Laboratório 03

Arquitetura de Computadores
Prof. Pedro Botelho
2º Semestre de 2025

1 Descrição

Seu trabalho será implementar um programa em linguagem C (um arquivo **imas.c** será fornecido com parte da implementação) que simule a execução de um processador baseado no IAS, o **IMAS** (*Institute of More Advanced Studies*), contendo as instruções mostradas na tabela fornecida na Seção 3.

Esse processador que você irá simular é muito semelhante ao IAS, porém trazendo algumas melhorias e sendo de 16 bits. Considere que o processador possui uma palavra de dados e de instrução de 16 bits, uma memória de 4KB, e os seguintes registradores de 16 bits:

- **PC** (*Program Counter*): Endereço da próxima instrução
- **MAR** (*Memory Address Register*): Endereço a ser acessado na memória
- **IBR** (*Instruction Buffer Register*): Mantém toda a instrução
- **IR** (*Instruction Register*): Mantém o *opcode* da instrução
- **MBR** (*Memory Buffer Register*): Dado a ser escrito/lido da memória
- **AC** (*Accumulator*): Acumulador usado em operações aritméticas
- **MQ** (*Multiplier Quocient*): Usado na multiplicação e divisão

O funcionamento do IMAS é muito semelhante ao IAS, porém com algumas diferenças: palavra de **16 bits**, instruções adicionais, um ciclo de execução de instruções contendo **três subciclos** ao invés de dois, sendo eles busca, decodificação e execução e apenas uma instrução por palavra, ao invés de duas. Assim, a instrução do IMAS é dividida da seguinte forma:

- 4 bits para ***opcode*** (bits 0 a 3 do IBR) → IR
- 12 bits para o **endereço do operando** (bits 4 a 15 do IBR) → MAR

Todas as interações com a memória usam **MAR** e **MBR**. Uma unidade de entrada e saída simples interfaceia com o MBR (veja a seção 2.3). Em cada subciclo o IMAS deve realizar operações com os seus registradores:

1. **Busca de Instrução**: Instrução é buscada na memória (move o endereço no **PC** para o **MAR**, incrementa **PC**, obtém a instrução no **MBR** e move-a para o **IBR**)
2. **Decodificação de Instrução**: Campos da instrução (no **IBR**) são divididos em opcode (para o **IR**) e endereço do operando (para o **MAR**)
3. **Execução de Instrução**: Instrução é executada na ULA (usando o **AC** e geralmente o **MBR**), sendo que pode ocorrer interação com a memória (dependendo da instrução, que utilizará o endereço no **MAR**)

Abaixo é possível ver a estrutura interna do IAS. O IMAS tem a mesma estrutura. Perceba que a direção do fluxo de dados pelos registradores do processador é indicado pelas setas. Utilize isso para se basear ao programar:

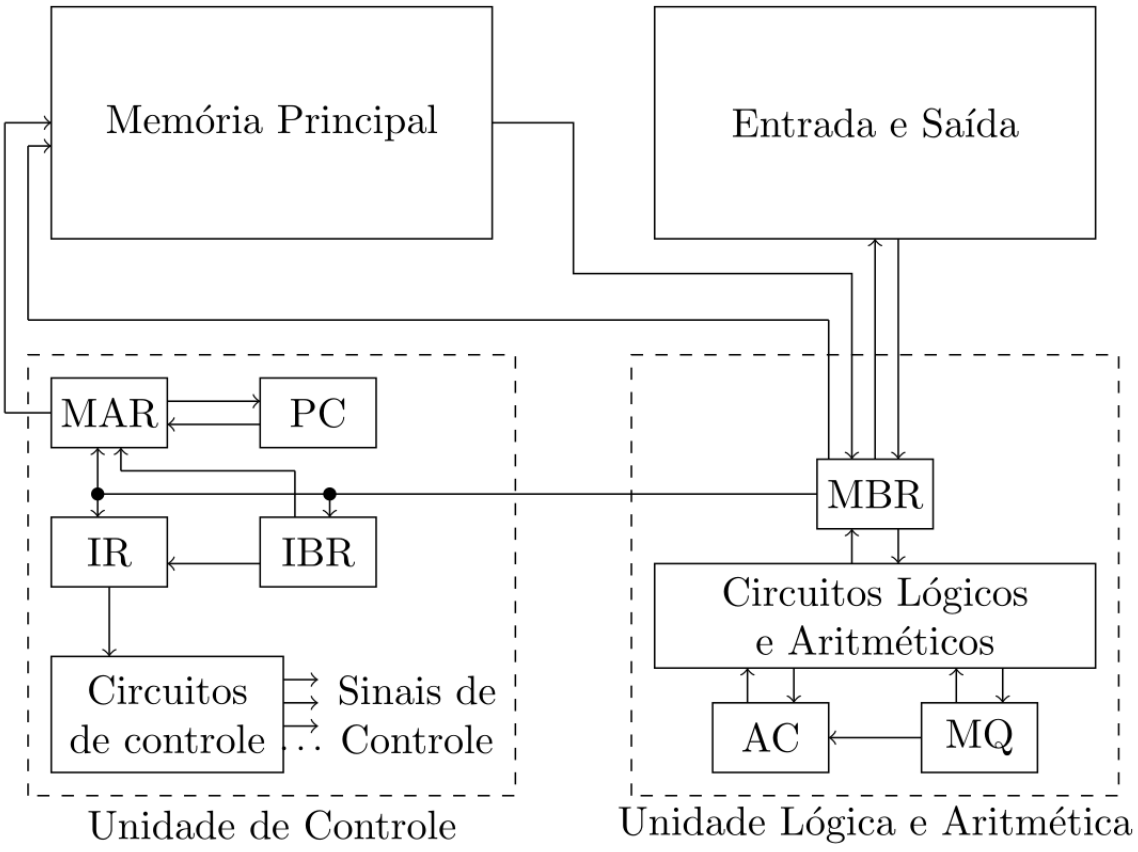

Figura 1: Estrutura do IAS

2 Especificação do Projeto

O simulador recebe como entrada um **arquivo de texto** contendo o conteúdo da memória do processador, em notação hexadecimal, onde cada linha corresponde a um endereço e a uma instrução, no formato "<**endereço**> <**conteúdo**>", sendo ambos em hexadecimal.

Cada posição de memória possui 16 bits, e os endereços devem sempre ser fornecidos, não necessariamente em ordem. Tudo que estiver após o hexadecimal da instrução será **ignorado**. Sendo assim, podemos colocar "comentários", facilitando a programação. Abaixo está um exemplo de código:

```
000 1100 LOAD   M(varA)
001 6101 ADD    M(varA)
002 7102 SUB    M(varA)
003 4103 STOR   M(varA)
004 0000 HALT
100 0015 @ varA
101 0020 @ varB
102 0025 @ varC
103 0000 @ varD
```

Para simplicidade, o caractere @ está sendo usado para delimitar uma etiqueta, ou *label*, que **marca um endereço**, assim como era feito no Assembler do IAS. Como o IMAS não tem Assembler, a tradução é feita manualmente, sendo esses artificios apenas facilitadores.

2.1 Estrutura do Processador

O processador simulado usa o modelo de **von Neumann**, isto é dados e instruções na mesma memória. Ainda, cada posição de memória guarda 16 bits.

2.2 Estrutura do Simulador

O programa deve executar até encontrar uma instrução **HALT**. É possível informar ao simulador endereços em que devem ser impressos o conteúdo dos registradores (chamados de *breakpoints*). Esses endereços podem ser passados em linha de comando, após o arquivo de entrada:

```
./imas ex1.txt 000 001
```

Assim, o simulador apresentará o conteúdo (em notação hexadecimal, indicada pelo prefixo '0x') dos registradores ao executar as instruções dos endereços 000 e 001, da seguinte forma:

```
<== IMAS Registers ==>
PC = 0x0000
PC+ = 0x0001
```

```
MAR = 0x0100
IBR = 0x1100
IR = 0x0001
MBR = 0x0015
AC = 0x0015
MQ = 0x0000
<== IMAS Registers ==>
PC = 0x0001
PC+ = 0x0002
MAR = 0x0101
IBR = 0x6101
IR = 0x0006
MBR = 0x0020
AC = 0x0035
MQ = 0x0000
```

Veja que o simulador imprimiu dois valores para PC. PC é o endereço da instrução, e PC+ é o endereço após sua execução (pode ser PC incrementado ou o resultado de um desvio).

Ao programar o simulador não use variáveis adicionais, apenas as variáveis definidas dentro do código! Foi definida no código uma estrutura contendo todos os registradores da CPU. Use a variável **imas** definida nas operações:

```c
/* IMAS registers and memory definitions */
typedef struct imas_t {
    /* UC */
    uint16_t pc;  /* Program Counter */
    uint16_t mar; /* Memory Address Register */
    uint16_t ibr; /* Instruction Buffer Register */
    uint16_t ir;  /* Instruction Register */

    /* ULA */
    int16_t mbr;  /* Memory Buffer Register */
    int16_t ac;   /* Accumulator */
    int16_t mq;   /* Multiplier Quocient */

    /* MEMORY */
    uint16_t memory[IMAS_MEM_SIZE];
} imas_t;

/* Initiate IMAS registers as zero */
imas_t imas = {0};
```

