

UNIVERSIDADE FEDERAL DE SANTA MARIA  
CENTRO DE TECNOLOGIA  
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Matheus Dalmolin da Silva  
Julia Gabriela Santi Acosta

**ORGANIZAÇÃO DE COMPUTADORES**

Santa Maria, RS  
2018

## SUMÁRIO

<b>1</b>	<b>QUESTÃO 1 .....</b>	<b>3</b>
1.1	PRIMEIRO NÚMERO.....	3
1.2	SEGUNDO NÚMERO .....	4
<b>2</b>	<b>QUESTÃO 2 .....</b>	<b>6</b>
<b>3</b>	<b>QUESTÃO 3 .....</b>	<b>9</b>
3.1	PRIMEIRO NÚMERO.....	9
3.2	SEGUNDO NÚMERO .....	10
3.3	SOMA DOS NÚMEROS .....	11
<b>4</b>	<b>QUESTÃO 4 .....</b>	<b>14</b>
<b>5</b>	<b>QUESTÃO 5 .....</b>	<b>16</b>
5.1	LETRA A.....	16
5.2	LETRA B.....	19
5.3	LETRA C.....	22
	<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>29</b>

## 1 QUESTÃO 1

### 1.1 PRIMEIRO NÚMERO

O primeiro número para converter de decimal para binário é **1,234567**.

O primeiro passo é pegar o módulo do número para converter. Nesse caso, temos um número positivo, o seu módulo terá o mesmo valor, logo.

$$1,234567$$

Em seguida, separamos as partes inteira e fracionária.

$$I = 1$$

$$F = 0,234567$$

Para descobrirmos o valor em binário do nosso número decimal, usamos o método das divisões sucessivas, com o ponto de parada Quociente = 0.

$$\frac{1}{2} = 0$$

, resto = 1.

Como o nosso número teve só uma divisão sucessiva, estendemos o sinal até que o número fique em 8 bits. Com isso, temos o nosso número **1** em binário.

$$I = 000000001$$

Para acharmos a nossa parte fracionária em binário, usamos o método das multiplicações sucessivas. E o nosso ponto de parada vai ser quando fizermos 8 iterações, porque a parte fracionária do nosso número tem apenas 8 bits.

$$0,234567 \times 2 = 0,469134$$

$$0,469134 \times 2 = 0,938268$$

$$0,938268 \times 2 = 1,876536$$

$$0,876536 \times 2 = 1,753072$$

$$0,753072 \times 2 = 1,506144$$

$$0,506144 \times 2 = 1,012288$$

$$0,012288 \times 2 = 0,024576$$

$$0,024576 \times 2 = 0,049152$$

Chegando na nossa oitava iteração, que é o nosso ponto de parada, a parte fracionária **0,234567** é  $F = 00111100$  em binário.

Depois de já ter encontrado os valores em binário para as partes inteira e fracionária, basta concatenar as duas para ter o resultado em 16 bits, onde 8 bits representam

a parte inteira e 8 bits representam a parte fracionária.

$$N = 00000000100111100$$

## 1.2 SEGUNDO NÚMERO

O segundo número para converter de decimal para binário é **-78,901234**.

Para a conversão de bases, não levamos em conta o sinal por enquanto. Trabalharemos com o módulo do número em questão.

$$78,901234$$

Em seguida, separamos as partes inteira e fracionária.

$$I = 78$$

$$F = 0,901234$$

Fazemos o método da divisão longa até atingir Quociente = 0 para achar a parte inteira em binário.

$$\frac{78}{2} = 39$$

, resto = 0

$$\frac{39}{2} = 19$$

, resto = 1

$$\frac{19}{2} = 9$$

, resto = 1

$$\frac{9}{2} = 4$$

, resto = 1

$$\frac{4}{2} = 2$$

, resto = 0

$$\frac{2}{2} = 1$$

, resto = 0

$$\frac{1}{2} = 0$$

, resto = 1.

O resultado é lido do Quociente = 0, última divisão, até a primeira. Temos o valor **78** sendo  $I = 01001110$ , em binário.

Para encontrar a parte fracionária em binário, usamos o método da multiplicação longa. O nosso ponto de parada é quando o resultado da multiplicação for 1,0. Nesse caso, usamos como ponto de parada, a oitava iteração, porque o nosso número binário é composto por 8 bits para a parte fracionária, apenas.

$$0,901234 \times 2 = 1,802468$$

$$0,802468 \times 2 = 1,604936$$

$$0,604936 \times 2 = 1,209872$$

$$0,209872 \times 2 = 0,419744$$

$$0,419744 \times 2 = 0,839488$$

$$0,839488 \times 2 = 1,678976$$

$$0,678976 \times 2 = 1,357952$$

$$0,357952 \times 2 = 0,715904$$

Chegando no nosso ponto de parada, a nossa parte fracionária **0,901234** fica  $F = 11100110$  em binário.

Tendo os valores inteiro e fracionário já convertidos para binário, concatenamos esses números e temos o nosso número binário. Temos, em binário, o número decimal **78,901234**.

$$N = 0100111011100110$$

Para acharmos o valor **-78,901234**, temos que fazer o complemento de 2 do nosso número binário encontrado.

Primeiro passo é complementar todos os bits, invertendo os valores.

$$N = 1011000100011001$$

Após complementar todos os bits, somamos 1 ao  $N$ , resultando em:

$$N = 1011000100011010$$

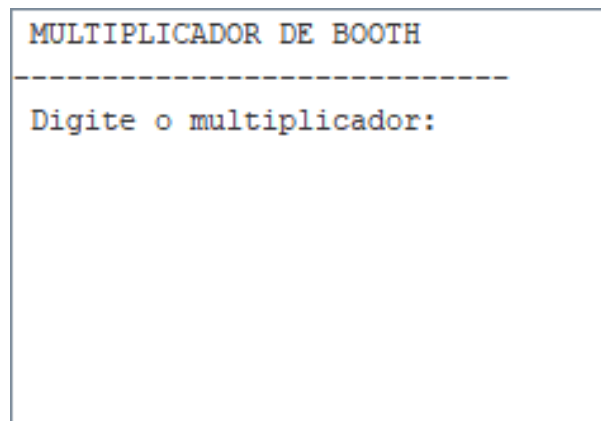
, que é a representação binária em complemento de 2 do número **-78,901234**.

## 2 QUESTÃO 2

O multiplicador de Booth que foi desenvolvido para essa questão, foram usados dois registradores básicos de 32 bits para armazenar o multiplicador e o multiplicando. Como todo algoritmo de multiplicação, se os números que estão sendo multiplicados são de  $N = 32$ , o resultado tem que ser representado por  $2N$ .

O programa desenvolvido para a questão 2 pede para o usuário os valores do multiplicador e do multiplicando, mostrados na Figuras 2.1 2.2.

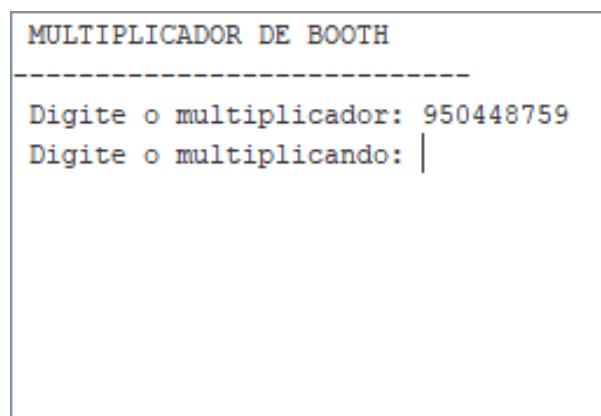
Figura 2.1 – Pede multiplicador para o usuário.



```
MULTPLICADOR DE BOOTH
-----
Digite o multiplicador:
```

Fonte: Arquivo pessoal.

Figura 2.2 – Pede multiplicando para o usuário.



```
MULTPLICADOR DE BOOTH
-----
Digite o multiplicador: 950448759
Digite o multiplicando: |
```

Fonte: Arquivo pessoal.

O programa lê números inteiros digitados pelo usuário, e está com um bug quando lê um valor maior que 2 bilhões. Números abaixo dessa magnitude, o multiplicador está funcionando perfeitamente, como mostra o exemplo resolvido nas imagens.

Após inserir o multiplicador e o multiplicando, o programa faz a multiplicação de acordo com o algoritmo de Booth, ilustrado na Figura 2.3.

E mostra o resultado em duas partes, o Produto HIGH e Produto LOW, como mostra a Figura 2.4.

Figura 2.3 – Algoritmo de Booth.

Iteração	Passo	Multiplicando	Produto	BA*
0 →	(0) valores Iniciais	1011	0000 1101	0
1 →	(1a) subtraia o multiplicando dos bits mais significativos do produto.	1011	0000 1101	0
	(1b) deslocamento aritmético para a direita	1011	0101 1101	0
2 →	(2a) Some o multiplicando dos bits mais significativos do produto.	1011	0010 1110	1
	(2b) deslocamento aritmético para a direita	1011	1101 1110	1
3 →	(3a) subtraia o multiplicando dos bits mais significativos do produto.	1011	1110 1111	0
	(3b) deslocamento aritmético para a direita	1011	0001 1111	1
4 →	(4a) nenhuma operação	1011	0001 1111	1
	(4b) deslocamento aritmético para a direita	1011	0000 1111	1
*Bit_Auxiliar			Resultado	

Fonte: Retirado de Giovani Baratto (2018).

