como se fosse o Sistema Operacional). A opção `-s` permite ativar a depuração através da porta tcp/ip 1234, com a qual podemos conectar com o depurador (gdb), o que fizemos através do comando `target extended-remote`:

```
qemu-system-arm -M virt -s -kernel kernel.elf &
# kernel.elf será carregado na memória
# stub gdb na porta 1234 (opção -s)
gdb-multiarch kernel.elf
(gdb) set architecture arm
(gdb) target extended-remote :1234
(gdb) load
# etc...
```

Você pode usar um `Makefile` como o seguinte para automatizar o processo de desenvolvimento[7]:

Makefile padrão

```
# Makefile
# Liste os arquivos fonte aqui:
FONTES = file1.s file2.s file3.s
# Você pode usar FONTES = $(wildcard *.s)
# todos os arquivos .s serão incluídos: cuidado!

# Arquivos de saída
EXEC = kernel.elf
MAP = kernel.map

PREFIXO = arm-none-eabi-
LDSCRIPT = kernel.ld
AS = ${PREFIXO}as
LD = ${PREFIXO}ld
OBJETOS = $(FONTES:.s=.o)

# Alvo: gerar executável
${EXEC}: ${OBJETOS}
	${LD} -T ${LDSCRIPT} -M=${MAP} -o $@ ${OBJETOS}

# Alvo: montar arquivos em assembler
.s.o:
	${AS} -g -o $@ $<

# Alvo: limpar tudo
clean:
	rm -f *.o ${EXEC} ${MAP}

# Alvo: iniciar qemu
qemu: ${EXEC}
	@if lsof -Pi :1234 >/dev/null ; then\
	   echo "qemu já está executando"; \
	else qemu-system-arm -s -M virt -kernel ${EXEC} & \
	fi

# Alvo: iniciar gdb
gdb: ${EXEC}
	gdb-multiarch -ex "set architecture arm" \
	              -ex "target extended-remote :1234" \
	              -ex "load"
	              ${EXEC}
```

linker script

O *linker* precisa do arquivo `kernel.ld` ("*linker script*", usando a opção `-T`) para especificar o **mapa de memória** da máquina virtual. A máquina

7 **Importante:** o `make` usa uma sintaxe antiga e precisa que os caracteres de tabulação sejam realmente '*tabs*' (código ASCII 0x09 ou '\t') e não espaços. Seu editor de texto pode não saber disso.

"virt" é bastante simples, e para nós só interessa que a memória RAM começa no endereço 0x40000000. O arquivo `kernel.ld` pode ser, simplesmente:

```
/* Arquivo kernel.ld: linker script */
SECTIONS {
  .text 0x40000000 : {
    *(.text)
  }

  .data : {
    *(.data)
  }
}
```

Com o `Makefile` anterior, você pode executar os alvos:

- `make` – monta os fontes com o `gnu-as` e gera o arquivo binário executável de saída (`kernel.elf`, no exemplo) com o `gnu-ld`. Gera também um arquivo de mapa do *linker* (`kernel.map`);
- `make qemu` – monta o executável de saída e carrega `kernel.elf` com a máquina virtual `qemu` (caso ainda não esteja em execução)
- `make gdb` – monta o executável de saída e executa o `gdb` junto com os comandos "*set architecture*" e "*target*" (a máquina virtual `qemu` precisa estar em execução), recarregando `kernel.elf`;
- `make clean` – apaga todos os arquivos intermediários (para forçar a remontagem do zero).

Um exemplo completo

Use o arquivo fonte seguinte, no mesmo diretório do `Makefile` e do `kernel.ld`. Altere a primeira linha do `Makefile` para incluir o nome do arquivo (`teste.s`).

```
// Arquivo teste.s
.global start
.text
start:
  ldr r0, =0x12345678
  b start
```

Ao executar os comandos com o `make`, você deve obter algo do tipo:

```
bruno/temp> make
arm-none-eabi-as -g -o teste.o teste.s
arm-none-eabi-ld -T kernel.ld -M=kernel.map -o kernel.elf teste.o
bruno/temp> make qemu
bruno/temp> make gdb
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/...
Reading symbols from kernel.elf...
The target architecture is set to "arm".
Remote debugging using :1234
start () at teste.s:4
4       ldr r0, =0x12345678
(gdb) ni
5       b start
(gdb) p/x $r0
$1 = 0x12345678
(gdb)
```

GNU assembler (as)

O formato geral do comando do *assembler* é

```
as [opções] <arquivos-fonte>
```

Quando se utiliza a compilação cruzada, é introduzido o *prefixo* correspondente à arquitetura, sistema operacional e ABI do sistema alvo, por exemplo **`arm-none-gnueabi-as`**.

Arquivos-fonte

Um ou mais arquivos-fonte devem ser especificados na linha de comando e serão processados em ordem. Os arquivos-fonte são arquivos texto com a extensão `.s`, `.src`, `.a` ou `.asm`, sendo recomendada a extensão `.s`.

Arquivos objeto e listagem

O montador vai produzir como resultado um arquivo objeto binário realocável, que normalmente é processado pelo vinculador ("*linker*"). O **nome** do arquivo objeto pode ser especificado com a opção `-o`; caso essa opção não esteja presente, será gerado um arquivo de nome "a.out". O *assembler* pode gerar também um arquivo texto contendo a **listagem** (opção `-a`), que pode ser bastante útil durante o processo de desenvolvimento.

Informações de depuração

Normalmente, o *assembler* não inclui informações para depuração (descrição dos símbolos, números de linha, etc.) no arquivo objeto produzido; essas informações são utilizadas, por exemplo, por depuradores e analisadores de código. Durante o processo de desenvolvimento é interessante incluir a opção `-g` na linha de comando para que essas informações sejam incluídas.

Opções comuns

`-o` `<arquivo>`	Define o arquivo objeto (saída)
`-a`	Gera listagem em `stdout`
`-a=``<arquivo>`	Define o arquivo de listagem
`-g`	Inclui informações de depuração
`-I` `<diretório>`	Diretório para procurar `.include`
`--def-sym` `<simbolo>`*`[=<valor>]`*	Define um símbolo
`-mthumb`	Permitir instruções de 16 bits (T16)
`-march=``<arquitetura>`	Versão da arquitetura (`armv4`, `armv7`, ...)
`-EL`	Usar ordenação *little-endian*
`-EB`	Usar ordenação *big-endian*

Formato dos arquivos-fonte

Os arquivos-fonte do *assembler* contém mnemônicos de instruções e pseudo-instruções do processador alvo, nomes de registradores, diretivas, símbolos, expressões, comentários e informações para uso do *linker*. Um arquivo-fonte pode incluir outros arquivos com a diretiva `.include` e pode conter *macros* que são automaticamente expandidas durante o processo de montagem.

Comentários

Vários estilos de comentários são suportados pelo `gnu-as`:

```
@ comentário padrão do assembler ARM
;@ ou isso, para ajudar os editores de texto
# comentário padrão sh (o # deve ser o primeiro não branco)
/* comentário estilo C */
// comentário estilo C++
```

Símbolos

Símbolos

Símbolos são sequências de caracteres alfanuméricos, incluindo o ponto (“.”), o cifrão (“$”) e a barra de sublinhar (“_”), com a restrição de não começar com um dígito. Os símbolos no *assembler* são *sensíveis ao caso*, ou seja, “`loop`”, “`Loop`” e “`LOOP`” são todos símbolos diferentes.

O **valor** de um símbolo é um *número* (de 32 ou 64 bits, dependendo da arquitetura alvo), que geralmente representa um endereço ou uma constante. Símbolos não definidos são considerados com o valor (temporário) “zero”: eventualmente o *linker* tentará “corrigir” esses valores. Observe-se que todos os endereços para o *assembler* são **relativos ao primeiro endereço da seção** onde estão declarados e são sempre **realocados** pelo *linker*.

Rótulos

Um símbolo pode ser declarado como um sendo **rótulo** (“*label*”), que aponta um **endereço**, bastando para isso ser seguido por dois pontos `:`, por exemplo “`loop:`”, “`fim:`”. Símbolos também podem ser **atribuídos** usando `=` ou através das diretivas `.set` ou `.equ` com o resultado de uma **expressão**.

Endereço atual (“.”)

O símbolo `.` representa o **endereço atual** (sempre em relação ao primeiro endereço da seção atual); ele pode ser **atribuído**, o que tem efeito semelhante às diretivas `.org` e `.space`.

Expressões

Expressões

O `gnu-as` é capaz de processar **expressões** envolvendo operadores equivalentes aos da linguagem `C`, com as mesmas regras de precedência, incluindo o aninhamento de parênteses. O valor das expressões é avaliado em tempo de montagem, devendo resultar em um valor constante. Uma expressão vazia é avaliada com o valor zero. Observe-se que o resultado de uma expressão que contenha operadores de comparação (por exemplo, `==` ou `>=`) ou operadores lógicos (`&` ou `|`) será “zero” para significar “falso”, mas pode resultar em *qualquer outro valor* para significar “verdadeiro”.

Constantes

As expressões podem conter contantes numéricas, no mesmo formato da linguagem C:

- Decimal: `116`
- Hexadecimal: `0x74`
- Octal: `0164`
- Binário: `0b1110100`
- Caractere: `'t'`

Diretivas ou "pseudo-ops"

Diretivas e parâmetros

Diretivas são comandos para o *assembler*, e sempre começam com um um **ponto**. Uma diretiva pode ter um ou mais **parâmetros**; caso possua mais do que um parâmetro, eles devem ser separados por **vírgulas**.

.include

A diretiva `.include` permite incluir um arquivo, cujo nome deve aparecer entre aspas duplas (*string*). Os diretórios onde os arquivos a incluir são procurados podem ser especificados pela opção `-I` na linha de comando do *assembler*.

```
.include "declr.inc"   // inclui arquivo com declarações
```

Definição de Seções

Seções

Seções são áreas de memória tratadas como unidades pelo *linker* e *loader*, geralmente associadas a determinadas funções (como ".text" para conter código executável e ".data" para conter informações modificáveis) e que serão alocadas a endereços absolutos de memória contíguos e com determinadas propriedades (por exemplo, ".text" em uma região de memória com permissão de execução ou na memória *flash* de programa de um microcontrolador). O *assembler* pode especificar o nome da seção na qual instruções ou dados serão inseridos, porém não tem controle sobre o endereçamento absoluto das seções, cuja atribuição é função do *linker*.

.text
.data
.section

As diretivas `.data` e `.text` informam ao *assembler* que os próximos endereços estão nessas seções, respectivamente. Um valor numérico após a diretiva permite definir uma subseção. A diretiva `.section` pode especificar uma seção de nome arbitrário (além de `.data` e `.text`).

```
.text
.section .text   ;@ equivalente
ldr r0, =a
ldr r0, [r0]

.data
a: .word
```

Atribuição de Símbolos

As diretivas `.equ` e `.set` permitem atribuir o resultado de uma **expressão** a um **símbolo**:

.equ
.set

```
.set TRUE, 1    // formas equivalentes para dar um valor
.equ FALSE, 0   // a um símbolo
ON = 1
```

A diretiva `.global` **exporta** um símbolo para que seu valor seja conhecido pelo *linker*; observe que não é necessário importar símbolos no *assembler*.

Introdução de dados

Um conjunto arbitrário de *bytes* pode ser inserido no código objeto pelo *assembler* usando as diretivas com nomes de tipos (`.byte`, `.word`, `.hword`, `.int`, `.float`, etc.). Esses comandos alocam espaço suficiente para o tipo especificado no código objeto na posição especificada pelo endereço atual, na seção atual. Caso tenham parâmetros (uma lista de expressões), os valores são incluídos na memória em sequência, de acordo com o tipo especificado. Observe que o tamanho dos "*word*" e "*hword*" depende da arquitetura alvo.

As diretivas no formato `.dc.b`, `.dc.l`, `.dc.w`, `.dc.s`, etc., são semelhantes (*byte*, *long*, *word* e *single precision*, respectivamente). Neste formato, "*word*" sempre se refere a 16 bits.

.dc

```
.int              ;@ aloca 4 bytes na memória
.int -32          ;@ aloca e inicializa
.byte 0, 1, 2, 3  ;@ vetor de bytes
.float 1.3e-12
.dc.s 1.3e-12     ;@ o mesmo que o anterior
```

A diretiva `.ds` permite alocar espaço para vetores, opcionalmente preenchendo com valores iniciais:

.ds

```
.ds.b 20, 0xff      ;@ aloca 20 bytes e preenche com 0xff
.ds.w 10            ;@ aloca 10 half-words (20 bytes)
.ds.l 5, 0xaaaaaaaa ;@ aloca 5 words (20 bytes) e preenche
```

A diretiva `.ascii` permite introduzir *strings*. Observe que os *strings* não são terminados com zero automaticamente, você deve introduzir o caractere \0, caso necessário.

.ascii

```
.ascii "Hello ", "world", "!\0"
```

Código condicional

.if
.else
.endif

As diretivas `.if`, `.else` e `.endif` permitem condicionar a inclusão de código no arquivo de saída, de acordo com o valor de uma expressão:

```
.if reserva < 16    ;@ reserva no mínimo 16 bytes
.ds.b 16
.else
.ds.b reserva
.endif
```

Posicionar o endereço atual

.space
.skip
.org

As diretivas `.space` e `.skip` são equivalentes e permitem pular uma quantidade de endereços da seção atual, preenchendo ou não com um valor. A diretiva `.org` é semelhante, porém tenta posicionar o endereço no valor especificado (lembre-se, sempre relativamente ao endereço inicial da seção).

