Arquitetura de Computadores

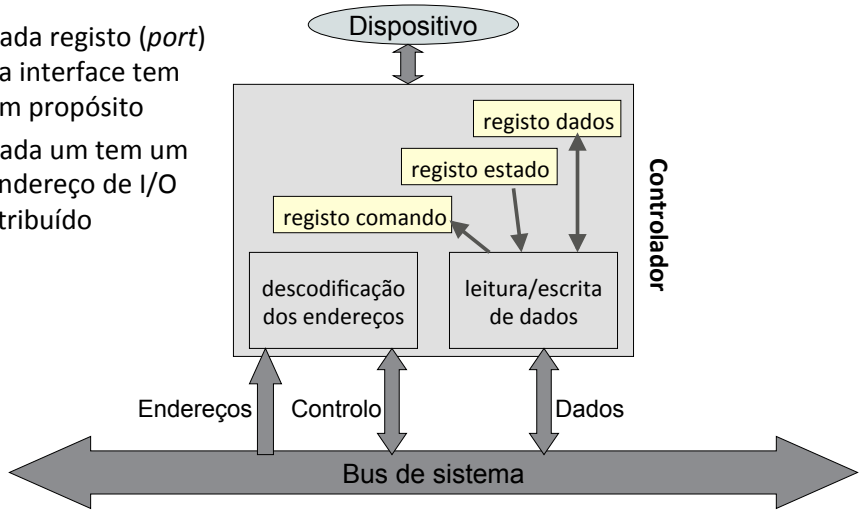
MiEI – 2016/17

DI-FCT/UNL

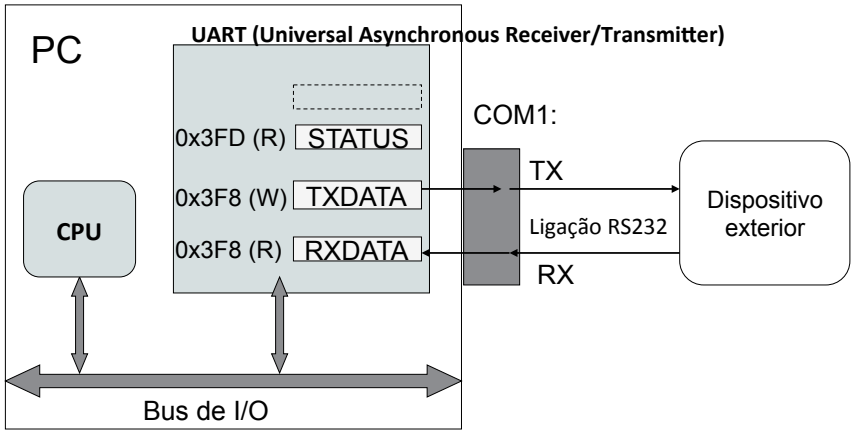
Aula 14

Interface genérica de periférico

- Cada registo (*port*) da interface tem um propósito
- Cada um tem um endereço de I/O atribuído

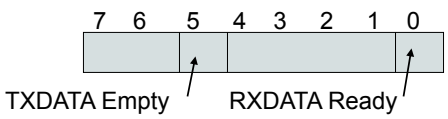


Interface do controlador



Interface da UART

- A interface de programação oferecida esconde os detalhes da comunicação série
  - Interacção com o controlador por bytes de dados
  - Suporta *full-duplex*: 1 reg dados a enviar (TXDATA) e 1 reg. de dados recebidos (RXDATA)
  - Registo de estado (STATUS)



  - Vários registos de comando/controlo para programar as características da comunicação

Porta série: envio

- Suponha-se a existência das duas seguintes funções C que são equivalentes às instruções máquina IN e OUT:

```
unsigned char in(unsigned short port);
void out(unsigned int port,unsigned char byte);
```

- Para enviar um byte pela porta série teremos:

```
void send_serial( unsigned char b )
{ unsigned char s;
  do {
      s = in(0x3fd); /* polling STATUS*/
  } while ((s & 0x20) == 0);

   out(0x3f8, b);
}
```


