2o. Trabalho de Arquitetura de Computadores I

- **Objetivo**:
  - Desenvolvimento de um programa simulador da arquitetura do MIPS, para executar programas em linguagem de máquina MIPS
  - **Simulador funcional**: reproduz comportamento da arquitetura do MIPS
- **Simulador**:
  - Recebe como entrada um programa em linguagem de máquina MIPS (instruções e dados)
  - Executa instruções do programa, uma após a outra

Instruções de Máquina Simuladas

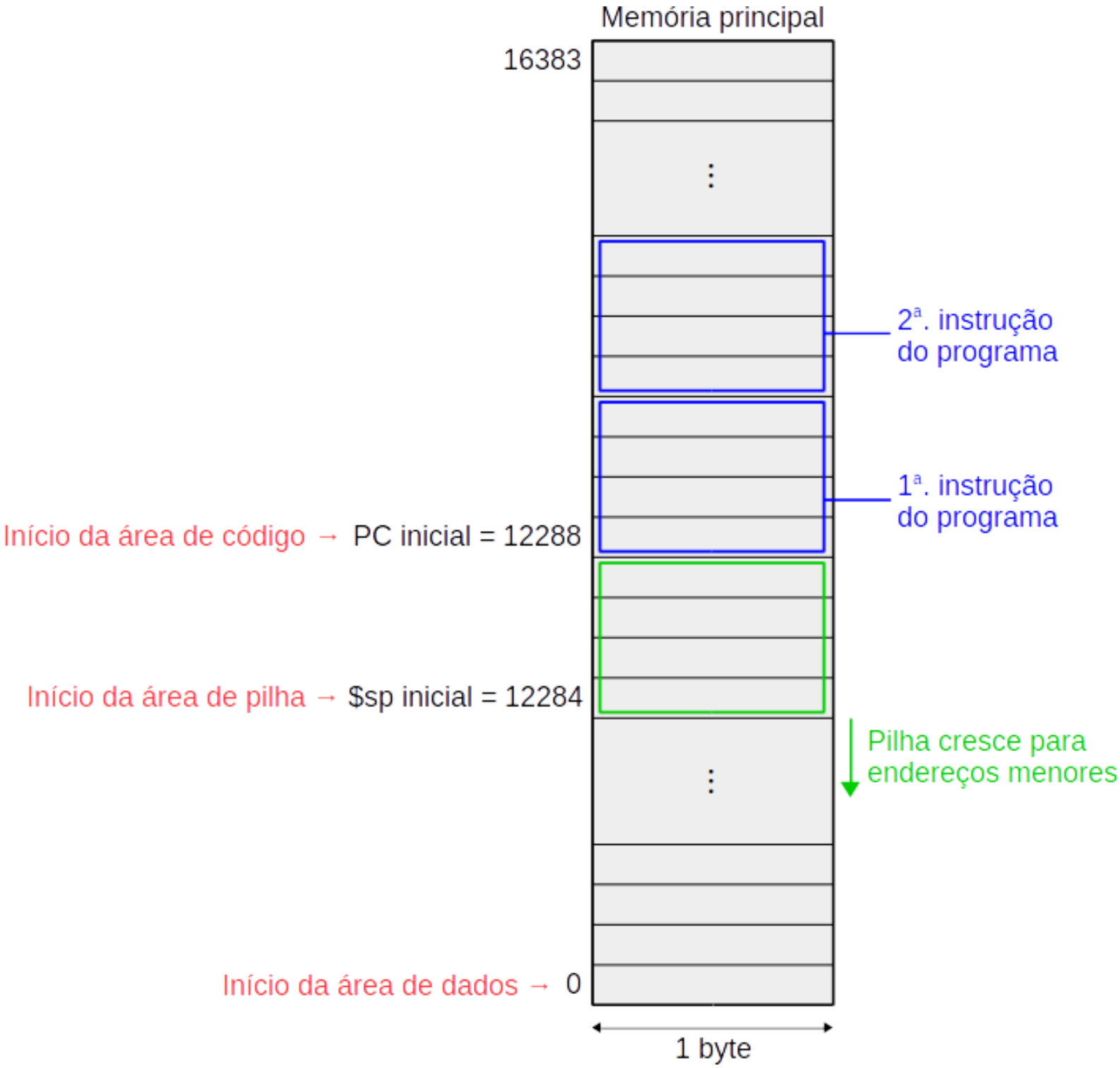
- Lógico-aritméticas:
  - `add`, `sub`, `addi`, `and`, `or`, `xor`, `nor`, `andi`, `ori`, `sll`, `srl`, `slt`, `slti`
- Multiplicação e divisão: `mult`, `div`
- Acesso à memória: `lw`, `sw`, `lb`, `sb`
- Desvios condicionais: `beq`, `bne`
- Desvios incondicionais: `j`, `jr`, `jal`
- Chamada ao sistema operacional: `syscall`
  - Serviços simulados:
    - 1 (*print integer*)
    - 4 (*print string*)
    - 5 (*read integer*)
    - 8 (*read string*)
    - 10 (exit)
- Outras: `mfhi`, `mflo`, `lui`

