

MIPS



Grupo TQA:

- ▷ José Hélio Paiva Neto
- ▷ Pedro Vero Fontes
- ▷ Rafael de Melo Almada

Professor: Paulo Carlos Santos

Repositório do projeto:

<https://github.com/slimkaki/mips>

Índice:

Índice:

Introdução

Instruções

Instruções do tipo R

Instruções do tipo I

Instruções do tipo J

O nosso MIPS...

Instruction Fetch

Instruction Decode

Execute

Memory Access
Write-Back
Testando
Teste Subconjunto A
Teste Subconjunto AB
Interação com o usuário

Introdução

A Arquitetura MIPS foi criada com o intuito de implementar *pipelines* na arquitetura RISC que em 1981 já estava ganhando sua segunda versão. O MIPS foi concebido na Universidade de Stanford por John Hennessy. Seu objetivo era implementar *pipelines* profundas em cima da arquitetura RISC que já era focada em registradores.

Essa melhoria possibilitaria dividir as partes de execução de uma tarefa para que vários processos possam ocorrer de forma sobreposta, e assim otimizar o tempo de execução das instruções. E apesar de limitações ao necessitar que as operações sejam todas concluídas dentro de cada ciclo o MIPS se mostrou uma das arquiteturas baseadas em RISC mais promissoras e utilizadas no mundo.

Instruções

No MIPS foram estabelecidos 3 tipos de diferentes instruções, onde cada tipo tem uma especificidade para interpretação (*decode*) da instrução.

Instruções do tipo R

As instruções do tipo R são operações que utilizam até 3 registradores por instrução e não há espaço para imediato.

Estrutura dos Campos da Instrução					
opcode	Rs	Rt	Rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
MSB (b31)					LSB (b0)

Um detalhe importante desse tipo de instrução é que o *OpCode* de todas as instruções sempre será igual a 0, enquanto que o espaço denominado para *funct* é quem irá diferenciar cada instrução particularmente.

```
-- Add  
add $t0, $s1, $s2    -- R[s2] = R[t0] + R[s1]  
  
-- Sub  
sub $t0, $s1, $s2    -- R[s2] = R[t0] - R[s1]
```

Instruções do tipo I

As instruções do tipo I são operações que permitem que um valor seja colocado como imediato, assim é possível recuperar dados da memória de dados (*memória RAM*), escrever dados na memória e até fazer contas com números que não estão em nenhum registrador.

Formato da Instrução do Tipo I

opcode	Rs	Rt	imediato
6 bits	5 bits	5 bits	16 bits
31~26	25~21	20~16	15~0

Esse tipo de instrução se diferencia pelo *OpCode*.

```
-- Branch on Equal  
beq $rs, $rt, imediato; -- PC += (R[rs] == R[rt] ? 4 + distância : 4)  
                           -- sendo a distância = sinalExtendido(imediato)<<2  
  
-- Store Word  
sw $rt, imediato($rs); -- M[R[rs] + sinalExtendido[imediato]] = R[rt]  
  
-- Load Word  
lw $rt, imediato($rs); -- R[rt] = M[R[rs] + sinalExtendido[imediato]]
```

Instruções do tipo J

As instruções do tipo J foram

pensadas para os casos onde há *jump*, por conta disso a forma que a instrução é construída permite um grande espaço para o imediato.

opcode	imediato
6 bits	26 bits
31~26	25~0

Alguns exemplos para esse tipo de instrução, são:

```
-- Jump incondicional  
j endereço; -- PC = Endereço
```

O nosso MIPS...

Apesar de não ter o sistema de *pipelines*, este projeto trouxe como motivação implementar a ideia do MIPS, com a implementação de um processador *single-cycle*, porém com os três tipos de instruções do MIPS original e implementando as instruções originais.