.align

A diretiva `.align` soma ao endereço atual um valor suficiente para que esse endereço seja múltiplo da potência de dois do parâmetro (“1” endereços pares, “2” endereços múltiplos de quatro, “3” endereços múltiplos de oito, etc.).

```
.skip 10            ;@ pula dez bytes
. = . + 10          ;@ equivalente
.skip 10, 0x00      ;@ pula e zera dez bytes
.org 0x100          ;@ o próximo endereço deve ser o 0x100
                    ;@ (em relação ao início da seção)
. = 0x100           ;@ equivalente
.align 2            ;@ alinha em endereços de word (2**2)
```

Macros e repetições

.macro

Macros podem ser criadas usando a diretiva `.macro`. Podem ser especificados **parâmetros** (separados por vírgulas), eventualmente com valores *default* (usando o operador `=`). Os valores dos parâmetros somente são visíveis no interior da macro (entre `.macro` e `.endm`), sendo referenciados com uma barra invertida (`\`).

```
.macro incr, reg=r0
add \reg, \reg, #1
.endm

;@ "chama" a macro (expande)
incr r3
```

Pode-se utilizar a diretiva `.exitm` para encerrar a macro antes de `.endm` (por exemplo em código condicional). Para criar o equivalente a “*loops*” em macros pode-se usar *recursão*:

```
.macro  lista i=0, j=3        // vai produzir o código
.int \i                       // ".int 0 .int 1 .int 2 .int 3"
.if \j - \i
lista "(\i+1)", \j
.endif
.endm
```

.rept

As instruções ou diretivas colocadas entre as diretivas `.rept` e `.endr` são replicadas a quantidade de vezes especificada.

```
.rept 4
str r0, [r8], #4
.endr
```

.abort
.error
.warning

Finalmente, as diretivas `.abort`, `.error` e `.warning` permitem cancelar o processo de montagem, por exemplo, na situação de verificação de algum erro ou inconsistência.

```
.if !p
.abort                        // termina ou
.error "Valor inválido"       // exibe mensagem de erro
.endif
.ifndef qtd
.warning "Valor qtd não definido!"
.endif
```

GNU ld

Comando

`ld [opções] [arquivos-objeto]`

- ***Prefixo***: depende do processador, sistema operacional e ABI desejado (compilação cruzada), por exemplo, **`arm-none-gnueabi-`**`ld`
- A **ordem** dos arquivos objeto na linha de comando, incluindo os arquivos de biblioteca, é importante;
- Opção `-o [arquivo-executável]` – especifica arquivo de saída;
- Opção `-T [script]` – especifica o arquivo com o *script* para o *linker*;
- Opção `-l [biblioteca]` – inclui funções de uma biblioteca (estática ou dinâmica);
- Opção `-L [diretório]` – especifica diretório para procurar as bibliotecas;
- Opção `-M` – exibir mapa do *linker* em *stdout*. A opção `-M=[arquivo]` gera o mapa em um arquivo.

Arquivo script básico

```
SECTIONS {
  .text 0x40000000 : {
    *(.text)
  }

  .data : {
    *(.data)
  }
}
```

- A sintaxe dos *scripts* é semelhante àquela do *assembler*;
- Os *scripts* padrão estão em `/lib/[arquitetura]/ldscripts`

objdump

- Opção `-d` – mostrar o código *assembler*;
- Opção `-f` – cabeçalho de arquivo (formato ELF);
- Opção `-g` – informações para o depurador;
- Opção `-h` – cabeçalhos das seções;
- Opção `-t` – listar símbolos;
- Opção `-T` – listar símbolos dinâmicos;
- Opção `-r` – informação sobre realocação.

O Depurador do Gnu (gdb)

O depurador do sistema Gnu (Gnu *Debugger* ou `gdb`) é um software com muitos recursos para analisar, depurar e alterar um outro programa, na maior parte das vezes *enquanto esse programa é executado*. Dessa forma, é possível observar a ocorrência de eventos (frequentemente erros) em tempo real e analisar as condições que favoreceram tais eventos. Alguns dos recursos oferecidos pelo `gdb` são:

- Sincronização entre o ponto de execução atual e o código-fonte do programa, caso disponível;
- Interrupção da execução do programa a qualquer momento (*break*);
- Interrupção do programa depurado em pontos de parada (*breakpoints*) definidos pelo usuário em endereços de memória arbitrários;
- Monitoramento do estado de posições de memória (variáveis, símbolos em geral), podendo provocar interrupção do programa depurado, conforme seu valor seja alterado (*watchpoints*), de forma semelhante aos *breakpoints*;
- Execução passo a passo, seja por linhas do código-fonte (em `C` ou outra linguagem), seja por instruções de máquina individuais (*assembly*);
- Visualização da memória em diversos formatos: mapa de memória, caracteres, *strings*, números inteiros (hexadecimal, octal, decimal ou binário), números em ponto flutuante, instruções de máquina (*disassembly*), etc.;
- Visualização do *contexto* do processador através dos registradores internos;
- Visualização do estado do programa a partir da pilha de chamadas de sub-rotinas;
- Alteração (edição) do ponto de execução do programa: execução de saltos, reinícios (*reset*), encerramento do programa (*exit*), etc.;
- Alteração (edição) de qualquer posição de memória (desde que a escrita nessa posição seja permitida) e dos registradores do processador.

O depurador pode ser um programa único, executado pela mesma máquina e sistema operacional que executa o programa depurado ou pode ser dividido em *duas partes* que se comunicam (por exemplo, através de uma rede de computadores, uma porta USB ou um canal serial). Nesse último caso, o programa depurado pode estar em *outra máquina*, eventualmente em outra arquitetura e por um sistema operacional diferente (ou mesmo não possuir qualquer sistema operacional envolvido). A parte do `gdb` que é executada na máquina de teste, juntamente com o programa depurado, é chamado de *stub*. Essa é uma situação comum quando depuramos um programa embarcado (em uma placa com um microcontrolador ou um aplicativo em um telefone celular, por exemplo) ou quando depuramos o próprio *kernel* do Linux executando em uma outra máquina. No caso do sistema emulado com o `qemu`, o *stub* está incluído no próprio programa do emulador.

stub

O `gdb` ou o seu *stub* podem ser controlados por qualquer programa diretamente, através de troca de mensagens, mas também existem *bibliotecas* de sistema (tais como `libgdb.so`) que oferecem uma interface de programação (API) de nível mais alto, que facilita a integração. Porém o uso mais comum é através de um aplicativo de linha de comando (utilitário "`gdb`"): esse é um aplicativo tipo "*read-eval-print-loop*" ou REPL: aguarda um comando do usuário via terminal, executa o comando, mostra de volta o resultado do comando e repete o processo indefinidamente.

Alguns dos comandos (tais como "*run*" e "*continue*") transferem o controle ao programa em depuração e somente vão retornar com algum resultado quando esse programa for encerrado ou interrompido por uma exceção ou por um *breakpoint*. Outros comandos são *interativos*, respondendo imediatamente. Em qualquer momento, a interrupção do programa em execução pode ser forçada pelo usuário do `gdb` com a combinação de teclas *break* (geralmente Control + C): neste caso, o *loop* de interpretação de comandos retorna.

É importante notar que enquanto o depurador aguarda um comando do usuário, o processo em teste não está sendo executado e permanece em seu último estado e contexto. Os comandos mais importantes do `gdb` em linha de comando serão descritos nos próximos parágrafos.

Execução e controle do processo

Comandos de execução e controle de processo

- "*run*" ou "r" – executa o programa a partir do início, somente retornando quando terminar, ocorrer uma exceção, encontrar um *breakpoint* ou receber um comando de interrupção (Control + C);
- "*continue*" ou "c" – executa o programa a partir da posição atual, somente retornando quando terminar, ocorrer uma exceção, encontrar um *breakpoint* ou receber um comando de interrupção (Control + C);
- "*step*" ou "s" – executa a próxima linha do código-fonte, retornando em seguida;
- "*stepi*" ou "si" – executa a próxima instrução de máquina, retornando em seguida;
- "*next*" ou "n"– executa a próxima linha do código-fonte, retornando em seguida. Caso seja uma *chamada de sub-rotina* (função, procedimento, método, etc.), executa a sub-rotina inteira antes de retornar;
- "*nexti*" ou "ni"– executa a próxima instrução de máquina, retornando em seguida. Caso seja uma *chamada de sub-rotina* (função, procedimento, método, etc.), executa a sub-rotina inteira antes de retornar;

Repetições

Os comandos "*continue*", "*step*" ou "*next*" podem incluir um *número de repetições*. Nesse caso, o comando será repetido esse número de vezes antes de retornar ao *loop* de comandos. *Exemplos*:

`step 3`	Executa as próximas três linhas
`nexti 5`	Executa as próximas cinco instruções, pulando sub-rotinas
`c 10`	Somente pára após dez interrupções do programa (*breakpoints*)

- "*finish*" ou "fin" – executa a sub-rotina atual até o final e retorna;

- "*kill*" – encerra o programa;
- "*backtrace*", "ba" ou "*where*" – mostra o ponto de execução atual, incluindo todas as chamadas de sub-rotinas na pilha do sistema.
- Se for enviado um comando *vazio* (ou seja, pressionar a tecla <enter> sem digitar nenhum comando), o `gdb` vai **repetir** o comando anterior, o que é muito útil para executar sucessivas linhas com "*step*", "*continue*" ou "*next*", ou ainda para continuar a visualização da memória com os comandos "*list*" ou "x".

Breakpoints

Comando "break"

Breakpoints

- O comando "*break* <local>" ou "b <local>" introduz um novo ponto de parada (*breakpoint*) no local especificado.

  O parâmetro <local> pode ser um *endereço* em memória (com um asterisco à frente), o *nome* de uma função, um *rótulo*, um *número de linha* referente ao arquivo-fonte atual ou de outro arquivo, no formato "nome do arquivo:número de linha". Se <local> for omitido, assume-se a posição atual.

Exemplos:

`b`	*Breakpoint* na posição atual
`break +3`	*Breakpoint* daqui a três linhas (ou três instruções)
`break main`	*Breakpoint* na entrada da função main()
`b main+2`	*Breakpoint* a segunda linha da função main()
`b *0x555555555e4`	*Breakpoint* no endereço virtual
`break teste.c:54`	*Breakpoint* na linha 54 do arquivo teste.c
`b 63`	*Breakpoint* na linha 63 do arquivo atual

Informações de depuração

É importante frisar que, para que o depurador possa encontrar os símbolos definidos no código-fonte (por exemplo, "*main*") é necessário que *a tabela de símbolos de depuração esteja presente no arquivo executável*: algo que normalmente é evitado, para economizar espaço. O programa depurado deve ter sido compilado ou montado com as opções `–g` ou `–ggdb`, que obrigam o compilador ou montador a copiar dados para o depurador no arquivo de saída. Além disso, é conveniente reduzir – ou mesmo desabilitar – a realização de *otimizações* pelo compilador, que podem eventualmente alterar a disposição das instruções no código-objeto, dificultando o entendimento da saída do depurador. Para isso, pode-se utilizar a opção `–O` do compilador.

Comando "break" condicional

Breakpoints condicionais

- O comando "*break* <local> *if* <condição>" permite definir uma **condição** para que a parada no *breakpoint* aconteça (*breakpoint* ***condicional***).

  O parâmetro <local> pode ser um *endereço* em memória, o *nome* de uma função, um *rótulo*, um *número de linha* referente ao arquivo-fonte atual ou de outro arquivo, no formato "nome do arquivo:número de linha". Se <local> for omitido, assume-se a posição atual.

O parâmetro <condição> pode ser qualquer expressão da linguagem C, envolvendo quaisquer símbolos definidos (no escopo global ou local, em relação à posição do *breakpoint*), ponteiros (endereços), registradores (precedidos com um cifrão "$") e constantes. Caso a expressão seja avaliada com um valor *diferente de zero*, o programa será interrompido no ponto de parada definido; do contrário, o ponto de parada é ignorado.

Exemplos:

`break if ok==0`	*Breakpoint* condicional na posição atual
`b loop+5 if data[4]>=8`	*Breakpoint* condicional em um rótulo
`b teste.c:32 if (a>0) && (i>8)`	*Breakpoint* condicional na linha 32 do arquivo

Edição de breakpoints

- "*info breakpoints*" ou "i b" mostra todos os *breakpoints* definidos, ativos ou não.

A informação mais importante da lista de *breakpoints* é o **índice** de cada *breakpoint*. É a partir desse número que os comandos a seguir podem identificar um *breakpoint* específico para alterá-lo, desabilitá-lo ou removê-lo completamente:

Edição de breakpoints

- "*condition* <índice> <condição>" – muda ou acrescenta uma condição ao *breakpoint* cujo índice é especificado no comando;
- "*ignore* <índice> <número>" – ignora o "<número>" de ocorrências do *breakpoint* **antes** de interromper o programa e retornar ao *loop* de comandos. Isso permite que um *breakpoint* somente seja acionado após uma quantidade de ocorrências;
- "*disable* <índices>" – desabilita o(s) *breakpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando. O(s) *breakpoint*(s) poderá(ão) ser reabilitado(s) no futuro;
- "*enable* <índices>" – habilita o(s) *breakpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando;
- "*delete* <índices>" – remove permanentemente o(s) *breakpoint*(s).

Caso o valor de <índices> seja omitido nos comandos "*disable*", "*enable*" ou "*delete*", a operação afetará **todos** os *breakpoints* existentes.

Exemplos:

`ignore 1 10`	Somente aciona o *breakpoint* 1 após dez ocorrências
`ignore 2 2`	Aciona o *breakpoint* 2 uma vez sim, uma vez não
`disable 3 4 5`	Desabilita os *breakpoints* de índices 3, 4 e 5
`enable 4`	Reabilita o *breakpoint* 4
`delete 3 5`	Remove os *breakpoints* 3 e 5

Variáveis e conteúdo da memória

Comandos "print" e "display"

Comando print

- "*print* <expressão>" ou "p <expressão>" – avalia e mostra o valor da expressão, que pode incluir endereços, constantes e o nome de variáveis que sejam visíveis no escopo corrente;

Comando display

- "*display* <expressão>" ou "d <expressão>" – o mesmo que "*print*", porém **memoriza** a expressão, recalcula e mostra o seu valor atualizado a cada passo da execução na linha de comando do depurador;

Formatos

- Os comandos "*print*" e "*display*" podem especificar o **formato** no qual a expressão será exibida, utilizando a notação "*print*/<formato> <expressão>". Os valores permitidos para <formato> são:
  - 'a' = ponteiro, 'c' = caractere, 'd' = inteiro com sinal, 'u' = inteiro sem sinal;
  - 'o' = inteiro em octal, 't' = inteiro em binário, 'x' = inteiro em hexadecimal;
  - 'f' = ponto flutuante;
  - 's' = *string*.

