Universidade Federal do Ceará

Relatório - Arquitetura II

Equipe: GUSTAVO DAMASCENO DE CAMPOS 403407

Daniel de Moura Mascarenhas 403340

Para: Roberto Cabral

Quixadá - CE

2019

Sumário

Sumário..... **2**

Introdução..... **4**

Lendo o arquivo..... **6**

LSL..... **6**

LSL(1)..... 6

LSL(2)..... 7

LSR..... **8**

LSR(1)..... 8

LSR(2)..... 8

ASR..... **9**

ASR(1)..... 9

ASR(2)..... 10

ADD..... **10**

ADD(1)..... 10

ADD(2)..... 11

ADD(3)..... 11

ADD(4)..... 11

ADD(5)..... 12

ADD(6)..... 12

ADD(7)..... 12

SUB..... **12**

SUB(1)..... 13

Outros SUBs..... 13

MOV..... **13**

MOV(1)..... 13

MOV(2)..... 14

MOV(3)..... 14
CMP.....14
AND.....14
EOR.....15
ASR.....15
ADC.....15
SBC.....15
ROR.....16
TST..... 16
NEG.....16
CMN..... 17
ORR.....17
MUL.....17
MVN.....18
BIC..... 18
Outros casos.....18
Conclusão.....19

Introdução

Este trabalho consiste em implementar um programa (em python) que decodifique um código Thumb em seu respectivo mapa de memória.

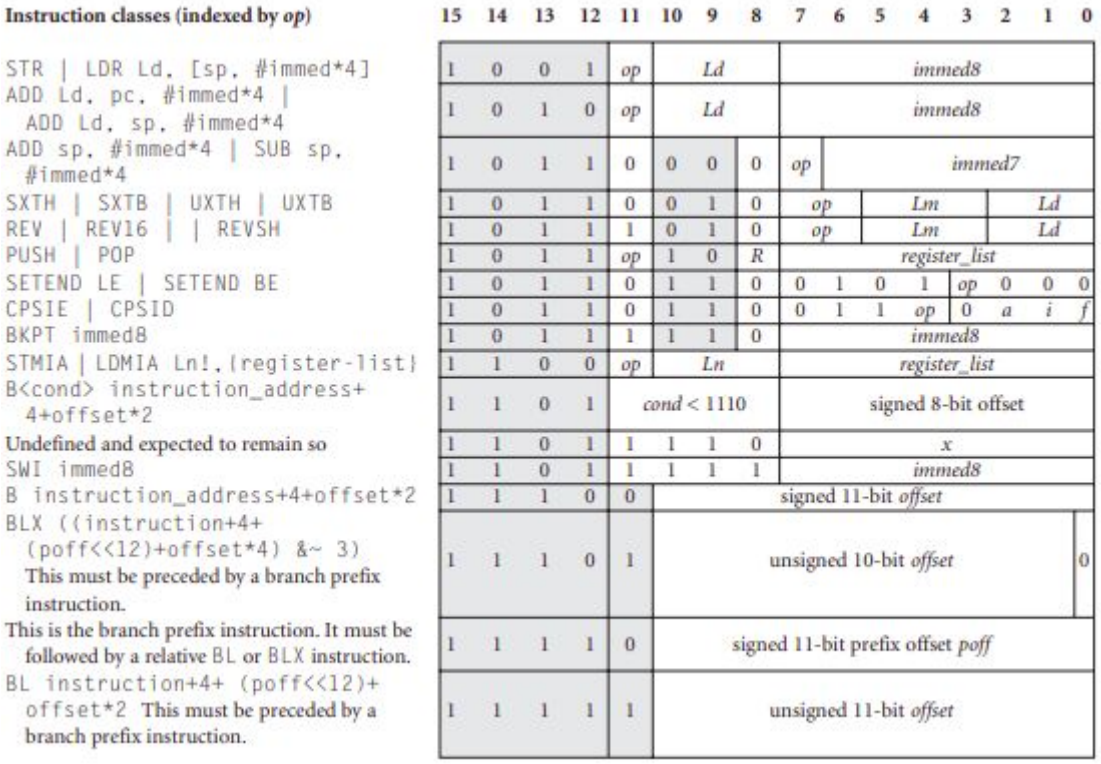
O decodificador deverá receber como entrada um arquivo de texto contendo o código a ser executado, em Thumb, e retornar seu respectivo mapa de memória. O arquivo de saída deve representar a memória de programa, onde cada linha corresponde a um endereço com alinhamento de 32 bits, no formato:

