Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores

Licenciatura em Engenharia Informática e de Computadores

Arquitetura de Computadores

Estudo do funcionamento de um processador

Alun@s:
A49418 - Roberto Petrisoru
A49447 - Francisco Castelo
A49506 - Pedro Malafaia

Docentes:
Rui Policarpo

Novembro 2022

Conteúdo

1 Análise da microarquitetura

1.1 Tipo da microarquitectura

"A microarquitetura do processador é do tipo von Neumann."

Esta afirmação é falsa, visto que a arquitetura deste processador não armazena os dados do programa e os dados das instruções na mesma memória. Assim, como este processador armazena os dados e as instruções em memórias diferentes, trata-se de uma arquitetura de Harvard.

1.2 Blocos Ext, LExt e RExt

Os três blocos têm como função estender o sinal.

- Ext - Estende o sinal de uma imediata.
- LExt - Estende o sinal de uma label.
- RExt - Estende o sinal de um registo (endereço) para ser carregado no *Program Counter*.

1.3 Operações da ALU

A ALU consegue fazer três tipos de operações:

- Soma (Figura 1)
- Subtração (Figura 2)
- Deslocamento (Figura 3)

A ALU tem também uma "operação especial", em que basicamente só passa o operando B (Figura 4).

OPALU	Tipo	Operação
00	soma	A+B
01	subtração	A-B
10	deslocamento/*shift*	A»B
11	operando B	B

Podemos diretamente ver no circuito, o resultado é guardado em `R.`:

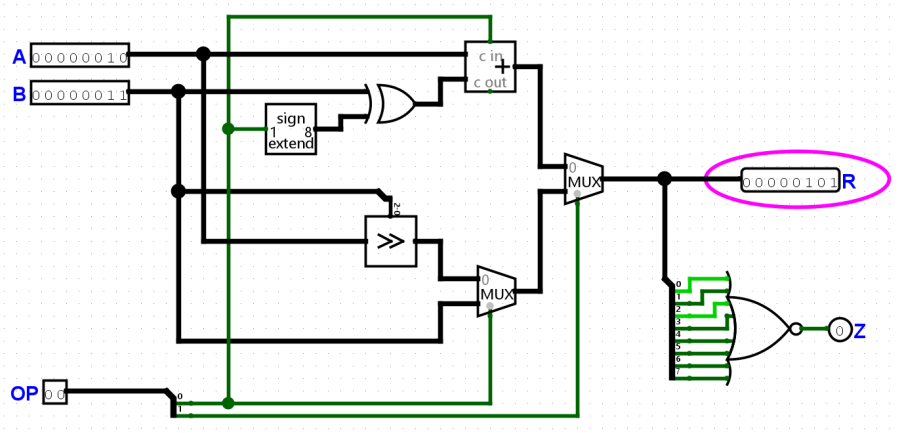

Figura 1: ALU - Soma

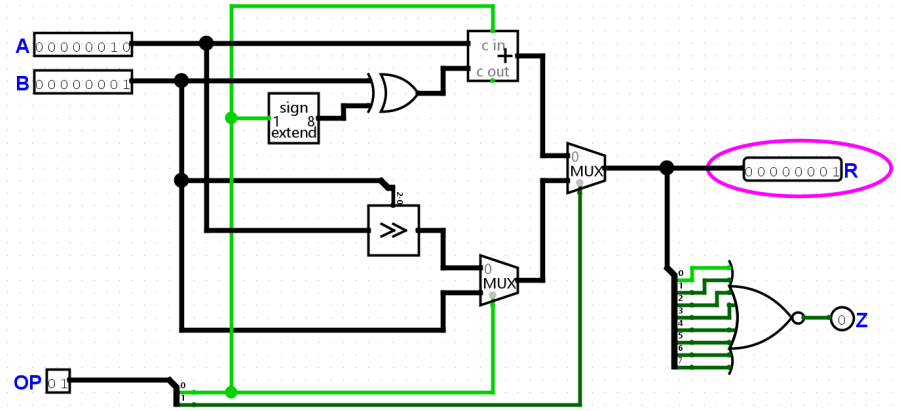

Figura 2: ALU - Subtração

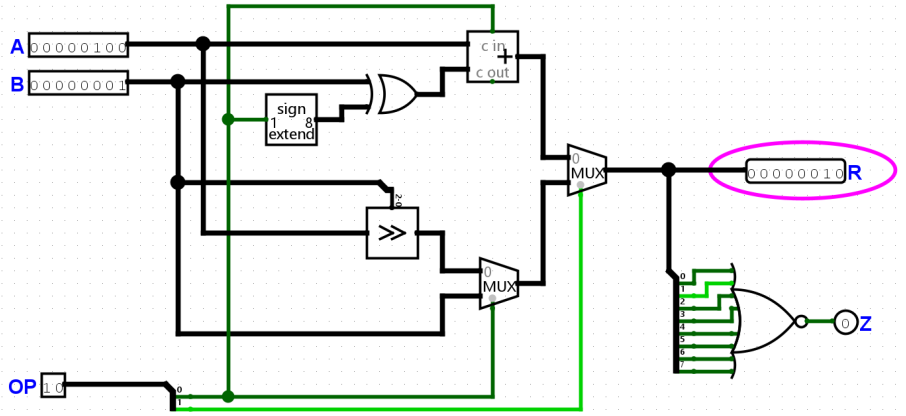

Figura 3: ALU - Deslocamento

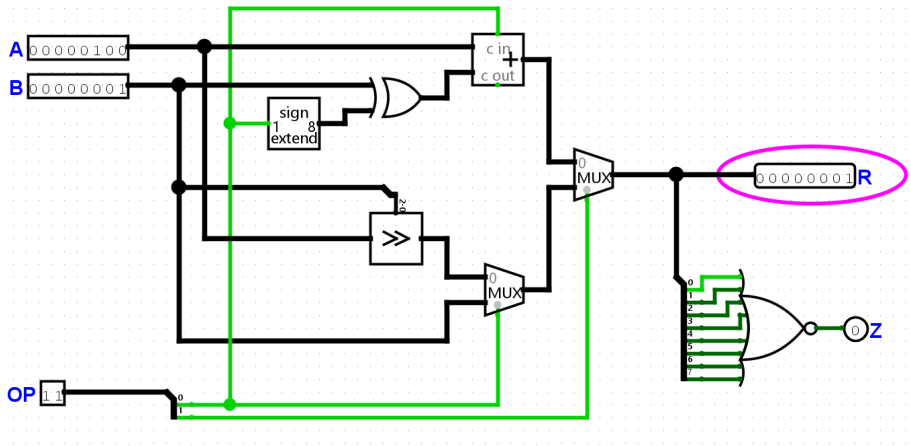

Figura 4: ALU - Operando B

2 Codificação das instruções

2.1 Mapa de codificação

O mapa de codificação que achamos mais adequado foi o que está apresentado na Figura 5.

	11	10	9	8	7	6	5	4	3	2	1	0
add rd, rn, rm	rd			rn			rm			opcode		
b label	label									opcode		
bne rn	-	-	-	rn			-	-	-	opcode		
cmp rn, rm	-	-	-	rn			rm			opcode		
ldr rd, [rn, #imm3]	rd			rn			imm3			opcode		
lsr rd, rn, #imm3	rd			rn			imm3			opcode		
mov rd, #imm6	rd			imm6						opcode		
str rd, [rn]	-	-	-	rd			rn			opcode		

Figura 5: Mapa de codificação

2.2 Código das instruções

Os valores dos `OPCODEs` e dos `OPALU` (seletor da operação que a ALU tem que fazer) estão apresentados na Figura 6.

Sabemos que as instruções `add` e `ldr` irão precisar de fazer somas, logo o `OPALU` das mesmas será `00`.

- `add rd, rn, rm : rd ← rn + rm; rd == 0 ? CPSR.Z ← 1 : CPSR.Z ← 0`
- `ldr rd, [rn, #imm3] : rd ← M[rn + imm3]`

A instrução `cmp` irá precisar de fazer uma subtração, para caso o resultado dê `0`, a flag `Z` é ativada. Sabemos assim que o seu `OPALU` é `01`.

- `cmp rn, rm : rn  rm == 0 ? CPSR.Z ← 1 : CPSR.Z ← 0`

A instrução `lsr` irá precisar de fazer um deslocamento para a direita, logo o seu `OPALU` é `10`.

