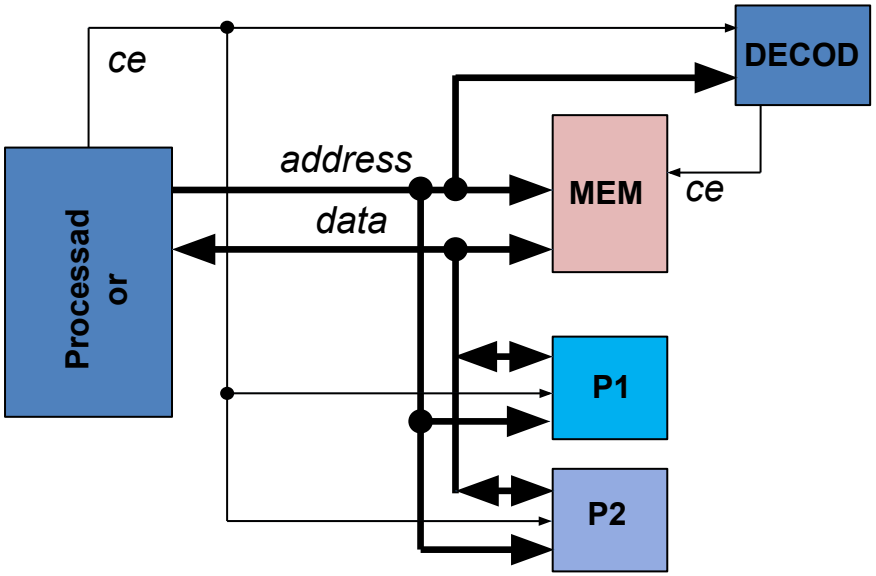
Interface entre Processador e Periféricos

- Transferência de dados
  - Existem basicamente três técnicas envolvidas na transferência de dados entre processador e periféricos
    - *Polling*
    - Interrupção
    - Acesso direto à memória (DMA – *Direct Memory Access*)

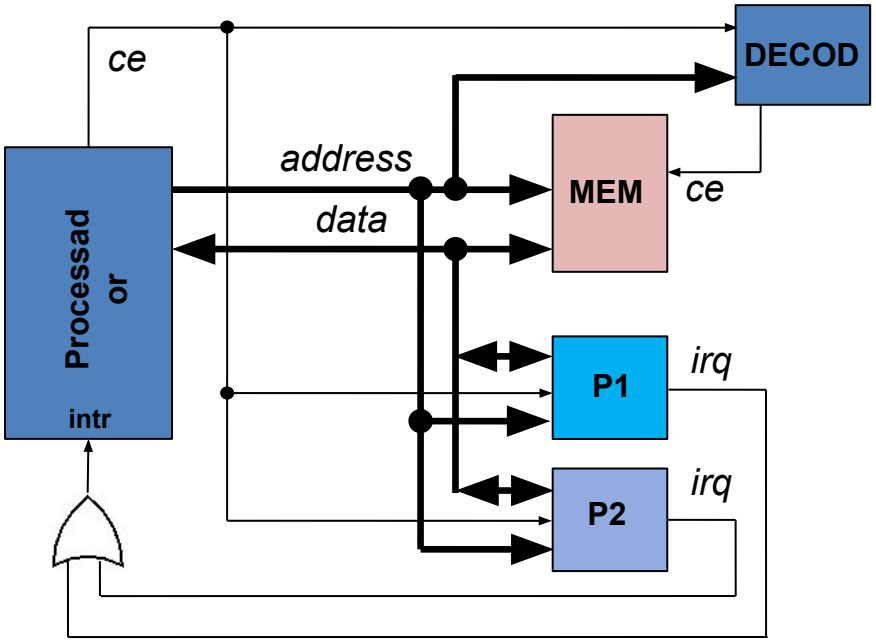


Interface entre Processador e Periféricos

- Interrupção
  - Criada para solucionar o problema do tempo desperdiçado com a verificação de estado do periférico inerente ao *polling*
  - O periférico é responsável por "chamar a atenção" do processador quando
    - Existem dados para serem lidos pelo processador
    - Periférico está pronto para receber dados do processador
    - "Lembrar" o processador que alguma tarefa periódica deve ser executada
  - Quando o processador atende ao pedido de interrupção, ele executa a rotina de tratamento de interrupção (*ISR – Interrupt Service Routine*)
    - Trecho de código que verifica o periférico origem da interrupção e desvia para o manipulador correspondente (*handler*)
    - Para cada perférico existe um *handler*
      - Código que interage com o periférico (*device driver*)