Exemplos:

`print i`	Mostra o valor atual da variável "i"
`print i/x`	Mostra o valor atual da variável "i" em hexadecimal
`display i`	Mostra o valor da variável "i" a cada iteração do depurador
`p x+y`	Avalia e mostra a expressão "x+y"

- A lista de expressões incluídas pelo comando "*display*" pode ser visualizada e alterada a partir do comando "*info display*" (ou "i *display*"), de forma semelhante ao apresentado anteriormente para os *breakpoints*.
  - "*disable display* <índice>" – interrompe a exibição da expressão identificada por <índice>;
  - "*enable display* <índice>" – ativa novamente a exibição, anteriormente desabilitada por "*disable display ...*";
  - "*undisplay* <índice>" – remove permanentemente a expressão identificada por <índice> da lista de expressões.

Comando "set"

Comando set

- O comando "*set*" permite alterar o valor de uma variável, registrador ou de posições de memória, no formato "*set* <destino> = <expressão>"
- <destino> pode ser um símbolo ou um endereço (precedido por um asterisco). Neste último caso, é recomendado incluir um *cast* para definir o tipo da expressão, entre parênteses, como em C (por exemplo, "(int)", "(float)", etc.)

Exemplos:

`set $cpsr = 0`	Muda o valor do registrador
`set i = 5`	Modifica o valor atual da variável "i"
`set a = b + 2`	Modifica o valor da variável "a"
`set *0xfeff0000 = (int)25`	Modifica a memória

Watchpoints

Watchpoints

- "*watch* <dado>" ou "w <dado>" cria um ponto de observação (*watchpoint*) relacionado a <dado>, que pode ser um símbolo do programa ou um endereço de memória. Sempre que o **valor** de <dado> for modificado pelo programa em depuração, o processo será interrompido, de forma equivalente ao que acontece um com um *breakpoint*;
- Os pontos de observação ativos podem ser visualizados e alterados a partir do comando "*info watch*" (ou "i *watch*"). Os *watchpoints* também aparecem na lista de *breakpoints* (com "info break").
  - "*disable* <índices>" – desabilita o(s) *watchpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando. O(s) *watchpoint*(s) poderá(ão) ser reabilitado(s) no futuro;
  - "*enable* <índices>" – habilita o(s) *watchpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando;
  - "*delete* <índices>" – remove permanentemente o(s) *watchpoint*(s).

  Caso o valor de <índices> seja omitido nos comandos "*disable*", "*enable*" ou "*delete*", a operação afetará **todos** os *watchpoints* e *breakpoints* existentes.

Exemplos:

`watch res`	Interrompe o processo se a variável res for alterada
`info watch`	Lista de watchpoints
`disable 1`	Desabilita o primeiro *breakpoint*/*watchpoint*

Obtendo informação sobre símbolos e variáveis

- Use o comando "*info scope* <local>" ou "i *scope* <local>" para saber quais são os símbolos visíveis em um determinado escopo: <local> pode ser o nome de uma função ou um endereço de uma instrução;
- O comando "*whatis* <símbolo>" permite identificar o tipo que foi declarado para um símbolo (inteiro, ponteiro, ponto flutuante, etc.);

Visualizar e alterar o conteúdo da memória

Conteúdo da memória

- O comando "x/<formato> <endereço>" permite visualizar o conteúdo de uma ou várias posições de memória, em diversos formatos diferentes.
  - Se <formato> incluir um número, ele especifica a quantidade de **registros** na memória a ser exibidos pelo comando;
  - Um caractere permite definir o tipo dos registros a exibir:
    - 'u' = ignorar sinal;
    - 'd' = decimal, 'x' = hexadecimal, 'o' = octal, 't' = binário;
    - 'a' = endereço, 'i' = instrução (*disassembly*);
    - 'c' = caractere, 's' = *string*, 'f' = ponto flutuante.
  - Outro caractere permite definir o tamanho de cada registro individual:
    - 'b' = *byte* – um registro por endereço;
    - 'h' = *half-word* – um registro a cada dois endereços (16 bits);
    - 'w' = *word* – um registro a cada quatro endereços (32 bits);
    - 'g' = *giant* – um registro a cada oito endereços (64 bits).

Exemplos:

`x/16xb 0x7fff0000`	Mostra 16 *bytes* em hexadecimal a partir do endereço
`x/16xb &vetor`	Mostra 16 *bytes* em hexadecimal da memória alocada para a variável "vetor"
`x/1fw &a`	Mostra o valor de "a" como ponto flutuante de 32 bits (*float*)
`x/1fg &a`	Mostra o valor de "a" como ponto flutuante de 64 bits (*double*)
`x/20i &calcula`	Mostra as primeiras 20 instruções da função "calcula"
`x/20c str`	Mostra os primeiros 20 caracteres do *string* "str"
`x/s str`	Mostra o *string* "str" completo (terminado em zero)

- O comando "x" – sem a especificação do endereço – repete o comando "x" anterior, com o mesmo formato, a partir do **último** endereço mostrado;
- Observe-se que o parâmetro deve ser sempre um **endereço**. No caso de símbolos declarados em C, os comandos abaixo são equivalentes:

```
p/x var
x/wx &var
```

- O comando "*set*" também pode ser usado para alterar posições de memória a partir de um endereço, prefixado com um asterisco. Neste caso, é sempre conveniente incluir um *cast* para definir o tipo de dados correto.

Controle da interface do usuário

Layout

O programa gdb pode exibir várias informações simultaneamente, dependendo do suporte do terminal utilizado pelo usuário. Caso o terminal permita, informações do depurador podem ser exibidas em diferentes "janelas" no mesmo terminal.

Layouts

- O comando "*layout*" permite definir o modelo de exibição do estado do depurador ao usuário:
  - "*layout src*" – mostra o código-fonte juntamente com a linha de comando do depurador;
  - "*layout asm*" – mostra o programa executável em assembler juntamente com a linha de comando do depurador;
  - "*layout regs*" – mostra o contexto do processador (registradores).

Comandos Comuns

Controle de execução	load		Carrega um arquivo na memória
	^C		Interrompe a execução
	run	r	Executa a partir do início
	continue	c	Continua execução a partir da posição atual
	step	s	Executa a próxima linha
	stepi	si	Executa a próxima instrução
	next	n	Executa próxima linha (pula sub-rotinas)
	nexti	ni	Executa a próxima instrução (pula sub-rotinas)
	finish	fin	Executa até o final da sub-rotina
	jump	j	Salta para um rótulo ou endereço
	backtrace	ba	Mostra a pilha de chamadas
Breakpoints	break	b	Define um *breakpoint*
	tbreak	tb	Define um *breakpoint* temporário
	watch	w	Define um *watchpoint*
	info break	i b	Mostra *breakpoints* atuais
	delete #	del	Remove um *breakpoint* (ou todos)
	condition #		Acrescenta/muda a condição para um *breakpoint*
	disable #		Desabilita um *breakpoint*
	enable #		Reabilita um *breakpoint*
	ignore # <n>		Ignora um *breakpoint* n vezes
Visualização	print	p	Avalia uma expressão
	display	d	Avalia uma expressão sempre que parar
	undisplay #		Cancela um "*display*" anterior
	x	x	Visualiza o conteúdo da memória
	list	l	Mostra o código-fonte
	set		Altera um registrador, variável ou posição de memória

Formatos

/	número	Mostra "n" valores (o padrão é um)
Tamanho	b	Byte (8 bits)
	h	Half word (16 bits)
	w	Word (32 bits)
	g	Double word (64 bits)
Formato	a	Ponteiro
	i	Instrução (disassembler)
	c	Caractere
	u	Valor sem sinal
	f	Ponto flutuante
	s	string
	d	Decimal
	o	Octal
	t	Binário
	x	Hexadecimal

Placa de desenvolvimento Evaluator-7T

A placa de desenvolvimento utilizada no laboratório é baseada no componente **S3C4510B** da Samsung, que é um processador ARM da arquitetura v.4 (ARM7TDMI). Ele é compatível com os conjuntos de instruções A32 (v.4) e T16 (v.1), possuindo funções de depuração (TAP/J-TAG) e *cache* interno. A placa possui 1 MiB de memória total, dividida entre SRAM e *flash*, implementadas com componentes externos ao processador.

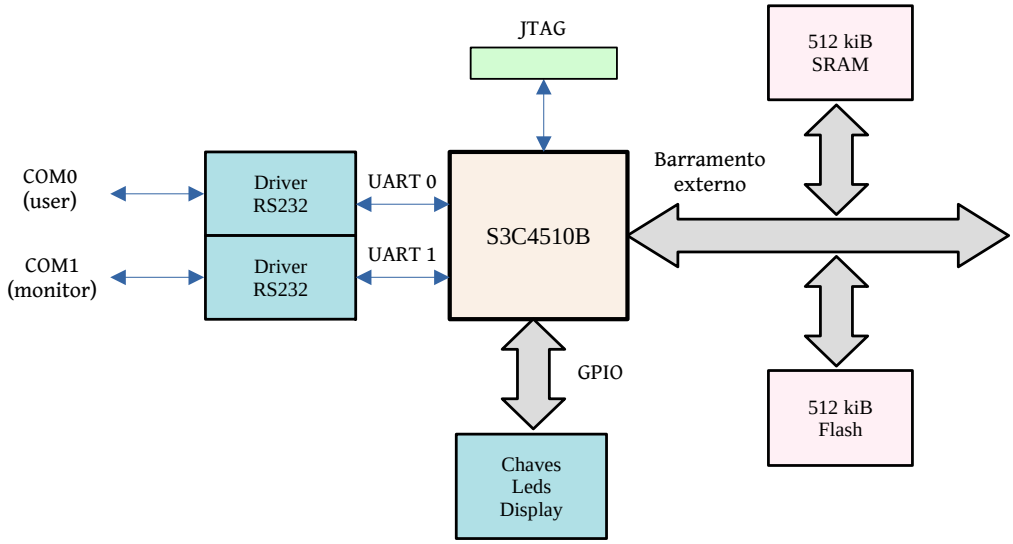
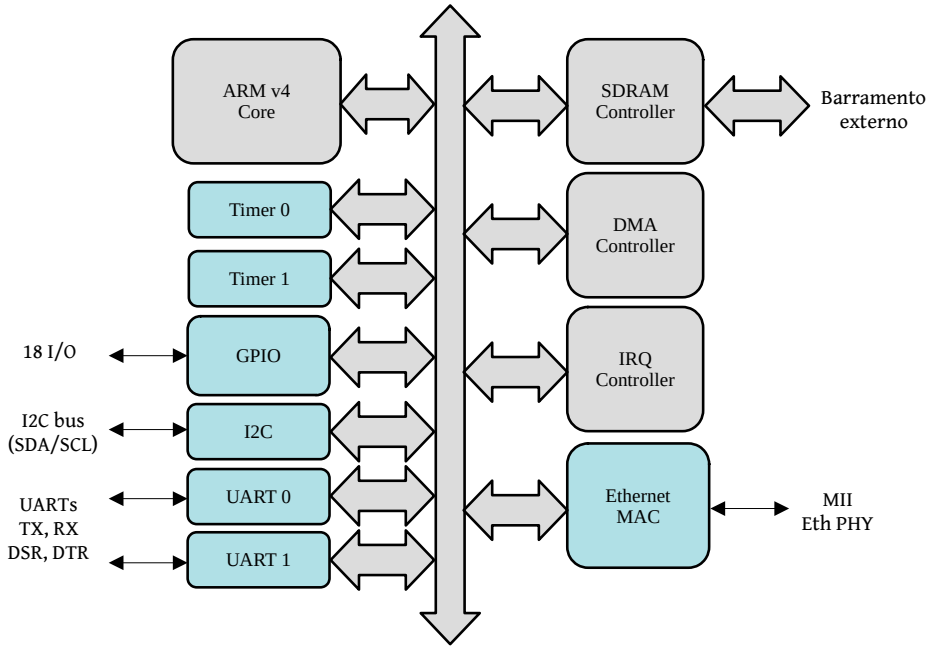


diagrama de blocos do Evaluator-7T

O processador S3C4510B inclui entradas e saídas digitais para uso geral (GPIOs), algumas delas são utilizadas na placa para acionar *leds* e um *display* de sete segmentos, bem como para ler o estado de chaves. Além disso, duas interfaces seriais UART estão disponíveis como portas RS232. Nós vamos utilizar a interface de depuração J-TAG para carregar, executar e depurar programas.