- `lsr rd, rn, #imm3 : rd ← rn  imm3; rd == 0 ? CPSR.Z ← 1 : CPSR.Z ← 0`

	11	10	9	8	7	6	5	4	3	2	1	0
add rd, rn, rm	rd			rn			rm			0	0	0
ldr rd, [rn, #imm3]	rd			rn			imm3			0	0	1
cmp rn, rm	-	-	-	rn			rm			0	1	0
b label	label									0	1	1
lsr rd, rn, #imm3	rd			rn			imm3			1	0	0
bne rn	-	-	-	rn			-	-	-	1	0	1
mov rd, #imm6	rd			imm6						1	1	0
str rd, [rn]	-	-	-	rd			rn			1	1	1
										OPCODE		
										OPALU		

Figura 6: OPCODE e OPALU das instruções

A instrução `str` precisa de enviar o registo A (`rd`) para o *Data Bus* (vindo do banco de registos) e o registo B (`rn`) para a ALU, sendo este o endereço onde o registo A irá ser escrito na memória. Sendo assim, o registo B tem que ir para a ALU e usarmos o `OPALU 11`, para passar só B para a memória. Como o sinal `nWR` vai estar ativo, o registo A é escrito.

- `str rd, [rn] : M[rn] ← rd`

As instruções `mov`, `bne` e `b` não precisam de fazer uma operação na ALU em específico, porque neste caso, exercem sobre imediatas. Neste caso por exemplo, atribuimos o `OPALU 11` ao `mov` e o `OPALU 10` ao `b` e `bne` pois por exemplo no `mov` queremos que o operando B seja a imediata, no resto das intruções foi uma questão de organização.

- `mov rd, #imm6 : rd ← imm6`
- `b label : PC ← label`
- `bne label : CPSR.Z == 0 ? PC ← rn : PC ← PC + 1`

3 Descodificador de instruções

3.1 Sinais do descodificador

A tabela que achamos mais adequada para os sinais de cada instrução está representada na Figura 7.

	0/1	2	3	4	5	6	7	8	9	
	SD	nRD	nWR	EF	SC	SE	ER	SI	SO	
add	"10"	1	1	1	0	-	1	0	1	
ldr	"00"	0	1	0	1	0	1	0	1	
cmp	"10"	1	1	1	0	-	0	0	1	
str	"00"	1	0	0	0	-	0	0	1	
lsr	"10"	1	1	1	1	0	1	0	1	
bne	-	1	1	0	-	-	-	0	1	*SO = flag*
bne	-	1	1	0	-	-	-	1	1	*SO = flag*
mov	"01"	1	1	0	-	1	1	0	1	
b	-	1	1	0	-	-	-	0	0	

Figura 7: Tabela com os sinais do descodificador

- O sinal `nRD` está ativo nas instruções que **NÃO** precisam de ler da memória, isto é, porque é negado. Portanto as instruções que precisam de ler (`ldr`), tem o sinal a 0.
- O sinal `nWR` está ativo nas instruções que **NÃO** precisam de escrever da memória, isto é, porque é negado. Portanto as instruções que precisam de armazenar na memória (`str`), tem o sinal a 0.
- O sinal `EF` está ativo nas instruções que atualizam as flags da ALU (neste caso `Z`).
- O sinal `SC` está ativo nas instruções que dependem de uma immediata para o registo B na ALU.
- O sinal `SE` está ativo nas instruções que dependem de uma immediata que tem que ser estendida.
- O sinal `ER` está ativo nas instruções que precisam de armazenar valores no banco de registos.
- O sinal `SI` está ativo nas instruções que precisam de mudar o valor do *Program Counter* para um valor de um registo (A).
- O sinal `SO` está ativo nas instruções que não alteram o *Program Counter* diretamente.

Quando um dos sinais está a *Don't Care* (-), significa que não interfere na instrução portanto tanto faz o seu valor.

Sendo assim, a ROM do descodificador implementada tem as seguintes instruções:

Endereço	Valor
0x00	0x29D
0x01	0x29D
0x02	0x2A8
0x03	0x2A8
0x04	0x21D
0x05	0x21D
0x06	0x006
0x07	0x006
0x08	0x2BD
0x09	0x2BD
0x0A	0x30C
0x0B	0x20C
0x0C	0x2CE
0x0D	0x2CE
0x0E	0x204
0x0F	0x204

Portanto o tamanho da ROM será $10 * 16 = 160$ bits.

4 Codificação de programas em linguagem máquina

```
    mov r0, #0
    mov r1, #0
    mov r2, #4
    mov r4, #1
    ldr r3, [ r0, #0 ]
    add r1, r1, r3
    add r0, r0, r4
    cmp r0, r2
    bne r2
    lsr r1, r1, #2
    str r1, [ r2 ]
loop:
    b loop
```

O programa tem como objetivo testar as várias instruções, começando por mover valores imeditados para registos, depois carrega em `r3` o valor que está no endereço de memória `r0 + 0`. Soma esse valor ao valor que esta em `r1`, soma também a `r0` o valor de `r4`, depois compara `r0` com `r2` e caso sejam diferentes salta para o endereço de memória que esta em `r2`. De seguida entra num loop em que carrega em `r3` o valor do endereço incrementado de `r0`, que é somado a `r1`, este ciclo repete-se até o valor do `r0` seja igual ao valor do `r2`, ou seja, 4. Como estes são iguais, desloca `r1` dois bits para a direita e armazena esse valor no endereço de memória de `r2`. Depois de isto tudo, a aplicação corre repetidamente a mesma instrução graças ao `b loop`.

Traduzindo este código para código máquina, podemos ver os seus valores hexadecimais na tabela da Figura 8.

	11	10	9	8	7	6	5	4	3	2	1	0	HEX	LEGENDA
mov r0, #0	0	0	0	0	0	0	0	0	0	1	1	0	0x6	rd
mov r1, #0	0	0	1	0	0	0	0	0	0	1	1	0	0x206	rn
mov r2, #4	0	1	0	0	0	0	1	0	0	1	1	0	0x426	rm
mov r4, #1	1	0	0	0	0	0	0	0	1	1	1	0	0x80E	imm
ldr r3, [r0, #0]	0	1	1	0	0	0	0	0	0	0	0	1	0x601	don't care
add r1, r1, r3	0	0	1	0	0	0	0	1	1	0	0	0	0x258	opcode
add r0, r0, r4	0	0	0	0	0	0	1	0	0	0	0	0	0x20	label
cmp r0, r2	0	0	0	0	0	0	0	1	0	0	1	0	0x12	
bne r2	0	0	0	0	1	0	0	0	0	1	0	1	0x85	
lsr r1, r1, #2	0	0	1	0	0	1	0	1	0	1	0	0	0x254	
str r1, [r2]	0	0	0	0	0	1	0	1	0	1	1	1	0x57	
b loop	0	0	0	0	0	0	0	0	0	0	1	1	0x3	

Figura 8: Tabela com os código máquina (HEX)

Quanto a alteração proposta, não é possível fazer esta alteração pois já existem duas instruções em que nos seu `OPCODE` têm o valor `OPA = 00`, que é o valor para a `ALU` realizar a soma.

Quanto às vantages e desvantagens:

- *i)* Ficaríamos com menos bits para endereçar registos e constantes imediatas.
- *ii)* Algumas instruções ficam mais limitadas.

- *iii)* O bloco Ext teria de ser modificado para suportar imediatas de dois e cinco bits, o identificador de AB também seria necessário de passar a usar 2 bits.

	11	10	9	8	7	6	5	4	3	2	1	0
add rd, rn, rm	rd			rn			rm		0	0	0	0
ldr rd, [rn, #imm3]	rd			rn			imm2		0	0	0	1
cmp rn, rm	-	-	-	rn			rm		0	0	1	0
b label	label								0	0	1	1
lsr rd, [rn, rm]	rd			rn			rm		1	0	0	0
bne rn	-	-	-	rn			-	-	0	1	0	1
mov rd, #imm6	rd			imm5					0	1	1	0
str rd, [rn]	-	-	-	rd			rn		0	1	1	1
									OPCODE			
										OPALU		

Figura 9: Conversão do mapa para suportar a instrução proposta

5 Conclusão

Com este trabalho conseguimos aprofundar os nossos conhecimentos sobre microarquiteturas e também o funcionamento de um processador.