Instrução	Exemplo	Significado	Formato	Campo Op	Campo Funct
add	`add rd, rs, rt`	Reg[rd] = Reg[rs] + Reg[rt]	R	000000	100000
subtract	`sub rd, rs, rt`	Reg[rd] = Reg[rs] − Reg[rt]	R	000000	100010
add immediate	`addi rt, rs, immediate`	Reg[rt] = Reg[rs] + sign_ext(immediate)	I	001000	—
multiply	`mult rs, rt`	Hi, Lo = Reg[rs] × Reg[rt]	R	000000	011000
divide	`div rs, rt`	Lo = Reg[rs] div Reg[rt] ; Hi = Reg[rs] mod Reg[rt]	R	000000	011010
move from Hi	`mfhi rd`	Reg[rd] = Hi	R	000000	010000
move from Lo	`mflo rd`	Reg[rd] = Lo	R	000000	010010
and	`and rd, rs, rt`	Reg[rd] = Reg[rs] and Reg[rt]	R	000000	100100
or	`or rd, rs, rt`	Reg[rd] = Reg[rs] or Reg[rt]	R	000000	100101
xor	`xor rd, rs, rt`	Reg[rd] = Reg[rs] xor Reg[rt]	R	000000	100110
nor	`nor rd, rs, rt`	Reg[rd] = not (Reg[rs] or Reg[rt])	R	000000	100111
and immediate	`andi rt, rs, immediate`	Reg[rt] = Reg[rs] and zero_ext(immediate)	I	001100	—
or immediate	`ori rt, rs, immediate`	Reg[rt] = Reg[rs] or zero_ext(immediate)	I	001101	—
shift left logical	`sll rd, rt, shamt`	Reg[rd] = Reg[rt] ≪ shamt	R	000000	000000
shift right logical	`srl rd, rt, shamt`	Reg[rd] = Reg[rt] ≫ shamt	R	000000	000010
load word	`lw rt, immediate (rs)`	$\text{Reg[rt]} = \text{Mem[Reg[rs]} + \text{sign\_ext(immediate)]}_{\text{w}}$	I	100011	—
store word	`sw rt, immediate (rs)`	$\text{Mem[Reg[rs]} + \text{sign\_ext(immediate)]}_{\text{w}} = \text{Reg[rt]}$	I	101011	—
load byte	`lb rt, immediate (rs)`	$\text{Reg[rt]} = \text{sign\_ext(Mem[Reg[rs]} + \text{sign\_ext(immediate)]}_{\text{b}})$	I	100000	—
store byte	`sb rt, immediate (rs)`	$\text{Mem[Reg[rs]} + \text{sign\_ext(immediate)]}_{\text{b}} = \text{Reg[rt]}_{7\text{-}0}$	I	101000	—
load upper immediate	`lui rt, immediate`	Reg[rt] = immediate::0000000000000000	I	001111	—
branch on equal	`beq rs, rt, immediate`	if Reg[rs] == Reg[rt] then PC = PC + 4 + (sign_ext(immediate) ≪ 2)	I	000100	—
branch on not equal	`bne rs, rt, immediate`	if Reg[rs] ≠ Reg[rt] then PC = PC + 4 + (sign_ext(immediate) ≪ 2)	I	000101	—
set on less than	`slt rd, rs, rt`	if Reg[rs] < Reg[rt] then Reg[rd] = 1 else Reg[rd] = 0	R	000000	101010
set on less than immediate	`slti rt, rs, immediate`	if Reg[rs] < sign_ext(immediate) then Reg[rt] = 1 else Reg[rt] = 0	I	001010	—
jump	`j address`	$\text{PC} = (\text{PC+4})_{31\text{-}28}$::address::00	J	000010	—
jump register	`jr rs`	PC = Reg[rs]	R	000000	001000
jump and link	`jal address`	$\text{Reg[31]} = \text{PC} + 4$; $\text{PC} = (\text{PC+4})_{31\text{-}28}$::address::00	J	000011	—
system call	`syscall`	Faz chamada ao sistema solicitando serviço cujo nº está em $v0	R	000000	001100

Simulador

- **Estruturas de dados do simulador**:
  - **Conjunto de registradores**: vetor de 32 inteiros
  - **Registradores PC, IR, Hi, Lo**: inteiros
  - **Memória principal**: vetor de 16.384 **bytes** (2^{14})
- **Entradas do simulador**:
  - Arquivo com instruções de máquina do programa a ser executado
  - Arquivo com dados do programa a ser executado
- **Saída do simulador**:
  - Arquivo com valores dos registradores ao final da execução do programa