***chip* Samsung S3C4510B**

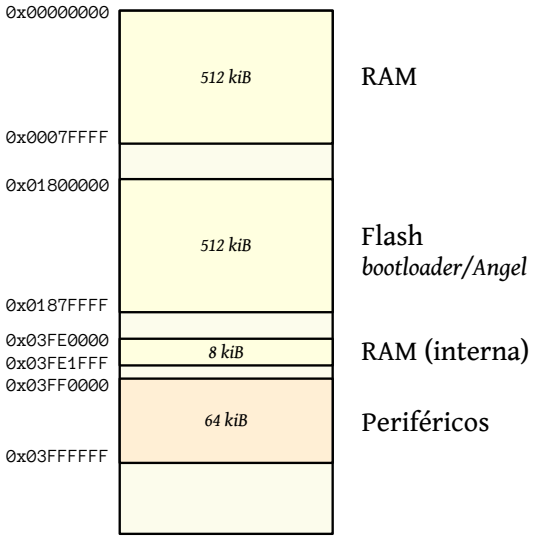


De forma semelhante a um microcontrolador, o componente S3C4510B inclui periféricos *internos*: os já mencionados GPIO e duas UARTs, dois *timers*, um controlador de barramento I2C, duas interfaces HDLC e um controlador de enlace *ethernet* (não utilizados na placa Evaluator-7T). Ao contrário dos

microcontroladores, por outro lado, o S3C4510B não possui memórias internas (à exceção da memória *cache*), dependendo de um barramento externo para a leitura das instruções e dados. O *chip* possui um controlador de memória SDRAM com possibilidade para mapeamento de endereços e o controle do processo de *refresh* de memória dinâmica. Além disso, ele possui um controlador de DMA (acesso direto à memória) e um controlador de interrupções, capaz de interromper o processador a partir de 21 fontes de interrupção diferentes.

organização da memória

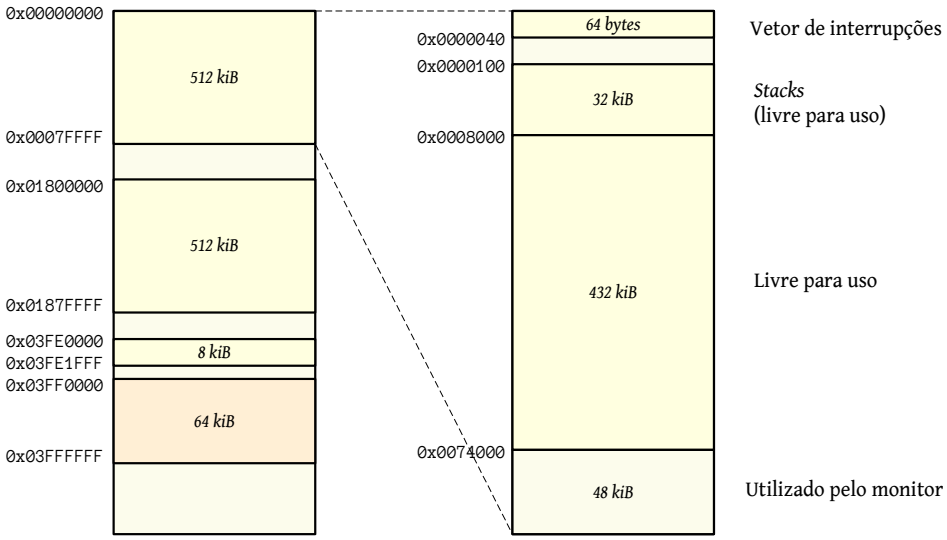
Visto pelo processador ARM, o mapa de memória da placa Evaluator-7T é apresentado na figura a seguir. Essa configuração de memória é realizada pelo programa *bootloader* armazenado na memória *flash*, que é executado quando a alimentação da placa é ligada.



Observe que, como em todas as máquinas de arquitetura ARM, os periféricos são *mapeados em memória*: para configurar e utilizar os dispositivos de entrada e saída, devem ser lidas e escritas posições de memória particulares.

memória RAM

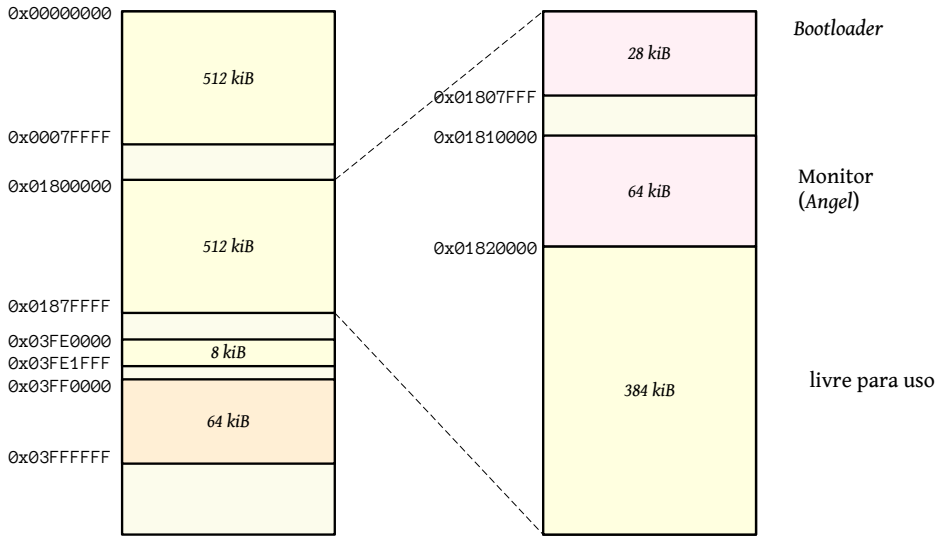
A memória RAM, com 512 kiB de tamanho, é mapeada a partir do endereço zero:



Os primeiros *bytes* da memória contém o vetor de interrupções, inicialmente preenchido pelo *bootloader*. Toda a memória RAM pode ser utilizada

para programas do usuário, mas os últimos 48 kiB são reservados ao programa monitor (“Angel”). Como vamos depurar usando a interface J-TAG, não vamos utilizar o monitor e essa restrição não se aplica.

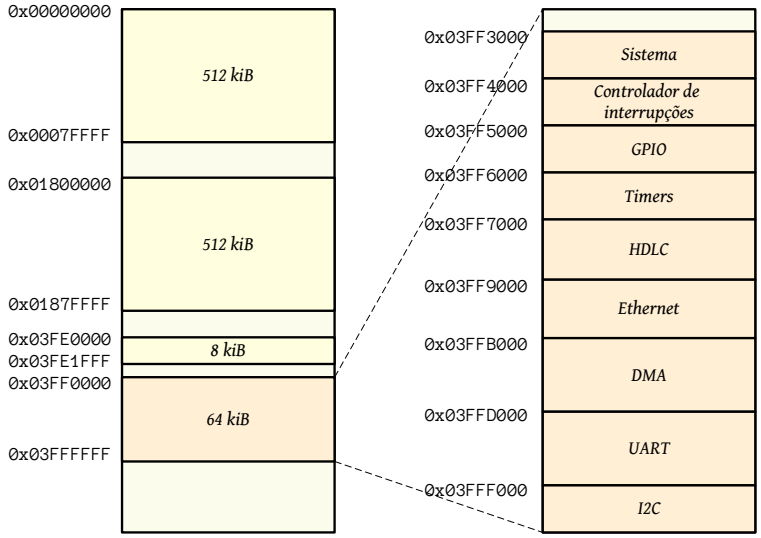
memória *flash*



Para referência, a memória *flash* é organizada em três seções a partir dos endereços 0x1800000 (*bootloader*), 0x1810000 (monitor *angel*) e 0x1820000 (livre). Não vamos utilizar a memória *flash* no laboratório, portanto ela deve ser considerada como uma memória apenas para leitura.

Finalmente, os dispositivos de entrada e saída são mapeados nos endereços a partir de 0x3ff0000, conforme a figura seguinte.

dispositivos mapeados em memória



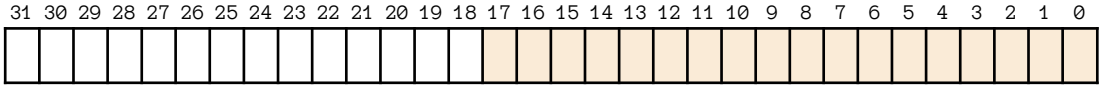
Portas de Entrada e Saída (GPIO)

O componente S3C4510B possui 18 pinos (P0 até P17) que podem ser utilizados como portas de entrada ou de saída, sob o controle do processador. Alguns desses pinos podem ter funções especiais: entradas de *interrupção* (P8 a P11), sinais de controle para o DMA (P12 a P15) e saídas de *clock* gerados pelos *timers* internos (P16 e P17).

funções especiais

Pino	Função	Placa Evaluator-7T
P0	GPIO 0	Chave 4
P1	GPIO 1	Chave 3
P2	GPIO 2	Chave 2
P3	GPIO 3	Chave 1
P4	GPIO 4	Led 4 (verde)
P5	GPIO 5	Led 3 (amarelo)
P6	GPIO 6	Led 2 (laranja)
P7	GPIO 7	Led 1 (verde)
P8	GPIO 8 / EIRQ0	Botão (“*user interrupt*”)
P9	GPIO 9 / EIRQ1	
P10	GPIO 10 / EIRQ2	Display- segmento “a”
P11	GPIO 11 / EIRQ3	Display- segmento “b”
P12	GPIO 12 / DMAREQ0	Display- segmento “c”
P13	GPIO 13 / DMAREQ1	Display- segmento “d”
P14	GPIO 14 / DMAACK0	Display- segmento “e”
P15	GPIO 15 / DMAACK1	Display- segmento “g”
P16	GPIO 16 / T0EN0	Display- segmento “f”
P17	GPIO 17 / T1EN1	

O funcionamento das entradas ou saídas digitais é controlado por três registradores: `IOPMOD` (endereço 0x3ff5000), `IOPCON` (endereço 0x3ff5004) e `IOPDATA` (endereço 0x3ff5008). O registrador `IOPMOD` especifica a *direção* de cada pino, no bit de mesmo índice: o valor “0” configura o pino como **entrada** digital e o valor “1” configura o pino como **saída** digital:

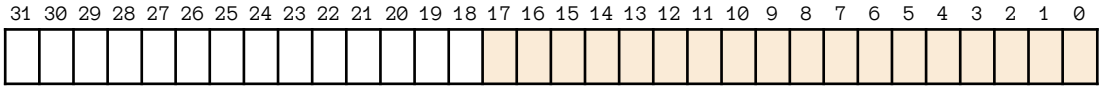


registrador IOPMOD

Após o *reset*, todos os GPIOs são configurados como **entradas** (o registrador IOPMOD vale *zero*).

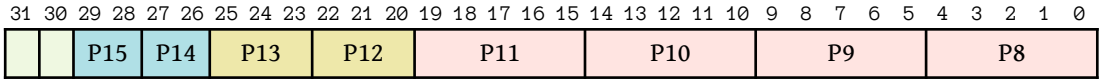
O registrador `IOPDATA` (endereço 0x3ff5008) contém o estado atual das 18 GPIOs; caso esse registrador seja **escrito**, o estado atual dos GPIOs configurados como saída serão atualizados. Um sinal alto ("H") no pino corresponde ao valor lógico "1" e um sinal baixo ("L") no pino corresponde ao valor lógico "0" no bit cujo índice é o mesmo da entrada ou saída.

registrador `IOPDATA`



As funções especiais dos pinos P8 a P17 são configuradas no registrador `IOPCON` (endereço 0x3ff5004):

registrador `IOPCON`



interrupções externas

Os pinos P8 a P11 podem ser configurados como entradas para produzir interrupções (interrupções externas 0 a 3). O formato dos seus campos de 5 bits no registrador `IOPCON` é o mesmo:

Bits	Significado
4	Habilita a interrupção (1)
3	Nível para verificar interrupção
2	Ativar *debounce* (1)
1-0	Evento a identificar como interrupção

00	Nível (conforme bit 1)
01	Borda de subida
10	Borda de descida
11	Qualquer transição

sinais de DMA

Os pinos P12 e P13 podem ser configurados como entradas de controle para o DMA, com um campo de três bits no registrador `IOPCON`:

Bits	Significado
2	Habilita sinal DMA *request* (1)
1	Ativar *debounce* (1)
0	Ativo alto (1) ou baixo (0)

Os pinos P14 e P15 podem ser configurados como saídas de controle para o DMA, com um campo de dois bits no registrador `IOPCON`:

Bits	Significado
1	Habilita sinal DMA *acknowledge* (1)
0	Ativo alto (1) ou baixo (0)

saídas dos *timers*

Finalmente, os bits 30 e 31 habilitam (bit igual a "um") ou desabilitam (bit igual a "zero") as saídas dos *timers* 0 e 1, respectivamente. Após um *reset*, o valor do registrador `IOPCON` é zero, desabilitando todas as funções especiais dos pinos.

Como exemplo, o programa a seguir configura o pino P3 como saída e liga a saída, escrevendo o valor lógico "1":

```
.set IOPMOD, 0x03ff5000
.set IOPCON, 0x03ff5004
.set IOPDATA, 0x03ff5008

  ldr r1, =IOPMOD
  ldr r0, [r1]
  orr r0, r0, #(1 << 3)     // configura IO 3 como saída
  str r0, [r1]

  ldr r1, =IOPDATA
  ldr r0, [r1]
  orr r0, r0, #(1 << 3)     // muda estado do IO 3
                            // ou bic r0, r0, #(1 << 3) para zerar
  str r0, [r1]
```

Gerenciador de Interrupções

O gerenciador de interrupções do S3C4510B pode reconhecer 21 fontes diferentes de interrupção e sinalizar à CPU com os sinais de interrupção "IRQ" (prioridade normal) ou "FIQ" (interrupção "rápida"), conforme a configuração. O serviço de interrupção deverá consultar o gerenciador de interrupções para determinar a causa da interrupção, realizar o processamento correspondente e finalmente limpar o *flag* de interrupção pendente, concluindo o processo.

índices das interrupções

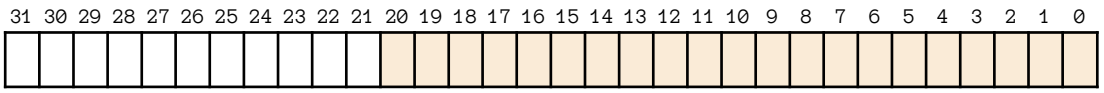
As interrupções que podem ser tratadas são correspondentes aos seguintes eventos, numeradas de zero a vinte:

Interrupção	Evento	Interrupção	Evento
0	Sinal externo (pino P8)	11	Temporização (*timer* 1)
1	Sinal externo (pino P9)	12	HDLC A transmissão
2	Sinal externo (pino P10)	13	HDLC A Recepção
3	Sinal externo (pino P11)	14	HDLC B transmissão
4	UART 0 Transmissão	15	HDLC B Recepção
5	UART 0 Recepção ou erro	16	DMA Ethernet Tx
6	UART 1 Transmissão	17	DMA Ethernet Rx
7	UART 1 Recepção ou erro	18	MAC Ethernet Tx
8	Conclusão DMA 0	19	MAC Ethernet Rx
9	Conclusão DMA 1	20	Evento I2C
10	Temporização (*timer* 0)		

O gerenciador de interrupções é controlado através de três registradores: `INTMOD` (endereço 0x3ff4000), `INTPEND` (endereço 0x3ff4004) e `INTMSK` (endereço 0x3ff4008). Outros registradores existem para a configuração de prioridades, mas não serão descritos aqui.

