Arquitetura de Computadores

MIEI – 2017/18

DI-FCT/UNL

Aula 17

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
    }
```


Tipos de interrupções/excepções

- **Externas ao CPU**: ligadas ao *hardware:*
  - Dispositivos de entrada/saída (I/O)
  - Temporizador (clock)
  - Falha no *hardware*
- **Internas ao CPU**: ligadas ao *software* (devido à instrução executada)
  - **Interrupção por software:** uma instrução que provoca deliberadamente uma interrupção: **INT**
    - Pode ser usada, p.e., para fazer chamadas ao SO
  - **Excepções** (exemplos: divisão por 0, acesso a memória não autorizada, execução de instrução privilegiada em modo utilizador, ...)

"Interrupt Flag"

- O CPU contém uma "flag" Interrupt Flag (IF) que indica se deve aceitar as interrupções externas (se estão ou não habilitadas)
  - Inicialmente a IF está a 0
- 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

Problemas a resolver

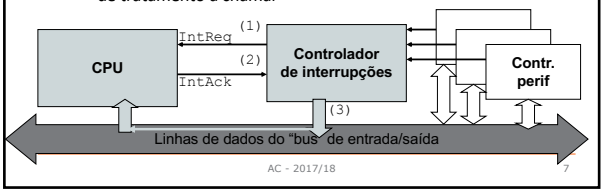
- Como identificar a interrupção?
  - Motivo da interrupção
- Como identificar a rotina a chamar?
  - Registar as rotinas para cada interrupção possível
- Como terminar a rotina de tratamento?
  - RET especial que também restaura FLAGS
    - E logo restaura modo utilizador e interrupções
- Como lidar 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

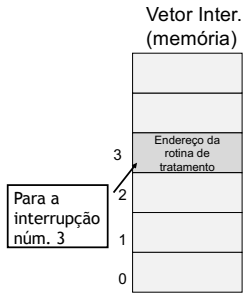
Controlador de Interrupções

- Árbitro da linha de interrupção do CPU
- Funcionamento do CPU e Controlador de Interrupções:
  1. Quando de um interrupção o CI ativa a linha de interrupção do CPU (IntReq)
  2. Se CPU aceita a interrupção, ativa a linha de "interrupt acknowledge" (IntAck)
  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



Como identificar a rotina 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 (pelo software) para o começarem a fazer
- Tipicamente é o SO que inicializa o vetor de interrupções, com as suas rotinas para tratar cada periférico
  - Se o SO não tem o código para determinado periférico (device driver) este não será configurado para lançar interrupções

Como terminar a rotina de tratamento?

- A rotina de tratamento de uma interrupção deve terminar executando uma instrução máquina para *Interrupt Return*
- Esta instrução faz o equivalente:

```
pop flags     // repõe IF e SupMode
ret
```

  - 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 IP 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
```


Como lidar com múltiplas interrupções?

- Impedir que uma 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 IRET
- Mas as necessidades dos controladores diferem:
  - Tempo máximo ao fim do qual têm de receber atenção
  - Duração do processamento da interrupção
- O processamento da interrupção de um periférico "lento" pode levar mais tempo do que o tempo máximo entre interrupções de um periférico "rápido"

Interrupções de múltiplos controladores

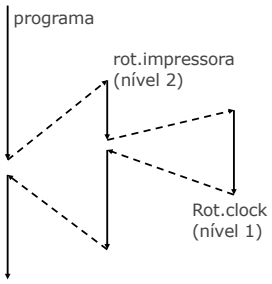
- As interrupções devem estar desligadas o menor tempo possível
  - Podem-se perder dados que cheguem ou tornar a sua saída mais lenta.
  - A rotina de tratamento deve fazer STI (IF ← 1) assim que seja seguro ser interrompida
- Para satisfazer estas diferenças são atribuídas prioridades diferentes aos controladores:
  - Interrupções mais prioritárias interrompem as rotinas de tratamento das menos prioritárias

Prioridades nas interrupções (1)

- Múltiplos níveis de interrupção ou múltiplas prioridades para as interrupções
- O CPU (e o seu controlador de interrupções) definem vários níveis para as prioridades
- A cada periférico é atribuída um nível (*N*):
  - por exemplo: *maior N* → prioridade mínima; 0 → prioridade máxima
- A rotina de atendimento de interrupções do nível *i* pode ser interrompida para executar a rotina de atendimento de um periférico de nível *j,* se *j < i*

Prioridade das interrupções (2)

- Exemplo:
  - Periférico rápido (clock) prioridade/nível 1
  - Periférico lento (impressora) prioridade/nível 2
  - Se estamos a executar a rotina de tratamento da impressora e chega uma interrupção do clock: vai-se executar a do clock.
  - Quando esta acaba voltamos à rotina da impressora; quando esta acaba volta-se ao programa que estava a correr