Uso da Memória pelo Programa Executável MIPS



Estrutura do Simulador: Programa Principal

```
inicializa() ;

// Laço que realiza ciclo de execução de uma instrução
// A cada iteração do laço, uma instrução é executada
while (! fim_execucao)
{
    // Lê da memória instrução a executar (apontada por PC) e guarda em IR
    IR = busca_instrucao(PC) ;

    // Atualiza PC
    PC = PC + 4 ;

    // Decodifica instrução em IR: determina campo op e demais campos
    decodifica_instrucao(IR, ...) ;

    // Executa instrução, com base em seus campos
    // Dependendo da instrução, PC pode ser alterado
    executa_instrucao(...) ;
}

finaliza() ;
```

Estrutura do Simulador: Rotinas Inicializa e Finaliza

- **Inicializa**:
  - **Carrega instruções do programa na memória**:
    - Lê arquivo de entrada com instruções, byte a byte, copiando para memória, **a partir do endereço 12288**
    - Arquivo **binário** com **sequência de bytes** (não é arquivo texto)
      - 4 bytes por instrução
      - Usar `fread`
  - **Carrega dados do programa na memória**:
    - Lê arquivo binário de entrada com dados, byte a byte, copiando para memória, **a partir do endereço 0**
    - Arquivo **binário** com **sequência de bytes** (não é arquivo texto)
      - Usar `fread`
  - **Inicializa registradores**:
    - `PC` = **12288** (0x00003000) (endereço da 1ª instrução do programa na memória)
    - `$zero` = 0
    - `$sp` = **12284** (0x00002FFC) (endereço do início da pilha na memória)
- **Finaliza**:
  - **Faz dump do conjunto de registradores**:
    - Escreve valor dos 32 registradores do conjunto de registradores em arquivo **texto** de saída, ao final da execução do programa

Estrutura do Simulador: Rotinas Busca / Decodifica Instrução

```
... busca_instrucao(int PC)
{
    ...
    // Lê 4 bytes da instrução da memória e junta 4 bytes em IR (palavra)
    // (através de manipulação de bits)
    IR = ... memoria[PC+3] ... memoria[PC+2]
         ... memoria[PC+1] ... memoria[PC] ;
    return IR ;
}

decodifica_instrucao(int IR, ...)
{
    // Extrai campos da instrução de IR (através de manipulação de bits)
    op          = IR ...
    rs          = IR ...
    rt          = IR ...
    rd          = IR ...
    shamt       = IR ...
    funct       = IR ...
    immediate   = IR ...
    address     = IR ...
}
```

Importante: Palavras na memória organizadas em **little-endian**
(1º byte é o byte menos significativo da palavra)
Atenção: leitura da instrução, instruções `lw` e `sw`

Estrutura do Simulador: Rotina Executa_Instrução

```
executa_instrucao(...)
{
    switch (op)
    {
        case 0: // formato R
            switch (funct)
            {
                case 0: // instrução sll
                    registrador[rd] = registrador[rt] << shamt ;
                    break ;
                ...
                case 8: // instrução jr
                    PC = registrador[rs] ;
                    break ;
                ...
                case 32: // instrução add
                    registrador[rd] = registrador[rs] + registrador[rt] ;
                    break ;
                ...
            }
            break ;
        case 2:  // instrução j
            ...
            break ;
        ...
        case 35: // instrução lw
            endereco = registrador[rs] + immediate ;
            registrador[rt] = ...memoria[endereco+3]...memoria[endereco+2]
                              ...memoria[endereco+1]...memoria[endereco]
            break ;
        ...
    }
}
```

Arquivos de Entrada para Simulador

- **Gerados usando MARS**:
  - Abrir programa `<programa>.asm` no MARS
  - **Ir no menu *Settings*, opção *Memory Configuration***:
    - Selecionar *Configuration*: *Compact, Data at address 0*
    - Dar *Apply*
  - **Montar programa**
  - **Ir no menu *File*, opção *Dump Memory***:
    - Selecionar *Memory Segment*: *.text*
    - Selecionar *Dump Format*: *Binary*
    - Dar *Dump To File*: `<programa>.text`
  - **Ir no menu *File*, opção *Dump Memory***:
    - Selecionar *Memory Segment*: *.data*
    - Selecionar *Dump Format*: *Binary*
    - Dar *Dump To File*: `<programa>.data`
- **3 programas de teste fornecidos**:
  - `soma_vetor`, `potencias2`, `fatorial`

Para visualizar arquivos binários:
- `ghex <programa>.text`
- `hexdump -C <programa>.text`

Importante

- **Interface de execução do simulador**:

  `simmips <programa>.text <programa>.data <programa>.reg`

- **Programa simulador**:
  - Deve ser desenvolvido em: C, c++ ou java
- **Grupos**: de 2 alunos
- **Submissão no Moodle**:
  - Submeter um único arquivo `trab.zip`, contendo:
    - Arquivos **fontes** que compõem o programa desenvolvido
  - No cabeçalho do programa fonte principal, incluir em comentários:
    - Nomes dos integrantes do grupo
    - Instruções de como compilar o programa
  - **Não** submeter programa executável, arquivos de entrada do simulador, ...
- **Data de entrega**: