Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 16

Espaço endereçamento separado vs. único

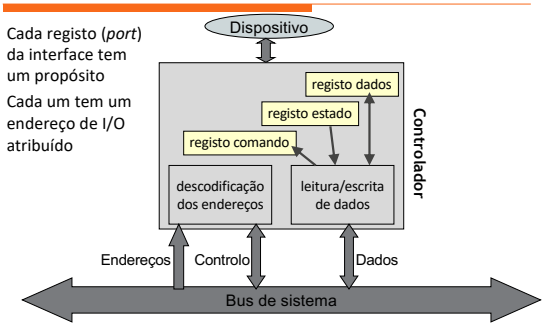
- Endereçamento separado
  - Há instruções máquina dedicadas ao acesso do espaço de endereços de entrada/saída (chamados *ports*)
  - IN, OUT
- Endereçamento único
  - Não há instruções máquina especiais para acesso aos controladores. São usadas *LOAD/STORE* (MOV) para endereços que pertencem à faixa de reservada para entrada/saída

Ponto de vista do CPU (dos programas)

- O controlador do dispositivo fornece uma interface de programação, que tem apenas duas operações:
  - ler e escrever, também chamadas entrada e saída
- Cada periférico possui um controlador com uma faixa de endereços **distinta**
- Os valores escritos em cada *port* têm significados especiais ou desencadeiam ações no controlador
  - Definem **uma linguagem** de operação/controlo específica do periférico
- Para o programador, o bus é invisível

Interface genérica de periférico

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



Controlador genérico

- Em geral, as interfaces têm três tipos de registos (portos ou portas):
  1. Porto de dados: para guardar os dados enviados pelo CPU ou recebidos do exterior pelo dispositivo periférico
  2. Porto de estado: indica informação sobre o estado corrente do periférico
     - existência de dados para o CPU ler
     - disponível para receber dados do CPU
     - erros e outras informações
  3. Porto de controlo: para receber comandos vindos do CPU (inicializar interface, configurar a interface para receber ou enviar, e outros comandos que desencadeiam a operação do periférico)

Programação dos controladores de I/O

- Cada controlador oferece uma interface de operação:
  - Uma "linguagem" específica desse hardware
- Podem ser mais ou menos autónomos em relação ao CPU
  - Controladores com pouca "autonomia" precisam de grande interação do CPU (necessita de mais software e mais complicado)
  - Controladores com grande "autonomia" oferecem operações mais elaboradas
    - Por exemplo: só necessitam que o programa mande iniciar a transferência

Ritmos de operação assincronos

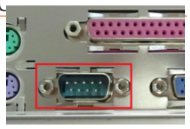
- Cada periférico trabalha ao seu ritmo, independente do CPU
  - Trabalham em paralelo com o CPU
  - São muito mais lentos do que o CPU
- O CPU (o software) tem de interatuar com o periférico para saber o seu estado. Ex:
  - se este já está disponível para receber novo comando
  - se já tem o dado pretendido
  - Etc.

Programação por espera ativa (*busy waiting* ou *polling*)

1. O CPU requer uma operação
2. O CPU interroga **repetidamente** o controlador do periférico (bits de ESTADO) para saber se a operação anterior já terminou
   - O CPU pode esperar (com grande desperdício de tempo de CPU) ou executar outro código, voltando a testar mais tarde

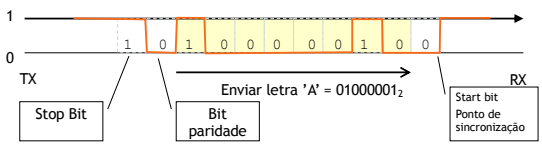
- O controlador fica a efectuar a operação
- O controlador posiciona os seus bits de estado (reg. ESTADO)
  - O controlador de I/O não informa o CPU diretamente e não interrompe o CPU

Exemplo: Porta série do PC

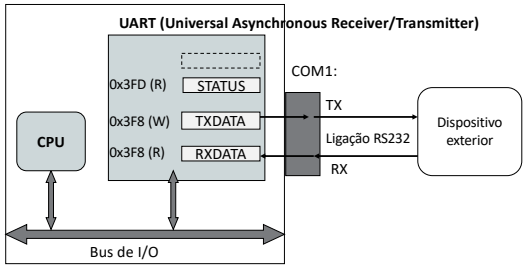


RS-232 ou V.24

- A norma inclui aspectos "físicos": fichas, cabos, tensões (Volts), etc…
- Características da comunicação:
  - Taxa de transferência, sincronização , detecção de erros, etc…
  - Exemplo simples:
    Grupo de 8 bits, 1 start bit, 1 stop bit e 1 bit paridade par

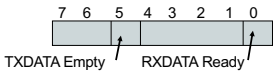


Interface do controlador (UART)



Interface da UART

- A interface de programação oferecida esconde os detalhes da comunicação série
  - Interação com o controlador orientada ao **byte**
  - 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: 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 ) {
  while ((in(0x3fd) & 0x20) == 0)
   ;                  /* polling STATUS*/
   out(0x3f8, b);
}
```


Porta série: recepção

- Para receber um byte teremos:

```
unsigned char receive_serial( )
{
  /* polling STATUS */
  while ((in(0x3fd) & 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

Testando todos os periféricos

- O SO, no limite, além de o pôr todos os programas em execução, tem de testar o estado de todos os periféricos e tratar de cada um
  - Pode perder dados, executa pouco dos processos, etc

```
Ciclo:
   se testaA  então  handleA
   se testaB  então  handleB
   …
   Executa um pouco de um processo . . .
```

Objetivo: I/O sem espera activa

- Um programa (SO) 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

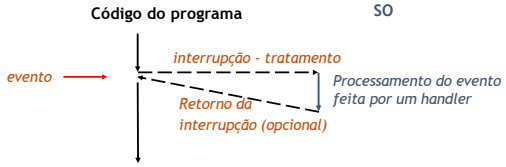
- O controlador recebe o pedido e fica a efetuar a operação
- O controlador quando terminar a operação pedida pelo programa, **notifica** o CPU (o programa)

Alteração do fluxo de execução

- Até agora três mecanismos para quebrar o fluxo sequencial de instruções :
  - Saltos (incondicionais e condicionais)
  - Call e return (usando o stack)
  - INT (visto brevemente)
- É difícil responder a alterações exteriores ao processo corrente. Exemplos:
  - Dados que chegam do disco ou da rede
  - Instrução faz divisão por zero
  - Utilizador carrega em ctl-c no teclado
  - O relógio hardware dá um impulso
- É preciso um mecanismo para alterar o fluxo de execução para atender eventos assíncronos exteriores ao programa!

Interrupções, eventos e excepções

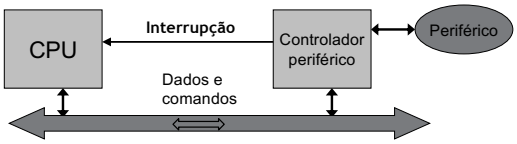
- Transferir o controlo (p.e. para código do SO) em resposta a um evento ou excepção
  - Exemplos: mudança no estado de um periférico; anomalia na execução de uma instrução; etc.



  - Semelhante à execução de **`int`** *nº*, mas pelo *hardware* e em qualquer altura

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 ativa, testando se já chegou um *byte*
  - mas temos de indicar a rotina para o tratamento desse evento: o código para ler do controlador o *byte* que chegou
  - *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 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 atua 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)

Proteção das interrupções

- O Sistema de Operação não pode perder o controlo dos periféricos
  - O tratamento das interrupções, tal como a programação de I/O, está reservada ao código do SO (tipicamente num módulo de código específico do periférico: *device driver*)
- Ao receber um sinal, o CPU interrompe a sequência normal de execução de instruções:
  - Comuta para modo supervisor e força um salto para uma subrotina de tratamento específica do evento (no SO)
    - ***INT nº***
  - Terminada a subrotina, o CPU retoma ao modo anterior e ao código anterior

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* prédefinidos 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 (inicial) de execução do CPU

```
ciclo de funcionamento:
    fetch     //obter instrução da memória
    decode    //descodifica
    execute   //executa
```


Ciclo de execução do CPU c/modo supervisor

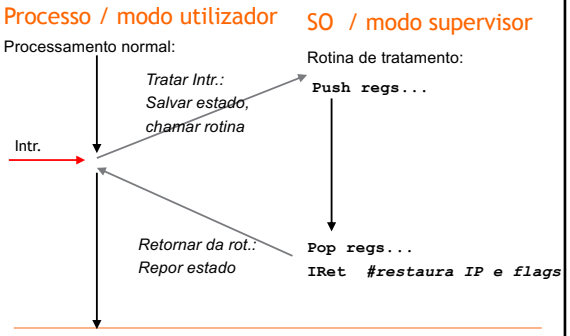
```
ciclo de funcionamento:
    fetch     //obter instrução da memória
    decode    //descodifica
    if (instrução privilegiada && SupFlag==0)
        exceção!  → interrupção
    else
        execute    //executa
```

Ciclo de execução do CPU c/interrupções

ciclo de funcionamento:

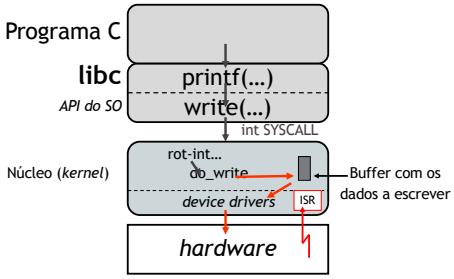
```
fetch     //obter instrução da memória
decode    //descodifica
if (instrução privilegiada && SupFlag==0)
    exceção!  → interrupção
else
    execute   //executa
 if ( IntRequest && IntFlag==1 ) {
    ID=identifica a interrupção
    execute( INT ID ) →
          salva flags  //semelhante a push EFLAGS
          IntFlag ← 0  //desliga novas interrupções
          SupFlag ← 1  //passa a modo supervisor
          chama rotina de tratamento ID
 }
```


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