Interface entre Processador e Periféricos

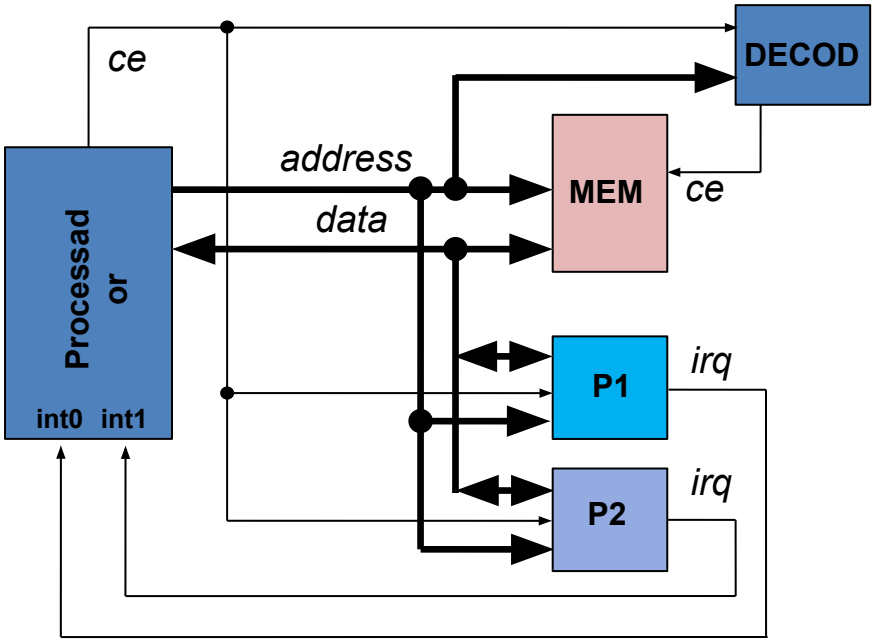
- Interrupção
  - O processador possui uma entrada de interrupção (e.g. *intr*), a qual é utilizada pelos periféricos para "chamar a atenção"
  - A interrupção é gerada pelos periféricos através de um bit de interrupção (*irq – interrupt request*)
  - Interrupções podem ocorrer a qualquer instante (assíncronas)



Interface entre Processador e Periféricos

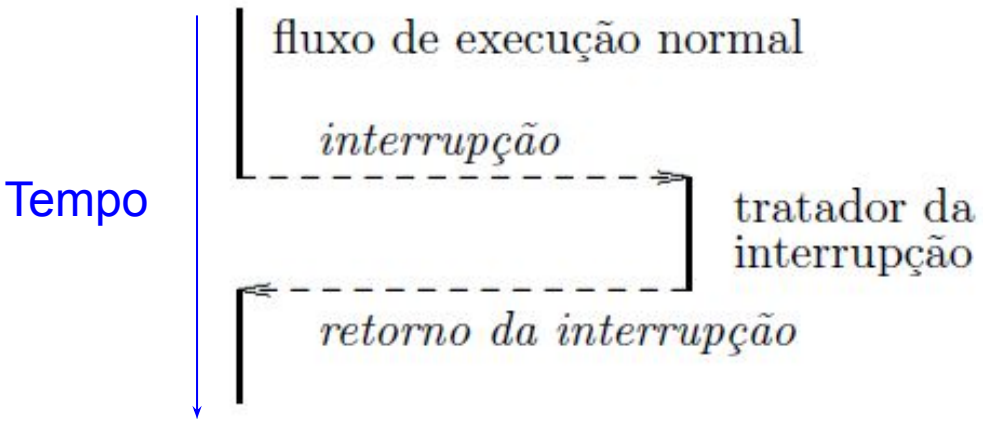
- Interrupção
  - O processador possui uma entrada de interrupção (e.g. *intr*), a qual é utilizada pelos periféricos para "chamar a atenção"
  - A interrupção é gerada pelos periféricos através de um bit de interrupção (*irq – interrupt request*)
  - Interrupções podem ocorrer a qualquer instante (assíncronas)

Pode existir mais de uma entrada de interrupção no processador



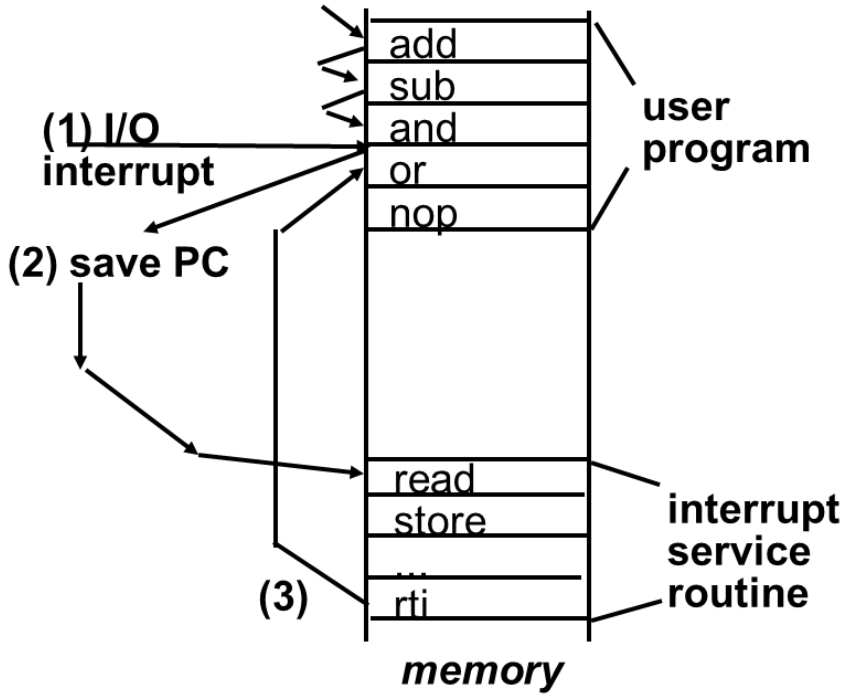
Interface entre Processador e Periféricos