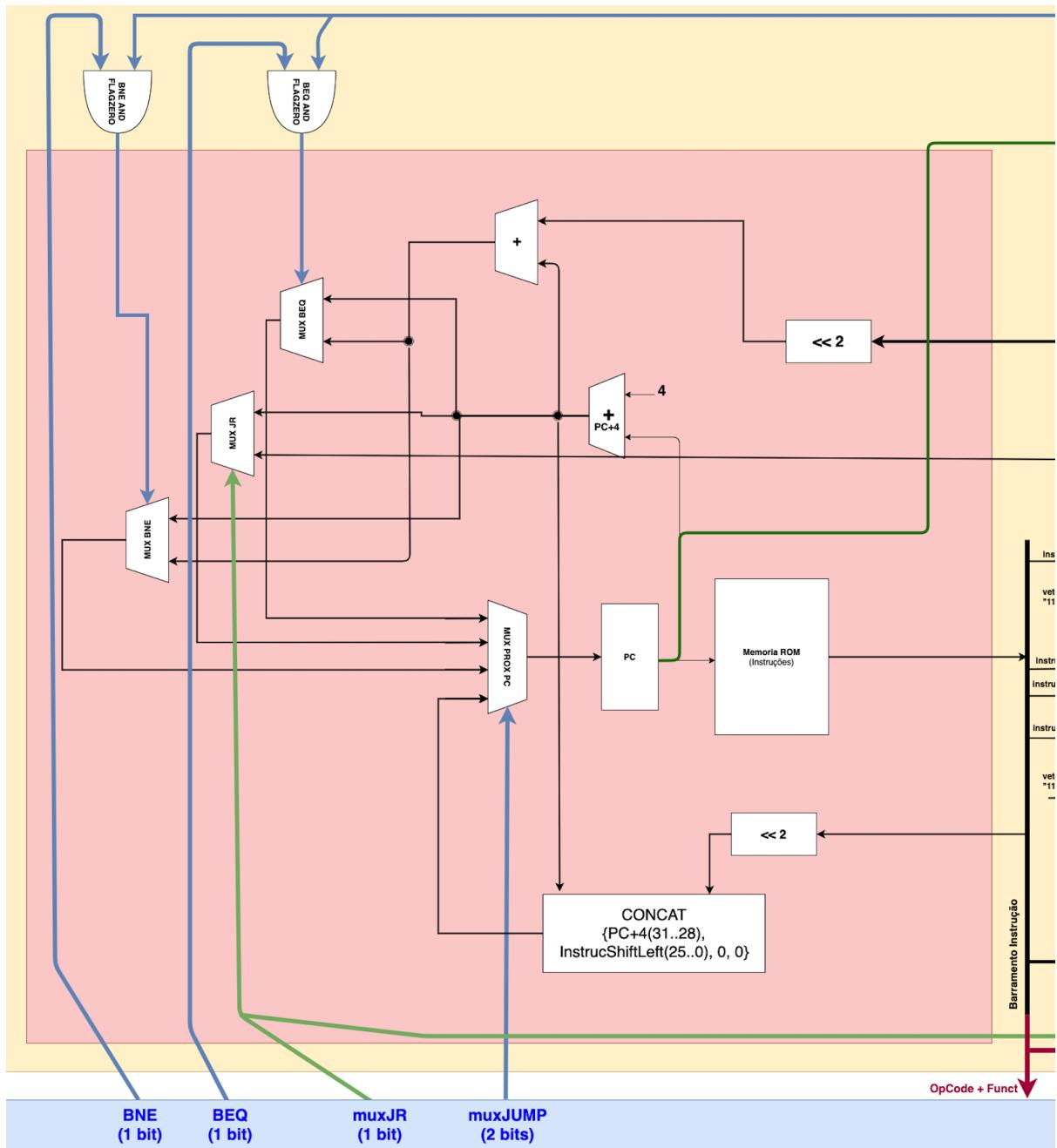
Nossa arquitetura completa:

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5b9ee130-149a-41c7-8e5c-db1a09c022e4/MIPS\_architecture.pdf
```

Arquitetura do nosso MIPS

Instruction Fetch

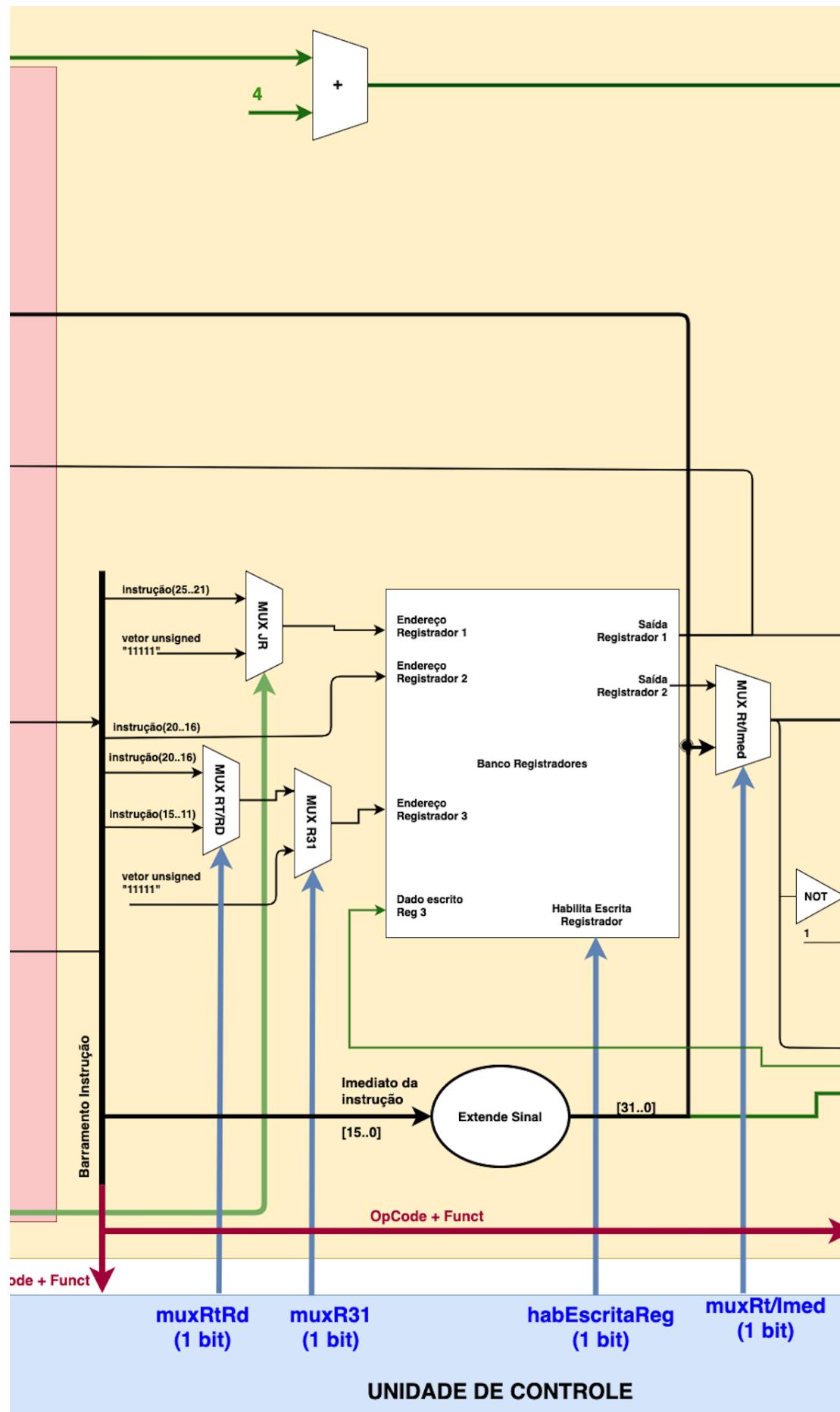
O *Instruction Fetch* é a parte do processador que se responsabiliza em achar a instrução atual e enviá-la para o resto do processador. Este *fetch*, no caso, precisa de alguns *muxes* para selecionar os casos em que é necessário realizar *jumps*, ou casos de *branch on equal* ou *branch on not equal*.



Zoom da etapa **Fecht**

Instruction Decode

A parte de *Instruction Decode* possui como principal tarefa entender a instrução que sai do *fetch* e então, com o auxílio da unidade de controle, inserir no banco de registradores os endereços a serem lidos na instrução atual.