Porta série: recepção

- Para receber um byte teremos:

```
unsigned char receive_serial( )
{ unsigned char s;

  do {
      s = in(0x3fd);   /* polling STATUS */
  } while ((s & 0x01) == 0);

  return in(0x3f8);
}
```

Exemplos em assembly

```
            mov $0x3fd, %dx
esperaOUT:  in %dx, %al        # lê o RegEstado para o CPU (reg AL)
            and $0x20, %al     # testar bit 5 (TXReady)
            jz esperaOUT
            movb (byte), %al
            mov $0x3f8, %dx
            out  %al, %dx      # envia o carácter para TXDados
```

```
            mov $0x3fd, %dx
esperaIN:   in %dx, %al        # lê o RegEstado para o CPU (reg AL)
            and $1, %al        # testar bit 0 (RXReady)
           jz esperaIN
           mov 0x3f8, %dx
           in %dx, %al         # lê o carácter do RXDados
          movb %al, (byte)
```


Inconvenientes do uso da espera activa

- Os dispositivos de E/S são muito lentos e o CPU vai passar grande parte do tempo nos ciclos de espera
  - Os controladores que só suportam espera activa são muito simples, mas implicam um grande desperdício de tempo de CPU
- Enquanto se espera há potencial para o CPU executar muitas instruções
- Se o CPU executa outras operações o periférico pode entretanto ficar disponível e logo, desocupado → **desperdício do periférico**
- Se o periférico recebe dados e o CPU (programa) não está pronto a recebê-los → **perda de dados**
- Mau para o controlo "simultâneo" de múltiplos periféricos

Objetivo: I/O sem espera activa

- Um programa executa código no CPU que requer uma operação num controlador
- O CPU continua a executar código do programa ou de outro programa
- Quando a notificação chega, o CPU pode executar o código para a próxima operação de I/O

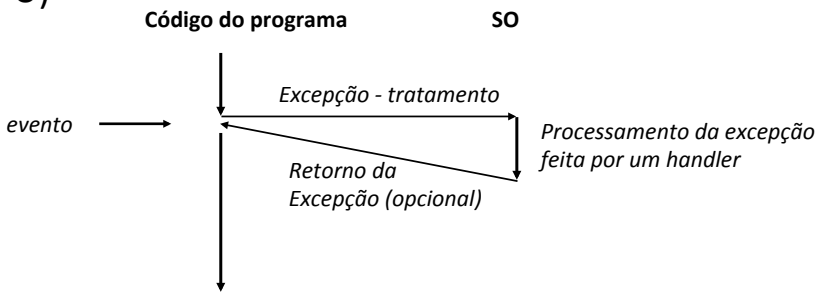
- O controlador recebe o pedido e fica a efetuar a operação
- O controlador quando terminar a operação pedida pelo programa, **notifica** o CPU

Alteração do fluxo de controlo

- Até agora três mecanismos para quebrar o fluxo sequencial de instruções :
  - Saltos (incondicionais e condicionais)
  - Call e return, que usam o stack.
  - INT (visto brevemente)
- É difícil para o CPU responder a alterações exteriores ao processo corrente.
  - Dados que chegam do disco ou da rede
  - Instrução faz divisão por zero
  - Utilizador carega em ctl-c no teclado
  - O relógio hardware dá um impulso
- São precisos mecanismos que forcem alterações do fluxo de controlo para atender eventos exteriores ao CPU

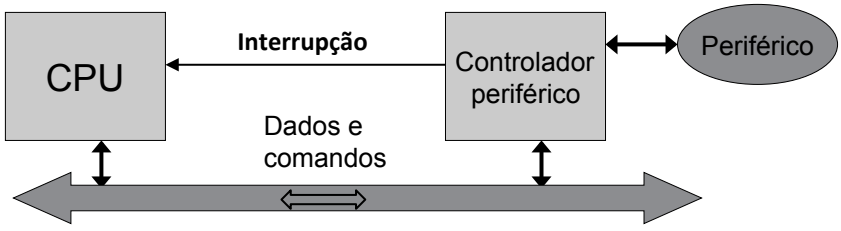
Excepções e Eventos

- Tranferir o controlo (p.e. para o SO) em resposta a um evento ou excepção (exemplos: mudança no estado de um periférico; anomalia na execução no CPU)



Notificação de eventos por interrupções

- Perante uma alteração no estado do periférico, o controlador notifica o CPU, interrompendo a sua normal execução
- O CPU passa a executar o código indicado para o tratamento desse evento



Exemplo para a Porta Série …

- Se o controlador enviar um sinal de cada vez que chega um *byte*:
  - não é necessário ficar em espera activa, testando se já chegou um *byte*
  - mas temos de indicar a rotina para o tratamento desse evento, com o código para ler do controlador o *byte* chegado
  - *e o que fazer com os dados recebidos?*
    - *guarda num buffer de dados recebidos, para tratamento posterior*

O mecanismo de interrupções

- A invenção do mecanismo de interrupções (~1960) permitiu resolver a questão da desadequação das velocidades do CPU e dos periféricos e tratar eventos excepcionais
- Permite que o CPU continue a efetuar computações enquanto decorrem transferências de dados, ou estar atento a vários periféricos
- Cada controlador actua autonomamente depois do CPU (programa) iniciar a operação; quando esta termina o controlador envia uma interrupção ao CPU.

I/O controlado por interrupções

- Exemplo de leitura (Entrada):
  - CPU (programa) emite um comando de leitura
  - O controlador obtém dados do periférico enquanto o CPU faz outra coisa
  - O controlador interrompe o CPU (notifica-o)
  - O CPU executado a rotina que lê os dados do controlador e pode efetuar logo novo pedido
  - O CPU retoma o que estava a executar quando foi interrompido
- Semelhante para a escrita (Saída)

Protecção do I/O

- O Sistema de Operação não pode perder o controlo dos periféricos
  - A programação de I/O e o tratamento das interrupções estão reservadas ao código do SO (tipicamente num módulo de código específico do periférico: *device driver*)
- O CPU suporta pelo menos dois modos de operação:
  - Modo utilizador vs. Modo supervisor (kernel mode or system mode)
  - Todas as instruções de I/O são privilegiadas! (IN e OUT nos Intel)
  - Um programa normal nunca execute em modo supervisor

Como implementar as Interrupções

- O controlador do periférico fica responsável por assinalar eventos ao CPU através de uma linha (Interrupt Request) :
  - estes eventos podem ser qualquer alteração no estado do periférico ou seu controlador
- Ao receber um sinal, o CPU interrompe a sequência normal de execução de instruções:
  - Guarda o **contexto de execução** do CPU
  - Comuta para modo supervisor e força um salto (como um call) para uma subrotina de tratamento do evento (do SO)
  - Terminada a subrotina, o CPU retoma o modo utilizador e o estado guardado

Contexto de execução

- Excepto pelo intervalo de tempo decorrido, o programa interrompido não se deve aperceber da ocorrência da interrupção!
- A quando do tratamento de uma interrupção o estado da computação interrompida não pode ser alterado
- O estado corrente duma computação: **estado corrente do CPU (conteúdo dos registos e flags), estado da memória**
- Que parte é guardada pelo mecanismo de interrupções e que parte tem de ser programada na rotina de tratamento?

Tratamento das Interrupções

- O *hardware* faz o mínimo essencial:
  - Guarda as *Flags e* o *Instruction Pointer (IP)*
  - O modo de execução é uma flag que é mudada para supervisor
  - Passa à execução do código para tratar o evento em causa
- Esta rotina de tratamento (ou serviço) não deve alterar o estado do programa interrompido
  - Se a rotina de tratamento vai usar registos do CPU, deve guardá-los antes de os usar e repor tudo no fim
  - Não pode alterar a memória do programa interrompido
- Os dados a enviar ou recebidos são trocados usando *buffers* em memória do SO
  - exemplo: quando chega um novo carácter, a rotina de tratamento para esta situação, põe este num buffer em vez de executar todo o programa que vai usar este carácter

Ciclo de execução do CPU

ciclo de funcionamento:
fetch (obter instrução da memória)
decode (descodifica)
execute (executa)

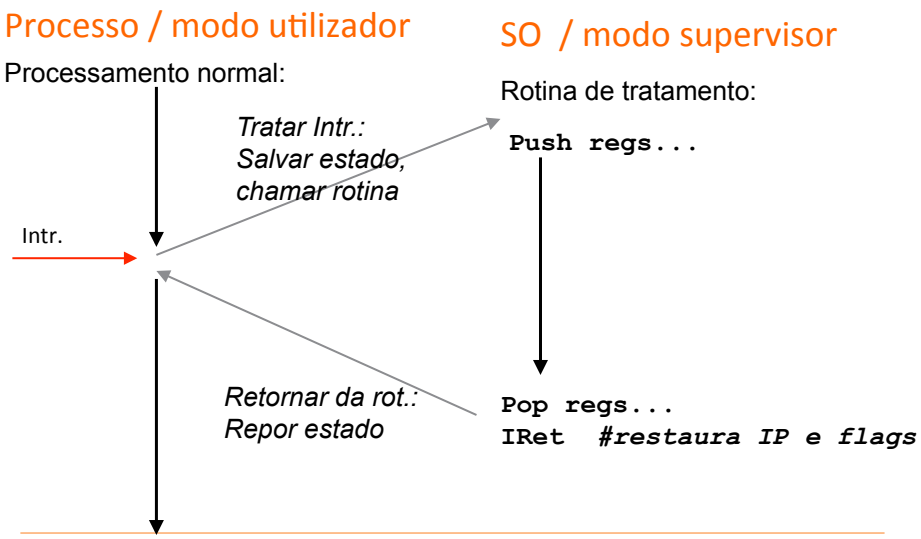
Ciclo de execução do CPU c/modo supervisor

ciclo de funcionamento:
- **fetch** (obter instrução da memória)
- **decode** (descodifica)
- if (instrução privilegiada && SupFlag==0)
  - exceção! → *interrupção*
- else
  - **execute** (executa)

Ciclo de execução do CPU c/interrupções

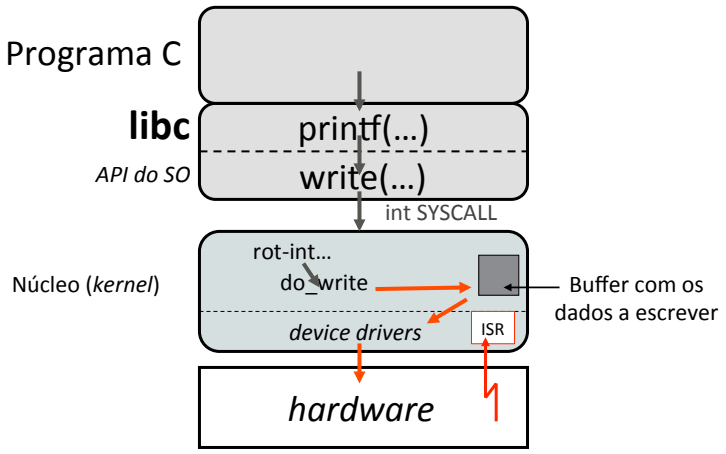
ciclo de funcionamento:
- **fetch** (obter instrução da memória)
- **decode** (descodifica)
- if (instrução privilegiada && SupFlag==0)
  - exceção! → *interrupção*
- else
  - **execute** (executa)
- if (IntFlag && IntRequest) {
  - salva flags (semelhante a push CFLAGS)
  - IntFlag ← 0 (desliga novas interrupções)
  - SupFlag ← 1 (passa a modo supervisor)
  - identifica a interrupção
  - **chama rotina de tratamento respetiva** (semelhante a um call)
- }

Tratamento de uma interrupção corresponde a uma troca de contexto



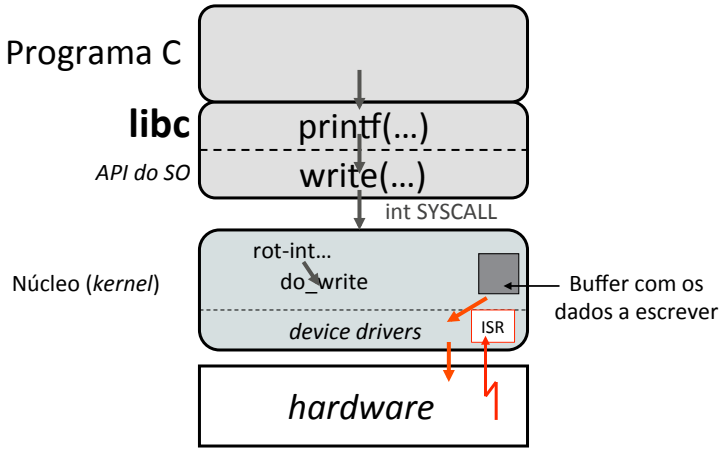
Chamadas ao sistema (I/O)

- Exemplo de saída de dados

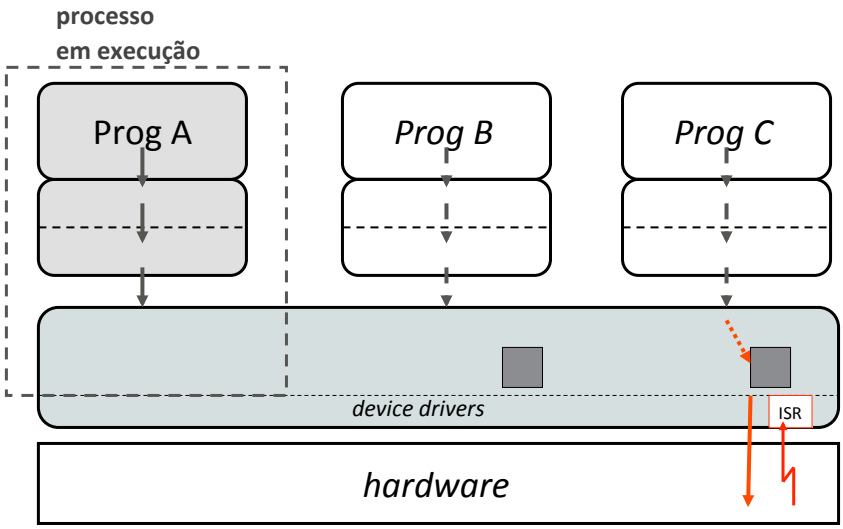


Chamadas ao sistema (I/O)

- Exemplo de saída de dados



Tratamento de saídas em simultâneo com execução de outro processo



Problemas a resolver

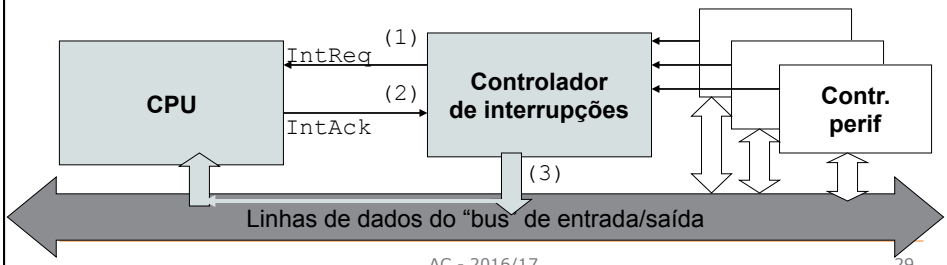
- Como identificar a interrupção?
  - Motivo da interrupção
- Como identificar a rotina a chamar?
  - Registar as rotinas para cada interrupção possível
- Como terminar a rotina de tratamento?
  - RET especial que também restaura CFLAGS
    - E logo restaura modo utilizador e interrupções
- Como lidar com múltiplas interrupções?
  - i.e. uma rotina de tratamento de interrupções pode ser, por sua vez, interrompida?

Como identificar a interrupção?

- Interrogação por *software?*
  - consulta-se o registo de estado de todos os controladores? Lento!
- Uma linha diferente para cada controlador?
  - Limita o número de controladores (e periféricos) ao número de linhas de interrupção no CPU
- Identificação por *hardware?*
  - existe um dispositivo capaz de identificar o controlador que interrompeu

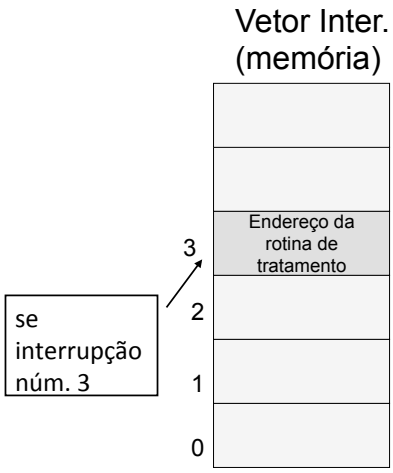
Controlador de Interrupções

- Árbitro da linha de interrupção do CPU
- Funcionamento do CPU e Controlador de Interrupções:
  1. Quando de um interrupção o CI activa a linha de interrupção do CPU
  2. CPU quando aceita a interrupção activa a linha de "interrupt acknowledge"
  3. O controlador coloca um valor no "bus" de dados; o CPU usa esse valor para identificar a interrupção e qual deve ser a rotina de tratamento a chamar



Interrupções vetorizadas

- O número da interrupção serve como índice numa tabela com endereços de rotinas: o **vetor de interrupções**
- Cada posição do vetor contém o endereço da rotina a chamar para essa interrupção



Inicialização do vetor de interrupções

- Por segurança, quando o CPU começa a funcionar o sistema de interrupções está desligado
- Os controladores só começam a lançar interrupções depois de instruídos pelo CPU (pelos programas) para o começarem a fazer
- Tipicamente é o SO que inicializa o vetor de interrupções, com rotinas suas, antes de ocorrer a 1a. interrupção

O estado do sistema de interrupções é guardado numa "flag"

- O estado do CPU contém uma "flag" Interrupt Flag (IF) que indica se as interrupções estão ou não habilitadas
- Inicialmente a IF está a 0 – interrupções não permitidas
- Há instruções máquina para ligar e desligar as interrupções. Nos Intel:
  - STI (Set Interrupt Flag) ou *Enable Interrupts* (IF ← 1);
  - CLI (Clear Interrupt Flag) ou *Disable Interrupts* (IF ← 0), não vai testar se há interrupções pendentes
  - **São naturalmente instruções privilegiadas**

Fim da rotina de tratamento de interrupções

- A rotina de tratamento de uma interrupção deve terminar executando uma instrução máquina para *Interrupt Return*
- Esta instrução
  - Restaura o valor das "flags". Isto permite:
    - A flag IF volta a abilitar as interrupções se ainda não foi feito; o bit de modo é restaurado, voltando a ser "modo utilizador"
  - Desempilha o IP, o que faz retomar a execução no ponto em que a execução "normal" foi interrompida
  - Assim, retoma a execução no ponto onde se encontrava com as "flags" da altura em ocorreu a interrupção
- Nos Intel a instrução máquina é:

  `IRET` - *Interrupt Return*

Salvaguarda/restauro do estado da computação

- O *hardware* só salva automaticamente o PC e as flags
- É a rotina de tratamento de interrupções que tem a obrigação de salvar os registos do CPU que "estraga". Alguns CPUs têm instruções máquina para empilhar / desempilhar todos os registos

```
Interrupt_handler:
            pusha      # empilha todos os regs. Gerais

            ...        # tratamento da interrupção

            popa       # restaura todos os registos
            iret       # interrupt return
```

Impedir que a rotina de tratamento de interrupções seja interrompida

- Quando se entra na rotina de tratamento, IF está a 0
- Se nada for feito, só será colocada a 1 quando do "interrupt return"
- Mas as interrupções devem estar desligadas o menor tempo possível –- podem-se perder dados que cheguem ou tornar a sua saída mais lenta.
- A rotina de tratamento deve fazer STI (IF ← 1) assim que seja seguro.