O registrador `INTMOD` (endereço 0x3ff4000) permite definir, para cada fonte de interrupção, qual o sinal de interrupção do processador ARM será utilizado: "IRQ" ou "FIQ".

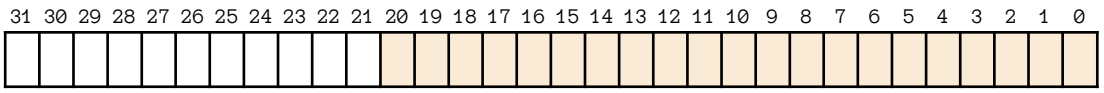
registrador `INTMOD`



Para cada um dos bits de 0 a 20 do registrador `INTMOD`, o valor "0" define a interrupção de mesmo índice como associada ao sinal "IRQ" e o valor "1" define essa interrupção como sendo "rápida" (sinal "FIQ").

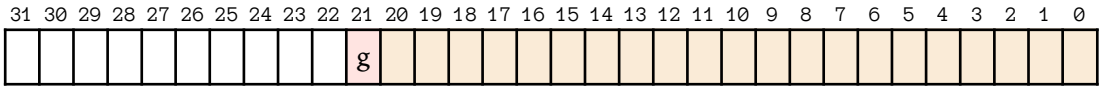
O registrador `INTPEND` (endereço 0x3ff4004) permite tanto *identificar* interrupções pendentes (operação de leitura) quanto *reconhecer* e *limpar* interrupções pendentes (escrevendo o valor lógico "1" nos bits individuais). Para cada um dos bits de 0 a 20, o valor "1" indica a ocorrência da interrupção de mesmo índice. O serviço de interrupção do ARM ("IRQ" ou "FIQ", dependendo da configuração do registrador `INTMOD`) deve primeiro consultar este registrador para identificar a(s) origem(ns) da(s) interrupção(ções) e, após o seu devido

processamento, escrever o valor lógico "1" nos bits correspondentes à(s) interrupção(ões) tratada(s), para reconhecê-la(s). Interrupções não reconhecidas vão causar sucessivos acionamentos do serviço de interrupção.



registrador INTPEND

O registrador INTMSK (endereço 0x3ff4008) é utilizado para habilitar ou desabilitar ("mascarar") interrupções. Além disso, o bit 21 é utilizado para habilitação (igual a "zero") ou mascaramento (igual a "um") de **todas** as interrupções simultaneamente ("global").



registrador INTMSK

Para cada um dos bits de 0 a 20 do registrador INTMSK, o valor "0" define a interrupção de mesmo índice como **habilitada** e o valor "1" define essa interrupção como desabilitada (mascarada). Seu valor inicial, após o *reset*, é 0x3fffff, ou seja, com todas as interrupções (incluindo o *flag* "global") desabilitadas.

Como exemplo, o trecho de código a seguir configura e trata a interrupção do *timer* 0 (índice 10):

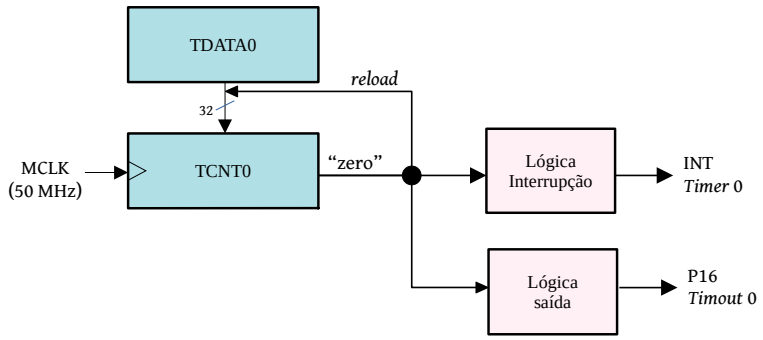
```
.set INTMOD, 0x03ff4000
.set INTPND, 0x03ff4004
.set INTMSK, 0x03ff4008

inicia_irq:
  ldr r1, =INTMOD
  ldr r0, [r1]
  bic r0, r0, #(1 << 10)   // configura como IRQ
  str r0, [r1]

  ldr r1, =INTMSK
  ldr r0, [r1]
  bic r0, r0, #(1 << 10)   // habilita interrupção 10 (timer 0)
  str r0, [r1]
  mov pc, lr

.global trata_irq
trata_irq:                 // chamado pelo vetor de interrupções
  push {r0-r1}
  ldr r1, =INTPND
  ldr r0, [r1]
  tst r0, #(1 << 10)       // verifica interrupção 10 (timer 0)
  beq ack
  /* ...
   * tratar interrupção do timer AQUI
   */
ack:
  str r0, [r1]             // reconhece todas as interrupções
  pop {r0-r1}
  subs pc, lr, #4          // retorna IRQ
```

Temporizadores (timers)

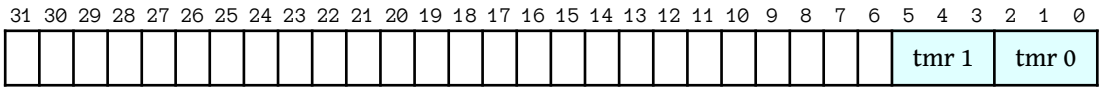
Um temporizador (*timer*) é um registrador interno do componente S3C4510B que é decrementado a cada ciclo do sinal *master clock* (MCLK). Quando esse registrador possui o valor "zero" e é decrementado novamente ("estouro"), seu estado é atualizado com um **valor de recarga** (*reload*), armazenado em outro registrador. A situação de estouro pode ser usada para gerar um **sinal** externo (pulso ou onda quadrada) ou provocar uma **interrupção** no processador. Assim, o valor armazenado no registrador de recarga é proporcional ao tempo entre cada "estouro" do temporizador: esse é um mecanismo útil para produzir interrupções periódicas.

valor de recarga

interrupção periódica



No caso da placa Evaluator-7T, o sinal MCLK tem a frequência de 50 MHz. O registrador `TMOD` (endereço 0x3ff6000) é usado para configurar ambos os temporizadores (*timer* 0 e *timer* 1). Os registradores `TDATA0` (endereço 0x3ff6004) e `TCNT0` (endereço 0x3ff600c) contém o valor de recarga e o valor atual do contador do *timer* 0, respectivamente. Os registradores `TDATA1` (endereço 0x3ff6008) e `TCNT1` (endereço 0x3ff6010) cumprem as mesmas funções, para o *timer* 1.



registrador `TMOD`

Ambos os temporizadores podem ser configurados com campos de três bits do registrador `TMOD` (endereço 0x3ff6000). O formato desses campos é o mesmo:

Bits	Significado
0	Habilita o *timer* (1)
1	*Toggle* (onda quadrada) (1) ou pulso (0)
2	Estado inicial do sinal de saída

sinais de saída

Os bits 1 e 2 (ou 4 e 5 para o *timer* 1) servem para configurar a saída especial produzida no pino P16 (ou P17 para o *timer* 1); observe-se que para obter uma saída (pulso ou onda quadrada) nesses pinos, ainda é necessário configurar os pinos como saída especial no registrador `IOPCON` (endereço 0x3ff5004).

Para que os *timers* produzam interrupções, além de habilitá-los, é necessário configurar e habilitar as interrupções correspondentes (índice 10 ou 11) através dos registradores `INTMOD` (endereço 0x3ff4000) e `INTMSK` (endereço 0x3ff4008).

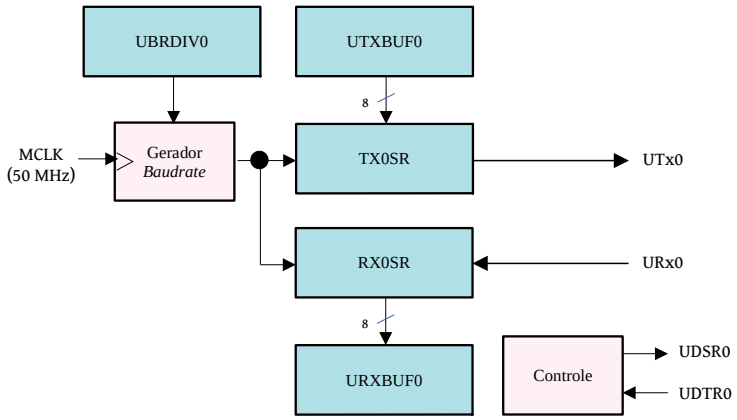
Como exemplo, a sub-rotina a seguir configura o *timer* 1 para produzir uma interrupção a cada 100 ms:

```
.set INTMOD, 0x03ff4000
.set INTPND, 0x03ff4004
.set INTMSK, 0x03ff4008
.set TMOD, 0x03ff6000
.set TDATA1, 0x03ff6008
.set TCNT1, 0x03ff6010

.set TEMPO, 4999999        // valor de recarga para 100 ms em 50 MHz

inicia_timer1:
  // configura interrupção 11 (timer 1)
  ldr r1, =INTMOD
  ldr r0, [r1]
  bic r0, r0, #(1 << 11)   // configura como IRQ
  str r0, [r1]

  ldr r1, =INTMSK
  ldr r0, [r1]
  bic r0, r0, #(1 << 11)   // habilita interrupção
  str r0, [r1]

  // configura valor de recarga (período)
  ldr r1, =TDATA1
  ldr r0, =TEMPO
  str r0, [r1]

  // liga o timer 1
  ldr r1, =TMOD
  ldr r0, [r1]
  bic r0, r0, #(0b111 << 3)
  orr r0, r0, #(0b001 << 3)
  str r0, [r1]

  mov pc, lr
```

UARTs

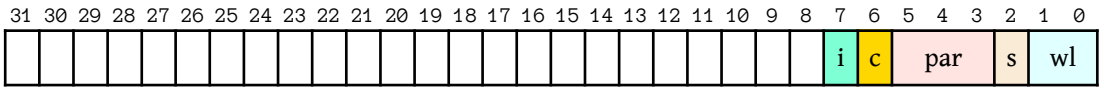
O S3C4510B possui duas unidades de comunicação serial tipo UART (*universal asynchronous receiver-transmitter*) capazes de enviar e receber caracteres de cinco a oito bits e gerenciar um bit de paridade. Na placa Evaluator-7T as duas UARTs estão conectadas a *transceivers* RS232 para conexão com portas seriais convencionais de 9 pinos (conectores DB-9).

Cada uma das UARTs é composta por três elementos: o transmissor, o receptor e um gerador de *baudrate*. O transmissor e o receptor são baseados em registradores de deslocamento, cujo sinal de *clock* é produzido pelo gerador de *baudrate*. O gerador de *baudrate* é semelhante a um *timer*, e pode usar o *clock* interno (o mesmo do processador) ou um sinal externo para obter taxas mais precisas.



Os registradores de deslocamento não são acessíveis ao programador. Uma vez concluída a recepção (*stop-bit* reconhecido), o conteúdo do registrador de deslocamento é copiado para o registrador `URXBUF0` (ou `URXBUF1`, endereços 0x3ffd010 e 0x3ffe010, respectivamente), para que possa ser lido pelo software. Além disso, ao ser recebido um caractere, a UART pode gerar uma interrupção (interrupções 5 ou 7). De forma análoga, para transmitir um caractere, deve-se escrevê-lo no registrador `UTXBUF0` (ou `UTXBUF1`, endereços 0x3ffd00c e 0x3ffe00c, respectivamente); esse registrador vai alimentar o registrador de deslocamento e a transmissão dos bits terá início. Ao final da transmissão completa, uma interrupção pode ser gerada (interrupções 4 ou 6).

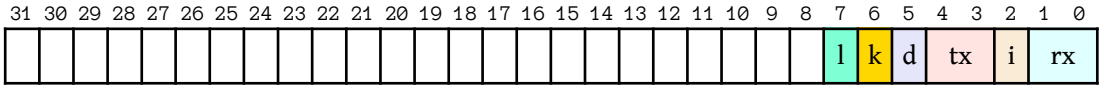
As UARTs são configuradas usando os registradores descritos a seguir.



registrador
ULCON0
ULCON1

Bits	Significado
0-1	Palavra com 5 bits (00), 6 bits (01), 7 bits (10) ou 8 bits (11)
2	Um *stop bit* (0) ou dois *stop bits* (1)
3-5	Paridade: nenhuma (0xx), ímpar (100), par (101), sempre “1” (110) ou sempre “0” (111)
6	Usar *clock* interno (0) ou externo (1)
7	Modo IR habilitado (1)

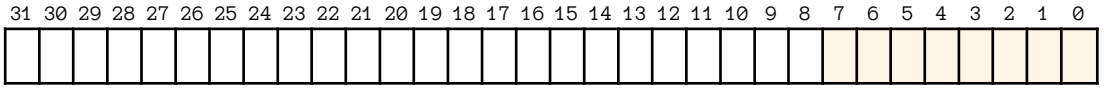
Os registradores ULCON0 e ULCON1 (endereços 0x3ffd000 e 0x3ffe000, respectivamente) permitem configurar o formato do sinal gerado e reconhecido pela UART, bem como a origem do sinal a ser utilizado pelo gerador de *baudrate*. Os registradores UCON0 e UCON1 (endereços 0x3ffd004 e 0x3ffe004, respectivamente) permitem habilitar e controlar o transmissor e o receptor:



registrador
UCON0
UCON1

Bits	Significado
0-1	Recepção: desabilitar (00), interrupção (01), DMA 0 (10) ou DMA 1 (11)
2	Habilita interrupção em caso de erro (1)
3-4	Transmissão: desabilitar (00), interrupção (01), DMA 0 (10) ou DMA 1 (11)
5	Estado do sinal DSR
6	Enviar um sinal *break* (1)
7	Modo *loopback* (1)

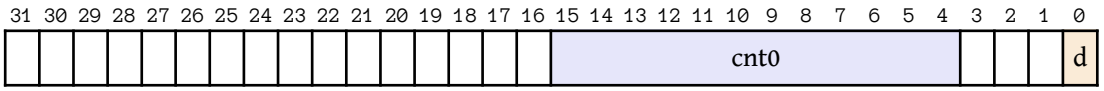
O estado atual das UARTs pode ser verificado consultando os registradores USTAT0 e USTAT1 (endereços 0x3ffd008 e 0x3ffe008, respectivamente):



registrador
USTAT0
USTAT1

Bits	Significado
0	Erro de *overrun* (dado recebido não lido a tempo)
1	Erro de paridade
2	Erro de *framing* (stop bit não detectado)
3	Sinal de *break* recebido
4	Estado do sinal DTR
5	Dados prontos para leitura
6	Transmissor pronto para enviar novo caractere
7	Transmissão completa

Finalmente, os geradores de *baudrate* são configurado pelos registradores UBRDIV0 e UBRDIV1 (endereços 0x3ffd014 e 0x3ffe014, respectivamente), cujos campos são usados para definir o divisor de frequência.



registrador
UBRDIV0
UBRDIV1

A frequência de entrada do gerador de *baudrate* pode ser a metade da frequência do *clock* (MCLK / 2) ou um sinal externo (UCLK). No caso de usar o *clock* interno, o *baudrate* final é dado pela expressão

$$baudrate = \frac{MCLK}{32 \cdot (1 + CNT0)}$$

caso o bit “d” de USTAT seja igual a “zero”. Caso seja “1”, o *clock* de entrada é previamente dividido por dezesseis e o *baudrate* é dado pela expressão

$$baudrate = \frac{MCLK}{512 \cdot (1 + CNT0)}$$

No caso da placa Evaluator-7T, usando o *clock* de 50 MHz, a tabela a seguir mostra os valores para “cnt0” (“d” é zero) para os *baudrates* mais comuns.