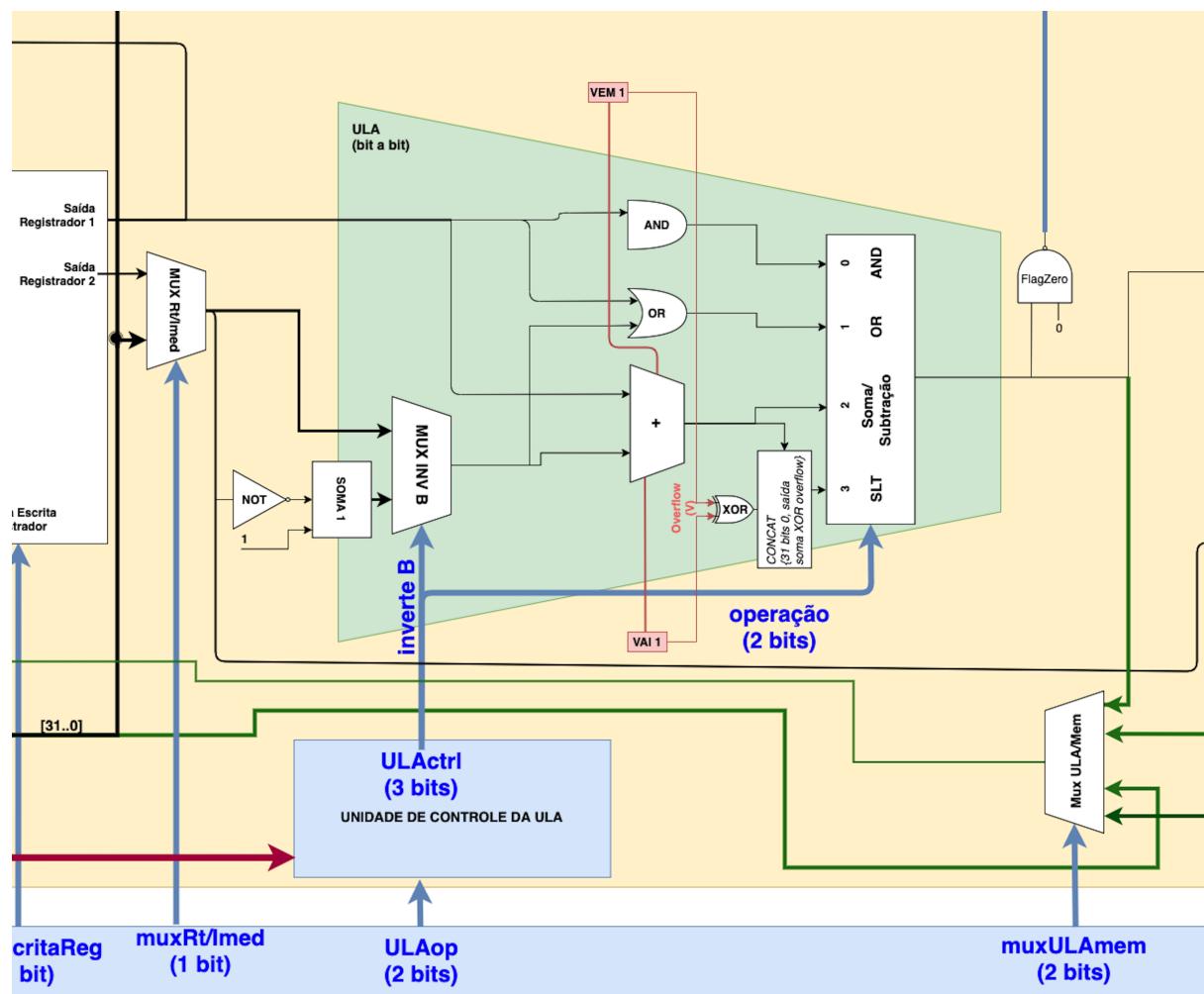


Zoom da etapa Decode

Execute

A etapa de *Execute*, como o nome já diz, irá executar o comando acionando a ULA (Unidade Lógica Aritmética) com as saídas do banco de registradores e auxílio da Unidade de Controle da ULA que faz a intermediação da ULA com a Unidade de Controle.

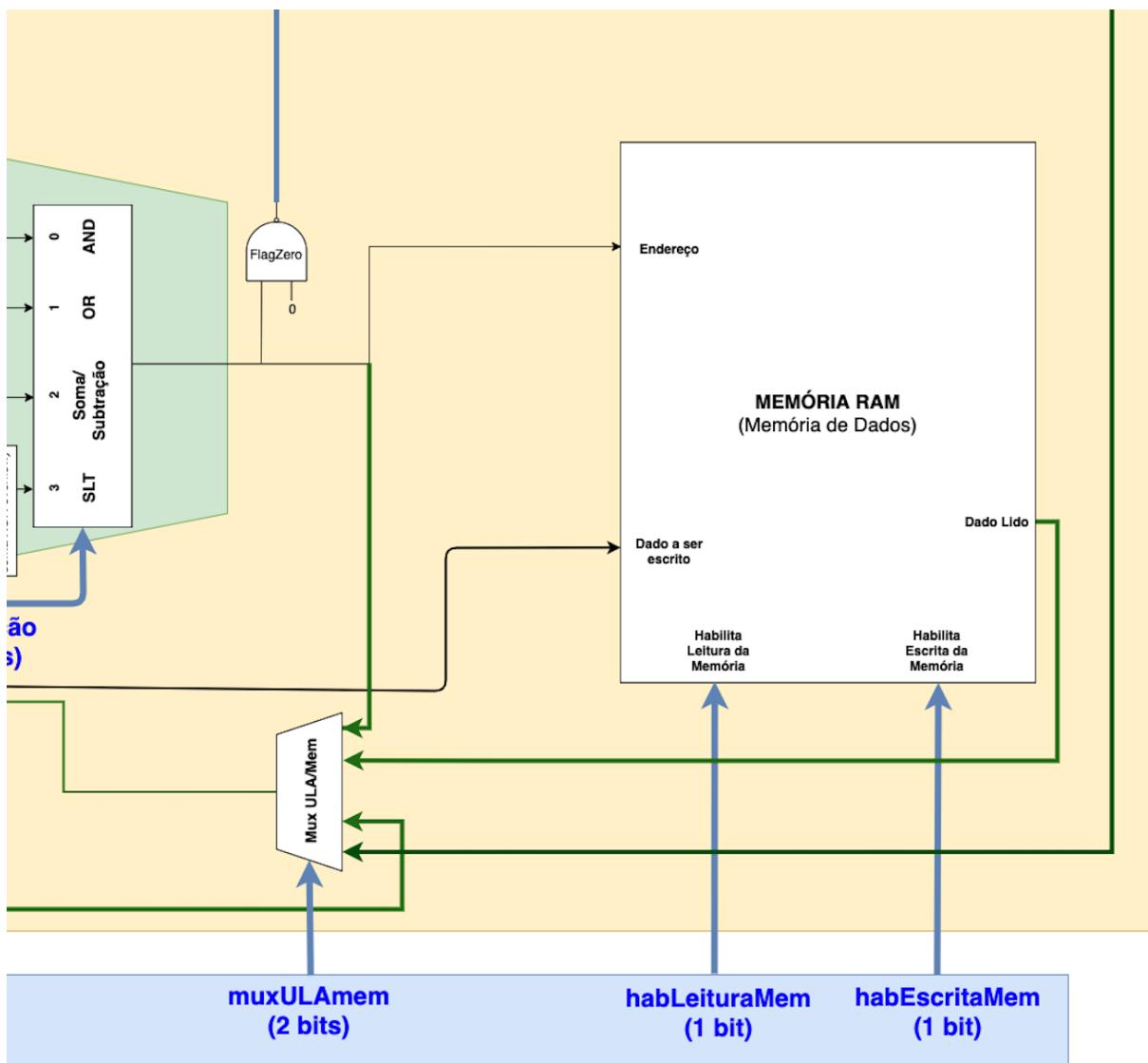
Além disso, é importante ressaltar que a ULA do nosso processador, por fins didáticos, foi feita bit a bit, logo é adicionado sinais para o que chamamos de "vem 1" e "vai 1", dessa forma ele consegue realizar o carry-on nas instruções que utilizam o somador e essa informação de carry-on é utilizado para encontrar o overflow da saída do somador.



