

**Trabalho N1 – Tipo 5 (versão 2) – Valor: 10%**

O objetivo deste trabalho é compreender como o ciclo de instrução da Máquina de von Neumann. Para isso, cada **dupla de alunos** deverá implementar um “simulador” de uma CPU com o conjunto de instruções (ISA – *Instruction Set Architecture*), o conjunto de registradores arquiteturais e o formato de instrução listados abaixo. Esse simulador deve mostrar o conteúdo dos registradores no fim de cada ciclo de máquina, quando haverá uma pausa até apertar uma tecla para iniciar o próximo ciclo.

O prazo de entrega é 16/06/2025, segunda-feira, às 23:59 via Moodle. Como de praxe, **cópia ou semelhança nas implementações serão punidas com nota zero** para todas as duplas em que isso for detectado. Passar o seu código para o colega é antiético, antiprofissional e está atrapalhando no desenvolvimento dele. Além disso, corre-se o risco da punição e, no fim do semestre, todo ponto faz a diferença entre ser aprovado ou não. Ainda, **duplas que não tiverem tirado nenhuma dúvida comigo até a entrega do trabalho terão a nota zero atribuída à atividade**. Isso é para evitar alunos que simplesmente apareçam com o trabalho pronto sem nunca ter tirado dúvidas, o que é extremamente improvável.

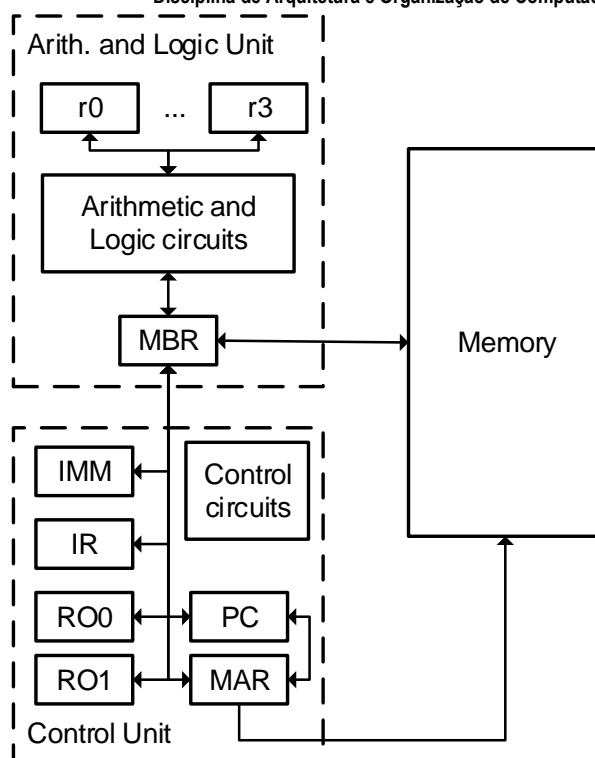
**Os entregáveis são: todos os códigos-fonte necessários para a compilação e o programa compilado, assim como instruções de compilação e de uso do programa.**

Ele deve ser implementado em linguagem C. Não é C++ e, sim, C. Portanto, sem orientação a objeto, pois só complica desnecessariamente o desenvolvimento deste trabalho em específico. Para facilitar, **a memória deve ser um vetor de palavras de oito bits** de 154 posições, o que, consequentemente, totalizará essa mesma quantidade em bytes. Em outras palavras, `unsigned char memoria[154]`. Como dica, implemente os registradores como explicado abaixo. **Não serão aceitas implementações onde os registradores sejam um vetor de caracteres ou a memória, uma matriz de caracteres.** O programa deve ser colocado na memória na forma binária respeitando os formatos das palavras de instrução abaixo. Outra característica dessa memória é que ela usa um barramento com oito linhas de dados – ou seja, um barramento com oito bits de largura. Sendo assim, **todas as transferências entre MBR e a memória devem ser de byte em byte.**

O simulador **deve** possuir uma maneira de ler um arquivo texto para carregar a memória com instruções e dados como apresentado na última página. É importante ressaltar, porém, que o simulador não precisa ser “bonito”, mas precisa ser fácil de usar.

Além disso, a CPU a ser implementada processa apenas números inteiros contidos em palavras de 16 bits e, portanto, não há nenhuma operação com ponto-flutuante, BCD ou números inteiros de outros tamanhos. Ademais, não é necessário implementar nenhuma representação de aritmética sinalizada. Embora o formato de instrução permita endereçar até  $2^{16} = 65.536$  palavras de oito bits na memória, a memória possui apenas 154 endereços, como apresentado acima, abrangendo os endereços de 0 (0x0) a 153 (0x99).

De maneira muito simplificada, o diagrama da CPU com os registradores arquiteturais é:



E a função de cada registrador arquitetural é:

1. **MBR** – *Memory Buffer Register* – contém a palavra a ser armazenada na memória. Também é o registrador usado para receber uma palavra lida da memória. Todo o tráfego de e para a memória RAM deve passar pelo MBR. Teoricamente, ele deveria ter o tamanho de 24 bits, mas como não existe variável de 24 bits na linguagem C, ele deve ser implementado como uma variável de 32 bits (`unsigned int mbr`);
2. **MAR** – *Memory Address Register* – especifica o endereço de memória da palavra a ser lida da ou escrita na memória. Todo endereço de memória deve ser indicado nesse registrador antes da execução da instrução. Deve ser implementado como uma variável de 16 bits (`unsigned short int mar`);
3. **IR** – *Instruction Register* – contém o *opcode* da instrução a ser executada. Teoricamente, ele deveria ter o tamanho de cinco bits, que não existe na linguagem C. Por isso, deve ser implementado como uma variável de oito bits (`unsigned char ir`);
4. **RO0** – *Register Operand 0* – contém o endereço do primeiro operando registrador da instrução. Teoricamente, ele deveria ter o tamanho de dois bits, que não existe na linguagem C. Por isso, deve ser implementado como uma variável de oito bits (`unsigned char ro0`);
5. **RO1** – *Register Operand 1* – contém o endereço do segundo operando registrador da instrução. Teoricamente, ele deveria ter o tamanho de dois bits, que não existe na linguagem C. Por isso, deve ser implementado como uma variável de oito bits (`unsigned char ro1`);
6. **IMM** – *Immediate* – contém o operando imediato da instrução. Deve ser uma variável de 16 bits (`unsigned short int imm`).
7. **PC** – *Program Counter* – contém o endereço da próxima palavra de instrução a ser buscada na memória. Caso não haja nenhum desvio, `halt` ou `nop`, o PC deve ser incrementado em cada ciclo de instrução tendo como referência o tamanho da instrução recém-executada. Deve ser uma variável de 16 bits (`unsigned short int pc`);
8. **E, L e G** – registradores internos que armazenam as *flags* 'equal to', 'lower than' e 'greater than'. Cada uma delas contém um bit indicando se o conteúdo do primeiro operando registrador é, ao ser comparado pela instrução `cmp`, respectivamente 1) igual a, 2) menor do que ou 3) maior do que o conteúdo do segundo operando registrador. Como não há maneira de implementar variáveis de um bit, devem ser implementados como variáveis de oito bits (`unsigned char e, unsigned char l, unsigned char g`);

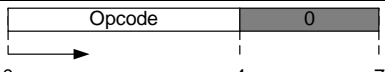

9. **r0 a r3** – registradores de propósito-geral (GPRs) utilizados para manter temporariamente os operandos na ALU. Devem ser implementados como um vetor com posições de 16 bits (unsigned short int reg[4]). Esses registradores são codificados nos campos de instrução `reg0` e `reg1` da seguinte forma:

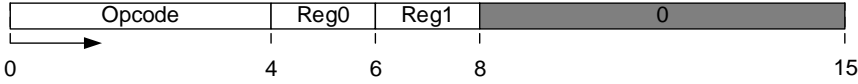
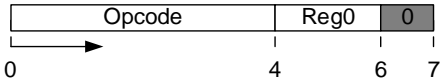
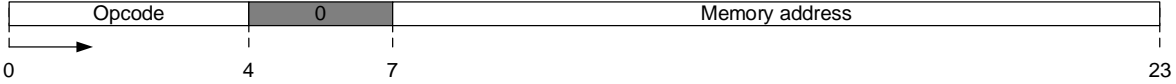
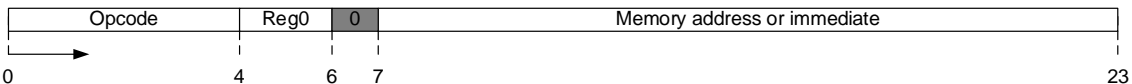
Reg.	Sequência de bits
r0	00
r1	01
r2	10
r3	11

Finalmente, o conjunto de instruções é:

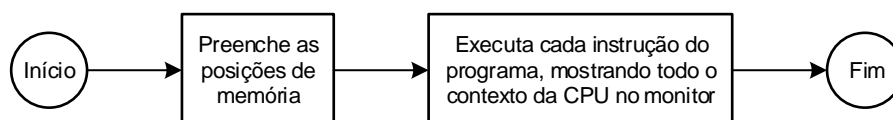
Mnemônico	Opcode	Descrição
hlt	0b00000	HALT: o processador não faz nada. Em outras palavras, <u>nenhum</u> registrador tem o seu valor alterado durante a <u>execução</u> de hlt. Deve-se colocar no fim do programa.
nop	0b00001	NO OPERATION: O PC é incrementado, mas <u>nenhum</u> outro registrador tem seu valor alterado durante a execução de nop.
ldr rX, rY	0b00010	LOAD VIA REGISTER: $rX = *rY$
str rX, rY	0b00011	STORE VIA REGISTER: $*rY = rX$
add rX, rY	0b00100	ADD REGISTER: $rX = rX + rY$
sub rX, rY	0b00101	SUBTRACT REGISTER: $rX = rX - rY$
mul rX, rY	0b00110	MULTIPLY REGISTER: $rX = rX \times rY$
div rX, rY	0b00111	DIVIDE REGISTER: $rX = rX \div rY$
cmp rX, rY	0b01000	COMPARE REGISTER: compara a palavra no registrador X com a palavra no registrador Y e preenche os registradores internos E, L e G os valores fazendo sequencialmente os seguintes testes: 1. Se $rX = rY$ , então $E = 1$ ; senão $E = 0$ ; 2. Se $rX < rY$ , então $L = 1$ ; senão $L = 0$ ; 3. Se $rX > rY$ , então $G = 1$ ; senão $G = 0$ .
movr rX, rY	0b01001	MOVE REGISTER: $rX = rY$
and rX, rY	0b01010	LOGICAL-AND ON REGISTER: $rX = rX \& rY$
or rX, rY	0b01011	LOGICAL-OR ON REGISTER: $rX = rX   rY$
xor rX, rY	0b01100	LOGICAL-XOR ON REGISTER: $rX = rX \wedge rY$
not rX	0b01101	LOGICAL-NOT ON REGISTER: $rX = !rX$
je Z	0b01110	JUMP IF EQUAL TO: muda o registrador PC para o endereço de memória Z caso $E = 1$ .
jne Z	0b01111	JUMP IF NOT EQUAL TO: muda o registrador PC para o endereço de memória Z caso $E = 0$ .
jl Z	0b10000	JUMP IF LOWER THAN: muda o registrador PC para o endereço de memória Z caso $L = 1$ .
jle Z	0b10001	JUMP IF LOWER THAN OR EQUAL TO: muda o registrador PC para o endereço de memória Z caso $E = 1$ ou $L = 1$ .
jg Z	0b10010	JUMP IF GREATER THAN: muda o registrador PC para o endereço de memória Z caso $G = 1$ .
jge Z	0b10011	JUMP IF GREATER THAN OR EQUAL TO: muda o registrador PC para o endereço de memória Z caso $E = 1$ ou $G = 1$ .
jmp Z	0b10100	JUMP: muda o registrador PC para o endereço de memória Z.
ld rX, Z	0b10101	LOAD: carrega o registrador X com uma palavra da memória de 16 bits que se inicia no endereço Z, ou seja, $rX = *Z$ .
st rX, Z	0b10110	STORE: armazena uma palavra de 16 bits que começa a partir do endereço de memória Z o conteúdo do registrador X, ou seja, $*Z = rX$ .
movi rX, IMM	0b10111	MOVE IMMEDIATE: $rX = IMM$
addi rX, IMM	0b11000	ADD IMMEDIATE: $rX = rX + IMM$
subi rX, IMM	0b11001	SUBTRACT IMMEDIATE: $rX = rX - IMM$
muli rX, IMM	0b11010	MULTIPLY IMMEDIATE: $rX = rX \times IMM$
divi rX, IMM	0b11011	DIVIDE IMMEDIATE: $rX = rX \div IMM$
lsh rX, IMM	0b11100	LEFT SHIFT: desloca a palavra no registrador X em IMM bits à esquerda.
rsh rX, IMM	0b11101	RIGHT SHIFT: desloca a palavra no registrador X em IMM bits à direita.

As instruções são codificadas da seguinte forma:

Mnemônico	Formato da palavra de instrução
hlt	
nop	
ldr rX, rY	
str rX, rY	
add rX, rY	

Mnemônico	Formato da palavra de instrução
sub rX, rY mul rX, rY div rX, rY cmp rX, rY movrr rX, rY and rX, rY or rX, rY xor rX, rY	
not rX	
je Z jne Z jl Z jle Z jg Z jge Z jmp Z	
ld rX, Z st rX, Z movi rX, IMM addi rX, IMM subi rX, IMM muli rX, IMM divi rX, IMM lsh rX, IMM rsh rX, IMM	

Observe que a CPU simulada deverá ser capaz de executar **qualquer** programa que for adicionado à memória. O programa deve ser colocado na memória na forma binária respeitando os formatos das palavras de instrução ilustradas na tabela acima. Dessa forma, o funcionamento do programa deve seguir a seguinte sequência:



Para a primeira etapa – carregar a memória – o simulador deve ler um arquivo texto como apresentado na última página. Essa etapa deve continuar enquanto o programa não encontrar a linha contendo 'hlt' (sem os apóstrofes). A conversão preencherá a memória em uma maneira semelhante à observada abaixo.