<endereço>:<conteúdo>

O programa deve também executar até encontrar um loop com a instrução “b .” OU uma instrução de formato indefinido. Além de tudo, o programa deverá ler a entrada de um **arquivo.s** e salvar o resultado em um **arquivo.out**

Vale ressaltar que o trabalho foi baseado da tabela B.5 segue abaixo:

Instruction classes (indexed by *op*)	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSL | LSR	0	0	0	0	*op*	*immed5*					*Lm*			*Ld*		
ASR	0	0	0	1	0	*immed5*					*Lm*			*Ld*		
ADD | SUB	0	0	0	1	1	0	*op*	*Lm*			*Ln*			*Ld*		
ADD | SUB	0	0	0	1	1	1	*op*	*immed3*			*Ln*			*Ld*		
MOV | CMP	0	0	1	0	*op*	*Ld/Ln*			*immed8*							
ADD | SUB	0	0	1	1	*op*	*Ld*			*immed8*							
AND | EOR | LSL | LSR	0	1	0	0	0	0	0	0	*op*		*Lm/Ls*			*Ld*		
ASR | ADC | SBC | ROR	0	1	0	0	0	0	0	1	*op*		*Lm/Ls*			*Ld*		
TST | NEG | CMP | CMN	0	1	0	0	0	0	1	0	*op*		*Lm*			*Ld/Ln*		
ORR | MUL | BIC | MVN	0	1	0	0	0	0	1	1	*op*		*Lm*			*Ld*		
CPY Ld, Lm	0	1	0	0	0	1	1	0	0	0	*Lm*			*Ld*		
ADD | MOV Ld, Hm	0	1	0	0	0	1	*op*	0	0	1	*Hm & 7*			*Ld*		
ADD | MOV Hd, Lm	0	1	0	0	0	1	*op*	0	1	0	*Lm*			*Hd & 7*		
ADD | MOV Hd, Hm	0	1	0	0	0	1	*op*	0	1	1	*Hm & 7*			*Hd & 7*		
CMP	0	1	0	0	0	1	0	1	0	1	*Hm & 7*			*Ln*		
CMP	0	1	0	0	0	1	0	1	1	0	*Lm*			*Hn & 7*		
CMP	0	1	0	0	0	1	0	1	1	1	*Hm & 7*			*Hn & 7*		
BX | BLX	0	1	0	0	0	1	1	1	*op*		*Rm*			0	0	0
LDR Ld, [pc, #immed*4]	0	1	0	0	1	*Ld*			*immed8*							
STR | STRH | STRB | LDRSB *pre*	0	1	0	1	0	*op*		*Lm*			*Ln*			*Ld*		
LDR | LDRH | LDRB | LDRSH *pre*	0	1	0	1	1	*op*		*Lm*			*Ln*			*Ld*		
STR | LDR Ld, [Ln, #immed*4]	0	1	1	0	*op*	*immed5*					*Ln*			*Ld*		
STRB | LDRB Ld, [Ln, #immed]	0	1	1	1	*op*	*immed5*					*Ln*			*Ld*		
STRH | LDRH Ld, [Ln, #immed*2]	1	0	0	0	*op*	*immed5*					*Ln*			*Ld*		



Instruction classes (indexed by *op*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	*op*	*Ld*			*immed8*								`STR | LDR Ld, [sp, #immed*4]`
1	0	1	0	*op*	*Ld*			*immed8*								`ADD Ld, pc, #immed*4 | ADD Ld, sp, #immed*4`
1	0	1	1	0	0	0	0	*op*	*immed7*							`ADD sp, #immed*4 | SUB sp, #immed*4`
1	0	1	1	0	0	1	0	*op*		*Lm*			*Ld*			`SXTH | SXTB | UXTH | UXTB`
1	0	1	1	1	0	1	0	*op*		*Lm*			*Ld*			`REV | REV16 | | REVSH`
1	0	1	1	*op*	1	0	*R*	*register_list*								`PUSH | POP`
1	0	1	1	0	1	1	0	0	1	0	1	*op*	0	0	0	`SETEND LE | SETEND BE`
1	0	1	1	0	1	1	0	0	1	1	*op*	0	*a*	*i*	*f*	`CPSIE | CPSID`
1	0	1	1	1	1	1	0	*immed8*								`BKPT immed8`
1	1	0	0	*op*	*Ln*			*register_list*								`STMIA | LDMIA Ln!,{register-list}`
1	1	0	1	*cond* < 1110				signed 8-bit offset								`B<cond> instruction_address+4+offset*2`
1	1	0	1	1	1	1	0	*x*								Undefined and expected to remain so
1	1	0	1	1	1	1	1	*immed8*								`SWI immed8`
1	1	1	0	0	signed 11-bit *offset*											`B instruction_address+4+offset*2`
1	1	1	0	1	unsigned 10-bit *offset*										0	`BLX ((instruction+4+ (poff<<12)+offset*4) &~ 3)` This must be preceded by a branch prefix instruction.
1	1	1	1	0	signed 11-bit prefix offset *poff*											This is the branch prefix instruction. It must be followed by a relative `BL` or `BLX` instruction.
1	1	1	1	1	unsigned 11-bit *offset*											`BL instruction+4+ (poff<<12)+ offset*2` This must be preceded by a branch prefix instruction.