- Interrupção
  - Este mecanismo se chama de interrupcão porque a sequência normal de execução de um programa é interrompida para que um periférico seja atendido
  - O tratamento de uma interrupção é similar a uma chamada de sub-rotina
  - Ao final do tratamento da interrupção, o processador retorna ao fluxo de intruções que era executado antes da interrupção



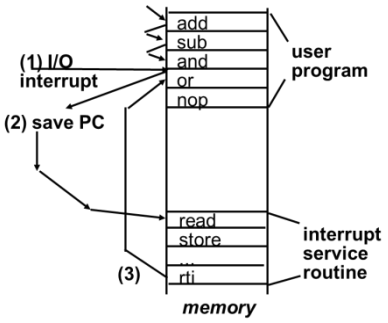
Interface entre Processador e Periféricos

- Interrupção
  - Ao invés de o programador inserir no código uma chamada para a ISR (e.g. *jump*), é o periférico que ao interromper dispara a sua execução
    - Processamento muito parecido com uma chamada de sub-rotina



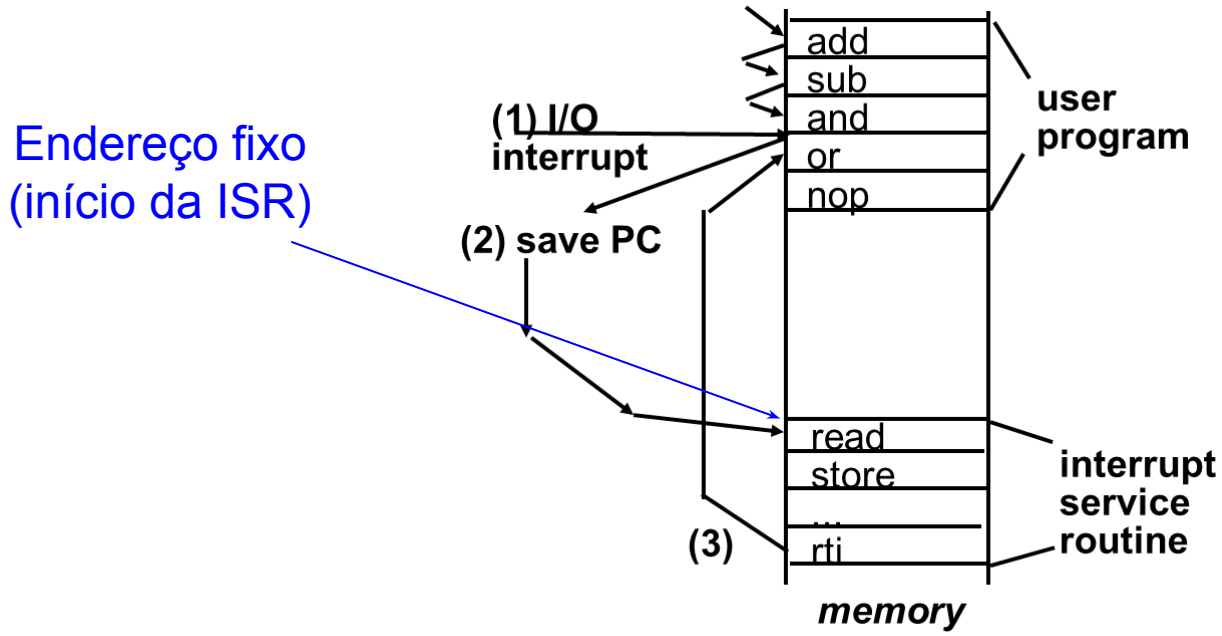
Interface entre Processador e Periféricos

- Interrupção
  - Antes de buscar a próxima instrução do programa em execução, o processador verifica se há um pedido de interrupção (*intr = 1*)
    - Se não há nenhum pedido de interrupção, a próxima instrução do programa é buscada
    - Se há um pedido de interrupção, o processador interrompe a execucão do programa e salta para a ISR
      - Antes de saltar, o endereço da instrução do programa que seria executada é armazenado (*link*)
      - O PC é setado com o endereço da ISR (salto)
      - A execução da ISR é inciada
      - Ao final da execução da ISR, o processador salta de volta (*return*) para o endereço da instrução do programa que não foi executada em virtude da interrupção e continua a execução do programa interrompido