Baudrate	cnt0
1200	1301
2400	650
4800	324
9600	162
19200	80
38400	40
57600	26
115200	13

Implementação da Linguagem C

A linguagem "C" foi criada da década de 70 por Dennis Ritchie, a partir do protótipo (linguagem "B") desenvolvido por Ken Thompson para o novo sistema operacional Unix. Mais tarde, a própria linguagem passou a ser utilizada para escrever o código-fonte do sistema operacional e torná-lo mais facilmente portável para outras arquiteturas. A linguagem C começou a ser padronizada em 1985 (ANSI C); hoje existem dois padrões principais: C99 e C11.

Algumas características dessa linguagem são especialmente importantes:

- Possui uma estrutura simples e muito próxima ao *hardware* (de fato, nas primeiras implementações somente os tipos de dados e operações efetivamente suportados pela máquina alvo eram implementados na linguagem), tornando seus compiladores mais eficientes e possibilitando a implementação de funcionalidades de baixo nível semântico com menor necessidade de se recorrer à linguagem de máquina;
- "C" é uma linguagem de modelo *funcional*, na tradição do ALGOL. De fato, a entidade central na linguagem C é a "função": com escopo próprio, que pode receber um conjunto variável de parâmetros e pode ser acionada de forma **recursiva**. Esse tipo de funcionalidade é geralmente implementado através de uma estrutura de dados em **pilha**, que possui um papel fundamental no funcionamento de um programa codificado em C;

a pilha

- As entidades declaradas em C são mapeadas diretamente na memória. Seus endereços de memória e os valores contidos na memória podem ser usados alternativamente com facilidade, através de operações de **referência** e **derreferência** ("ponteiros"). Todos os operandos da linguagem são igualmente aplicáveis a ponteiros; um ponteiro, além do endereço em memória, carrega também informação sobre o **tipo** de dados apontado;

ponteiros

- Em sua grande maioria, dados de tipos diferentes são intercambiáveis em C e têm sua conversão realizada de forma implícita pelo compilador. Operandos de uma expressão são *promovidos* para tipos mais completos (por exemplo, um `int` para um `float`, ou um `unsigned` para um `signed`) e valores atribuídos são convertidos para o tipo correspondente à variável ou ponteiro de destino. Além do tratamento padrão, a conversão de dados pode ser diretamente controlada pelo programador, através de operadores de *casting*;

struct

- Permite a representação eficiente de estruturas binárias complexas através do uso de tipos especiais (`struct` e `union`), que podem conter todos os tipos elementares da linguagem, vetores, campos de bit e outros tipos especiais aninhados. Campos de bit em uma estrutura permitem definir uma quantidade precisa de bits para a informação, e o compilador automaticamente gera código para sua leitura e atualização (máscaras, deslocamentos, etc.);

- O valor **nulo** (todos os bits iguais a "zero") tem significados especiais em C:
  - Em expressões **condicionais** (operador `?`, `if()`, `while()`, etc.) o valor zero representa o valor lógico "falso" ou "não", enquanto que qualquer outro valor é considerado "verdadeiro"; **valores lógicos**
  - Utilizado como um **índice**, entre colchetes (`[0]`), o valor zero aponta sempre o **primeiro** elemento de um vetor (deslocamento zero a partir da base);
  - O caractere nulo (código ASCII `NUL`) é utilizado como **terminador** de *strings* pela biblioteca C padrão;
  - Um ponteiro nulo (`NULL`) é sempre **inválido**: derreferenciar um ponteiro nulo causa uma exceção ou *abort*. Assim sendo, é comum que o ponteiro nulo seja utilizado como marcador em situações especiais ("vazio", "inválido", "não utilizado", "fim de lista", "erro", etc.); **NULL pointer**
  - Só há dois **tamanhos** nulos: o tipo `void` (`sizeof(void)`) e o vetor sem dimensão (`int x[0]`);
  - Todas as variáveis globais não inicializadas no código-fonte têm seus bits zerados antes da chamada à função `main()`.

Como a linguagem C possui uma grande flexibilidade, aderência a padrões de portabilidade e uma longa história de utilização, sendo muito utilizada na construção de *firmware*, sistemas operacionais e compiladores para outras linguagens, ela se tornou um padrão "de fato", tendo impacto inclusive no projeto de novos processadores. Diversas características inerentes ao processador ARM, por exemplo, têm influência direta de funcionalidades necessárias para o suporte à linguagem C padronizada.

C-Runtime

O código objeto produzido por um compilador C inclui, além de sequências de instruções de máquina da arquitetura alvo, correspondentes aos algoritmos descritos no programa, funções especiais anexadas pelo *linker* para o suporte a funcionalidades de nível semântico mais alto oferecidas pela linguagem. As funções do *Runtime* que são adicionadas ao objeto são escolhidas conforme a necessidade do compilador em relação aos recursos nativos oferecidos pela arquitetura alvo. Uma lista (incompleta) de funções oferecidas pelo *runtime* é apresentada a seguir:

- Prólogos e epílogos de funções: alocação de espaço para variáveis locais, salvamento de registradores e manutenção da pilha durante chamadas de funções;
- Funções de inicialização: montagem e configuração da pilha, suporte aos prólogos e epílogos das funções;
- Funções de finalização: *cleanup*, finalização de processo no sistema operacional, etc.;

- Funções de extensão aritmética: operações com tipos diferentes, tipos estendidos ("`long long`", etc.), operações internas (em ponto fixo, precisão arbitrária, ponto flutuante decimal, etc.), emulação de instruções de ponto flutuante (para aquelas que a arquitetura não ofereça suporte);
- Funções de conversão de tipos: transformação entre os tipos inteiros, precisão de ponto flutuante, conversão e conservação de sinal, etc.;
- Funções de comparação: comparação de valores de tipos diferentes, com ou sem conversão prévia;
- Funções de manipulação binária: tratamento de campos de bit, deslocamentos, operações lógicas e operações lógicas bit a bit;
- Funções para tratamento de situações especiais: controle de execução, exceções, sinais enviados pelo sistema operacional, *caches*, etc.

As funções do *runtime* são oferecidas pelo compilador ao *linker*, contidas em arquivos de biblioteca como `libgcc.a` e `libgcc_s.a`. Você pode verificar as funções de baixo nível existentes na versão do `gcc` que está utilizando com o comando:

```
arm-none-eabi-nm /usr/lib/gcc/arm-none-eabi/<sua versão>/libgcc.a
```

Biblioteca C padrão

Os compiladores `C` geralmente oferecem bibliotecas de funções de alto nível, algumas delas definidas por padrões (como `stdlib`).

Muitas das funções padrão são interfaces para o acionamento de serviços de um sistema operacional (como as funções para manipulação de arquivos e *sockets*). Na quase totalidade dos casos, os processos não acessam diretamente as chamadas do sistema operacional, mas o fazem indiretamente a partir das funções oferecidas pela biblioteca `C` (ainda que tais programas tenham sido escritos em outras linguagens!), por simplicidade e familiaridade, mas também por ser uma interface padronizada, ajudando no processo de portabilidade entre diferentes sistemas operacionais.

Funções mais básicas como a inicialização de um processo, com a alocação e inicialização dos segmentos de dados, a chamada e o tratamento do valor retornado pela função `main()`, também estão incluídas nas bibliotecas padrão: o ponto de entrada definido pelo *linker* é a função `start()`, definida no arquivo `crt0.o`, que realiza esse procedimento. Ao criar um programa em `C` para ser executado em uma máquina sem um sistema operacional, o programador é responsável pela realização dessas tarefas.

crt0.o

As bibliotecas padrão oferecem também várias ferramentas úteis ao programador, como tratamento de funções variádicas (`stdarg`), funções matemáticas, gerenciamento de memória dinâmica, manipulação de *strings*, data e hora, formatação de *strings* com localização ("*locales*"), saltos longos, expressões regulares, etc.

A inclusão das funções das bibliotecas padrão é opcional (ao contrário das funções do *runtime*) e pode ser feita tanto estaticamente pelo *linker* (por exemplo, incluindo o arquivo `libc.a`) quanto dinamicamente pelo *loader* do sistema operacional (carregando `libc.so` junto com o processo).

De um modo geral, todas as declarações das funções presentes na biblioteca padrão são acessíveis para inclusão no código-fonte com o comando `#include` do pré-processador, em arquivos cabeçalho como `stdlib.h`, `stdio.h`, `math.h`, `string.h`, etc. As bibliotecas a serem utilizadas também devem incluídas entre os arquivos a serem consultados pelo *linker* (`libm.a`, `libpthread.a`, etc.).

Você pode verificar as funções de biblioteca existentes na versão do `gcc` que está utilizando com os comandos:

```
ls /usr/lib/arm-none-eabi/lib/
arm-none-eabi-nm /usr/lib/arm-none-eabi/lib/libc.a
arm-none-eabi-nm /usr/lib/arm-none-eabi/lib/libm.a
arm-none-eabi-nm /usr/lib/arm-none-eabi/lib/<etc>
```

Existem várias implementações da biblioteca `C` que podem ser utilizadas pelo compilador `gcc` (mas isso geralmente requer uma versão especial do `gcc` para cada biblioteca: às vezes o nome da biblioteca é incluído no prefixo do executável do compilador), tais como `glibc` (do próprio GNU), BSD (usado no macOS), `uclibc` (projeto uClinux), `newlib` (Red Hat), `musl` (usado no Gentoo e Alpine) e `bionic` (usado no Android). A versão do `arm-none-eabi-gcc` que estamos utilizando instala a biblioteca `C` do projeto `newlib`[8], frequentemente utilizada sem o suporte de um sistema operacional (aplicações "*bare metal*", por isso o "*none*" no prefixo) e em sistemas sem suporte a memória virtual.

8 Veja o *site* do projeto `newlib` em http://www.sourceware.org/newlib/

Dados em C

Variáveis declaradas em C são alocadas pelo compilador em diferentes *segmentos* de memória, de acordo com o seu escopo e modo de inicialização:

variáveis globais

- Variáveis **globais** com algum *valor inicial* são incluídos no segmento `.data` e o valor inicial é escrito na posição de memória correspondente[9];

segmento .bss

- Variáveis **globais** sem inicialização são incluídas no segmento *.bss*[10] (ou simplesmente como símbolos "comuns"). A memória onde reside o segmento `.bss` é inicializada com zeros pelo *startup* da biblioteca C (`crt0.o`), antes da chamada à função `main()`;

- Variáveis **locais** (declaradas no interior de uma função) marcadas com **`static`** também são alocadas nos mesmos segmentos `.data` ou `.bss`, conforme a existência de inicializações, possivelmente com um símbolo alterado (*name mangling*) para evitar conflito com o nome de outras variáveis;

variáveis locais

- Variáveis **locais** (declaradas no interior de uma função) são criadas (pelo *prólogo* da função) e destruídas (pelo *epílogo* da função) **dinamicamente** pelo *runtime* no segmento de **pilha** e são, portanto, invisíveis para o *linker*. Instâncias de variáveis locais pertencentes a chamadas recursivas de uma mesma função estão em *frames* diferentes da pilha. Preste atenção que não existe nenhum tipo de inicialização para variáveis criadas na pilha: jamais assuma o valor "zero", por exemplo.

vetores

Vetores são alocados em posições de memória consecutivas, com um tamanho correspondente ao produto do número de elementos pelo tamanho de cada elemento. O acesso aos elementos individuais de um vetor é realizado diretamente em código de máquina, em geral usando modos de endereçamento indexados: o *runtime* da linguagem C não faz nenhum tipo de verificação de endereçamento (*boundary violations*), portanto é perfeitamente viável invadir áreas não alocadas por um vetor.

Operações realizadas com ponteiros para elementos de um vetor levam em consideração o *tamanho* de cada elemento, o que normalmente corresponde ao comportamento desejado. A forma como um vetor é disposto na memória pode também ser afetado por características inerentes à arquitetura, como por exemplo, obrigatoriedade de alinhamento em endereços pares, múltiplos de quatro, etc.

estruturas

A definição de uma estrutura (`struct`) ou união (`union`) não ocasiona a alocação de posições de memória, o que somente vai acontecer quando a estrutura for declarada como uma variável, parâmetro de uma função ou retorno de função. A disposição dos campos de uma estrutura segue endereços contíguos de memória, de acordo com o tamanho dos campos; no entanto, o compilador

9 Em situações em que os valores iniciais não podem ser mantidos em RAM, por exemplo em sistemas embarcados, eles são salvos em um segmento chamado `.rodata`, que reside em memória não-volátil. Os valores desse segmento são copiados para a RAM pelo código *startup* (`crt0.o`) antes da chamada à função `main()`.

10 A sigla vem de "*block started by symbol*", que era uma diretiva dos antigos *assemblers*. O nome foi mantido como tradição, mas poderia muito bem significar "*better save space*", segundo Peter van Linden.

geralmente tem a liberdade de redistribuir os campos para otimizar espaço ou incluir espaços não utilizados para forçar alinhamentos e com isso otimizar o tempo de acesso, a menos que você peça para ele não fazer isso. Da mesma forma que para vetores, o cálculo dos endereços efetivos dos campos de uma estrutura é feito em código de máquina, usando modos de endereçamento indexados.

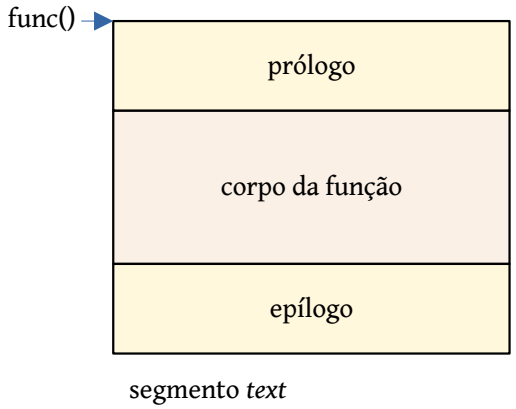
Variáveis globais declaradas de tipos `struct` ou `union` e não inicializadas explicitamente também são inicializadas com zero, por residirem na área de memória do segmento `.bss`.

Implementação de funções em C

Uma função compilada em C ganha um endereço no código objeto (que pode ser referenciado em um ponteiro de função com uma notação especial) e um símbolo no segmento `.text`. Esse símbolo normalmente é exportado (como "`.global`" em *assembler*), a menos que a função seja marcada como `static`: neste caso, o símbolo somente é visível no arquivo onde a função foi definida. Outras funções podem usar esse símbolo para chamá-la como uma sub-rotina (instrução `bl`, no caso do ARM).

prólogo e epílogo

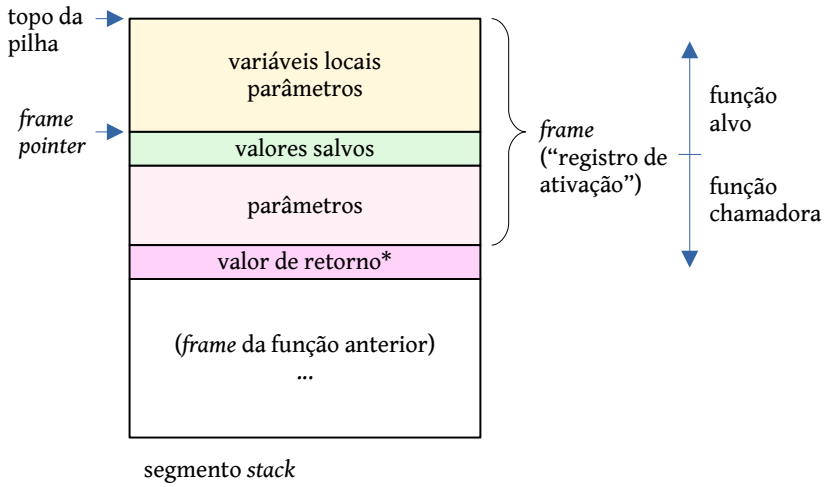
O código de uma função C geralmente inclui um *prólogo* criado pelo compilador para gerenciar a pilha, alocar espaço para as variáveis locais e salvar o estado de registradores que sejam alterados dentro da função. Eventuais alterações na pilha são desfeitas por um *epílogo* introduzido no final da função, que também é responsável por realizar o retorno à função chamadora.



Todas as funções em C têm a obrigação de retornar a pilha no mesmo estado que receberam, bem como manter certos registradores com seus valores inalterados. Funções especificamente declaradas como assíncronas (por exemplo, implementando serviços de interrupção) devem manter os valores originais de **todos** os registradores.

frame

As funções em C organizam o seu espaço na pilha através de uma estrutura de dados chamada *frame* ou "registro de ativação":



Como regra geral, os **parâmetros** da função a chamar são colocados na pilha pela função chamadora, enquanto que a alocação de variáveis locais e de cópias dos parâmetros são realizadas pela função chamada. Após o retorno, a função chamadora é responsável pela eventual remoção dos parâmetros colocados na pilha, caso necessário.

frame pointer

Um registrador especial, geralmente chamado de *frame pointer* (fp) é utilizado para armazenar um endereço do *frame*, que serve como base para acessar tanto os parâmetros quanto as variáveis locais através de endereçamento indexado (como se fossem campos de um `struct`). O endereço armazenado no *frame pointer* também serve para atualizar o ponteiro de pilha para o valor anterior, de forma a retornar a pilha inalterada à função chamadora.

Vários tipos de otimizações podem ser utilizados, conforme convenções e configurações feitas pelo programador:

- Passagem de parâmetros em **registradores** específicos da arquitetura, sem usar a pilha. Caso o número de parâmetros ou a quantidade de bits necessários ultrapasse a capacidade dos registradores, os parâmetros extra são colocados na pilha, segundo o modelo padrão;
- Caso a função chamada não altere os valores de algum parâmetro ou esteja autorizada a fazer **alterações** nos valores dos parâmetros introduzidos no *frame* pela função chamadora, a cópia desses parâmetros junto com as variáveis locais da função pode ser evitada. Esse tipo de otimização pode ser inibido pela especificação do parâmetro com a palavra reservada `const`;
- Variáveis locais também podem ser alocadas em registradores (ou reutilizando os registradores eventualmente utilizados para a passagem de parâmetros), evitando o acesso à pilha. Essa otimização pode ser *sugerida* ao compilador usando a palavra reservada `register`. Caso a variável local use algum registrador que não possa ser alterado, seu valor original deve ser salvo e recuperado após a execução (provavelmente usando a pilha).

valor de retorno

O **valor de retorno** de uma função é passado de volta à função chamadora em um registrador específico ou, no caso de valores de retorno maiores, em um grupo de registradores. Um caso especial, no qual o valor é grande demais para ser retornado dessa forma[11], usa um truque: a função chamadora reserva espaço suficiente em seu próprio *frame* e passa um endereço à função chamada em um parâmetro "oculto". A função chamada fará uma cópia do valor a ser retornado no endereço especificado pelo parâmetro oculto.

uso dos registradores

No caso particular do ARM32, o uso dos registradores nas funções é convencionado conforme a tabela a seguir:

r0	a1	Parâmetros Valor de retorno Rascunho
r1	a2	
r2	a3	
r3	a4	
r4	v1	Variáveis locais (o valor original deve ser preservado)
r5	v2	
r6	v3	
r7	v4	
r8	v5	
r9	v6	
r10	v7	
r11	v8/fp	
r12	fp	*Frame pointer* (deve ser preservado)
r13	sp	*Stack pointer* (deve ser preservado)
r14	lr	*Link register* (endereço de retorno)
r15	pc	*Program counter*

O valor anterior do *frame pointer* sempre é salvo na pilha pelo prólogo da função chamada; caso essa função chame outras funções, o valor do *link register* também é obrigatoriamente salvo.

Os parâmetros são passados em ordem, nos registradores r0-r3. Por exemplo:

- Um único parâmetro do tipo "int", "char" ou ponteiro é passado no registrador r0;
- Dois parâmetros "int" ou "char" são passados em r0 (primeiro parâmetro) e r1 (segundo parâmetro);
- Um parâmetro "long" é passado em r0 (32 bits menos significativos) e r1 (32 bits mais significativos);
- Dois parâmetros "long" usam r0-r1 para o primeiro parâmetro e r2-r3 para o segundo parâmetro;

11 Uma das excentricidades mais inesperadas da linguagem C é a possibilidade de que funções retornem estruturas!

- Vetores usados em parâmetros são normalmente substituídos por ponteiros.

Caso existam mais parâmetros, os parâmetros excedentes serão introduzidos no *frame* na pilha, conforme descrito anteriormente. Os mesmos registradores r0-r3 são usados para armazenar o valor de retorno da função, até o máximo de 128 bits (usando todos os quatro registradores). Caso esse conjunto de registradores não seja suficiente (ou em situações especiais, como por exemplo o retorno de uma estrutura), o mecanismo descrito anteriormente é utilizado.

GNU C Compiler (gcc)

O formato geral do comando do compilador C é

```
gcc [opções] <arquivos-fonte>
```

Quando se utiliza a compilação cruzada, é introduzido o *prefixo* correspondente à arquitetura, sistema operacional e ABI do sistema alvo, por exemplo **`arm-none-gnueabi-gcc`**.

Arquivos-fonte

Um ou mais arquivos-fonte devem ser especificados na linha de comando e serão processados em ordem. O programa `gcc` reconhece arquivos em C (`.c`, `.i`), arquivos em *assembler* (`.s`) e arquivos objeto binários realocáveis (`.o`, `.a`). Os arquivos contendo código-fonte na linguagem C são processados diretamente pelo `gcc` (eventualmente envolvendo o preprocessador, `cpp`); arquivos em *assembler* são tratados pelo programa `as` e os arquivos objeto diretamente pelo programa `ld`. Esses programas são chamados automaticamente pelo compilador, conforme a necessidade.

Resultados da compilação

A saída do compilador depende das **opções** especificadas na linha de comando:

- No caso de nenhuma opção de formato de saída esteja presente, o compilador vai executar o *linker* e produzir um arquivo executável único; se o nome para o arquivo de saída não tiver sido especificado, através da opção -o, o compilador vai gerar um arquivo com o nome `a.out` (a "saída do *assembler*");
- Caso a opção `-S` esteja presente, o compilador vai produzir um arquivo texto com a tradução no programa em *assembler* (`.s`), sem chamar o `gnu-as`. Se não for especificado um nome para o arquivo de saída, o arquivo produzido terá o mesmo nome que o arquivo-fonte, com a extensão `.s`;
- Caso a opção `-c` esteja presente, o compilador vai produzir um arquivo objeto realocável (`.o`) para ser posteriormente vinculado a uma biblioteca ou a um arquivo executável pelo *linker*. Se não for especificado um nome para o arquivo de saída, o arquivo produzido terá o mesmo nome que o arquivo-fonte, com a extensão `.o`.

Informações de depuração

Normalmente, informações de depuração (descrição dos símbolos, números de linha, etc.) não são incluídas nos arquivos produzidos; essas informações são utilizadas, por exemplo, por depuradores e analisadores de código. Utilize a opção `-g` na linha de comando para que essas informações sejam incluídas.

Opções comuns

-o ‹arquivo›	Define o nome do arquivo de saída
-c	Gera um arquivo objeto realocável (.o)
-S	Traduz para um arquivo fonte em *assembler* (.s)
-g	Inclui informações de depuração
-I ‹diretório›	Diretório para procurar `#include`
-D ‹simbolo›*[=‹valor›]*	Define um símbolo
-mthumb	Usar instruções de 16 bits (T16)
-mthumb-interwork	Usar tanto instruções de 16 bits (T16) quanto 32 bits (A32)
-march=‹arquitetura›	Versão da arquitetura (`armv4`, `armv7`, ...)
-mfloat-abi=‹nome›	Especifica ponto flutuante ("*soft*" para emular)
-mfpu=‹tipo›	Especifica tipo do processador de ponto flutuante
-mtune=‹opções›	Inclui mais opções sobre a arquitetura
-O ‹tipo›	Define o tipo ou nível de otimização a utilizar
-l ‹nome›	Comanda o *linker* para incluir uma biblioteca
-L ‹diretório›	Diretório para o *linker* procurar por bibliotecas
-Wl, ‹comandos›	Envia comandos específicos do *linker*
-Wa, ‹comandos›	Envia comandos específicos do *assembler*
-nostartfiles	Não introduzir inicializações ao *linker*
-static	Não usar bibliotecas dinâmicas
-nostdlib	Não incluir as bibliotecas padrão da linguagem C

Ambiente de desenvolvimento

Além dos programas utilizados anteriormente, vamos agora também compilar programas escritos na linguagem C, utilizando o compilador `arm-none-eabi-gcc`. Vamos utilizar o `Makefile` a seguir, que permite incluir tanto fontes em *assembler* (.s) quanto em C (.c). Você pode acrescentar os demais alvos que precisar ("ocd", "qemu", "gdb", etc.), copiando dos Makefiles anteriores.

Makefile

```
# Makefile
# Liste os arquivos fonte aqui:
FONTES = startup.s

# Arquivos de saída
EXEC = kernel.elf
MAP = kernel.map

PREFIXO = arm-none-eabi-
LDSCRIPT = kernel.ld
AS = ${PREFIXO}as
LD = ${PREFIXO}ld
GCC = ${PREFIXO}gcc
OBJ = $(FONTES:.s=.o)
OBJETOS = $(OBJ:.c=.o)

# Caminhos para as bibliotecas (ajuste!)
PATH_GCC = /usr/lib/gcc/arm-none-eabi/8.3.1
PATH_NEWLIB = /usr/lib/arm-none-eabi/lib

# Opções do compilador (ajuste!)
OPTS = -march=armv4 -g

# Opções do linker (ajuste!)
LDOPTS = -L${PATH_GCC} -L${PATH_NEWLIB}
LDOPTS += -lgcc -lc

# Alvo: Gerar executável
${EXEC}: ${OBJETOS}
	${LD} ${OBJETOS} -T ${LDSCRIPT} -M=${MAP} ${LDOPTS} -o $@

# Alvo: Compilar arquivos em C
.c.o:
	${GCC} ${OPTS} -c -o $@ $<

# Alvo: Montar arquivos em assembler
.s.o:
	${AS} -g -o $@ $<

# Alvo: Limpar tudo
clean:
	rm -f *.o ${EXEC} ${MAP}
```

Quando iniciarmos um programa em C a partir da função `main()` precisamos implementar algumas das funções do *runtime*, por exemplo para iniciar a pilha e zerar as variáveis que foram declaradas no segmento BSS. Faremos isso no arquivo `startup.s`:

startup.s

```
.text
.global start

// Ponto de entrada do programa.
start:
   // Configura modo do processador e pilha
   mov r0, #0b10011              // modo SVR
   msr cpsr, r0
   ldr r13, =inicio_stack

   // zera segmento bss
   mov r0, #0
   ldr r1, =inicio_bss
   ldr r2, =fim_bss
loop:
   cmp r1, r2
   bge cont
   str r0, [r1], #4
   b loop
cont:
   // executa a função main()
   bl main

stop:
   b stop  // se main() retornar...
```

Devemos criar o segmento `.bss` e vamos alocar espaço para a pilha e o *heap* no nosso *linker script* (`kernel.ld`):

kernel.ld

```
SECTIONS {   /* arquivo kernel.ld para placa Evaluator7T */
   /* Vetor de reset */
   . = 0;
   .reset : { *(.reset) }

   /* Segmentos text e data */
   . = 0x8000;
  .text : { *(.text) }
  .data : { *(.data) }

  /* Segmento bss */
  inicio_bss = .;
  .bss : { *(.bss) }
  . = ALIGN(4);
  fim_bss = .;

  /* Reserva espaço para a pilha e o heap */
  inicio_heap = .;
  . = . + 4096;
  . = ALIGN(8);
  inicio_stack = .;
}
```

Observe que alguns símbolos declarados dentro do *linker script* são utilizados pela função `start` em `startup.s`.