Lendo o arquivo

Começamos abrindo o arquivo e fazendo uma leitura para identificar todas as instruções e assim começar a fazer o programa com as saídas com o formato endereço e conteúdo, já mostrado acima. Bem, para abrir o arquivo usaremos a linha de código seguinte: arquivo = open('exemplo.s', 'r'), com esse ‘r’ já faremos a leitura de todo o arquivo. Em seguida criaremos um vetor e ainda não colocamos nada, ele servirá para armazenamos as linhas do código em assembly. Por exemplo, na posição zero do vetor ficará a primeira linha do código assembly que pode ser ‘mov r0, #1’. Agora que entendemos como vai funcionar o vetor, vamos entender como fazemos isso em python.

Bem, criamos nosso vetor instrução com a seguinte linha de código: ‘instrução = []’ logo depois percorremos o arquivo com um ‘*for’*, dentro do ‘for’ utilizamos a função strip e append para adicionar no nosso vetor. Assim, o nosso vetor instrução terá em cada posição uma linha.

Logo em seguida, precisamos fechar o arquivo que abrimos. Então para isso, utilizaremos a seguinte linha de código, para fechar o arquivo: ‘arquivo.close()’

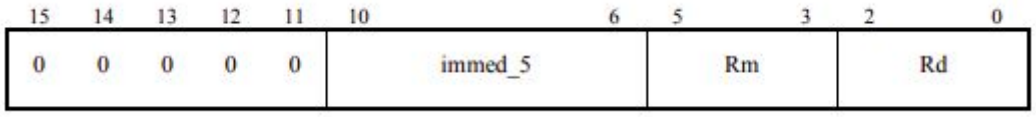
Agora que já temos cada linha armazenada em uma posição do vetor, precisamos pegar cada posição e colocar em um outro vetor chamado palavra. Esse vetor palavra armazenará em cada posição do vetor as palavras da linha. Com isso podemos saber qual instrução é daquela linha, como por exemplo, se é mov, add, sub, etc…

Bem, mas para isso iremos criar uma condição para saber se existe cada instrução no vetor palavra, então criamos um “if(‘mov’ in palavra):” para saber se “mov” está na palavra (vale ressaltar que estamos incrementando cada posição do vetor instrução) e faremos um “if” para todas as instruções mostradas na tabela acima.

LSL

Quando tiver a instrução LSL entrará na nossa condição “if(‘lsl’ in palavra):” nesse caso teremos dois casos para o lsl e antes de programar precisamos entender como funciona os dois tipos de lsl:

LSL(1)



Esta forma da instrução LSL (Logical Shift Left) é usada para fornecer diretamente o valor de um registro (LSL #0) ou o valor de um registro multiplicado por uma potência constante de dois. Zeros são inseridos no bit posições desocupadas pelo turno e os sinalizadores do código de condição são atualizados, com base no resultado.

E a sintaxe do lsl(1) será:

LSL <Rd>, <Rm>, #<immed_5>

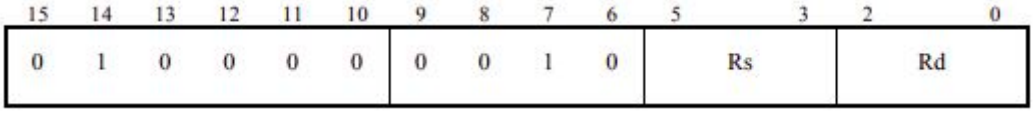
onde:

<Rd> É o registro que armazena o resultado da operação

<Rm> É o registro que armazena o resultado da operação

<immed_5> Especifica o valor do turno, no intervalo de 0 a 31

LSL(2)