Interface entre Processador e Periféricos

- Interrupção
  - O endereço de desvio quando ocorre uma interrupção é tipicamente fixo (*hardwired*) e varia de processador para processador
  - Portanto, o código da ISR deve sempre ser alocado na mesma posição na memória



Interface entre Processador e Periféricos

- Interrupção
  - Enquanto a técnica *polling* é puramente implementada em software, a E/S com interrupção requer um suporte de hardware
  - A vantagem é que o processador não precisa ficar verificando constantemente o estado do periférico
  - Entretanto, a transferência de dados continua sendo realizada pelo processador
    - O processador é responsável por ler dados da memória e enviar ao periférico ou
    - Ler dados do periférico e armazenar na memória
    - Para liberar o processador da transferência de dados entre periférico e memória utiliza-se um outro circuito periférico chamado DMA (*Direct Memory Access*)

Interface entre Processador e Periféricos

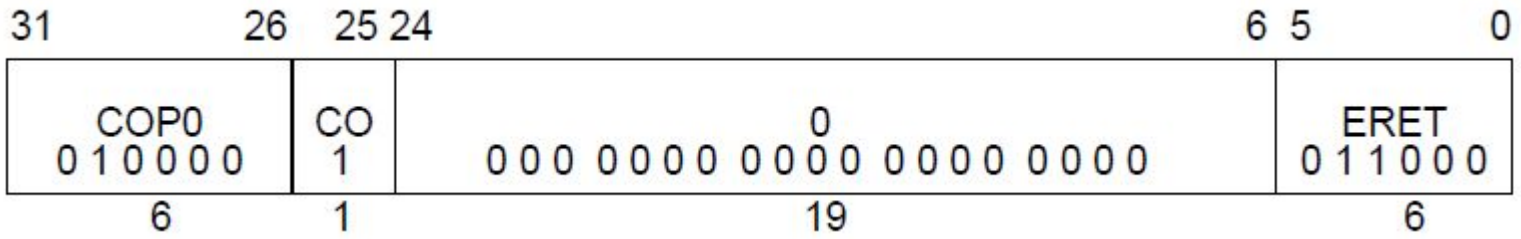
- Trabalho 3 – parte 1
  - Adicionar ao processador MIPS capacidade de atender interrupções
  - Sempre que um pedido de interrupção for feito, o processador deve saltar para uma ISR e executar o *handler* de acordo com o periférico que gerou a interrupção
  - Ao encerrar a ISR, a execução deve retornar de onde foi interrompida
  - Na interface do processador deve ser adicionado uma entrada de interrupção *intr* (*interrput request*)

Interface entre Processador e Periféricos

- Trabalho 3 – parte 1
  - Se *intr = 1* no meio da execução de uma instrução, a instrução deve terminar a execução e em seguida o processador deve saltar
    - Dica: verificar *intr* no estado *de fetch*
  - Deve-se adicionar um registrador (EPC – *Exception PC*) para armazenar o endereço de retorno (*link*)
    - Antes de saltar para a ISR, o PC deve ser gravado no registrador EPC a fim de possibilitar o retorno (*link*)
    - No jargão do MIPS, qualquer desvio do fluxo normal de intruções é chamado de exceção (e.g. interrupção gerada por um periférico)

Interface entre Processador e Periféricos

- Trabalho 3 – parte 1
  - Duranto o atendimento de um pedido de interrupção, o processador deve ignorar a entrada *intr*, de maneira a não ser interrompido novamente (ISR não reentrante)
  - O retorno da ISR para o programa é feito utilizando a instrução *ERET*
  - Formato



  - Ao executá-la, PC ← EPC (*return*) e o processador pode tratar novos pedidos de interrução