Atributos de função no GCC

O compilador C do GCC permite marcar funções com **atributos** especiais, que permitem maior controle sobre o prólogo e epílogo. Os atributos são definidos com uma notação especial (não portável):

```
void __attribute__((atributo)) funcao(void) { // ...
```

A tabela seguinte apresenta alguns desses atributos:

atributos no ARM

Atributo	Resultado
`interrupt("tipo")`	Define a função como *tratamento de interrupção*. "Tipo" pode ser "*IRQ*", "*FIQ*", "*ABORT*", "SWI" e "UNDEF". Gera prólogo e epílogo para salvar *todos os registradores* e retornar corretamente, conforme o tipo da interrupção.
`naked`	Não gera prólogo nem epílogo para a função. Útil para funções escritas totalmente em *assembler*.
`target("tipo")`	Escolhe o conjunto de instruções para a função. "Tipo" pode ser "*arm*", "*thumb*" ou especificar um processador ou coprocessador.

Por exemplo, o código a seguir define a função `timer()` como um serviço de interrupção para o evento "IRQ". Veja que não são recebidos parâmetros, todos os registradores usados são salvos e a instrução de retorno é diferente:

```
void __attribute__((interrupt("IRQ"))) timer(void) {
   ticks++;
}

// Código assembler produzido:
// timer:
//    push  {r2, r3}
//    ldr   r2, [pc, #16]    ; ticks
//    ldr   r3, [r2]
//    add   r3, r3, #1
//    str   r3, [r2]
//    pop   {r2, r3}
//    subs  pc, lr, #4
// .word   ticks
```

Assembler inline

O compilador `gcc` permite integrar código em *assembler* diretamente no código-fonte em `C`, sem a necessidade de montar arquivos `.s` e vinculá-los ao executável separadamente. Esse recurso é implementado através da função especial `asm()` (uma sintaxe específica do `gcc`, não sendo portável[12]). Além disso, a função `asm()` permite que o código escrito em *assembler* acesse símbolos definidos no código-fonte em `C`.

Quando a função `asm()` possui um único parâmetro (um *string*), este é copiado integralmente na saída do compilador para ser montado pelo *assembler*[13], juntamente com o código produzido pelo compilador.

assembler inline

```
void __attribute__((naked)) delay(int tempo) {
   // o parâmetro é passado em r0
   asm volatile("cmp r0, #0     \n\t"
                "moveq pc, lr   \n\t"
                "sub r0, r0, #1 \n\t"
                "b delay");
}
```

Neste exemplo, a função `delay()` é implementada inteiramente em *assembler*. Observe que é preciso incluir manualmente quebras de linha (‘\n’) para que o *assembler* consiga diferenciar duas instruções sucessivas. É opcional, mas comum, também incluir tabulações (‘\t’) para melhorar a legibilidade do código *assembler* gerado (opção `-S`).

parâmetros de saída

Quando o código no interior de `asm()` precisa **enviar** dados de volta ao compilador `C` é possível utilizar *parâmetros de saída*, que são especificados em `asm()` após dois pontos (“:”). No caso de processadores RISC, esses valores precisam necessariamente estar em registradores (ou seja, o “tipo” é sempre “=r”):

```
// retorna o modo atual do processador ARM
int modo(void) {
   int res;
   asm("mrs %0, cpsr       \n\t"
       "and %0, %0, #0x1f"
       : "=r" (res));
   return res;
}
```

O exemplo anterior solicita ao compilador que informe um registrador (“=r”) que o *assembler* pode **alterar**: vai corresponder à variável local “res”. O *assembler* consulta o nome do registrador através da macro “%0” (que substitui o “primeiro parâmetro”, “%1” substitui o “segundo parâmetro”, etc.). Observe o uso de dois pontos (“:”) na função `asm()`.

12 O compilador `clang` imita a sintaxe do `gcc` e também aceita esse tipo de notação.

13 Observe que esse código está sujeito a otimizações; caso o programador não queira que o processo de otimização seja realizado, deve incluir a palavra reservada `volatile`.

parâmetros de entrada

De forma análoga, para o *assembler* **ler** valores provenientes da linguagem C são usados *parâmetros de entrada*, que são especificados após um segundo dois pontos (":") em `asm()`. No caso de processadores RISC, esses valores precisam necessariamente estar em registradores (ou seja, o "tipo" é sempre "r").

```
// muda o modo do processador ARM
void seta_modo(int modo) {
   asm("msr cpsr, %0" : : "r" (modo));
}
```

Veja neste exemplo que `asm()` solicita ao compilador que informe o registrador ("r") onde o *assembler* pode encontrar o valor de "modo". O *assembler* consulta o nome desse registrador através da macro "%0" (que substitui o "primeiro parâmetro", "%1" substitui o "segundo parâmetro", etc.). Observe o uso dos dois pontos (":") na função `asm()`: neste caso não há parâmetros de saída, correspondendo ao espaço vazio após o primeiro ":".

"*clobber list*"

Finalmente, caso o código em *assembler* inserido faça alguma **modificação** no contexto do processador, além dos parâmetros de entrada ou saída, é necessário informar ao compilador, após um terceiro dois pontos (":") em `asm()` (chamado de "*clobber*"):

```
asm volatile("mov r0, #0xff          \n\t"
             "loop: subs r0, r0, #1  \n\t"
             "bne loop               \n\t"
             : : : "r0");
```

O código neste exemplo introduz um pequeno atraso através de um laço ocioso. Como o registrador r0 é alterado internamente, ele é introduzido na lista "*clobber*", apesar de não existirem parâmetros de entrada ou de saída. Um caso particular importante a ser lembrado é introduzir a palavra "*memory*" na lista, sempre que o código em *assembler* fizer alguma alteração na memória (usando as instruções `str`, `strb`, `strh`, `stm` ou `stc`): isso é importante para o compilador considerar otimizações na parte da função que está escrita em C.

O runtime do gcc

libgcc

As funções do *runtime* do `gcc` estão definidas no arquivo `libgcc.a`, que é instalado juntamente com o compilador. Como estamos usando o *linker* (`gnu-ld`) separadamente, precisamos informar a ele o nome (opção `-l`) e a localização (opção `-L`) dessa biblioteca. A biblioteca `libgcc.a` normalmente é instalada no diretório `/usr/lib/gcc/arm-none-eabi/`*<versão do gcc>*.

Um exemplo de comando do *linker* para incluir o *runtime* é:

```
arm-none-eabi-ld arq.o -L/usr/lib/gcc/arm-none-eabi/8.3.1 -lgcc
```

Biblioteca C padrão: newlib

O compilador cruzado `arm-none-eabi-gcc` normalmente é compilado junto com a biblioteca C "*newlib*", criada pela Red Hat. Essa biblioteca é preferida em relação à biblioteca padrão do GNU (`glibc`) por ser mais compacta e não depender de um sistema operacional ou de um sistema de proteção ou virtualização de memória. Uma versão ainda menor pode ser utilizada em aplicações com pouca memória, como em microcontroladores, chamada de "*newlib-nano*".

libc
libm

Os arquivos de biblioteca do *newlib* normalmente estão instalados no diretório `/usr/lib/arm-none-eabi/lib` (arquivos `libc.a`, `libm.a`, `libc_nano.c`, `libg.a`, etc.). O *newlib* implementa as funções definidas pelos padrões, entre elas:

- Funções de entrada e saída ("*stdio*"): `fopen`, `fread`, `puts`, `gets`, `getc`, `printf` e sua família;
- Funções padrão da linguagem C ("*stdlib*"): `malloc`, `free`, `atof`, `atoi`, `rand`, etc.;
- Funções de tempo ("*time*"): `time`, `ctime`, `localtime`, etc.;
- Funções matemáticas ("*math*"): `sin`, `cos`, `pow`, `round`, etc.;
- Funções de manipulação de *strings* ("*string*"): `strlen`, `strcat`, `strcpy`, `memcpy`, etc.;
- Funções de tipos ("*ctype*"): `isdigit`, `isalpha`, `isprint`, `toupper`, etc.;
- Tratamento de parâmetros em funções variádicas ("*stdarg*"): macros `va_start`, `va_arg`, etc;
- Saltos longos ("*setjmp*"): `setjmp`, `longjmp`.

Para que as funções do *newlib* sejam incluídas no arquivo executável, deve-se informar ao *linker*. Como estamos usando o *linker* (`gnu-ld`) separadamente, precisamos informar a ele o nome (opção `-l`) e a localização (opção `-L`) das bibliotecas. Um exemplo de comando do *linker* para incluir o *newlib* é:

```
arm-none-eabi-ld arq.o -L/usr/lib/arm-none-eabi -lc -lm
```

Esse comando, contudo, vai *falhar* na maioria das vezes. As funções da biblioteca *newlib* vão em algum momento chamar determinadas funções, que correspondem aos "pontos de entrada" de um suposto sistema operacional: funções para ler ou escrever dados de arquivos, acessar o *hardware* do relógio de tempo real, alterar a área de memória no *heap*, etc. Uma vez que não existe um sistema operacional, o programador é responsável por fornecer essas funções, mesmo que em muitos casos não executem nenhum processamento.

funções *stub*

A tabela a seguir apresenta as funções "*stub*" que podem ser necessárias ao funcionamento do *newlib*.

Símbolo	Finalidade	Implementação trivial
`_exit`	Encerrar o processo atual	*Loop* infinito
`_close`	Fechar um arquivo	Retornar erro
`environ`	Aponta uma lista de *strings*	Pode ser `NULL`
`_execve`	Executar um arquivo	Retornar erro
`_fork`	Duplicar o processo atual	Retornar erro
`_fstat`	Obter estado de arquivo	Retornar arquivo tipo "char"
`_getpid`	Obter o identificador o processo	Retornar o mesmo número
`_isatty`	Verifica se o arquivo é um terminal	Retornar sempre "1"
`_kill`	Enviar um sinal	Retornar erro
`_link`	Criar um link para um arquivo	Retornar erro
`_lseek`	Mover o ponteiro de arquivo	Retornar sucesso
`_open`	Abrir arquivo	Retornar erro
`_read`	Ler dados de arquivo	Retornar zero (ler "zero" bytes)
`_sbrk`	Alocar/desalocar memória	Alterar ponteiro `brk`
`_times`	Ler informação de tempo	Retornar zero
`_unlink`	Remover link/arquivo	Retornar erro
`_wait`	Esperar por processo	Retornar sucesso
`_write`	Escrever dados em arquivo	Retornar sucesso

sbrk

O único "*stub*" que precisa ser implementado de forma não trivial é `_sbrk()`, para que as funções de alocação de memória dinâmica funcionem (`malloc`, `realloc`, `free`, etc.). Essa função pode ser bastante simples, considerando o *linker script* que estamos usando, que define o símbolo `inicio_heap`:

```
// arquivo sbrk.c
#include <stdint.h>
extern uint8_t *inicio_heap;  // definido em kernel.ld

void *_sbrk(int incr) {
   static uint8_t *brk = inicio_heap;
   uint8_t *prev;

   prev = brk;
   brk += incr;

   return (void *)prev;
}
```

A implementação anterior não verifica se existe memória disponível: simplesmente movimenta o ponteiro `brk` para frente ou para trás, conforme a quantidade de *bytes* solicitada (parâmetro `incr`). À medida que o ponteiro `brk` se move para frente, mais memória é alocada para a *heap* (que cresce, aproximando-se do *stack*): uma função mais realista precisaria verificar, por exemplo, se houve

colisão com o *stack* e retornar um valor de erro (valor negativo, geralmente –ENOMEM)[14].

Outros "*stubs*" para a biblioteca *newlib* que são interessantes para serem implementados são `_times()` - para consultar algum tipo de base de tempo do sistema e assim possibilitar o uso das funções de data e hora - e `_read()` / `_write()` para os arquivos padrão do sistema ("*stdin*" e "*stdout*") para, por exemplo, enviar a saída do "terminal" ou "console" através de um canal serial (UART ou USB), usando funções de alto nível como `printf()`.

```
int _write(int handle, char *data, int size) {
   int i;
   if(handle != 1) return -1;   // 1 = stdout

   for(i=0; i<size; i++) {
      envia_um_caractere_pela_uart(data[i]);
   }

   return i;
}
```

O "*stub*" `_write()` é responsável por enviar caracteres pelo canal serial, quando o arquivo de destino é "*stdout*" ("*handle*" ou "número de arquivo" igual a um). Por exemplo, na placa Evaluator-7T, uma função para enviar um caractere pela UART 0 poderia ser:

```
void envia_um_caractere_pela_uart(char c) {
   // espera se o transmissor estiver ocupado
   while(bit_is_set(USTAT0, 6) == 0) ;

   UTXBUF0 = c;  // inicia envio de caractere
}
```

Assim, ao ser executada uma função como

```
printf("Numero[%d] = %2.2f\n", i, num[i]);
```

O *string* formatado é enviado pela UART 0.

14 O que normalmente não é de grande ajuda, pois quando isso acontece já não há muito mais o que fazer para impedir o desastre.