Recapitulando

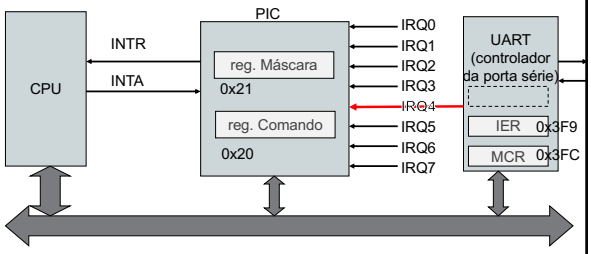
- O uso de interrupções permite sobrepor (*overlap*) a computação e a realização de entradas/saídas pelos periféricos
- Permitem adaptar a velocidade relativa de CPUs e periféricos, tirando melhor partido do CPU e dos periféricos
- Permite estar atento e tratar várias situações em simultâneo, sem ter de as verificar e sem correr o risco de as não detetar
- Um computador com interrupções permite ter melhor desempenho do que um que não tem

Exemplo – Intel nos PC

- Suporta até 255 tipos de interrupções vetorizadas
  - o vetor de endereços está na memória
- O CPU tem uma linha de interrupções gerais externas
  - a flag IF é alterada pelas instruções STI e CLI
- Usa um controlador de interrupções programável -- PIC (8259A ou equivalente)
  - Define as prioridades entre interrupções
  - Identifica o tipo de cada interrupção (nº), depois usado como índice no vetor de interrupções
- Suporta instrução de interrupção por software: INT nº
- O retorno da rotina de tratamento é feito pela instrução IRET

Interrupções na Porta Série 1

- COM1: tem atribuído o IRQ4, com tipo nº 12



Interrupções na Porta Série 1

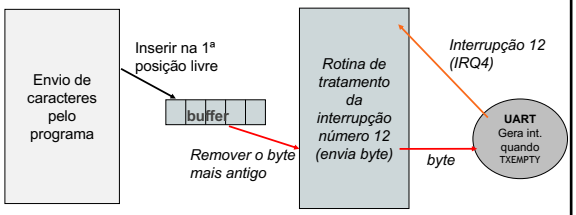
- Programação da UART para gerar interrupções:
  - Ligar interrupções no MCR (0x3FC):
    - Colocar a 1 o bit 3 do registo
  - Programar quais as interrupções pretendidas no registo IER (*Interrupt Enable Register*)(0x3F9):
    - quando chega um carácter (RXREADY): Colocar a 1 o bit 0
    - quando o registo de transmissão fica vazio (TXEMPTY) ou seja, pronto a enviar: Colocar a 1 o bit 1

Operação por interrupções

- Inicia-se o vetor de interrupções, o PIC e o UART
  - A porta série 1 tem associada a interrupção nº 12
- Quando existe uma transferência para fazer, o controlador da porta série passa a indicar (por interrupção) quando se pode fazer a leitura/escrita nos registos de dados
- É a rotina de tratamento da interrupção que escreve (OUT) no porto de Dados TX ou lê (IN) do porto de Dados RX
- A rotina usa um **buffer** como origem ou destino do próximo byte a enviar/recebido
- O programa principal usa o mesmo buffer para enviar/receber os dados
  - **Buffer** é o meio de comunicação entre programa principal e a rotina das interrupções

Transmissão com interrupções

- Em qualquer instante o *buffer* pode ser manipulado pelo programa ou pela rotina de atendimento → problemas?



Transmissão com interrupções

```
send_serial(ch){
   while bufFull()
       esperar ou liberta CPU;
   bufPut(ch);
   if inter. desligadas
       ligar inter. TXEMPTY
}
```

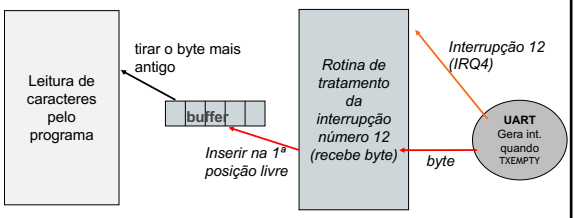
```
Rotina de tratamento de interrupções TXEMPTY{
   ch = bufGet();
   outByte( TXDATA, ch );
   if bufEmpty()
      desligar inter. TXEMPTY
   outByte(PIC_COMMAND, EOI);
}
```

Notas:

- A rotina send_serial necessita de espaço no "buffer" (esta situação não deve ser comum)
- A rotina send_serial verifica se as interrupções necessárias estão desligadas; se tal acontecer liga as interrupções na UART
- A rotina de interrupções desliga as interrupções da UART se o "buffer" ficar vazio

Recepção com interrupções

- Em qualquer instante o *buffer* pode ser manipulado pelo programa ou pela rotina de atendimento → problemas?