Interface entre Processador e Periféricos

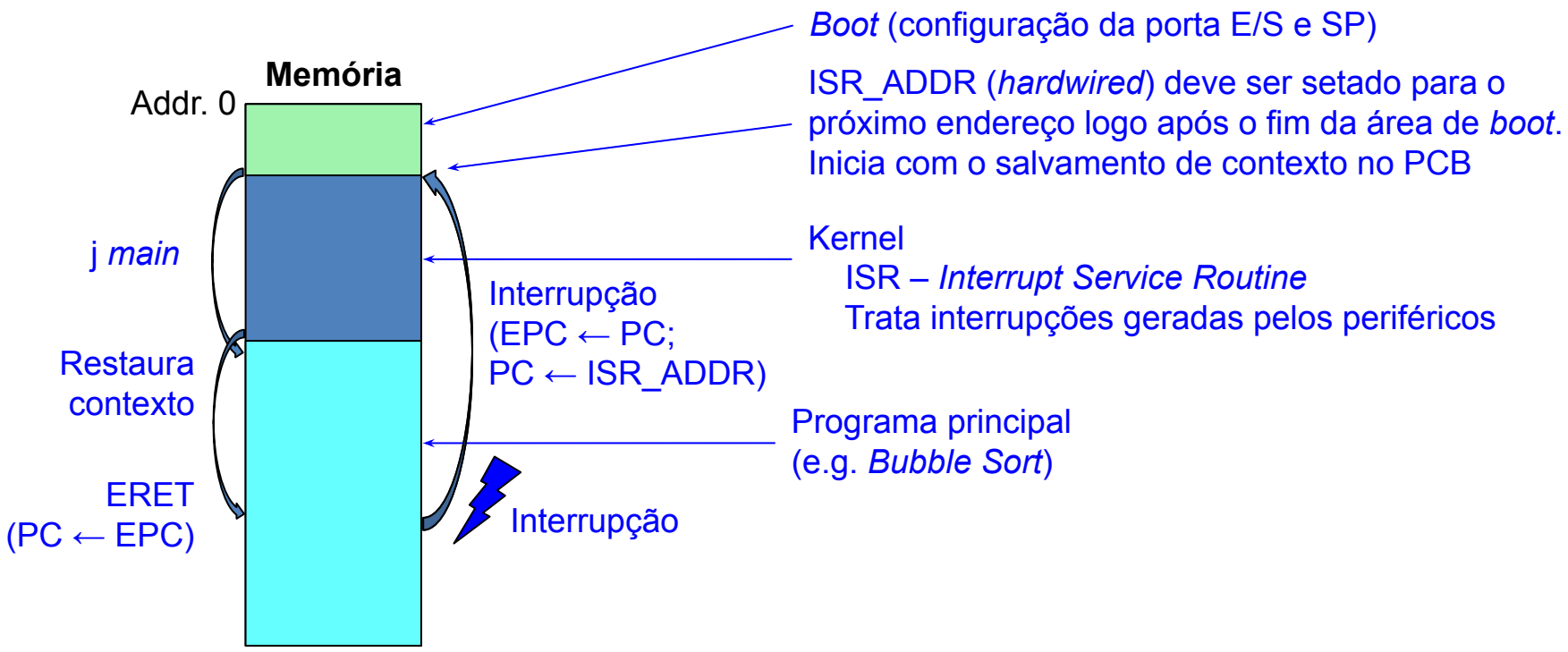
- Trabalho 3 – parte 1
  - A partir deste trabalho começaremos a trabalhar com a noção de kernel
  - O kernel é parte do sistema operacional (e.g. Windows, MAC OS, Linux…) responsável por funções de baixo nível como alocação de memória para programas, salvamento/recuperação de contexto, comunicação com periféricos (*handlers*), tratamento de interrupções (ISR) entre várias outras coisas
  - A ISR faz parte do kernel, o qual começará a ser construído neste trabalho

Trabalho 3 – parte 1

- Salvamento de contexto
  - Em caso de interrupção, o contexto do programa em execução (processo) deve ser armazenado em uma variável (*array*)
  - A variável que armazena o contexto de um programa em execução é chamada de PCB – *Process Control Block*
    - Deve ter capacidade para armazenar todos registradores do banco de registradores a exceção do $k0 ($26) e $k1 ($27)
    - Os registradores $k0 e $k1 devem ser utilizados pelo kernel para manipular o PCB sem alterar os demais registradores os quais podem estar sendo usados pelo processo interrompido
  - PCB (*Process Control Block*)
    - $at, $v0, $v1, $a0-$a3, $t0-$t9, $s0-$s7, $gp, $fp, **$sp** e $ra
    - 29 registradores
    - 112 bytes
    - PCB: .space 112 (*array* de 29 *words*)

Trabalho 3 – parte 1

- Recuperação de contexto
  - Imediatamente antes do retorno da ISR (ERET) o conteúdo dos registradores deve ser restaurado
- Estrutura do código na memória de instruções