End. mem.	Conteúdo	Conteúdo codificado em binário	Conteúdo codificado em hexadecimal
0x0	ld r0, 1E	1010 1000 0000 0000 0001 1110	A8 00 1E
0x3	ld r1, 20	1010 1010 0000 0000 0010 0000	AA 00 20
0x6	add r0, r1	0010 0000 1000 0000	20 80
0x8	addi r0, 20	1100 0000 0000 0000 0001 0100	C0 00 14
0xB	st r0, 22	1011 0000 0000 0000 0010 0010	B0 00 22
0xE	hlt	0000 0000	00 00
...	...	...	...
0x1E	15	0000 0000 0000 1111	00 0F
0x20	8	0000 0000 0000 1000	00 08

Você pode observar na última coluna da tabela acima que os valores estão agrupados em pares de valores hexadecimais, onde cada um deles é um byte (ou o equivalente a uma posição de memória). Dessa forma, o mapa de memória fica assim (os endereços de memória e os seus conteúdos estão em hexadecimal):

Endereço	Conteúdo	Endereço	Conteúdo	Endereço	Conteúdo	Endereço	Conteúdo
0	90	A	14	14	...	1E	00
1	00	B	98	15	...	1F	0F
2	1E	C	00	16	...	20	00
3	92	D	22	17	...	21	08

4	00	E	00	18	...	22	...
5	20	F	00	19	...	23	...
6	08	10	...	1A	...	24	...
7	80	11	...	1B	...	25	...
8	A8	12	...	1C	...	26	...
9	00	13	...	1D	...	27	...

O que estará na memória é o que está nessa tabela.

Na segunda etapa – que é exibir o funcionamento da CPU executando o programa – o trabalho deve exibir algo assim (considerando que todos os conteúdos de registradores e da memória, incluindo os seus endereços, estão em hexadecimal):

```
CPU:
R0: 0xFFFF R1: 0xFFFF R2: 0xFFFF R3: 0xFFFF
MBR: 0xFFFFFFFF MAR: 0xFFFF IMB: 0xFFFF PC: 0xFFFF
IR: 0xFF R00: 0xF R01: 0xF
E: 0xF L: 0xF G: 0xF

Memória:
00: 0xFF 01: 0xFF 02: 0xFF 03: 0xFF
... .. 99: 0xFF

Pressione uma tecla para iniciar o próximo ciclo de máquina ou aperte CTRL+C para finalizar a execução do trabalho.
```

Deve-se exibir o conteúdo dos registradores e da memória ao fim de cada ciclo de máquina, com o usuário devendo pressionar uma tecla para iniciar a execução do próximo ciclo.

Para ajudar no desenvolvimento do trabalho, teste-o com os dois programas abaixo, considerando que eles já estão no formato esperado de arquivo esperado de arquivo a ser lido pelo trabalho. Todos os dados, endereços e imediatos estão representados em hexadecimal. Observe que as linhas que estão no formato endereço;instrução/dado;palavra de instrução ou palavra de dado. Em todo caso, o endereço inicial de memória que aquela instrução ou dado ocupará é dado em hexadecimal, assim como os valores numéricos nas instruções. Além disso, observe que as instruções possuem tamanho variável – de um a três bytes. Os dados sempre ocupam dois bytes.

$A = 32 + 3 \times \frac{4}{5 - 3}$	$A = \sum_{1}^{10} \frac{10}{5} + 3 \times (2 - 1)$
<pre>0;i;ld r0, 96 3;i;ld r1, 98 6;i;sub r0, r1 8;i;ld r1, 94 b;i;div r1, r0 d;i;ld r2, 92 10;i;mul r2, r1 12;i;ld r1, 90 15;i;add r1, r2 17;i;st r1, 8e 1a;i;hlt 90;d;20 92;d;3 94;d;4 96;d;5 98;d;3</pre>	<pre>0;i;ld r0, 90 3;i;ld r1, 92 6;i;div r0, r1 8;i;ld r1, 96 b;i;ld r2, 98 e;i;sub r1, r2 10;i;ld r2, 94 13;i;mul r1, r2 15;i;add r0, r1 17;i;ld r1, 8a 1a;i;add r0, r1 1c;i;st r0, 8a 1f;i;ld r0, 8c 22;i;ld r1, 8e 25;i;addi r0, 1 28;i;st r0, 8c 2b;i;cmp r0, r1 2d;i;jle 30 30;i;hlt 8a;d;0 8c;d;1 8e;d;a 90;d;a 92;d;5 94;d;3</pre>

96;d;2  
98;d;1