Recepção com interrupções

```
unsigned char recv_serial(){
   while bufEmpty()
       esperar ou libertar CPU;
   ch = bufGet();
   return ch;
}
```

```
Rotina de tratamento de interrupções RXREADY{
   ch = inByte( RXDATA );
   if bufFull()
      /*Perde o byte!
        Não há espaço no buffer mas
        não se pode esperar … */
   else bufPut(ch);

   outByte( PIC_COMMAND, EOI)
}
```

Notas:

- a rotina recv_serial espera (com as interrupções ligadas) que haja um elemento no "buffer"
- A rotina de interrupções pode encontrar o " buffer" cheio; se tal acontece não pode ficar em espera ativa (porquê ?)… perde-se o byte que chegou…

Implementação do Buffer de I/O

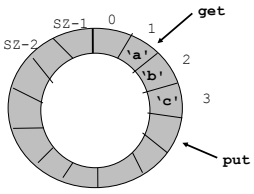
- Características:
  - Zona de memória acessível ao programa principal e à rotina de tratamento da interrupção
    - Pode ser indiretamente (p.ex. o programa recorre a chamadas ao sistema)
  - Capacidade suficiente para que o programa e o periférico trabalhem dessincronizados
    - Não perder os dados que "entram" do periférico
    - Manter o periférico ocupado com as "saídas"
  - Disciplina FIFO (First-In-First-Out)
  - Operações de pôr e tirar: bufPut() e bufGet(); testar buffer vazio/cheio: bufEmpty() e bufFull()

"Buffer" circular - exemplo

```
unsigned char buffer[SZ];
int put = 0;  // 1ª casa livre
int get = 0;  // casa ocupada há mais tempo
int nc = 0;   // nº de bytes no buffer

void bufPut( unsigned char c)
{ /* assume que buf não está cheio */
  buffer[put] = c;
  put = (put + 1) % SZ;
  nc ++;
}

unsigned char bufGet(void)
{ /* assume que buf não está vazio */
  unsigned char x = buffer[get];
  get = (get + 1) % SZ;
  nc --;
  return x;
}
```




```
int bufFull()
{ return (nc == SZ);
}

int bufEmpty()
{ return (nc == 0);
}
```


Possível problema: Actualização do nº de bytes no "buffer" (1)

- bufPut incrementa o número de bytes no buffer: *nc ++*
  - Suponhamos que o compilador traduz para

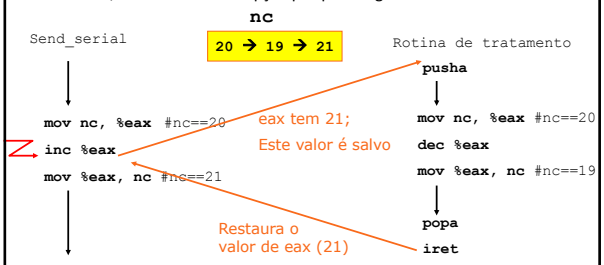
```
mov nc, %eax
inc %eax
mov %eax, nc
```

- A **rotina de atendimento de interrupções** usa bufGet que decrementa o número de bytes no buffer: *nc --*
  - Suponhamos que o compilador traduz para

```
mov nc, %eax
dec %eax
mov %eax, nc
```


Possível problema: Actualização do nº de bytes no "buffer" (2)

- Operações **concorrentes** alterando variáveis:
- Suponhamos que ***nc*** é 20; enquanto a rotina escrever_série deposita um carácter, ocorre uma interrupção porque o registo TX ficou vazio …



Atualização do nº de bytes no "buffer" (3)

- A *nc* ficou com 21 o que está errado e vai provocar problemas no futuro para *bufEmpty()* e *bufFull()*
- Isto aconteceu porque a acção de atualização da variável foi interrompida a meio e foi chamada uma rotina que atualiza a mesma variável → ***race condition (corrida)***
- A variável *nc* é partilhada pela rotina send_serial/recv_serial e pela rotina de tratamento de interrupções. A sua actualização não pode ser interrompida – constitui o que se chama uma **secção crítica**
- Para resolver isto é preciso alterar as rotinas para que não sejam interrompidas na alteração de *nc*:
  - Por ex. usando as instruções máquina CLI e STI
  - Assim garante-se que *nc* é atualizado corretamente

Transferência de dados conduzida por interrupções

- Nas transferências de com periféricos, o CPU só está envolvido quando é realmente necessário
- Ao ritmo necessário e sem desperdício de CPU
  - Executando o código da rotina de tratamento, que faz IN/OUT, apenas quando necessário
  - Executando o código da rotina de tratamento sempre que necessário (em qualquer altura e para qualquer evento)