Interface entre Processador e Periféricos

- Trabalho 3 – parte 1
  - Estrutura da ISR

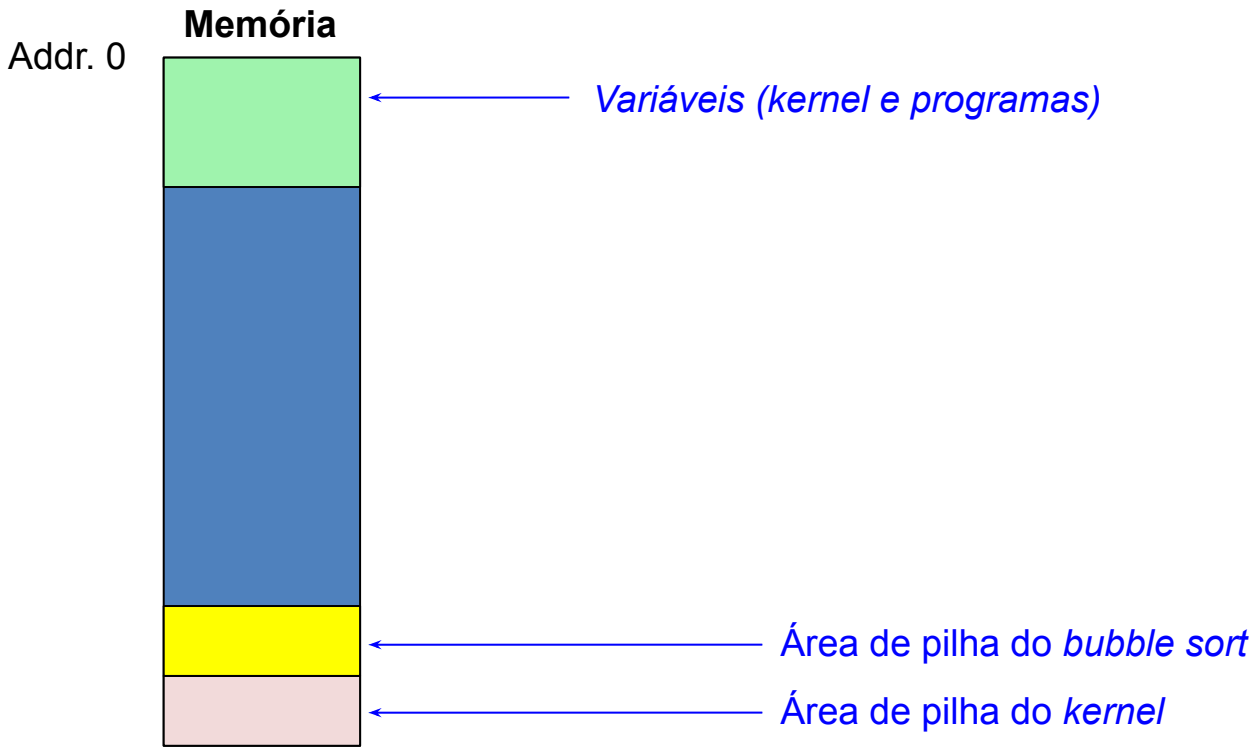
```
InterruptionServiceRoutine:
    1. Salvamento de contexto no PCB
   (todos regs menos 0, 26 e 27)
    2. Verificação do periférico origem da interrupção e
   salto para handler correspondente (jal)
    3. Recuperação de contexto
    4. Retorno (eret)

Periférico1Handler:
    ...
    jr ra

Periférico2Handler:
    ...
    jr ra
...
```

Trabalho 3 – parte 1

- Pilha
  - O kernel e o programa em execução devem ter cada um seu espaço de pilha (memória de dados)
    - Deve-se definir um *$sp* inicial para o kernel e outro para o programa em execução (*bubble sort)*
    - No *boot*, antes de saltar para *bubble sort*, deve-se setar o *$sp* do *bubble sort*
    - Sempre que a execução passar para a ISR (*kernel)*, o *$sp* tem de ser setado para a área de pilha do *kernel*
    - Ao retonar para o programa interrompido (*bubble sort)*, o *$sp* é restaurado com o valor que foi gravado no PCB na entrada da ISR