Zoom da etapa Execute

Memory Access

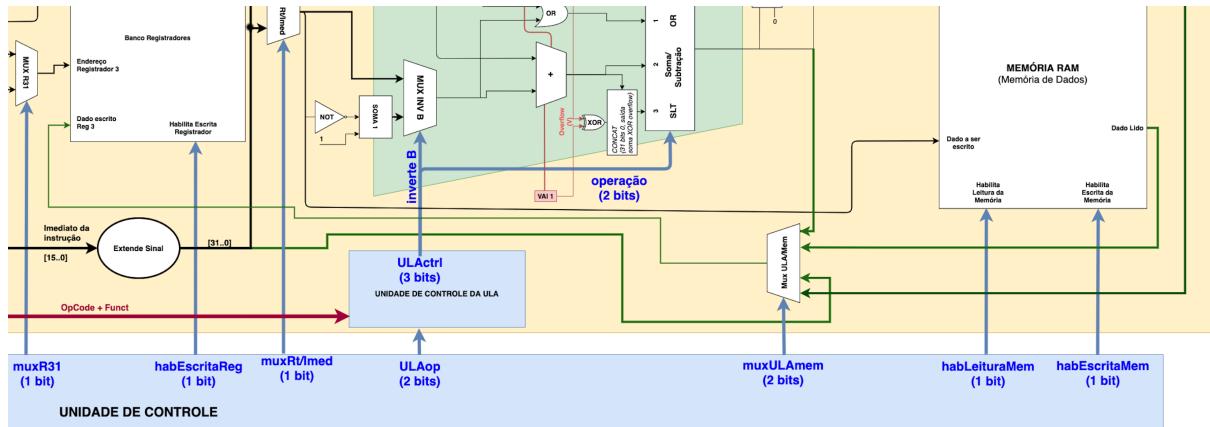
A etapa de *Memory Access*, permite o acesso à memória de Dados e, caso seja necessário ler ou escrever algum dado na memória, essa é a etapa para tal. Nem todas as instruções de fato utilizam esta memória (como por exemplo instruções do tipo R). Ela finaliza entregando o dado lido para um mux de seleção para caso seja necessário escrever o dado no banco de registradores. Outras opções para este mesmo MUX é escrever no banco de registradores diretamente a saída da ULA, ou então escrever o sinal do imediato extendido para 32 bits no banco de registradores ou também há a opção de escrever o endereço da próxima instrução em um registrador.



Zoom da etapa Memory Access

Write-Back

A etapa de *Write-Back* consiste em escrever no banco de registradores a saída do MUX ULA/Memória de dados. Esta etapa é essencial para que as instruções que realizam cálculos sejam computadas, dado que o resultado será salvo no registrador especificado pelo programador.



Zoom da etapa Write-Back

Testando

Para os testes de funcionamento, utilizamos os assembly's feito pelo professor da disciplina, onde foi verificado que todas as instruções estavam retornando valor esperado.

Além disso foi necessário iniciar alguns registradores no banco de registradores para que os testes funcionassem:

```
-- $zero (#0) := 0x00
-- $t0  (#8) := 0x00
-- $t1  (#9) := 0x0A
-- $t2  (#10) := 0x0B
-- $t3  (#11) := 0x0C
-- $t4  (#12) := 0x0D
-- $t5  (#13) := 0x16
```

Teste Subconjunto A

Este teste foi elaborado para testar as instruções mais básicas do MIPS de *single-cycle*.

```
sw $t1 8($zero)
lw $t0 8($zero)
sub $t0 $t1 $t2
```

```

and $t0 $t1 $t2
or $t0 $t1 $t2
slt $t0 $t1 $t2
add $t0 $t0 $t2
beq $t0 $t3 0xFFFFE
j 0x0000000

```

Resultados da Execução: Ciclo Único e Conjunto A

PC Dec	PC Hex	Saída ULA	Instrução	Comentário
00	0x00	0x08	sw \$t1 8(\$zero)	M(8) := 0x0000000A
04	0x04	0x08	lw \$t0 8(\$zero)	\$t0 := 0x0000000A
08	0x08	0xFFFFFFFF	sub \$t0 \$t1 \$t2	\$t0 := 0xFFFFFFFF
12	0x0C	0x0000000A	and \$t0 \$t1 \$t2	\$t0 := 0x0000000A
16	0x10	0x0000000B	or \$t0 \$t1 \$t2	\$t0 := 0x0000000B
20	0x14	0x00000001	slt \$t0 \$t1 \$t2	\$t0 := 0x00000001
24	0x18	0x0000000C	add \$t0 \$t0 \$t2	\$t0 := 0x0000000C
28	0x1C	0x00000000	beq \$t0 \$t3 0xFFFFE	pc := 0x18
24	0x18	0x00000017	add \$t0 \$t0 \$t2	\$t0 := 0x00000017
28	0x1C	0x0000000B	beq \$t0 \$t3 0xFFFFE	pc := 0x20
32	0x20	0x00000000	j 0x0000000	Volta ao Início

Na simulação do *ModelSim* do projeto, foi observado que todos os resultados bateram com o esperado.

Teste Subconjunto AB

Este teste foi elaborado para testar as outras instruções implementadas do MIPS de *single-cycle*

```

sw $t1 8($zero)
lw $t0 8($zero)
sub $t0 $t1 $t2
and $t0 $t1 $t2
or $t0 $t1 $t2
lui $t0 0xFFFF
addi $t0 $t1 0x000A
andi $t0 $t0 0x0013
ori $t0 $t4 0x0007
slti $t0 $t1 0xFFFF
add $t0 $t0 $t2
bne $t0 $t5 0xFFFFE
slt $t0 $t1 $t2
add $t0 $t0 $t2
beq $t0 $t3 0xFFFFE
jal 0x000001F

```