Durante a programação do simulador, apenas complete os espaços marcados como “TODO”. No trecho de código abaixo, por exemplo, você deverá implementar o ciclo de busca, decodificação e execução de instruções usando os registradores definidos acima:

```c
/* Fetch subcycle */
// TODO: Fetch instruction from memory (like in IAS)

```

```
/* Decode subcycle */
// TODO: Put instruction fields in registers

/* Execute subcycle */
switch(imas.ir) {
case IMAS_HALT:
    // TODO
    break;
case IMAS_LOAD_M:
    // TODO
    break;
```

Mas atenção, **todos** os acessos a memória (i.e. busca de instrução e execução de instrução) e ao sistema de entrada e saída (instruções IN e OUT) devem utilizar as instruções definidas para tal. Implemente-as também!

```
/* Executes a read from memory */
void memory_read(imas_t *imas) {
  // TODO
}

/* Executes a write into memory */
void memory_write(imas_t *imas, bool modify_address) {
  if(modify_address) {
    // TODO: Write only operand address field
  }
  else {
    // TODO
  }
}

/* Reads an integer from user */
void io_read(imas_t *imas) {
  printf("IN => ");
  // TODO: scanf("%hd", &<?>);
}

/* Outputs an integer to user */
void io_write(imas_t *imas) {
  // TODO: printf("OUT => %hd\n", <?>);
}
```

2.3 Sistema de Entrada e Saída (E/S)

O simulador fornece ao usuário um sistema básico de entrada e saída. Ao usar a instrução **IN** o usuário poderá inserir um valor inteiro dentro de AC. Ao usar a instrução **OUT** o usuário poderá ver um valor inteiro que estava no AC.

Segue abaixo um exemplo em código que acessa o sistema de E/S:

```
000 E000 IN                 # Le entrada do usuario
001 4100 STOR   M(temp)     # Guarda em temp
002 E000 IN                 # Le outra entrada do usuario
003 6100 ADD    M(temp)     # Soma com temp
004 F000 OUT                # Escreve na saida do usuario
005 0000 HALT               # Finaliza
100 0000 @ temp             # Variavel temporaria
```

3 Conjunto de Instruções

O processador que será simulado possui 16 instruções simples de 16 bits cada. Os 4 bits superiores das instruções indicam o código da operação (*opcode*), e os 12 bits inferiores indicam o endereço do operando. Essa divisão foi feita para facilitar a programação usando notação hexadecimal.

O IMAS consegue executar vários tipos de desvios (ou saltos), tanto condicionais como incondicionais:

- **JMP** (*Jump always*): Sempre salta (equivalente à instrução JUMP do IAS)
- **JZ** (*Jump if equal*): Salta quando AC é zero
- **JNZ** (*Jump if not equal*): Salta quando AC não é zero
- **JPOS** (*Jump if less than*): Salta quando AC é maior ou igual a zero (equivalente à instrução JUMP+ do IAS)

Abaixo estão as instruções do processador que o simulador deverá suportar:

Instrução	Descrição	Tipo	Opcode
HALT	Para a execução	CONTROL	0 0 0 0
LOAD M(X)	MBR ← Mem[MAR], AC ← MBR	LOAD	0 0 0 1
LOAD MQ	AC ← MQ	LOAD	0 0 1 0
LOAD MQ, M(X)	MBR ← Mem[MAR], MQ ← MBR	LOAD	0 0 1 1
STOR M(X)	MBR ← AC, Mem[MAR] ← MBR	STORE	0 1 0 0
STA M(X)	MBR ← AC, Mem[MAR](11:0) ← MBR	STORE	0 1 0 1
ADD M(X)	MBR ← Mem[MAR], AC ← AC + MBR	ALU	0 1 1 0
SUB M(X)	MBR ← Mem[MAR], AC ← AC - MBR	ALU	0 1 1 1
MUL M(X)	MBR ← Mem[MAR], AC:MQ ← MQ * MBR	ALU	1 0 0 0
DIV M(X)	MBR ← Mem[MAR], MQ ← MQ / MBR, MQ ← MQ % MBR	ALU	1 0 0 1
JMP M(X)	PC ← MAR	BRANCH	1 0 1 0
JZ M(X)	If AC == 0, PC ← MAR	BRANCH	1 0 1 1
JNZ M(X)	If AC != 0, PC ← MAR	BRANCH	1 1 0 0
JPOS M(X)	If AC >= 0, PC ← MAR	BRANCH	1 1 0 1
IN	MBR ← IO, AC ← MBR	IO	1 1 1 0
OUT	MBR ← AC, IO ← MBR	IO	1 1 1 1

Veja a descrição de cada instrução e implemente-a dentro do ***switch***, no código. Para conseguir separar os bits, estude operações bit-a-bit (ou *bitwise*) no C, mais especificamente AND, OR e NOT.