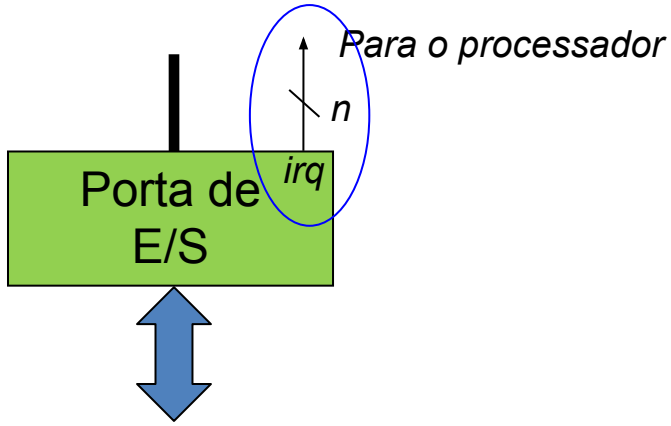
Trabalho 3 – parte 1

- Estrutura da memória de dados



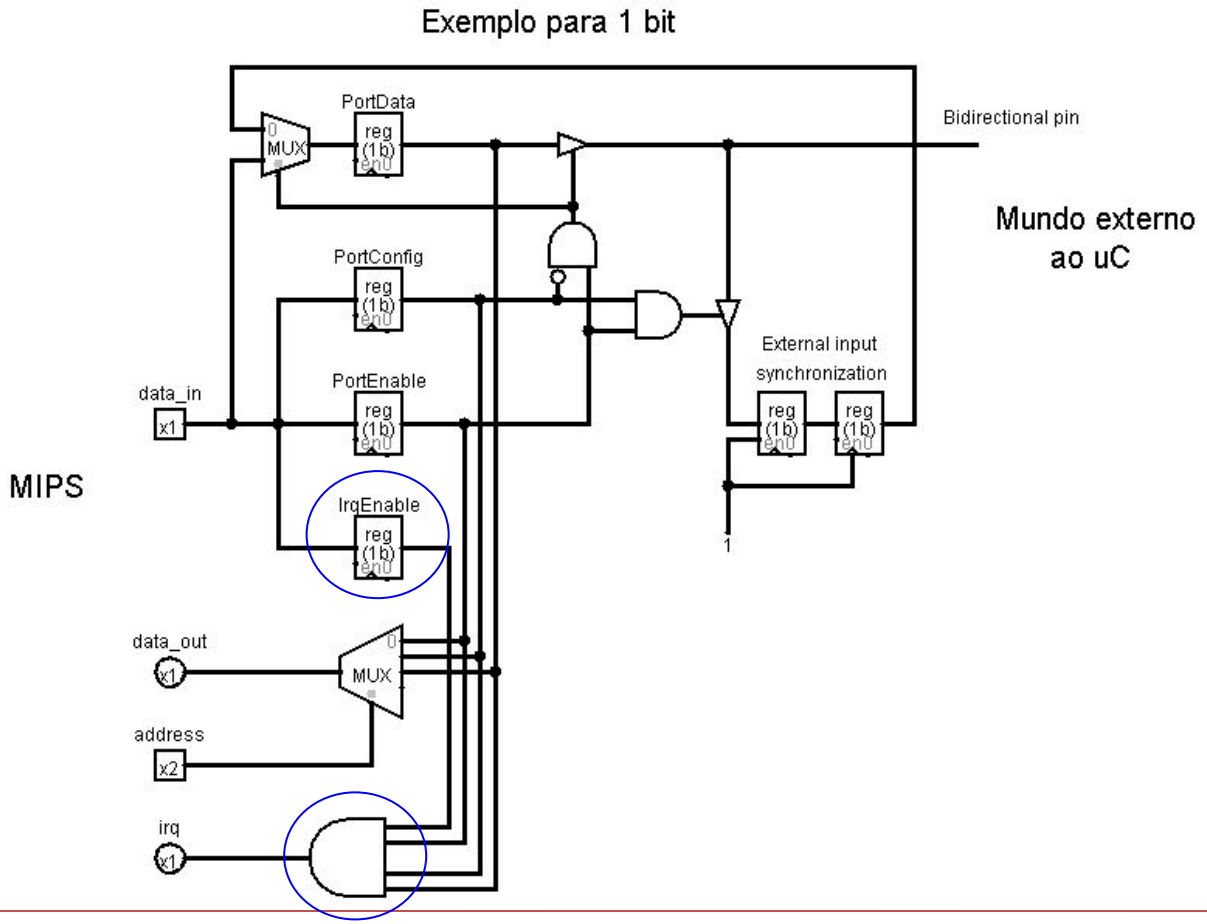
Interface entre Processador e Periféricos

- Trabalho 3 – parte 1
  - Adicionar saída de interrupção (*irq*) na porta de E/S
    - A saída *irq* deve ter a mesma largura da porta de E/S
    - Qualquer bit da porta de E/S pode ser configurado como entrada de interrupção externa ao MIPS_µC
    - Deve ser adicionado outro registrador à porta de E/S a fim de configurar entradas de interrupção
    - Bits da porta de E/S configurados como entrada de interrupção são ligados aos bits correspondentes da saída *irq*
      - *Exemplo: bits 1 e 3 da porta de E/S configurados como entada de interrupção*
        - *irq(1) ← port_io(1)*
        - *irq(3) ← port_io(3)*



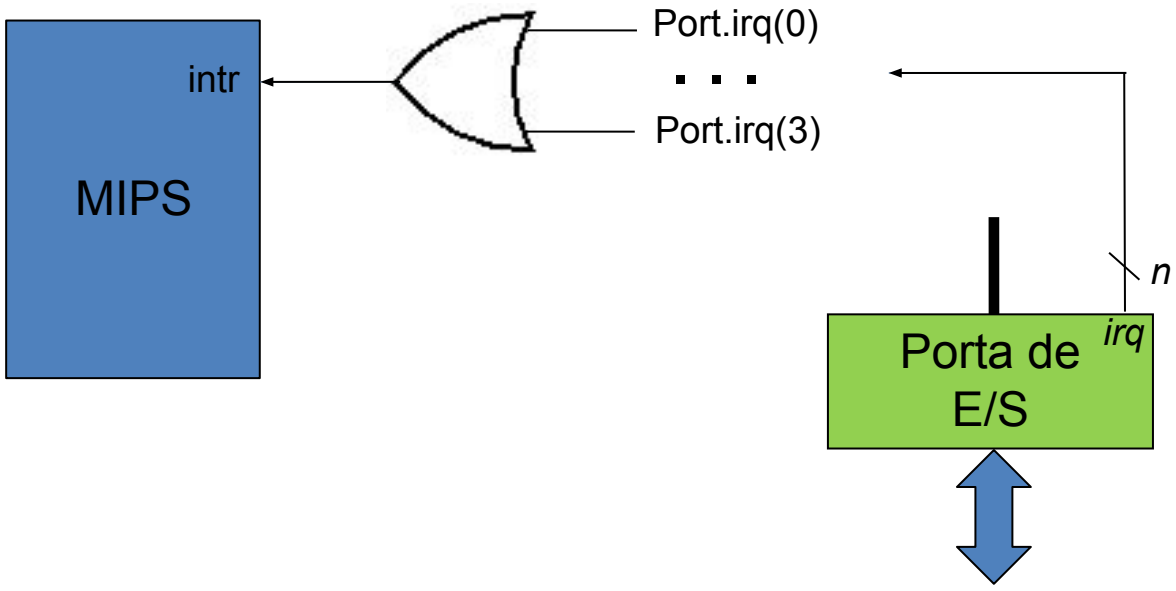
Interface entre Processador e Periféricos