Esta forma de LSL é usada para fornecer o valor de um registro multiplicado por uma potência variável de dois. Zeros são inserido nas posições de bit desocupadas. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do lsl(2) será:

LSL <Rd>, <Rs>

onde:

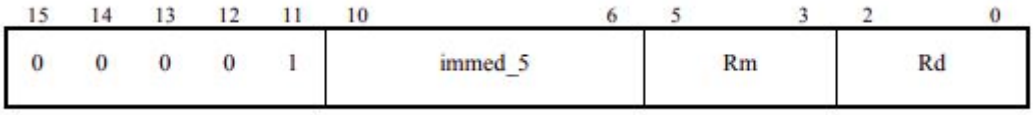
<Rd> Contém o valor a ser deslocado e é o registro de destino para o resultado da operação.

<Rs> É o registro que contém o valor do turno. O valor é mantido no byte menos significativo.

LSR

Quando tiver a instrução LSR entrará na nossa condição “if(‘lsr’ in palavra):” nesse caso teremos dois casos para o lsl e antes de programar precisamos entender como funciona os dois tipos de lsr:

LSR(1)



Este formulário da instrução LSR (Logical Shift Right) é usado para fornecer o valor não assinado de um registro, dividido por um poder constante de dois. O LSR executa um deslocamento lógico à direita do valor do registro <Rm>, e zeros são inseridos nas posições de bits desocupadas. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do lsr(1) será:

LSL <Rd>, <Rm>, #<immed_5>

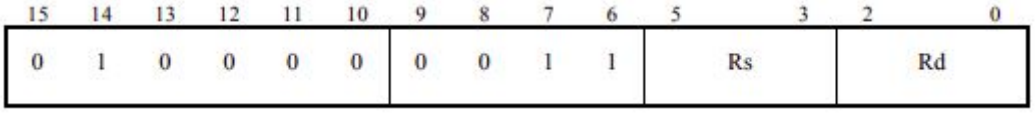
onde:

<Rd> É o registro que armazena o resultado da operação

<Rm> É o registro que armazena o resultado da operação

<immed_5> Especifica o valor do turno, no intervalo de 0 a 31

LSR(2)



Esta forma de LSR é usada para fornecer o valor não assinado de um registro dividido por uma potência variável de dois. Zeros são inseridos nas posições de bits desocupadas. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do lsr(2) será:

LSR <Rd>, <Rs>

onde:

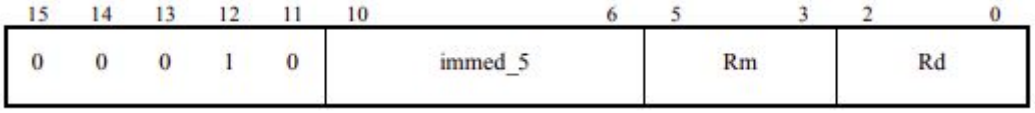
<Rd> Contém o valor a ser deslocado e é o registro de destino para o resultado da operação.

<Rs> É o registro que contém o valor do turno. O valor é mantido no byte menos significativo.

ASR

Quando tiver a instrução ASR entrará na nossa condição “if(‘asr’ in palavra):” nesse caso teremos dois casos para o asr e antes de programar precisamos entender como funciona os dois tipos de asr:

ASR(1)



Esta forma de instrução ASR (Arithmetic Shift Right) é usado para fornecer o valor assinado de um registro dividido por uma potência constante de dois. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do asr(1) será:

ASR <Rd>, <Rm>, #<immed_5>

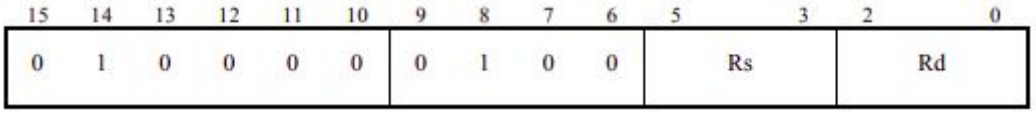
onde:

<Rd> É o registro que armazena o resultado da operação

<Rm> É o registro que armazena o resultado da operação

<immed_5> Especifica o valor do turno, no intervalo de 0 a 31

ASR(2)



Esta forma de ASR é usada para fornecer o valor assinado de um registro dividido por uma potência variável de 2. O sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do asr(2) será:

ASR <Rd>, <Rs>

onde:

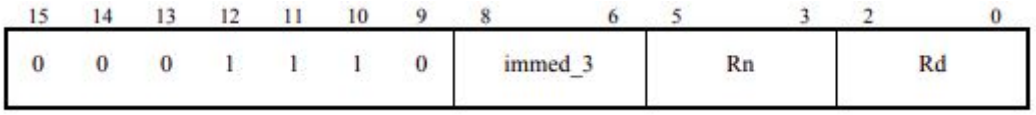
<Rd> Contém o valor a ser deslocado e é o registro de destino para o resultado da operação.