Figura 2.4 – Resultado da multiplicação.

```

MULTIPLICADOR DE BOOTH
-----
Digite o multiplicador: 950448759
Digite o multiplicando: 1548875962
-----
Resultado da multiplicacao:
Produto HIGH = 0x146e0beb
Produto LOW = 0xe1a7f276
-- program is finished running --

```

Fonte: Arquivo pessoal.

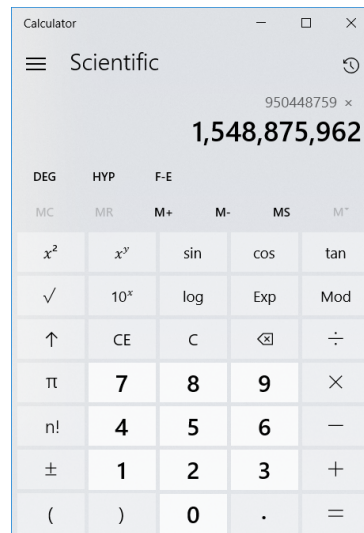
Se concatenarmos o Produto HIGH com o Produto LOW, temos o resultado da multiplicação em  $2N$  bits.

$0x146e0beb\_0xe1a7f276$

$0x146e0bebe1a7f276 = 1.472.127.235.927.831.158$

Podemos comprovar que a multiplicação está certa, fazendo a multiplicação na calculadora do sistema operacional. As Figuras 2.5 e 2.6 ilustram a multiplicação na calculadora.

Figura 2.5 – Multiplicação na calculadora do sistema operacional - Operadores.



Fonte: Arquivo pessoal.

Figura 2.6 – Multiplicação na calculadora do sistema operacional - Resultado.



Fonte: Arquivo pessoal.

Com os valores **0x12345678** e **0x9ABCDEF0**, disponibilizados pelo professor para testar o multiplicador, não funcionou. O problema está na leitura do número. Como são números com magnitude acima de 2 bilhões, acaba dando um erro e o compilador chama uma rota para encerrar o programa.

Para resolver esse problema, poderia ter lido o número como uma string, e a partir dessa string, converter para um número, e em seguida, para ponto flutuante.



### 3 QUESTÃO 3

Para convertermos um número decimal para ponto flutuante, precisamos, primeiramente, converter o número decimal para ponto fixo.

Como a conversão desses números já foi feita na **QUESTÃO 1**, não realizarei o passo-a-passo aqui nessa questão.

A representação em ponto flutuante de um número contém 3 partes. Usando o padrão IEEE-754, a primeira parte, representada pelo bit mais significativo, é a parte do sinal. Seguida por 8 bits, ou 11 para precisão dupla, que representam o expoente do número, é a parte do expoente polarizado. E por fim, os últimos 23, ou 52 para precisão dupla, são usados para a fração do número.

#### 3.1 PRIMEIRO NÚMERO

O primeiro número para ser convertido de decimal para ponto flutuante é **1,234567**. Primeiramente, multiplicamos o número binário por  $2^0$ .

$$1,234567 = 00000001,00111100 \times 2^0$$

Em seguida, deslocamos a vírgula até que o número fique no formato  $1, F$ . Nesse caso, o nosso número já está no formato, então, apenas descartamos os 0 à esquerda do 1.

$$1,234567 = 1,00111100 \times 2^0$$

O  $F$  é toda a parte fracionária do número. Para a representação em ponto flutuante, o nosso  $F$  tem 23 bits, para representação simples, ou 52 bits, para representação dupla.

Nesse caso, nosso  $F$  terá apenas 23 bits, pois estamos trabalhando com representação simples. Então pegamos o valor de  $F$ , extraído da representação binária do número, e acrescentamos 0 à sua direita até completar 23 bits.

$$F = 00111100000000000000000$$

Para encontrarmos o expoente polarizado, pegamos o valor do expoente do  $2^0$ , convertemos ele para binário em 8 bits e somamos com o peso. Como estamos trabalhando com precisão simples, o peso do expoente é **127**.

Logo, o expoente polarizado (EP) do nosso número em ponto flutuante se dá pela soma:

$$EP = 011111111 + 00000000 = 01111111$$

O número **1,234567** é um número positivo, logo o nosso sinal será positivo, representado por 0.

$$S = 0$$

Tendo os valores do sinal (S), expoente polarizado (EP) e da nossa fração (F), temos o nosso número **1,234567** em ponto flutuante, ilustrado na Tabela 3.1.

Também podemos representar esse valor em hexadecimal para facilitar a leitura,

Tabela 3.1 – Representação de 1,234567 em PF.

S	EP	F
0	011111111	001 1110 0000 0000 0000 0000

Fonte: Arquivo pessoal

nesse caso fica da seguinte forma.

$$0x3F9E0000$$

### 3.2 SEGUNDO NÚMERO

O segundo número a ser convertido de decimal para ponto flutuante é **-78,901234**.

Na representação em ponto flutuante de um número, é usada a forma sinal-e-magnitude, onde o sinal é representado em um bit exclusivo, e a magnitude representa o valor do número. De acordo com o padrão IEEE-754, o bit mais significativo é usado para representar o sinal.

Portanto, utilizaremos o módulo do número **-78,901234** para fazer a conversão, e no final, checaremos o sinal do número para ajustarmos o bit de sinal.

Como citado anteriormente, para converter um número decimal para ponto flutuante, precisamos, primeiro, converter esse número para ponto fixo.

$$78,901234 = 01001110,11100110$$

Em seguida, multiplicamos o número por  $2^0$ .

$$78,901234 = 01001110,11100110x2^0$$

Deslocamos a vírgula até que o número fique no formato  $1,F$ . E a cada vez que deslocamos a vírgula para a esquerda, é acrescentado 1 ao expoente de  $2^0$ . Nesse caso, deslocamos a vírgula 6 casas para a esquerda, resultando num acréscimo de 6 no expoente.

$$78,901234 = 01,00111011100110x2^6$$

Descartamos os zeros à esquerda do número, e temos:

$$78,901234 = 1,00111011100110x2^6$$

A fração do número fica com todo o valor após a vírgula. E como na representação de ponto flutuante em precisão simples, a nossa fração precisa ter 23 bits, adicionamos zeros à direita do número.

$$F = 00111011100110000000000$$

Para calcularmos o nosso expoente polarizado, pegamos o peso do nosso expoente, que no caso é 127, porque é precisão simples, e adicionamos o expoente do  $2^6$ .

Tabela 3.2 – Representação de -78,901234 em PF.

S	EP	F
1	10000101	001 1101 1100 1100 0000 0000

Fonte: Arquivo pessoal

Tabela 3.3 – Representação US em PF de 1,234567.

S	EP	US
0	01111111	001,001 1110 0000 0000 0000 0000

Fonte: Arquivo pessoal

Convertemos o expoente para binário, estendemos para 8 bits e fazemos a soma.

$$EP = 01111111 + 00000110 = 10000101$$

Como o número **-78,901234** é um número negativo, o bit de sinal será 1.

$$S = 1$$

Com os valores do sinal (S), expoente polarizado (EP) e fração (F) definidos, temos o número **-78,901234** em ponto flutuante, mostrado na Tabela 3.2.

Esse número também pode ser expresso em hexadecimal, da seguinte forma:

$$0xC29DCC00$$

### 3.3 SOMA DOS NÚMEROS

Para essa soma, defini  $N1 = 1,234567$  e  $N2 = -78,901234$ . Para somar os dois números em ponto flutuante, primeiramente precisamos igualar o expoente polarizado (EP). Com os valores definidos para  $N1$  e  $N2$ , usamos a seguinte fórmula para saber qual expoente temos que alterar.

$$d = EP(N1) - EP(N2)$$

$$d = 127 - 133$$

$$d = -6$$

Como temos  $d = -6$  sendo um valor negativo, ajustaremos o expoente de  $N1$ .

Convertemos as frações F para significandos sem sinal US, mostrados nas Tabelas 3.3 e 3.4.

Depois de ter as frações convertidas para significandos sem sinal, e sabendo que

Tabela 3.4 – Representação US em PF de -78,901234.

S	EP	US
1	10000101	001,001 1101 1100 1100 0000 0000

Fonte: Arquivo pessoal

Tabela 3.5 – Expoente ajustado.

S	EP	US
0	10000101	000,000 0010 0111 1000 0000 0000

Fonte: Arquivo pessoal

Tabela 3.6 – Representação de 1,234567 em PF com sinal.

S	EP	SS
0	10000101	000,000 0010 0111 1000 0000 0000

Fonte: Arquivo pessoal

temos que ajustar o expoente de  $N1$ .

Para igualar o expoente de  $N1$  com o de  $N2$ , temos que somar 6 unidades a ele. A cada unidade somada, a vírgula é deslocada uma cada para a direita. A Tabela 3.5 mostra o número com o expoente ajustado.

Após ajustarmos o expoente, temos que converter o significando sem sinal US para um significando com sinal SS. Verificamos os sinais para ver se algum número é negativo. Se for negativo, fazemos o complemento de 2 do significando, e se for positivo, não fazemos nada, pois o significando com sinal positivo tem a mesma representação de um significando sem sinal.

O número **1,234567** é positivo, logo, a sua representação com sinal SS será idêntica à representação sem sinal US, mostrada na Tabela 3.6

Como temos o valor **-78,901234**, está representado em ponto flutuante na Tabela 3.2, fazemos o complemento de dois do significando do número para ter o significando com sinal, mostrado na Tabela 3.7.

Depois de termos os significandos com sinal, fazemos a soma, ilustrada na Tabela 3.8.

Em seguida, pegamos o resultado da soma e complementamos novamente. Podemos ver na Tabela 3.9 o resultado complementado.

Como o significando já está no formato  $1,F$ , não precisamos ajustar a vírgula para normalizá-lo.

Com isso, a nossa soma chega ao fim com o resultado mostrado na Tabela 3.10.

Podemos representar o nosso resultado em hexadecimal pelo seguinte valor.

$0xC29B5400$

Para conferir se a soma foi feita de forma correta, convertemos o resultado em PF para decimal.

O resultado da soma é ilustrado na Tabela 3.10. Para converter esse valor para decimal, primeiramente verificamos os campos do expoente polarizado (EP) e da fração (F). Temos o valor  $EP = 10000101$ , que representa o valor **133** em decimal. Sendo um valor diferente de 0 e 255, o número em PF está normalizado.

Tabela 3.7 – Representação de -78,901234 em PF com sinal.

S	EP	SS
1	10000101	110,110 0010 0011 0100 0000 0000

Fonte: Arquivo pessoal

Tabela 3.8 – Soma dos números em PF.

	soma	SS
vem uns	0000 000 0100 1110 0000 0000 0000	
N1	000,000 0010 0111 1000 0000 0000	
N2	110,110 0010 0011 0100 0000 0000	
Resultado	110,110 0100 1010 1100 0000 0000	

Fonte: Arquivo pessoal

Tabela 3.9 – Resultado da soma complementado.

S	EP	US
1	10000101	001,001 1011 0101 0100 0000 0000

Fonte: Arquivo pessoal

Checando o bit de sinal, vemos que o sinal do número é negativo, representado pelo bit mais significativo, que tem valor **1**, nesse caso.

Passamos a fração para o formato  $1,F$  e em seguida convertemos para decimal.

$$1,001101101010100000000000 = 1,2135009765625$$

Com os valores do sinal (S), do expoente polarizado (EP) e da fração (F), calculamos o valor decimal com a seguinte fórmula.

$$N = (-1)^S \cdot (1, F) \cdot 2^{EP-Peso}$$

Substituindo na fórmula com os devidos valores, fazemos o cálculo.

$$N = (-1)^1 \cdot (1, 2135009765625) \cdot 2^{133-127}$$

$$N = -1,2135009765625 \cdot 2^6$$

$$N = -77.6640625$$

Aproximadamente o número esperado da subtração, que é **-77,666667**. A representação em ponto flutuante acaba sendo limitada porque não conseguimos representar certos valores com um número limitado de bits. Nesse caso, tínhamos apenas 8 bits na parte fracionária do número, o que acaba limitando muito o valor na conversão.

Tabela 3.10 – Resultado da soma de 1,234567 e -78,901234.

S	EP	F
1	10000101	001 1011 0101 0100 0000 0000

Fonte: Arquivo pessoal

## 4 QUESTÃO 4

O programa desenvolvido para a questão 4 calcula as raízes  $x'$  e  $x''$  de uma equação de segundo grau  $ax^2 + bx + c$ , dados  $a$ ,  $b$  e  $c$  informados pelo usuário, ilustrada na Figura 4.1.

Figura 4.1 – Pede os valores de  $a$ ,  $b$  e  $c$  para o usuário.

```
Programa que calcula as raizes de uma
funcao quadratica de segundo grau!
-----
Informe os valores de A, B e C!
A = 1
B = 4
C = -5|
```

Fonte: Arquivo pessoal.

Após receber os dados do usuário, o programa calcula as raízes e mostra o resultado para o usuário, como podemos ver na Figura 4.2.

Figura 4.2 – Resultado das raízes de  $ax^2 + bx + c$ .

```
Programa que calcula as raizes de uma
funcao quadratica de segundo grau!
-----
Informe os valores de A, B e C!
A = 1
B = 4
C = -5
-----
Resultados:
x' = 1.0
x'' = -5.0
-- program is finished running --
```

Fonte: Arquivo pessoal.

A questão 4 diz que os valores tem que ser representados em precisão dupla. Na implementação do programa, podemos ver que usamos os registradores  $\$f$ , localizados no coprocessador 1, usados para armazenar dados com informação acima de 32 bits. Juntamente com os registradores  $\$f$  temos que utilizar as instruções específicas para operar esses registradores.

Para pegar o conteúdo informado pelo usuário, leio o dado como um inteiro e guardo em um registrador básico. Em seguida, movo o dado para algum registrador do coprocessador para converter esse dado para precisão dupla.

A Figura 4.3 mostra a parte do código em que o dado é movido para o coprocessador e convertido para ponto flutuante, precisão dupla.

Figura 4.3 – Converte inteiro para PF - Parte do código.

```
# pede para o usuário os valores de A, B e C
li      $v0, 5 # "A"
syscall

mtc1    $v0, $f4 # move o valor de $t0 para $f6
cvt.d.w $f4, $f4 # converte 3 PI para 3 PF
la      $t1, A
sdcl    $f4, 0($t1) # guarda valor de "A" na memória
```

Fonte: Arquivo pessoal.

O programa pega um valor decimal, digitado pelo usuário, move para o coprocessador, para depois converter para precisão dupla. Em seguida, grava na memória o valor em PF.

Depois de fazer esses passos detalhados acima, o programa calcula o delta com o procedimento *calculaDelta* e grava na memória o retorno do procedimento.

Em seguida, compara o valor de delta para ver se é menor que zero. Se for menor que zero, o programa é encerrado. Caso contrário, ele calcula as raízes  $x'$  e  $x''$ .

A Figura 4.4 ilustra a comparação de  $\Delta < 0$ .

Figura 4.4 – Compara  $\Delta < 0$  - Parte do código.

```
li      $t0, 0
mtc1    $t0, $f8
cvt.d.w $f8, $f8
c.lt.d  $f0, $f8
bclt    deltaNegativo
```

Fonte: Arquivo pessoal.

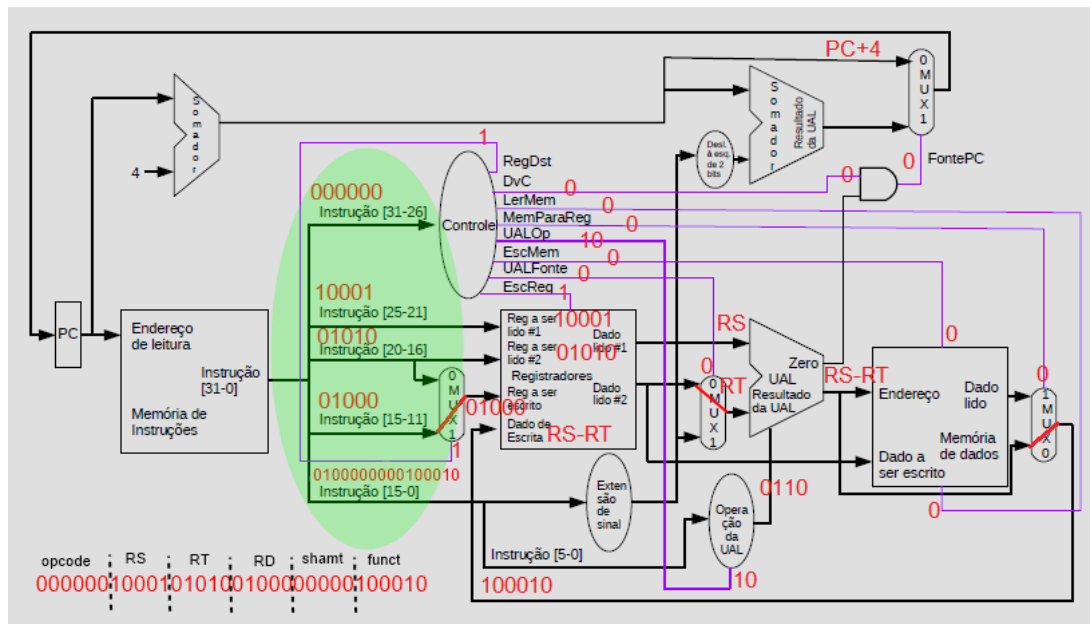
## 5 QUESTÃO 5

### 5.1 LETRA A

A instrução que será explicada nessa seção é **sub \$t0, \$s1, \$t2**, que é uma instrução que subtrai o conteúdo de \$s1 pelo conteúdo de \$t2 e armazena o resultado no registrador \$t0.

A Figura 5.1 mostra como a instrução é recebida a partir de um endereço, decodificada e separada para ser processada pelos caminhos de controle, registradores e memória.

Figura 5.1 – A - Separação da instrução nos caminhos.



Fonte: Arquivo pessoal.

Devido ao opcode 000000 (bits 31-26) o caminho de controle recebe os parâmetros demonstrados. A Figura 5.2 mostra a tabela de valores, dependendo do tipo de instrução, para as linhas de controle.

Figura 5.2 – Determinação das linhas de controle.

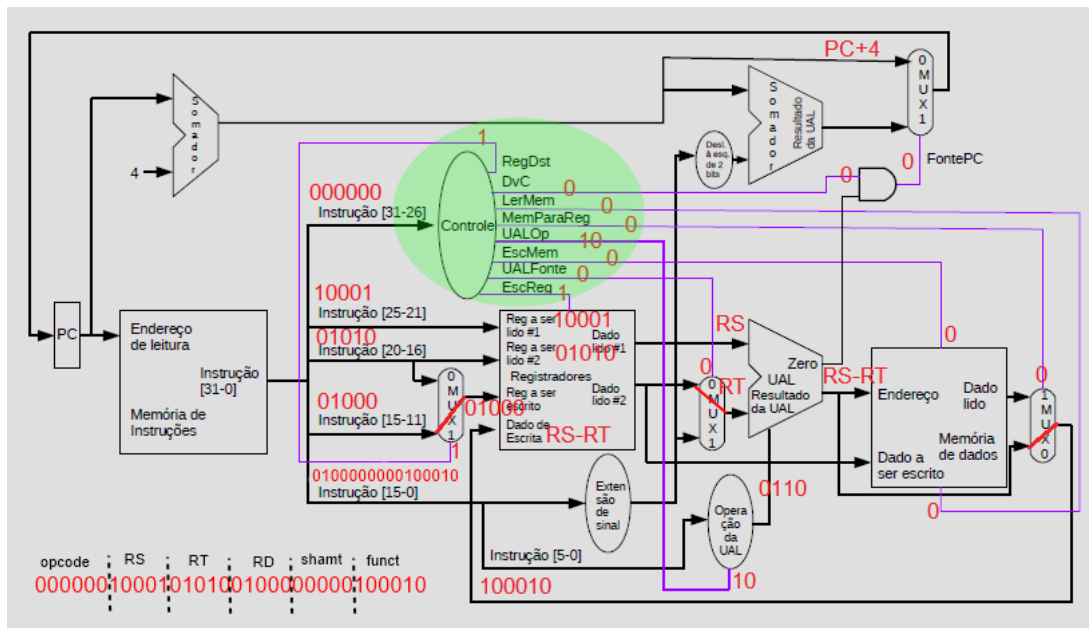
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Fonte: Retirado de Patterson e Hennessy (2011)

A Figura 5.3 mostra o caminho de controle preenchido com os valores da tabela citada anteriormente.



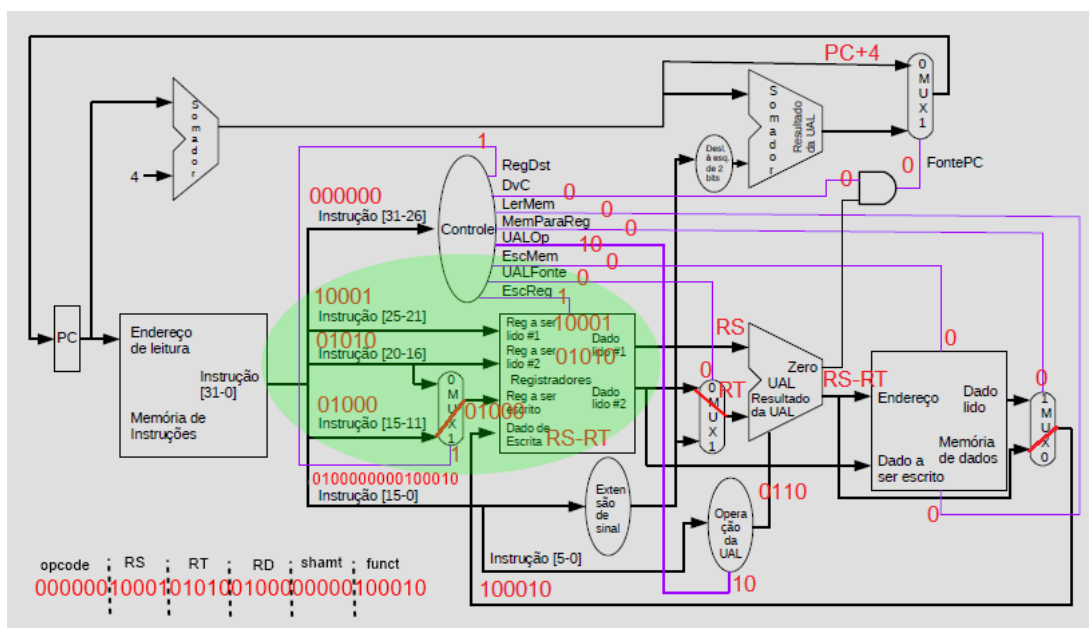
Figura 5.3 – A - Caminho de controle.



Fonte: Arquivo pessoal

A Figura 5.4 ilustra como o processador lê os registradores RS (bits 25-21) e RT (bits 20-16), mas a instrução [15-11] só é passada para o *Reg a ser escrito* porque o RegDst é 1, que controla o multiplexador destacado, confirmando que a instrução tem um registrador destino (RD) e é do tipo R.

Figura 5.4 – A - Leitura dos registradores.



Fonte: Arquivo pessoal

Os bits restantes [15-0] ainda podem ser uma instrução do tipo I. Esses 16 bits são

estendidos para 32 bits pela *Extensão de sinal*, e são enviados para o multiplexador que seleciona o segundo operador da UAL e para o somador do endereço de desvio.

Como a instrução em questão é do tipo R, não tem um endereço de desvio, nem um valor imediato. Logo, os bits [5-0] representam função, neste caso temos **100010**, que representa uma subtração. Com o código da função de subtração e  $UALOp = 10$  é gerado o código que representa a operação da UAL. Nesse caso temos uma subtração, dado por **0110**, extraído da tabela ilustrada na Figura 5.5.

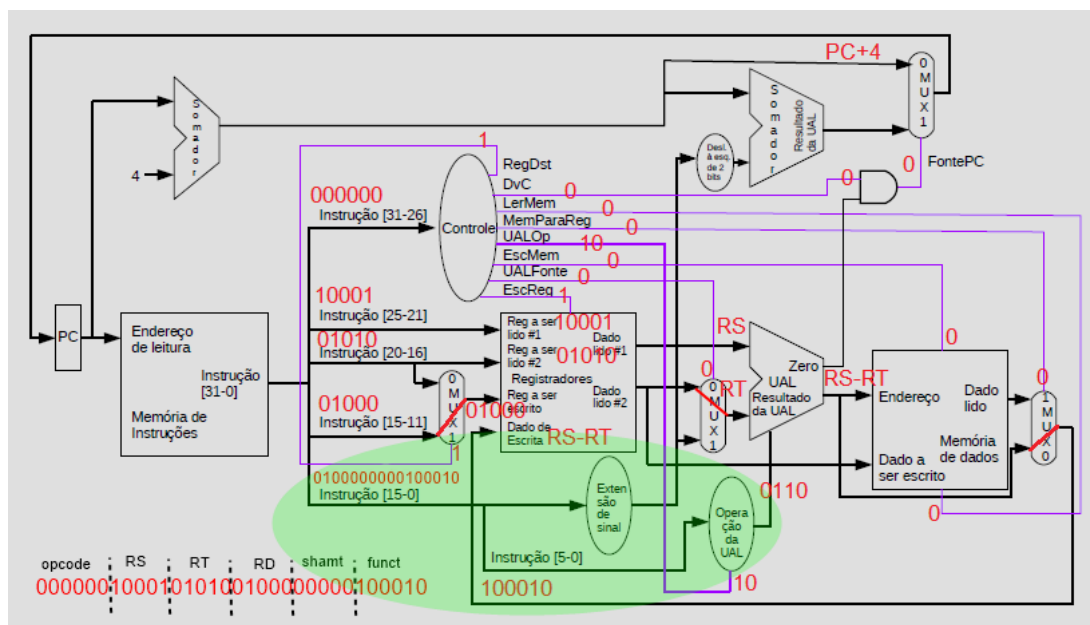
Figura 5.5 – Controle da UAL baseado na  $UALOp$

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Fonte: Retirado de Patterson e Hennessy (2011)

Na Figura 5.6 é possível visualizar os passos descritos acima.

Figura 5.6 – A - Bits 15-0.

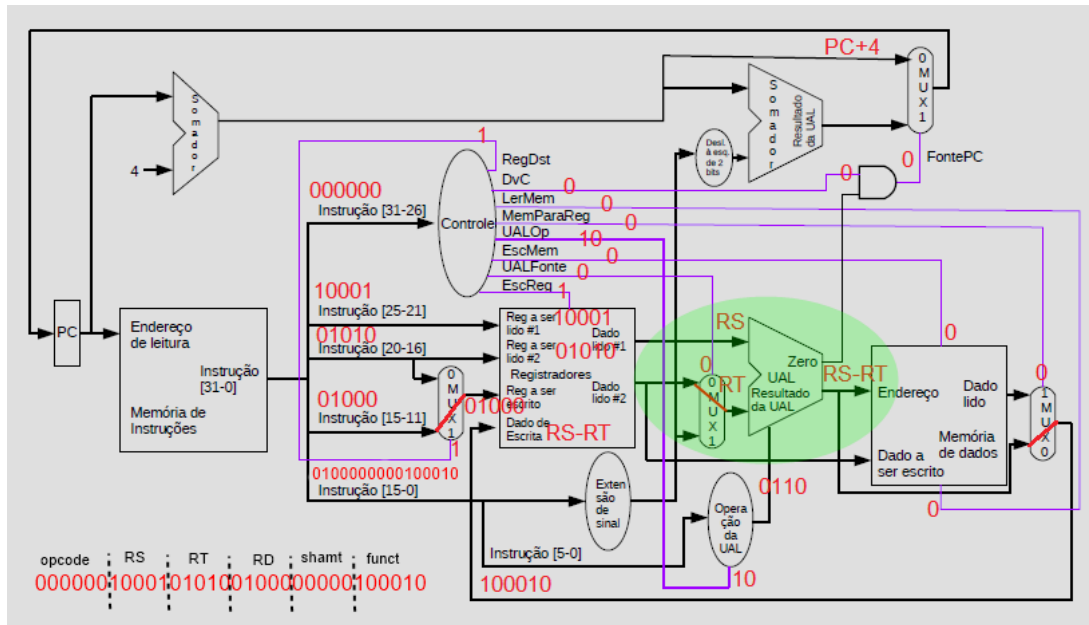


Fonte: Arquivo pessoal

Em seguida, os registradores lidos, RS e RT, são passados para a UAL para serem operados, ilustrado na Figura 5.7. O multiplexador que seleciona o registrador RT tem como controlador o **UALFonte** = 0, vindo do Controle, fazendo com que RT seja o *Reg*

a ser lido #2. A UAL vai esperar a operação que deve realizar, vinda da Operação da UAL.

Figura 5.7 – A - Operação da UAL.



Fonte: Arquivo pessoal

A Figura 5.8 apresenta o caminho de memória. Temos *EscMem* e *LerMem* nulos, logo, nosso dado não será escrito nem lido da memória. O resultado de **RS-RT** percorre o caminho até o multiplexador controlado por *MemParaReg*, que é 0, enviando o dado para *Dado de Escrita*.

Na próxima borda de subida, o RD terá o resultado de RS-RT, finalizando a execução da instrução.

## 5.2 LETRA B

A instrução que será explicada nessa seção é **sw \$v0, -8(\$sp)**, que é a instrução *store word*, utilizada para armazenar um dado na memória. O dado contido em \$v0 será armazenado no endereço de -8(\$sp).

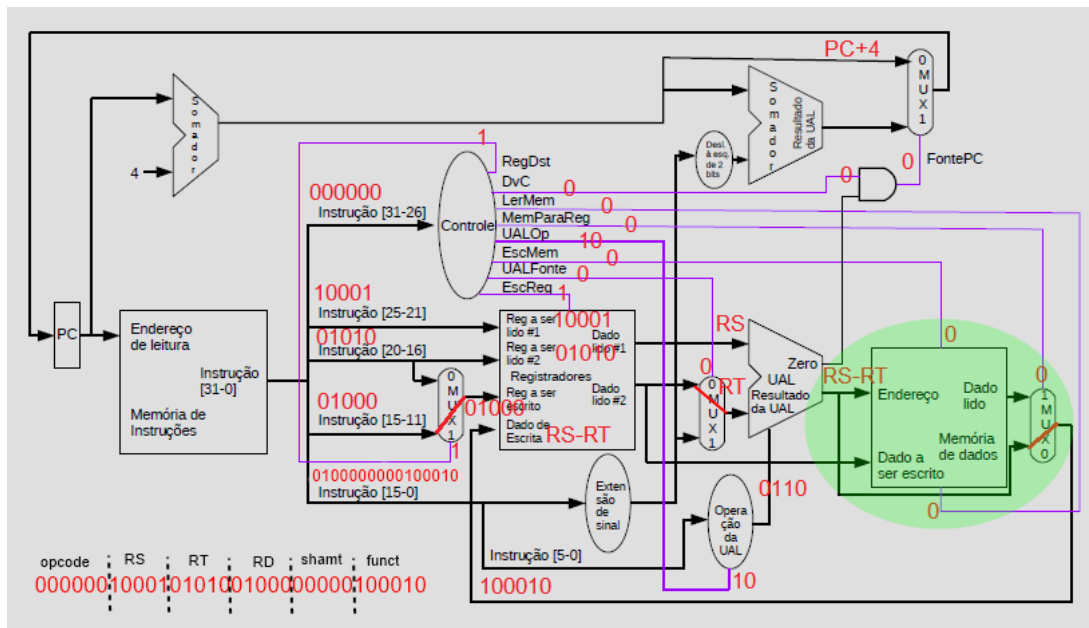
Na Figura 5.9 podemos visualizar que a instrução é decodificada para ser processada pelos caminhos do processador.

A partir do opcode 101011 (bits 31-26), o caminho de controle recebe os parâmetros demonstrados na tabela da Figura 5.2.

Podemos visualizar o caminho de controle preenchido com tais parâmetros na Figura 5.10.

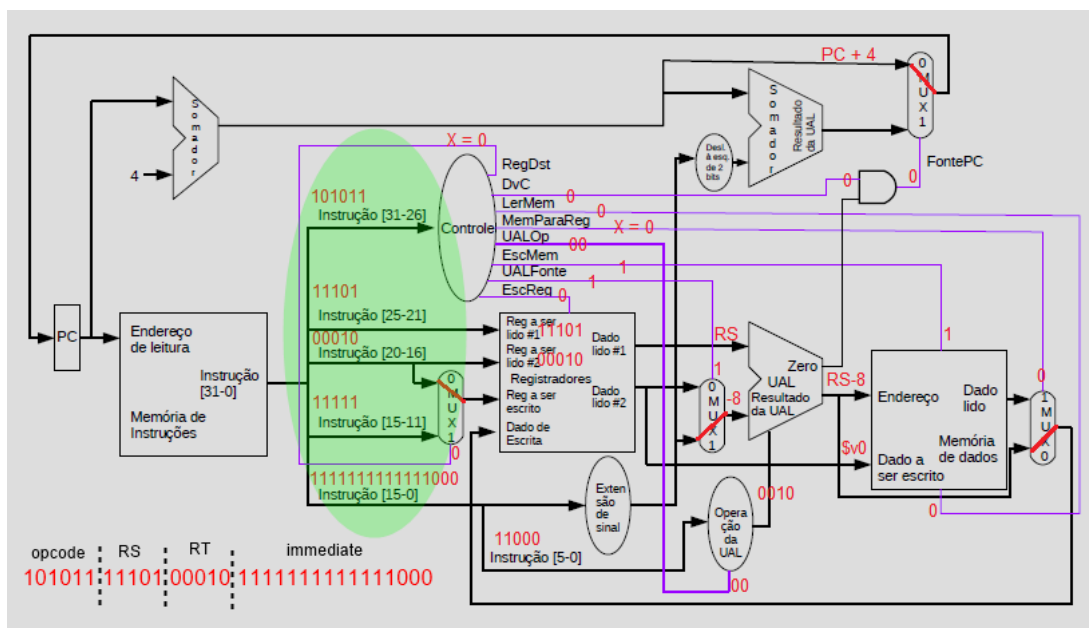
Os registradores \$v0 e \$sp são lidos pelo processador, representando RT e RS, respectivamente. O multiplexador fica esperando o sinal de *RedDst* para saber se é uma instrução do tipo R. Como o sinal de *RedDst* é 0, passa o valor de RT a ser escrito. A instrução não tem um registrador destino (RD), dando a entender que é uma instrução do tipo I.

Figura 5.8 – A - Caminho da memória



Fonte: Arquivo pessoal

Figura 5.9 – B - Separação da instrução nos caminhos.



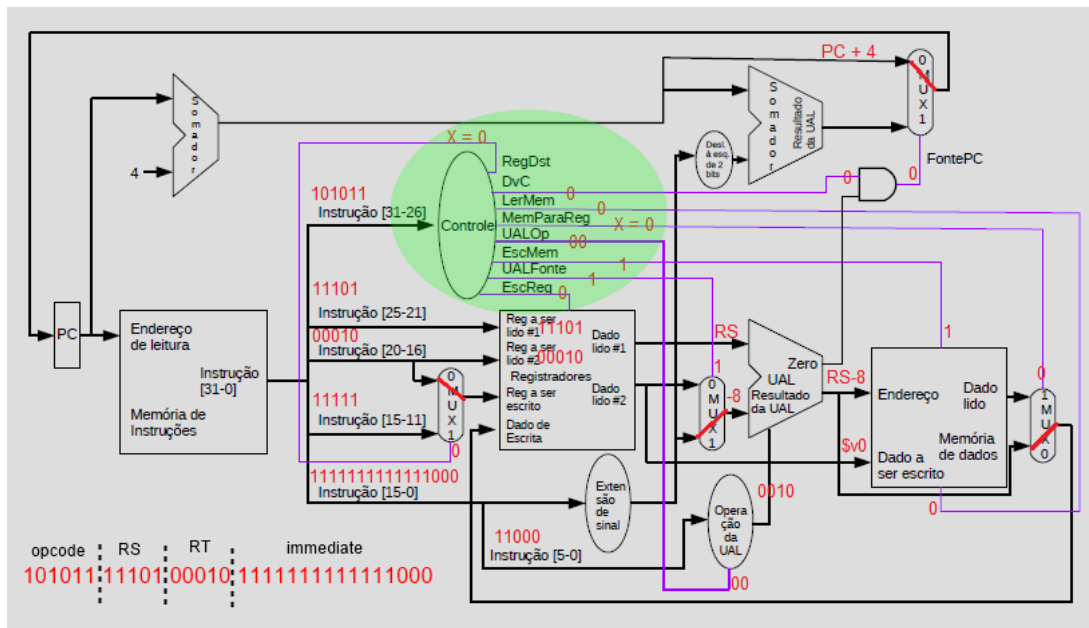
Fonte: Arquivo pessoal

O registrador RS é passado para a UAL e espera a operação que será feita na unidade.

A Figura 5.11 ilustra as explicações feitas anteriormente.

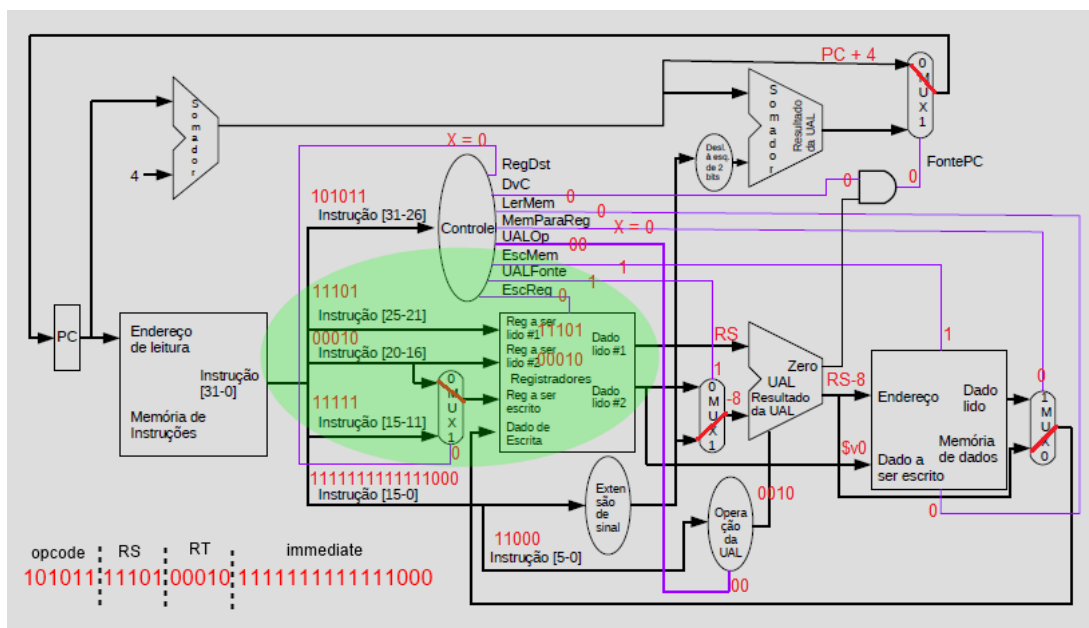
Os 16 bits restantes [15-0] correspondem ao valor imediato de deslocamento do endereço que será salvo o dado, que no caso, temos -8 estendido para 16 bits. Esses 16 bits são estendidos para 32 bits e enviados para o multiplexador de seleção do

Figura 5.10 – B - Caminho de controle.



Fonte: Arquivo pessoal

Figura 5.11 – B - Leitura dos registradores.



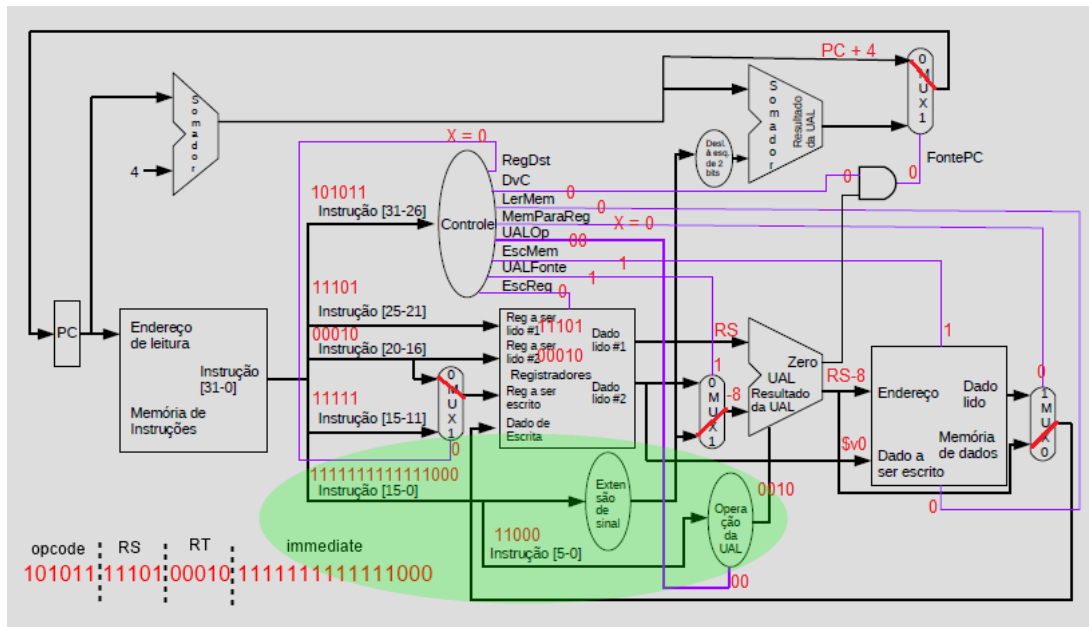
Fonte: Arquivo pessoal

segundo operador da UAL e para o somador do endereço de desvio, caso a instrução o tenha.

Os 6 bits menos significativos são separados e enviados para *Operação da UAL*, que recebe o sinal de controle de UALOp. Nesse caso temos UALOp sendo 00. Com base na tabela apresentada na Figura 5.5, é ignorado o que vem dos bits menos significativos que definem a função a ser operada pela UAL, a operação é definida

apenas pela *UALOp*, que nesse caso é 00, podendo ser uma instrução de *load word* ou *store word*. O que definirá se é uma instrução ou outra serão os sinais de controle *LerMem* e *EscMem*.

Figura 5.12 – B - Bits 15-0.



Fonte: Arquivo pessoal

No multiplexador em questão da Figura 5.13, vemos que o *UALFonte* seleciona o valor imediato estendido como segundo operador da UAL. Com a operação que será feita na UAL já definida, nesse caso será uma subtração, a unidade realiza a adição do endereço de **\$sp** e o valor imediato **-8**.

Após fazer as operações da UAL, o processador analisa se algo será escrito ou lido da memória. O endereço calculado pela UAL ( $RS - 8$ ) passa para o *Endereço* e para a entrada 0 do multiplexador em questão. O *Dado a ser escrito* recebe o conteúdo do registrador  $\$v0$ , que é o nosso RT. O controlador *EscMem* tem valor positivo, definindo a nossa instrução como uma *store word*. E, sendo assim, o nosso processador sabe que o conteúdo do nosso RT será salvo no endereço calculado pela UAL, que no caso temos **RS - 8**. O sinal de controle *MemParaReg* é X e atribuímos ele nulo, então o nosso multiplexador seleciona o 0 e envia para *Dado de Escrita* o endereço para salvar o conteúdo de  $\$v0$ , ilustrados na Figura 5.14.

Com o *Dado de Escrita* e sabendo qual instrução temos (sw), o processador aguarda a próxima borda de subida para que guarde o conteúdo de  $\$v0$  no endereço  $-8(\$sp)$ , finalizando a instrução.

### 5.3 LETRA C

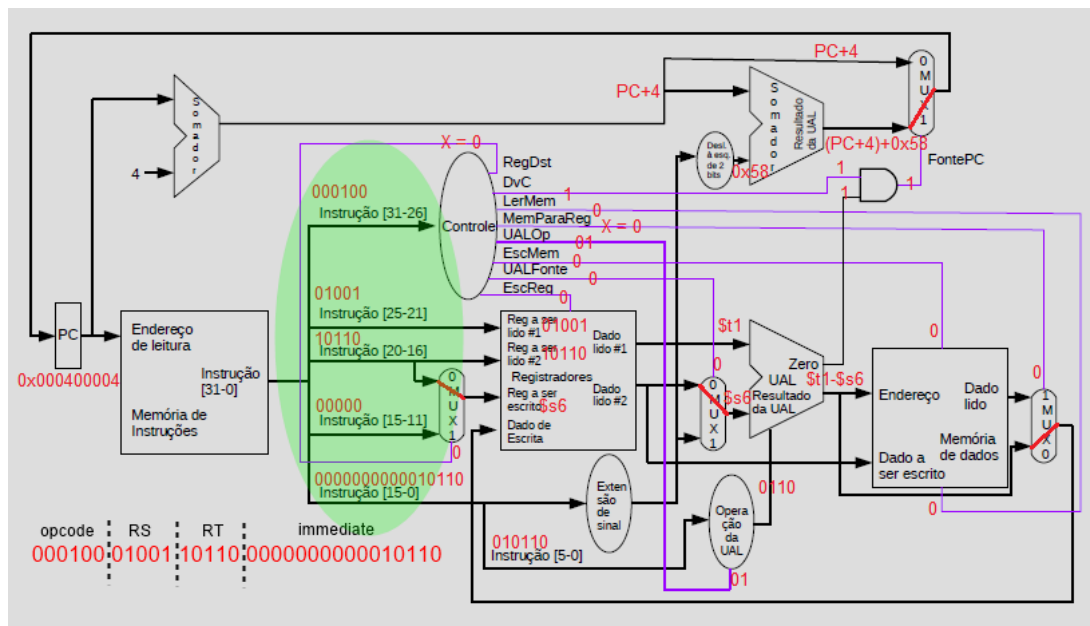
A instrução que será explicada nessa seção é *beq \$t1, \$s6, loop*, onde temos a instrução no endereço 0x00400004 e loop no endereço 0x00400060. A instrução *beq* representa um desvio condicional, onde a condição de desvio é se o conteúdo de  $\$t1$





ser processada, ilustrada na Figura 5.15.

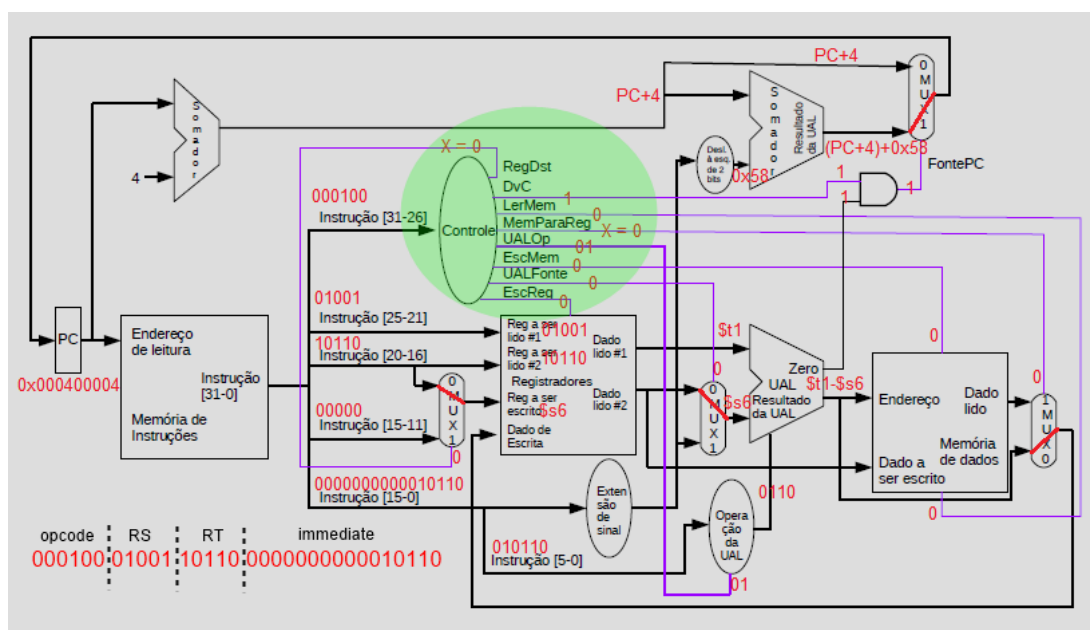
Figura 5.15 – C - Separação da instrução nos caminhos.



Fonte: Arquivo pessoal

Os bits mais significativos descrevem que tipo de instrução o processador irá receber. Nesse caso, o caminho de controle recebe **000100**, representando uma instrução *beq*, e decodifica ela ajustando todos os seus sinais de controle. Podemos consultar esses sinais na tabela da Figura 5.2 e ilustradas na Figura 5.16.

Figura 5.16 – C - Caminho de controle.

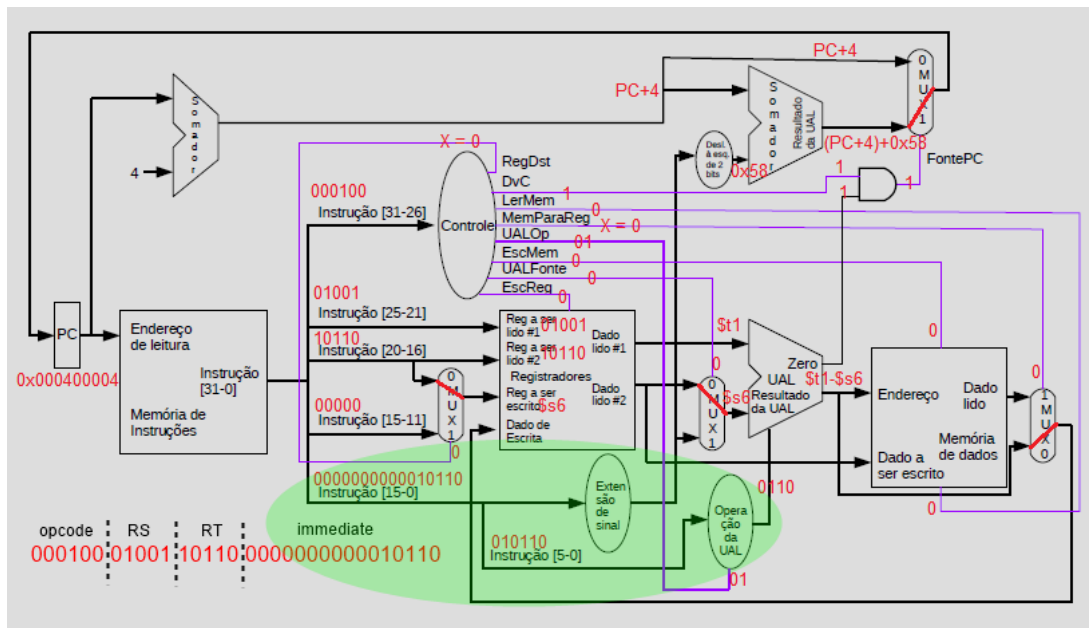


Fonte: Arquivo pessoal



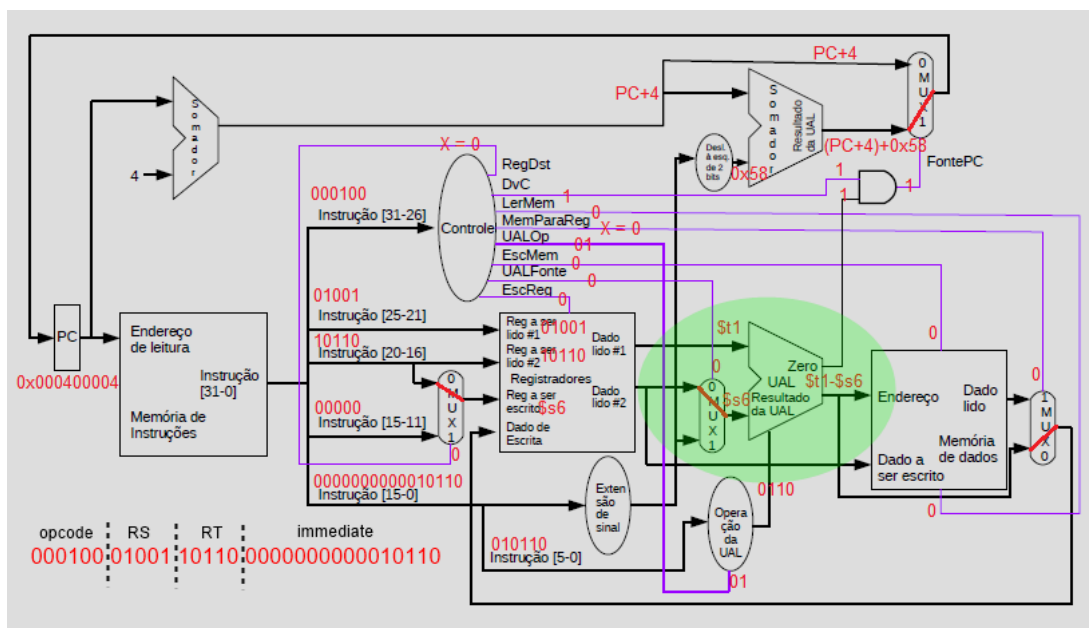


Figura 5.18 – C - Bits 15-0.