- Trabalho 3 – parte 1
  - Adicionar saída de interrupção (*irq*) na porta de E/S



Interface entre Processador e Periféricos

- Trabalho 3 – parte 1
  - A entrada de interrupção do processador deve receber o *or* de todas saídas *irq* da porta de E/S

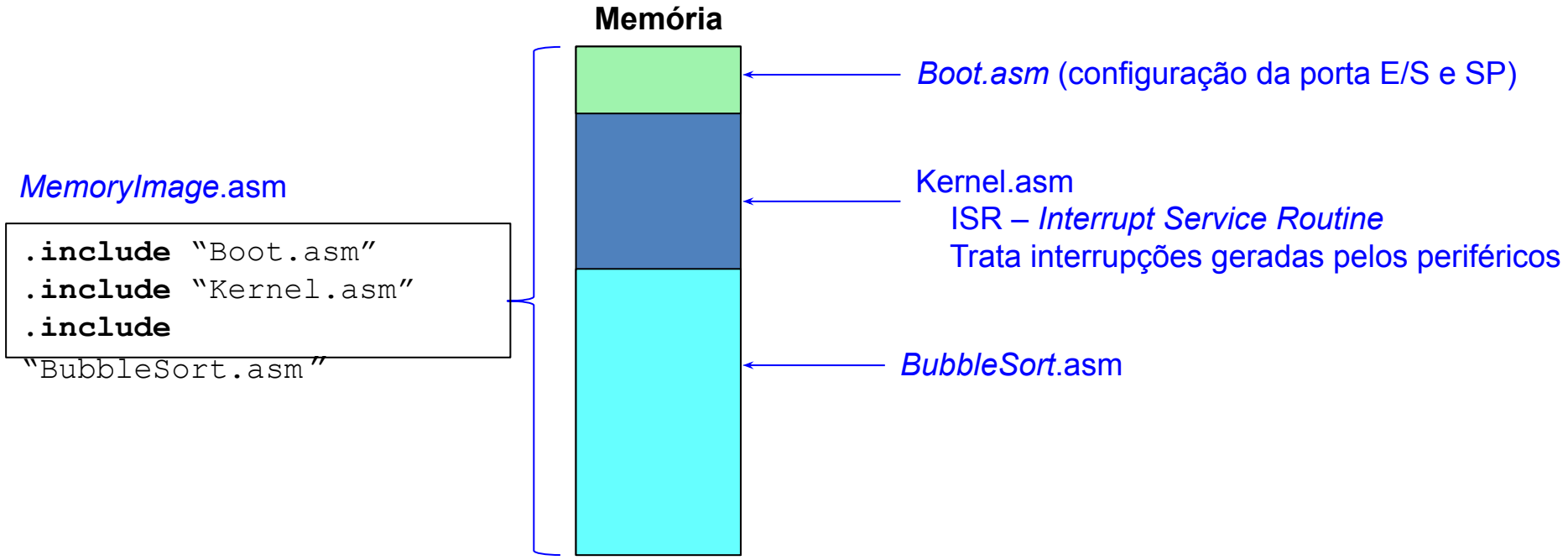


Trabalho 3 – parte 1

- Aplicação
  - Executar o *bubble sort* como programa em execução (processo)
    - Nova versão no *moodle* (utilizando pilha)
  - Implementar um *handler* que controla um contador de 8 bits na porta de E/S
    - A valor da contagem fica armazenado no registrador *portData* e deve aparecer em 8 bits da porta de E/S configurados como saídas
  - Configurar um bit da porta de E/S como entrada de interrupção
    - A cada interrução gerada o contador deve ser incrementado
    - As interrupções devem ser geradas via *test-bench*
    - A execução do *bubble sort* deve ser interrompida várias vezes a fim de verificar o funcionamento do suporte a interrupções (hardware/software)

Trabalho 3 – parte 1

- Aplicação
  - Dica 1
    - Visto que a memória de instruções está dividida em regiões, utilizar a diretiva *include* do MARS a fim de particionar os códigos em arquivos *assembly* separados (e.g. Boot.asm, Kernel.asm, BubbleSort.asm e MemoryImage.asm)



Trabalho 3 – parte 1

- Aplicação
  - Dica 2
    - Aprender a utilizar as macros do MARS (Help do MARS) a fim de tornar o código mais legível