4 Avaliação e Entrega

O trabalho poderá ser feito em equipes de duas pessoas, e deverá será enviado ao Moodle, onde será submetido a casos de teste específicos. Serão verificadas as informações mostradas na execução (ao encontrar um *breakpoint*).

A avaliação será feita com base em cinco testes, três mostrados na seção 4.1, e dois extras. O código, ainda, será analisado, de forma a verificar se condiz com o que se pede (por exemplo, se não define variáveis ou funções extras, e se manipula corretamente os registradores). A equipe, então, deve gravar um **vídeo** explicando suas atualizações no código, em no máximo 10 minutos. Suba o vídeo no YouTube, e envie o link no *Moodle* (em atividade própria).

4.1 Exemplo de Entrada

Seguem abaixo alguns exemplos de entrada em Assembly e seu hexadecimal equivalente:

4.1.1 Exemplo 1

Entrada:

```
000 1100 LOAD   M(varA)
001 6101 ADD    M(varA)
002 7102 SUB    M(varA)
003 4103 STOR   M(varA)
004 0000 HALT
100 0015 @ varA
101 0020 @ varB
102 0025 @ varC
103 0000 @ varD
```

Saída:
Para *breakpoints* na linha 001 e 002:

```
<== IMAS Registers ==>
PC = 0x0001
PC+ = 0x0002
MAR = 0x0101
IBR = 0x6101
IR = 0x0006
MBR = 0x0020
AC = 0x0035
MQ = 0x0000
<== IMAS Registers ==>
```

```
PC = 0x0002
PC+ = 0x0003
MAR = 0x0102
IBR = 0x7102
IR = 0x0007
MBR = 0x0025
AC = 0x0010
MQ = 0x0000
```

4.1.2 Exemplo 2

Entrada:

```
000 E000 IN                 # Le entrada do usuario
001 4100 STOR   M(temp)     # Guarda em temp
002 E000 IN                 # Le outra entrada do usuario
003 6100 ADD    M(temp)     # Soma com temp
004 F000 OUT                # Escreve na saida do usuario
005 0000 HALT               # Finaliza
100 0000 @ temp             # Variavel temporaria
```

Saída:
Para *breakpoint* na linha 003:

```
IN => 1
IN => 2
<== IMAS Registers ==>
PC = 0x0003
PC+ = 0x0004
MAR = 0x0100
IBR = 0x6100
IR = 0x0006
MBR = 0x0001
AC = 0x0003
MQ = 0x0000
OUT => 3
```

4.1.3 Exemplo 3

Entrada:

```
000 1101 LOAD   M(counter_1)    @ loop_1
001 7100 SUB    M(const_1)
002 4101 STOR   M(counter_1)
003 D005 JPOS   M(do_1)
004 A00C JMP    M(loop_2)
005 E000 IN                     @ do_1
006 4106 STOR   M(arr)          @ store_arr
007 1104 LOAD   M(arr_ptr_1)
008 6100 ADD    M(const_1)
009 4104 STOR   M(arr_ptr_1)
```

```
00A 5006 STA    M(store_arr)
00B A000 JMP    M(loop_1)
00C 1102 LOAD   M(counter_2)    @ loop_2
00D 7100 SUB    M(const_1)
00E 4102 STOR   M(counter_2)
00F D013 JPOS   M(load_arr)
010 1103 LOAD   M(result)
011 F000 OUT
012 0000 HALT
013 1106 LOAD   M(arr)          @ load_arr
014 6103 ADD    M(result)
015 4103 STOR   M(result)
016 1105 LOAD   M(array_ptr_2)
017 6100 ADD    M(const_1)
018 4105 STOR   M(array_ptr_2)
019 5013 STA    M(load_arr)
01A A00C JMP    M(loop_2)
100 1                           @ const_1
101 3                           @ counter_1
102 3                           @ counter_2
103 0                           @ result
104 106                         @ arr_ptr_1
105 106                         @ array_ptr_2
106 0                           @ arr (start)
```

Saída:

Para *breakpoint* na linha 010:

```
IN => 1
IN => 2
IN => 3
<== IMAS Registers ==>
PC = 0x0010
PC+ = 0x0011
MAR = 0x0103
IBR = 0x1103
IR = 0x0001
MBR = 0x0006
AC = 0x0006
MQ = 0x0000
OUT => 6
```