<Rs> É o registro que contém o valor do turno. O valor é mantido no byte menos significativo.

ADD

Quando tiver a instrução ADD entrará na nossa condição “if(‘add’ in palavra):” nesse caso teremos sete casos para o add e antes de programar precisamos entender como funciona os dois tipos de add:

ADD(1)

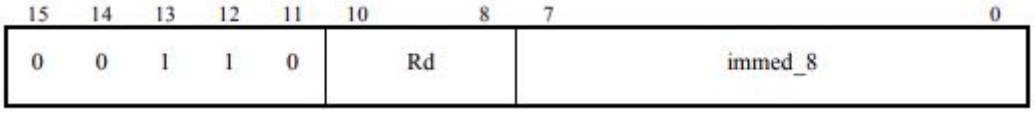


Essa forma de ADD adiciona um pequeno valor constante ao valor de um registro e armazena o resultado em um segundo registo. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do add(1) será:

ADD <Rd>, <Rm>, #<immed_5>

ADD(2)

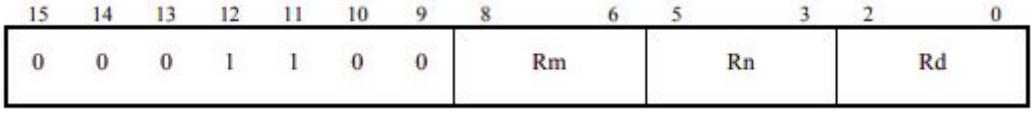


Essa forma de ADD adiciona um grande valor imediato ao valor de um registro e armazena o resultado novamente no mesmo registro. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do add(2) será:

ADD <Rd>, #<immed_5>

ADD(3)

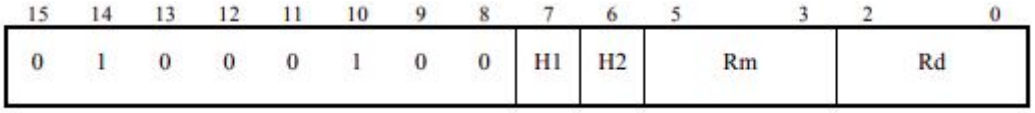


Essa forma de ADD adiciona o valor de um registro ao valor de um segundo registro e armazena o resultado em um terceiro registro. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do add(3) será:

ADD <Rd>, <Rn>, <Rm>

ADD(4)

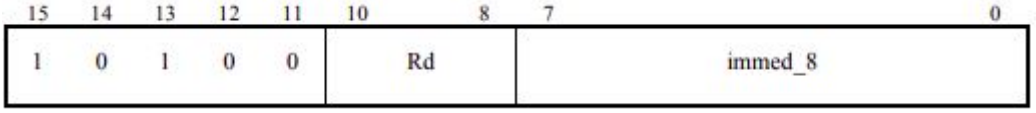


Esta forma de ADD adiciona os valores de dois registros, um ou ambos os quais são registros altos. Ao contrário do instrução ADD(3), esta instrução não altera os sinalizadores.

E a sintaxe do add(3) será:

ADD <Rd>, <Rm>

ADD(5)

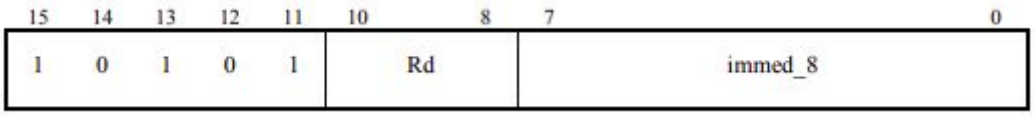


Esta forma de ADD agrega um valor imediato ao PC e grava o endereço relativo ao PC resultante em um registro de destino. O imediato pode ser qualquer múltiplo de 4 no intervalo de 0 a 1020. Os códigos de condição são não afetado.