Fonte: Arquivo pessoal

Figura 5.19 – C - Operação da UAL - Resulta zero.

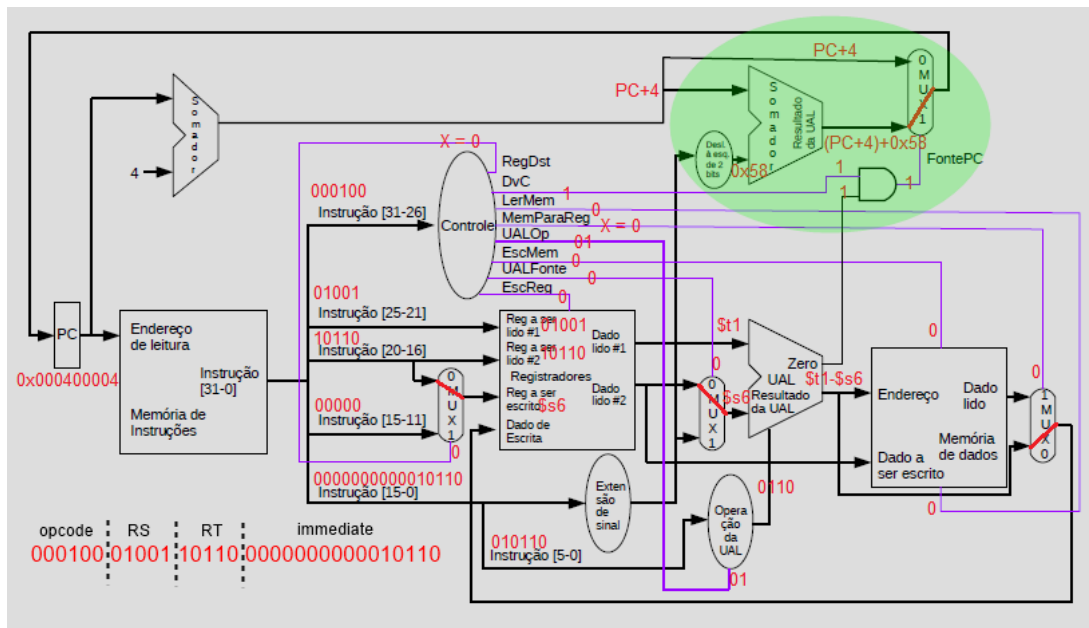


Fonte: Arquivo pessoal

para o caminho da memória e para o multiplexador operado por *MemParaReg* que é 0, selecionando o resultado da subtração e enviando para *Dado de Escrita*. Como temos uma instrução do tipo *beq*, os sinais de *LerMem* e *EscMem* serão nulos, ignorando o caminho da memória.

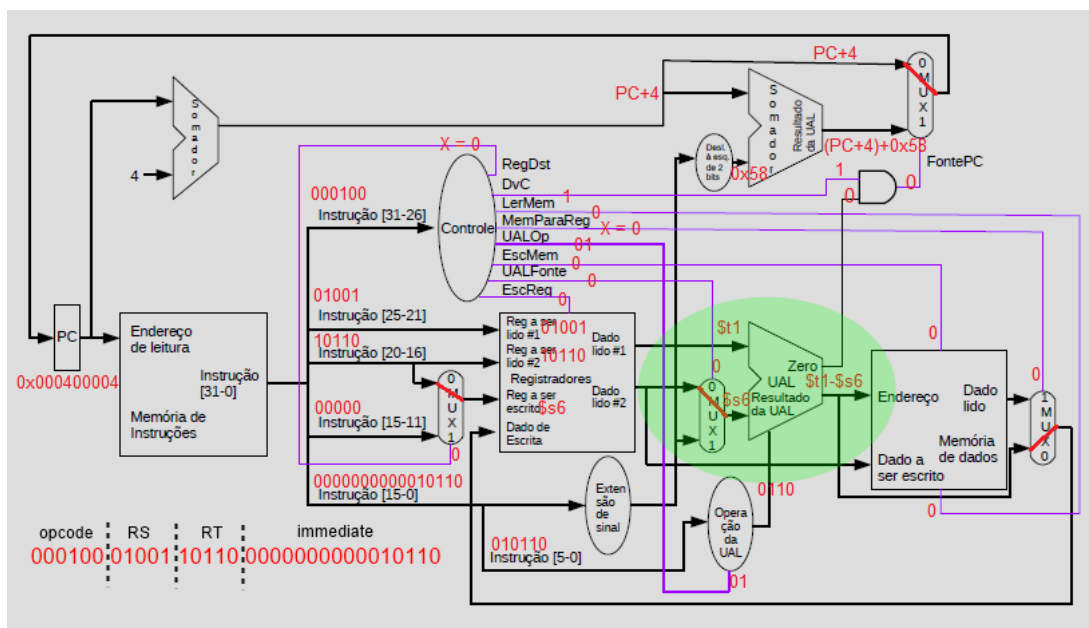
Depois de ter definido o endereço da próxima instrução, seja ele um endereço de desvio ou não, o processador espera a próxima borda de subida para que o possível

Figura 5.20 – C - Endereço de desvio.



Fonte: Arquivo pessoal

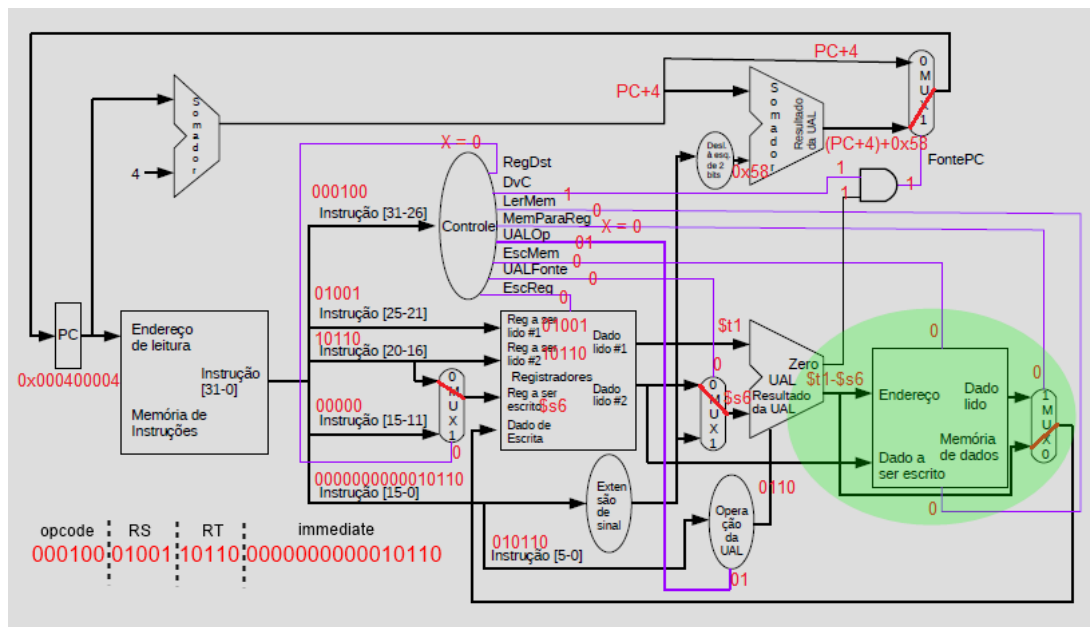
Figura 5.21 – C - Operação da UAL - Diferente de zero.



Fonte: Arquivo pessoal

desvio acontece. Acontecendo isso, a instrução é finalizada.

Figura 5.22 – C - Caminho da memória.



Fonte: Arquivo pessoal

## REFERÊNCIAS BIBLIOGRÁFICAS

Giovani Baratto. **Algoritmos para Multiplicação e Divisão de Números Binários**. 2018. Acesso em 15 jul. 2018. Disponível em: <[https://ead06.proj.ufsm.br/moodle/pluginfile.php/1361129/mod/\\_resource/content/0/algoritmoMultiplicacaoDivisao.pdf](https://ead06.proj.ufsm.br/moodle/pluginfile.php/1361129/mod/_resource/content/0/algoritmoMultiplicacaoDivisao.pdf)>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design**. Morgan Kaufmann, 2011. 919 p. Acesso em 14 de Julho de 2018. Disponível em: <<https://nsec.sjtu.edu.cn/data/MK.Computer.Organization.and.Design.4th.Edition.Oct.2011.pdf>>.