```
j 0x0000000
jr $ra
```

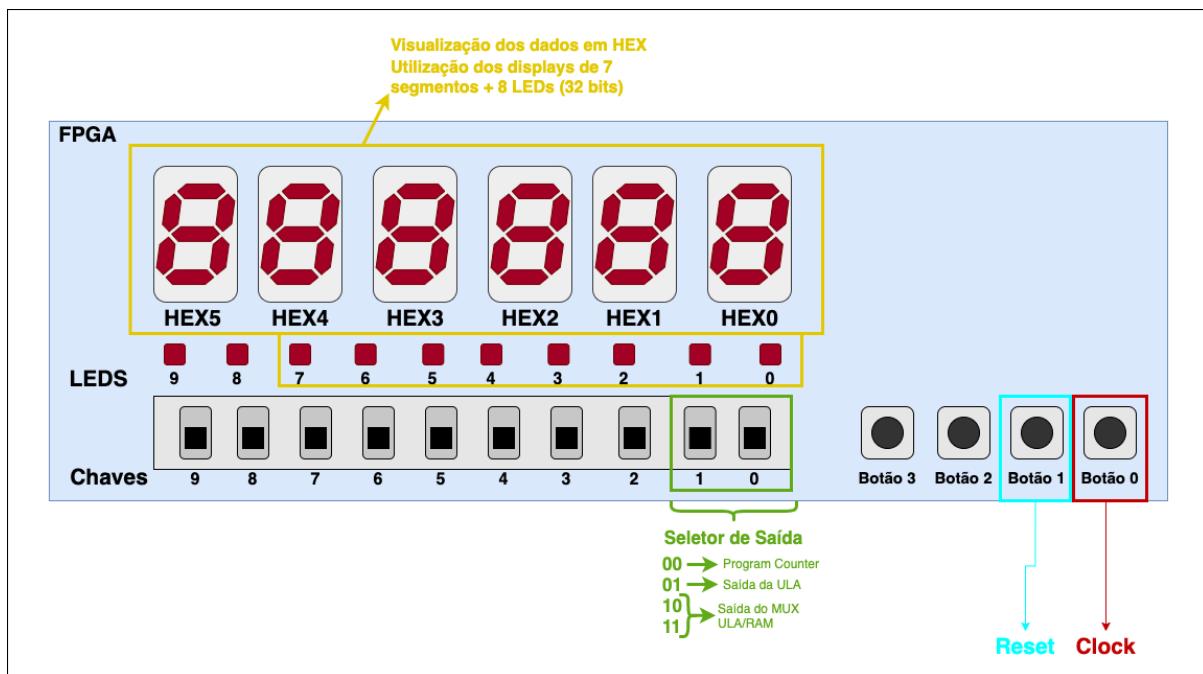
Resultados da Execução: Ciclo Único com Conjuntos A e B				
PC Dec	PC Hex	Saída ULA	Instrução	Comentário
00	0x00	0x00000008	sw \$t1 8(\$zero)	M(8) := 0x0000000A
04	0x04	0x00000008	lw \$t0 8(\$zero)	\$t0 := 0x0000000A
08	0x08	0xFFFFFFF	sub \$t0 \$t1 \$t2	\$t0 := 0xFFFFFFF
12	0x0C	0x0000000A	and \$t0 \$t1 \$t2	\$t0 := 0x0000000A
16	0x10	0x0000000B	or \$t0 \$t1 \$t2	\$t0 := 0x0000000B
20	0x14	0xFFFF0000	lui \$t0 0xFFFF	\$t0 := 0xFFFF0000
24	0x18	0x00000014	addi \$t0 \$t1 0x000A	\$t0 := 0x00000014
28	0x1C	0x00000010	andi \$t0 \$t0 0x0013	\$t0 := 0x00000010
32	0x20	0x0000000F	ori \$t0 \$t4 0x0007	\$t0 := 0x0000000F
36	0x24	0x00000000	slti \$t0 \$t1 0xFFFF	\$t0 := 0x00000000
40	0x28	0x0000000B	add \$t0 \$t0 \$t2	\$t0 := 0x0000000B
44	0x2C	0xFFFFFFF5	bne \$t0 \$t5 0xFFFFE	pc := 0x28
40	0x28	0x00000016	add \$t0 \$t0 \$t2	\$t0 := 0x00000016
44	0x2C	0x00000000	bne \$t0 \$t5 0xFFFFE	pc := 0x30
48	0x30	0x00000001	slt \$t0 \$t1 \$t2	\$t0 := 0x00000001
52	0x34	0x0000000C	add \$t0 \$t0 \$t2	\$t0 := 0x0000000C
56	0x38	0x00000000	beq \$t0 \$t3 0xFFFFE	pc := 0x34
52	0x34	0x00000017	add \$t0 \$t0 \$t2	\$t0 := 0x00000017
56	0x38	0x0000000B	beq \$t0 \$t3 0xFFFFE	pc := 0x3C
60	0x3C	0xXXXXXXXX	jal 0x00001F	<< 0x1F := 0x7C
124	0x7C	0xXXXXXXXX	jr \$ra	pc := 0x40
64	0x40	0xXXXXXXXX	nop	
68	0x44	0xXXXXXXXX	j 0x000000	Volta ao Início

Na simulação do *ModelSim* do projeto, foi observado que todos os resultados bateram com o esperado.

Interação com o usuário

Para demonstração do funcionamento do processador, sem que tenha de realizar simulações com o *ModelSim*, foi utilizado uma FPGA que, ao compilar o projeto e realizar o upload para o hardware, é mostrado no display de sete segmentos o dado desejado. Para selecionar o dado basta usar as chaves 1 e 0 da própria FPGA. O

botão 0 também foi colocado como o clock do processador e o botão 1 como reset, assim é possível andar pelas instruções com maior facilidade.



Desenho ilustrativo para visualização dos dados do MIPS