E a sintaxe do add(2) será:

ADD <Rd>, PC, #<immed_8> * 4

ADD(6)

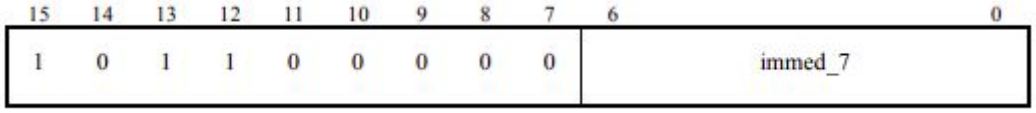


Essa forma de ADD adiciona um valor imediato ao SP e grava o endereço relativo ao SP resultante em um registro de destino. O imediato pode ser qualquer múltiplo de 4 no intervalo de 0 a 1020. Os códigos de condição são não afetado.

E a sintaxe do add(2) será:

ADD <Rd>, SP, #<immed_8> * 4

ADD(7)



Essa forma de ADD incrementa o SP em quatro vezes um imediato de 7 bits (ou seja, um múltiplo de 4 no intervalo 0 a 508). Os códigos de condição não são afetados.

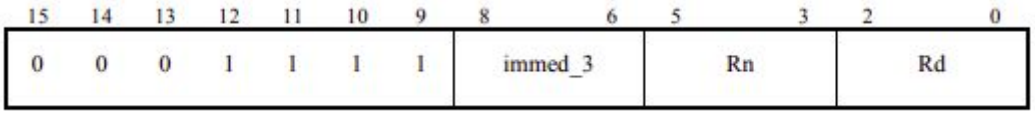
E a sintaxe do add(2) será:

ADD SP, #<immed_7> * 4

SUB

Quando tiver a instrução SUB entrará na nossa condição “if(‘sub’ in palavra):” nesse caso teremos quatro casos para o sub e antes de programar precisamos entender como funciona os dois tipos de sub:

SUB(1)



Esta forma da instrução SUB (Subtração) subtrai um pequeno valor constante do valor de um registro e armazena o resultado em um segundo registro. Os sinalizadores de código de condição são atualizados, com base no resultado.

E a sintaxe do sub(1) será:

ADD <Rd>, <Rm>, #<immed_3>

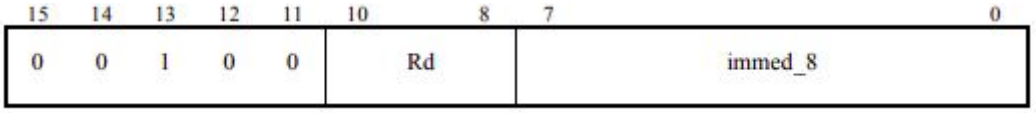
Outros SUBs

Bem, as outras instruções com o SUB será semelhante com as instruções que já criamos aqui, será apenas diferentes os valores que iremos setar, pois cada instrução tem os seus valores setados.

MOV

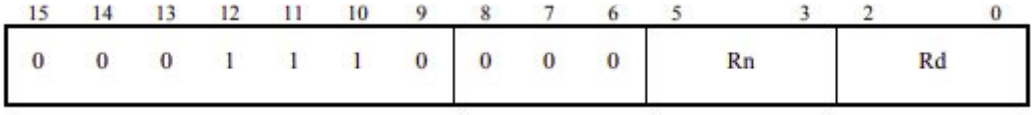
Quando tiver a instrução MOV entrará na nossa condição “if(‘mov’ in palavra):” nesse caso teremos três casos para o mov e antes de programar precisamos entender como funciona os dois tipos de mov:

MOV(1)



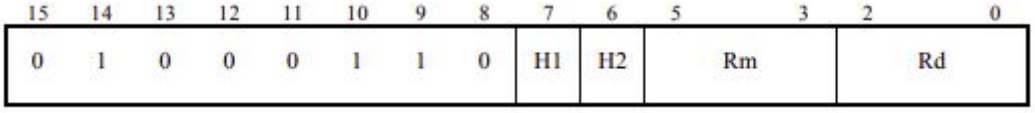
Esta forma da instrução MOV (Move) move um grande valor imediato para um registrador. O código de condição os sinalizadores são atualizados, com base no resultado. A sintaxe dele é um registrador e um imediato, ou seja estou colocando um valor para o meu registrador.

MOV(2)



Essa forma de MOV é usada para mover um valor de um registro baixo para outro, e os sinalizadores são definidos de acordo com para esse valor. A sintaxe é um registrador e outro registrador.

MOV(3)

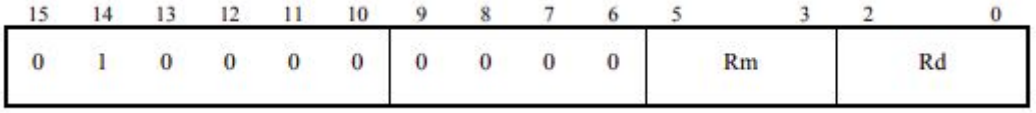


Essa forma de MOV é usada para mover um valor para, de ou entre registros altos. Ao contrário do MOV de baixo registro instrução descrita em MOV (2) na página A7-67, esta instrução não altera os sinalizadores. A sintaxe é um registrador e outro registrador.

CMP

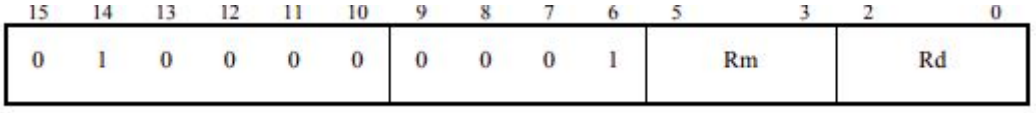
Os casos das instruções de comparação (que são quatro). São muito parecidas com o mov que já foi mostrada em cima, os diferenciais será os valores setados. Mas as sintaxes serão as mesmas já utilizadas.

AND



A instrução AND (AND lógico) executa um AND bit a bit dos valores em dois registradores. A condição sinalizadores de código são atualizados, com base no resultado.

EOR



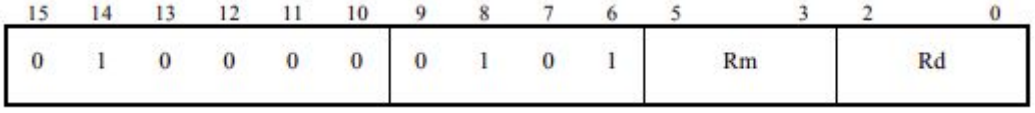
A instrução EOR (Exclusivo OR) executa um EOR bit a bit dos valores em dois registradores. A condição sinalizadores de código são atualizados, com base no resultado.

ASR

Quando tiver a instrução ASR entrará na nossa condição “if(‘asr’ in palavra):” nesse caso teremos dois casos para o asr e antes de programar precisamos entender como funciona os dois tipos de asr:

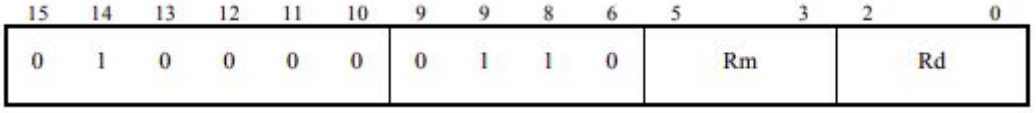
E os casos das instruções são semelhantes a algumas com quais já trabalhamos, só mudando os valores que setados, obviamente.

ADC



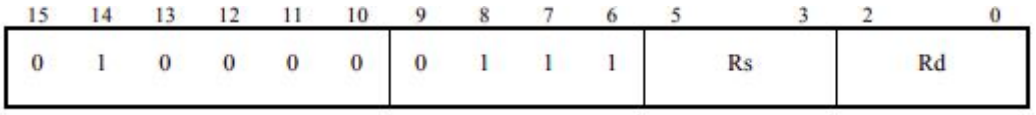
A instrução ADC (Adicionar com transporte) pode ser usada para sintetizar a adição de várias palavras. O código de condição os sinalizadores são atualizados, com base no resultado.

SBC



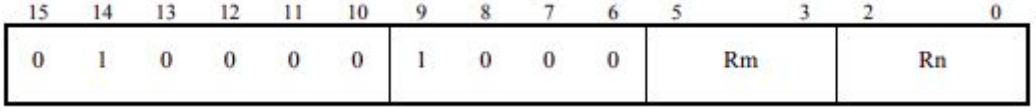
A instrução SBC (Subtrair com transporte) pode ser usada para sintetizar a subtração de várias palavras. Subtrai o valor do registro <Rm> e o valor de NOT (Carry Flag) do valor do registro <Rd>. A condição sinalizadores de código são atualizados, com base no resultado.

ROR



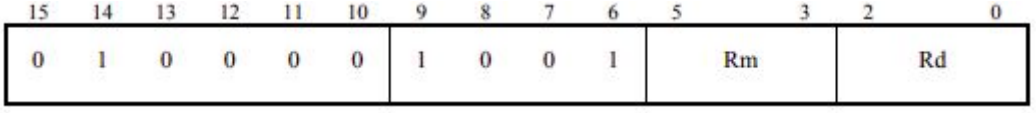
A instrução ROR (Rotate Right Register) é usada para fornecer o valor de um registro rotacionado por uma variável valor (de outro registro). Os bits que são rotacionados na extremidade direita são inseridos no bit desocupado posições à esquerda. Os sinalizadores de código de condição são atualizados, com base no resultado.

TST



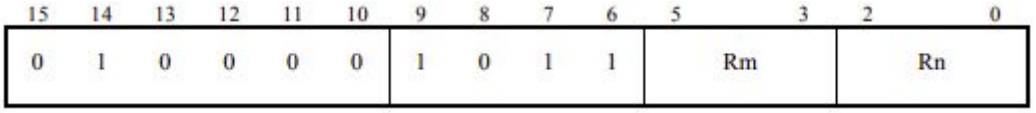
A instrução TST (Teste) é usada para determinar se um subconjunto específico de bits em um registro inclui em menos um bit definido. Um uso muito comum para o TST é testar se um único bit está definido ou não. A condição sinalizadores de código são atualizados, com base no resultado.

NEG



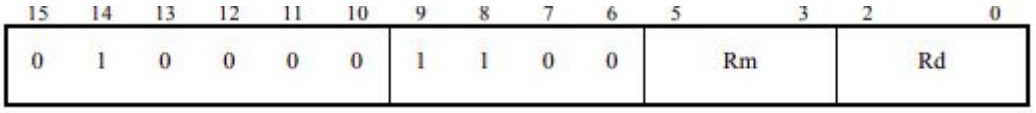
A instrução NEG (Negação) nega o valor de um registro e armazena o resultado em um segundo registro. o os sinalizadores de código de condição são atualizados (com base no resultado).

CMN



A instrução CMN (Compare Negative) compara um valor de registro com a negação de outro registro valor. Os sinalizadores de condição são atualizados, com base no resultado da adição dos dois valores do registro, para que instruções subsequentes podem ser executadas condicionalmente (usando uma ramificação condicional).

ORR

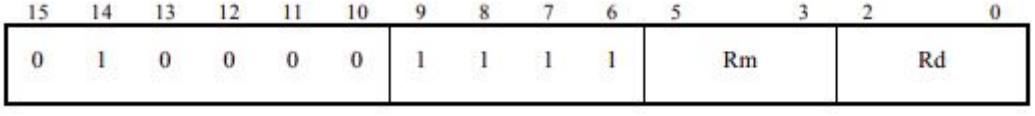


A instrução ORR (Logical OR) executa um OR bit a bit dos valores em dois registradores. O código de condição os sinalizadores são atualizados, com base no resultado.

MUL

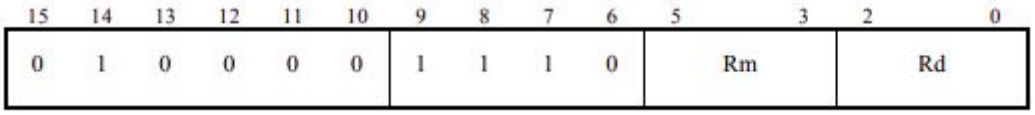
É uma instrução com a sintaxe semelhante com algumas instruções que nós já elaboramos e explicamos nesse relatório. Com a diferenciação dos bits setados, obviamente.

MVN



A instrução MVN (Move NOT) é usada para complementar um valor de registro, geralmente para formar uma máscara de bit. o sinalizadores de código de condição são atualizados, com base no resultado.

BIC



A instrução BIC (Bit Clear) executa um AND bit a bit do valor de um registro e a inversa bit a bit do valor de outro registro. Os sinalizadores de código de condição são atualizados, com base no resultado.

Outros casos

Bem as outras instruções que não apareceu no relatório é porque a sintaxe delas é semelhante e ficaria muito repetitivo colocar todas aqui, mas é importante sabermos algumas coisa, principalmente com as condições do branch que precisamos saber quais são os valores para poder setar. Segue abaixo os opcode das condições do branch:

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

E para os outros casos, basta analisarmos as tabelas que está disponibilizada no capítulo da Introdução.

Conclusão

Bem, esse trabalho consiste praticamente em fazer a leitura de um arquivo.s, após a leitura fazer um tratamento dos dados, para assim fazer a saída em um arquivo.out no formato:

<endereço>:<conteúdo>

O trabalho foi desenvolvido em python e faz a leitura de um código desenvolvido em assembly. O decodificador foi elaborado pelos alunos: Gustavo Damasceno e Daniel Mascarenhas.