UNIVERSIDADE FEDERAL DE SANTA MARIA
ENGENHARIA DE COMPUTAÇÃO
DISCIPLINA DE PROJETO DE PROCESSADORES

EDUARDO CAPELLARI CULAU
NELSON ROBERTO WEIRICH JUNIOR

# MANUAL/TUTORIAL DO MICROCONTROLADOR MIPS_uC

Prof. Everton Alceu Carara

Santa Maria
2018

# Sumário

# Overview

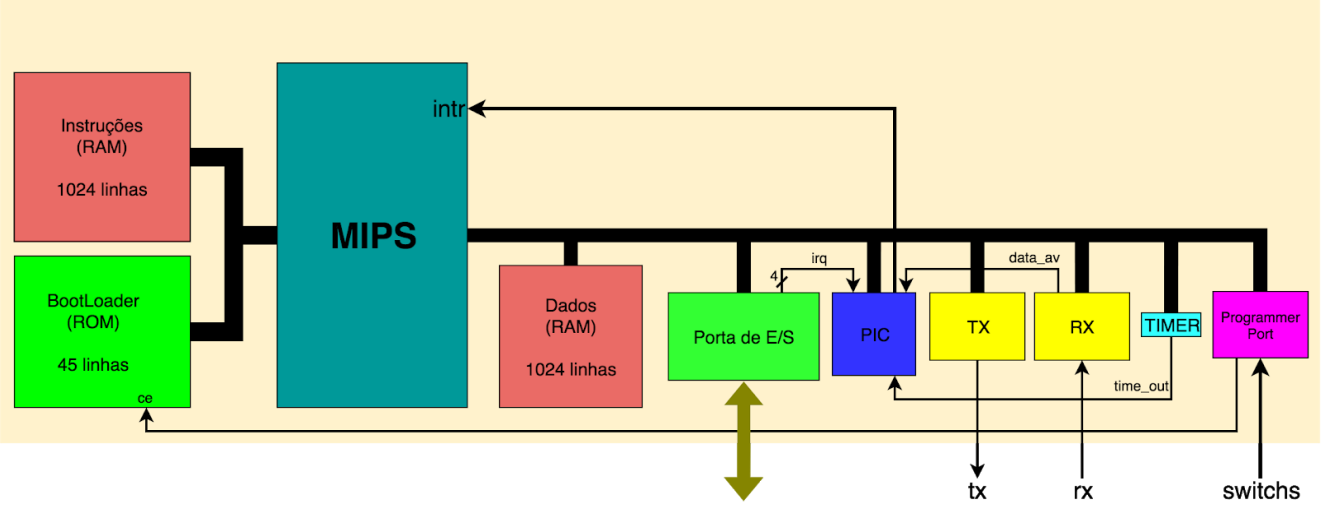
O microcontrolador de 32 bits MIPS_uC é um dispositivo implementado na disciplina de Projeto de Processadores. Ele possui uma frequência de operação de 5 MHz, 32 registradores de propósito geral, um controlador de interrupções, interrupções interna e externa, um timer, comunicação serial UART programável e uma porta de E/S.

O microcontrolador apresenta os seguintes parâmetros:

| Nome | Valor |
|---|---|
| Frequência de operação (MHz) | 5 |
| Periféricos | 6 |
| Tamanho da Memória de instruções (KB) | 4 |
| Tamanho da Memória de dados (KB) | 4 |
| Periférico de comunicação digital | 1-UART |
| Timers | 1 x 32-bit |
| Número de instruções | 40 |

## Diagrama de blocos

Na imagem abaixo temos um diagrama de blocos simplificado com todos os componentes do microcontrolador (uC). É apresentado o processador MIPS com as memórias RAM de dados e de instruções, além da memória somente para leitura, que armazena o bootloader. Os outros são periféricos que podem ser acessados pelo processador e o programmer port cuja função é permitir mudar o uC para o modo de programação por meios externos.



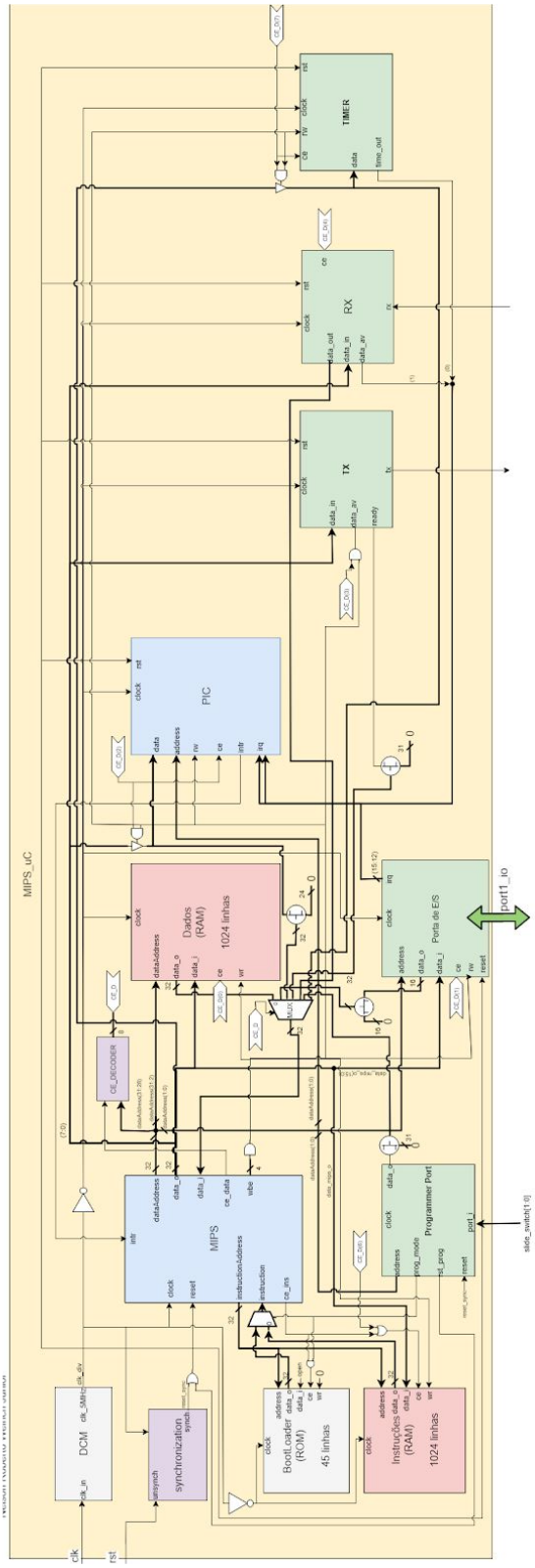
## Mapa de memória

Abaixo temos uma tabela com os endereços os quais o processador deve acessar para se comunicar com cada periférico (mais adiante será explicado e exemplificado cada um deles). Quando o processador realizar um load/store nesses endereços, ele estará salvando/carregando o dado nos periféricos, ao invés da memória.

| Endereço | Nome do periférico |
|---|---|
| 0x00000000 | Memória de Dados |
| 0x10000000 | Bidirectional PortIO |
| 0x10000000 | PortData |
| 0x10000001 | PortConfig |
| 0x10000002 | PortEnable |
| 0x10000003 | PortIrqEnable |
| 0x20000000 | PIC |
| 0x20000000 | Interrupt request |
| 0x20000001 | Interrupt Acknowledgment |
| 0x20000002 | Interrupt mask |
| 0x30000000 | TX |
| 0x30000000 | TX_data |
| 0x30000001 | RATE_FREQ_BAUD |
| 0x40000000 | RX |
| 0x40000000 | RX_data |
| 0x40000001 | RATE_FREQ_BAUD |
| 0x50000000 | Programmer PortIO |
| 0x50000000 | PortData |
| 0x60000000 | Memória de Instruções (escrita) |
| 0x70000000 | Timer |
| 0x70000000 | Counter |

# Diagrama detalhado

Segue abaixo o diagrama detalhado do uC. Para maiores detalhes verificar a imagem do diagrama ou o vetor que seguem junto com o manual.



# Periféricos

Nesta seção iremos explicar e exemplificar a utilização de cada um dos periféricos do uC. Abaixo temos uma tabela com os endereços de cada um e uma breve descrição da sua funcionalidade.

| Nome do periférico | Endereço do periférico | Funcionalidade |
|---|---|---|
| Bidirectional PortIO | 0x10000000 | entrada e saída de dados. |
| PIC | 0x20000000 | controlar interrupções externas. |
| TX | 0x30000000 | enviar dados via comunicação UART. |
| RX | 0x40000000 | receber dados via comunicação UART. |
| Programmer PortIO | 0x50000000 | controlar a programação das memórias. |
| Timer | 0x70000000 | contar um intervalo de tempo. |

## Bidirectional PortIO

Periférico responsável por fazer a comunicação externa, enviando ou recebendo dados de forma paralela. Contém 16 pinos, resultando em 16 bits que podem ser habilitados e configurados como entrada e saída, individualmente. Além disso pode ser configurado os 4 bits mais significativos da porta de forma a gerar uma interrupção no processador quando a conexão da porta ficar em nível alto (verificar o funcionamento do PIC).

### Registradores

| Nome do registrador | Endereço do registrador | Funcionalidade |
|---|---|---|
| PortData | 0x10000000 | armazenar dados recebidos pela porta. |
| PortConfig | 0x10000001 | armazenar a configuração do pinos da porta.<br>I : entrada<br>0 : saída |
| PortEnable | 0x10000002 | armazenar o estado de habilitação da porta. |
| PortIrqEnable | 0x10000003 | armazenar o estado de habilitação da interrupção. |

### Exemplo de acesso

Abaixo temos um exemplo de utilização da porta. Foi configurado para que a parte baixa seja saída e a parte alta seja entrada. Todos os bits foram habilitados, além disso o último bit foi configurado para gerar interrupção, caso fique em nível lógico alto. É importante notar que, ao salvar no registrador de dados, somente os bits configurados como saída recebem o valor, logo na última linha somente a parte baixa será colocada nos pinos externos.

```
li      $t0, 0x10000000  #Endereço base da porta.
li      $t1, 0xff00      #Bits[15:8] em 1 e Bits[7:0] em 0.
sw      $t1, 1 ($t0)     #Configurado a parte alta como entrada e
                         # a parte baixa como saída.
li      $t2, 0xffff      #Bits[15:0] em 1.
sw      $t2, 2 ($t0)     #Habilitando todos os bits da porta.
li      $t3, 0x8000      #Bit[15] = 1.
sw      $t3, 3 ($t0)     #Habilitado interrupção para o Bit[15].
li      $t4, 0xFFE8      #Carrega valor a ser colocado na porta.
sw      $t4, 0 ($t0)     #PortIO Bits[7:0] = 0xE8.
```

# PIC

Periférico responsável por controlar as interrupções externas do microcontrolador. O PIC tem 8 bits conectados nos periféricos que geram interrupção, sendo o bit 0 de maior prioridade e o 7 de menor. É necessário configurar uma mascará que indicará qual bits devem ser verificados, ou seja, qual periférico deve ser verificado e assim interromper o processador. O Timer é o que tem maior prioridade dentre todos os periféricos e Porta a menor. Os periféricos estão conectados da seguinte forma:

| Índice do bit | Conexão |
|---|---|
| 0 | Timer |
| 1 | RX |
| 4 | Port[12] |
| 5 | Port[13] |
| 6 | Port[14] |
| 7 | Port[15] |

Ao gerar uma  interrupção o processador irá saltar para o endereço especificado pelo registrador ISR_AD. Devemos ler do PIC qual interrupção foi requisitada, o qual disponibilizará o índice do bit que gerou a interrupção. Após tratar a interrupção devemos informar ao PIC qual a interrupção que foi tratada, enviando o valor lido de volta. Assim ele pode verificar outros periféricos de menor prioridade, caso requisitem interrupção.

## Registradores

| Nome do registrador | Endereço do registrador | Funcionalidade |
|---|---|---|
| Interrupt request | 0x20000000 | armazenar o bit de interrupção requisitada. |
| Interrupt ack | 0x20000001 | recebe a interrupção tratada pelo processador. |
| Interrupt mask | 0x20000002 | armazenar a máscara dos bits de interrupção. |

## Exemplo de acesso

Inicialmente devemos configurar a máscara, assim colocando 1 no bit que devemos ativar do PIC, sempre seguindo os bits correspondente dos periféricos. Após isso, podemos ficar processando qualquer outra coisa. Quando ocorrer uma interrupção o processador irá saltar para o endereço configurado no registrador ISR_AD. Neste endereço devemos tratar todas as interrupções. Primeiramente temos de salvar o contexto para não sobrescrever os

registradores que estavam sendo usados, depois lemos o *Interrupt request* do PIC para saber qual o ID da interrupção. Baseado no ID, tratamos a interrupção e ao final enviamos o ID para o endereço *Interrupt ack*, assim avisando o mesmo que aquela interrupção foi tratada. Ao final deve-se recuperar o contexto e executar a instrução '*eret*' para assim retornar ao ponto que estava sendo executado antes de ser interrompido.

```
li      $t0, 0x20000000  #Endereço base do PIC.
li      $t1, 0x13        #Bit[0] = 1 , Bit[1] = 1, Bit[4] = 1.
sw      $t1, 2 ($t0)     #PIC só processará interrupções do
                         # Timer, RX e Port[4].
#Aguardar Interrupção.
ISR_ADDR:
#Salvar o contexto na pilha.
li      $t0, 0x20000000  #Endereço base do PIC.
lw      $t1, 0 ($t0)     #Ler qual o ID da interrupção do PIC.
#Verificar e tratar a interrupção.
sw      $t1, 1 ($t0)     #Enviar o ACK para o PIC, sendo o mesmo valor lido.
#Recuperar o contexto da pilha.
eret                     #Retorna para o ponto que estava sendo executado.
```

# TX

Periférico responsável por enviar dados via comunicação serial UART. Por usar comunicação serial devemos configurar a frequência na qual o dado será enviado. O valor a ser salvo no registrador não é em bauds, mas sim uma relação entre bauds e o clock do periférico. A fórmula abaixo resulta no valor a ser salvo no RATE_FREQ_BAUD, tendo como entrada bauds.

$$RATE\_FREQ\_BAUD = \frac{5\,M}{Baud\ Rate}$$

Para enviar um dado devemos sempre verificar se o TX está pronto para receber um dado, ou seja, ele já enviou o último valor. Após isso simplesmente enviamos um caractere (8 bits) para o mesmo.

## Registradores

| Nome do registrador | Endereço do registrador | Funcionalidade |
|---|---|---|
| TX_data | 0x30000000 | armazenar o dado a ser transmitido. |
| RATE_FREQ_BAUD | 0x30000001 | armazenar o valor da relação frequência - taxa de bits de comunicação. |

## Exemplo de acesso

Primeiramente devemos configurar a frequência. Aplicando a fórmula anterior para o valor de 9600 baud, chega-se no valor de 520. Para configurar, simplesmente enviamos esse valor para o periférico. Ao utilizá-lo devemos ler o valor no endereço do periférico e verificar o mesmo. Caso o valor lido seja 0, quer dizer que o TX ainda não está pronto, se for 1 ele está pronto e pode-se enviar um novo dado para ele.

```
li    $t0, 0x30000000     #Endereço base do TX.
li    $t1, 520            #Valor para 9600 Bauds.
sw    $t1, 1 ($t0)        #Configura a frequência do TX.
readyTX:
lw    $t1, 0 ($t0)        #Lê o retorno do TX.
beq   $t1, $zero, readyTX #Espera até o TX estar pronto.
li    $t1, 0x41           #Caractere 'A'.
sw    $t1, 0 ($t0)        #Envia para o TX o 'A'.
```

# RX

Periférico responsável por receber dados via comunicação serial UART. Por usar comunicação serial devemos configurar a frequência na qual será recebido o dado. O valor a ser salvo no registrador não é em bauds, mas sim uma relação entre bauds e o clock do periférico. A fórmula abaixo resulta no valor a ser salvo no RATE_FREQ_BAUD, tendo como entrada bauds.

$$RATE\_FREQ\_BAUD = \frac{5\,M}{Baud\ Rate}$$

O RX por receber dados fica a maior parte do tempo sem fazer nada, logo só deve-se pegar o dado quando o mesmo chega. Por esse motivo o RX gerar uma interrupção para o processador quando um dado estiver pronto, assim o processador só verifica o RX quando o mesmo tiver um dado disponível. Para isso, deve-se configurar o PIC adequadamente para que o RX possa interromper o processador. Algo que deve-se cuidar ao enviar dados externos para o RX é a frequência de envio, pois caso não dê tempo do processador ler o dado do RX, antes de chegar o próximo, o caractere (8 bits) atual será perdido.

## Registradores

| Nome do registrador | Endereço do registrador | Funcionalidade |
|---|---|---|
| RX_data | 0x40000000 | armazenar o dado recebido. |
| RATE_FREQ_BAUD | 0x40000001 | armazenar o valor da relação frequência - taxa de bits de comunicação. |

## Exemplo de acesso

Primeiramente devemos configurar a frequência. Aplicando a fórmula anterior para o valor de 9600 baud, chega-se no valor de 520. Para configurar, simplesmente enviamos esse valor para o periférico. Quando um novo dado chegar o RX irá gerar uma interrupção, portanto no tratamento da interrupção deve-se ter um tratamento para o RX. Para ler o caractere, simplesmente deve-se carregar o valor no endereço do periférico.

```
li    $t0, 0x40000000   #Endereço base do RX.
li    $t1, 520          #Valor para 9600 Bauds.
sw    $t1, 1 ($t0)      #Configura a frequência do RX.
#Aguardar chegar um char e o RX gerar uma intr.
RX_Handler:
li    $t0, 0x40000000   #Endereço base do RX.
lw    $t1, 0 ($t0)      #Lê o char do RX.
#Processa o caractere lido.
jr    $ra               #Retorna para poder sair da intr.
```

# Programmer PortIO

Periférico destinado a armazenar os bits de controle do estado de programação recebidos dos slide switches. Utilizado principalmente pelo bootloader, porém o usuário pode utilizá-lo para pegar o valor do switch SW0 (o mais à direita) da placa.

## Registradores

| Nome do registrador | Endereço do registrador | Funcionalidade |
|---|---|---|
| PortData | 0x50000000 | armazenar os bits de controle do estado de programação. |

## Exemplo de acesso

Simplesmente carregue o valor no endereço correspondente.

```
li    $t0, 0x50000000   #Endereço base da porta.
lw    $t1, 0 ($t0)      #Ler o valor do switch.
#Processar esse dado.
```

# Timer

Periférico utilizado para marcação de tempo em uma aplicação. O Timer recebe um valor de até 32 bits e o decrementa, assim quando o valor chegar em 0 ele irá gerar uma interrupção, avisando que já se passou aquele tempo. Para isso deve-se configurar o PIC adequadamente para que o Timer possa interromper o processador. Quando o valor chegar em zero, o timer não contará novamente, logo deve recolocar o valor no mesmo. O valor que deve ser colocado segue a fórmula abaixo. Caso não queira mais usar o timer, deve-se mascarar a interrupção correspondente à ele no PIC, pois o mesmo fica interrompendo o processador o tempo todo, por estar em zero sempre.

$$Counter = 5K \times Tempo(ms)$$

Algumas vezes é necessário saber dois tempos diferentes, porém só temos um timer, logo uma dica de implementação é achar um tempo múltiplo entre os dois. Com isso pode-se criar um contador que a cada interrupção incrementa uma unidade. Assim podendo saber que contou X unidade de tempo quando o contador chegar no valor Y.

## Registradores

| Nome do registrador | Endereço do registrador | Uso |
|---|---|---|
| Counter | 0x70000000 | armazenar o valor da contagem do timer. |

## Exemplo de acesso

Inicialmente configuramos o timer, para o tempo desejado. Quando acabar o tempo o Timer irá gerar uma interrupção, portanto no tratamento da interrupção deve-se ter um tratamento para o Timer. Sabendo que se passou o tempo configurado, processamos o que se deseja e ao final configuramos o timer para interromper novamente após aquele tempo, ou não configuramos, caso não quisermos que interrompa novamente.

```
li    $t0, 0x70000000   #Endereço base do Timer.
li    $t1, 25000        #Valor 5ms.
sw    $t1, 0 ($t0)      #Configura o Timer para 5ms.

#Aguardar o tempo passar e o Timer gerar uma intr.
Timer_Handler:
#Deu 5ms, logo faz o que deve ser feito.
li    $t0, 0x70000000   #Endereço base do Timer.
li    $t1, 25000        #Valor 5ms.
sw    $t1, 0 ($t0)      #Reconfigura o timer para interromper após 5ms.
jr    $ra               #Retorna para poder sair da intr.
```

# Registradores do co-processador

Nesta seção iremos explicar e exemplificar a utilização de cada um dos registradores do co-processador do uC. Lembrando que para acessar os mesmos usa-se as instruções para mover para o co-processador '*mtc0'* e mover a partir do coprocessador *'mfc0'.* Abaixo temos uma tabela com o ID de cada registrador para usar nessas instruções.

| **Nome do registrador** | **Número do registrador** | **Funcionalidade** |
|---|---|---|
| Cause | 13 | tipo de exceção e bits de interrupções pendentes. |
| EPC | 14 | endereço da instrução que causou a exceção. |
| ESR_AD | 30 | endereço da  Exception Service Routine. |
| ISR_AD | 31 | endereço da Interrupt Service Routine. |

## Cause

Registrador que armazena o código da exceção ocorrida. Ao gerar uma exceção o processador irá saltar para o endereço especificado pelo registrador ESR_AD.


Código de exceção

Nos bits em destaque estará o código (ID) de qual a exceção que foi gerada. As exceções e os códigos respectivos seguem na tabela.

| Código de exceção | Cause | Nome | Causa da exceção |
|---|---|---|---|
| 01 | 4 | IBE | erro de barramento na busca da instrução (invalid instruction) |
| 08 | 32 | Sys | exceção de syscall |
| 12 | 48 | Ov | exceção de overflow aritmético (ADD, ADDI, SUB) |
| 15 | 60 | DZE | exceção de divisão por zero (DIVU) |

### Exemplo de acesso

Quando uma exceção for gerada, o processador salta para o endereço salvo no registrador, logo é preciso haver um tratamento para ela nesse local. Deve-se salvar o contexto na pilha (para não perder os valores dos registradores), pegar o ID da execução no registrador cause, verificar qual é a exceção, tratá-la e retornar para o ponto que estava sendo executado anteriormente.

```
#Aguardar a Exceção.
ESR_ADDR:
#Salvar o contexto na pilha.
mfc0    $t0, $13  #Move o valor do CAUSE (ID) para o registrador ‘t0’.
#Verificar e tratar a exceção.
#Recuperar o contexto da pilha.
eret              #Retorna para o ponto que estava sendo executado.
```

# EPC

Registrador que armazena o endereço da instrução que causou a exceção ou interrupção externa. Ele é utilizado pelo processador para retornar à execução após o tratamento das interrupções, por esse motivo é altamente não recomendado alterar o valor do mesmo. Uma utilização muito interessante do EPC para nível de software é de debug, pois o mesmo irá conter o endereço da instrução que causou a exceção, assim podendo verificar e corrigir o erro. Essa verificação deve ser feita no tratamento da exceção e um modo sugerido de verificar é enviar o valor do EPC através do TX para algo externo que mostre o valor dele, assim podendo verificar o endereço.

## Exemplo de acesso

Por não ser recomendado escrever um valor nele, a utilização mais básica do mesmo é mover o valor dele para um registrador do banco de registradores. Após isso processar algo com o valor pego do EPC.

```
mfc0  $t0, $14 #Move o valor do EPC para o registrador ‘t0’.
#Processar o que deve ser feito com o valor lido do EPC.
```

# ESR_AD

Registrador que armazena o endereço da  Exception Service Routine, ou seja, o local onde será tratado as exceções do processador. Verificar o registrador Cause para informações adicionais sobre exceção.

## Exemplo de acesso

Para salvar o endereço da rotina de tratamento de exceções, basta carregar o endereço da rotina, através da instrução ‘la’ para algum registrador e depois mover o valor desse registrador para o ESR_AD.

```
la      $t0, ExceptionServiceRoutine #Carrega o end. da rotina.
mtc0    $t0,  $30                    #Salva o end. no ESR_AD.
```

# ISR_AD

Registrador que armazena o endereço da  Interruption Service Routine, ou seja, o local onde será tratado as interrupções do processador. Verificar o periférico PIC para informações adicionais sobre interrupções.

## Exemplo de acesso

Para salvar o endereço da rotina de tratamento de interrupções, basta carregar o endereço da rotina, através da instrução ‘la’ para algum registrador e depois mover o valor desse registrador para o ISR_AD.

```
la      $t0, InterruptionServiceRoutine #Carrega o end. da rotina.
mtc0    $t0,  $31                       #Salva o end. no IRS_AD.
```

# Chamadas de sistema

As chamadas de sistema são rotinas implementadas pelo kernel (sistema) que implantam comunicação com alguns periféricos ou processam dados. Elas servem para transferir o controle do código de menor privilégio para o de maior, permitindo que o anterior possa requisitar o uso de periféricos e outras funções privilegiadas. Sendo também útil para aumentar o nível de abstração (tornar mais fácil para utilizar o sistema), assim não sendo necessário implementar toda uma rotina complexa para utilizar certos periféricos. As chamadas de sistema podem ser chamadas através da instrução '*syscall'* e passando um ID pelo registrador '$v0', indicando qual chamada de sistema está sendo solicitada. Abaixo temos uma tabela com todas as chamadas, seus IDs, os argumentos e os resultados.

| Serviço | ID | Argumentos | Resultados |
|---|---|---|---|
| PrintString | 0 | $a0 = endereço da string | |
| IntegerToString | 1 | $a0 = valor inteiro;<br>$a1 = endereço da string de retorno | |
| IntegerToHexString | 2 | $a0 = valor inteiro;<br>$a1 = endereço da string de retorno | |
| ReadString | 3 | $a0 = endereço da string | 1. $v0 = 0 se não pressionar <ENTER><br>2. $v0 = número de caracteres lidos da string, caso contrário |
| StringToInteger | 4 | $a0 = endereço da string contendo números. | $v0 = valor inteiro resentado pela string. |

Ao incluir o layer de abstração '*syscallLib'* (*.include "/lib/syscallsLib.asm"*) tem-se acesso às rotinas que chamam automaticamente as chamadas de sistema, assim não tendo de ficar verificando IDs. Verificar a Estrutura de diretórios para mais informações de onde encontrar esse arquivo e o APÊNDICE A, que contém uma descrição de cada arquivo.

## PrintString

Imprime uma string, enviando-a, através do TX, para o exterior do uC, assim mostrando a string em um monitor. Devesse enviar um ponteiro (endereço) de string, sempre lembrando de indicar o final da string (‘\0’ ou 0 no último caractere da string). Verificar os layers de compatibilidade stdio para verificar um modo mais prático de realizar essa operação.

### Exemplo de utilização

Primeiramente carrega-se o endereço da string, que deve conter o ‘\0’ final para o registrador ‘a0’. Move-se para o ‘v0’ o ID, sendo 0 o da PrintString.

```
la      $a0, string  #'a0' deve conter o endereço da string.
li      $v0, 0       #'v0' deve conter o ID da printString.
syscall              #Executar a chamada de sistema.
```

## IntegerToString

Converte um inteiro em string, salvando essa string em um endereço enviado junto com o inteiro. A string, na qual foi enviado o endereço, deve conter no mínimo 12 bytes. Ao final da string é colocado o '\0' indicando que a string terminou.

### Exemplo de utilização

Primeiramente carrega-se o valor a ser convertido para o registrador 'a0' e o endereço da string no registrador 'a1'. Move-se para o 'v0' o ID, sendo 1 o da IntegerToString.

```
li      $a0, 1234567 #'a0' deve conter o número a ser convertido.
la      $a1, string  #'a1' deve conter o endereço da string.
li      $v0, 1       #'v0' deve conter o ID da IntegerToString.
syscall              #Executar a chamada de sistema.
#Agora string contém o valor convertido para string.
```

## IntegerToHexString

Converte um inteiro em string na base hexadecimal, salvando essa string em um endereço enviado junto com o inteiro. A string, na qual foi enviado o endereço, deve conter no mínimo 12 bytes. Ao final da string é colocado o '\0' indicando que a string terminou.

### Exemplo de utilização

Primeiramente carrega-se o valor a ser convertido para o registrador 'a0' e o endereço da string no registrador 'a1'. Move-se para o 'v0' o ID, sendo 2 o da IntegerToHexString.

```
li      $a0, 1234567 #'a0' deve conter o número a ser convertido.
la      $a1, string  #'a1' deve conter o endereço da string.
li      $v0, 2       #'v0' deve conter o ID da IntegerToHexString.
syscall              #Executar a chamada de sistema.
#Agora string contém o valor convertido para string na base hexadecimal.
```

## ReadString

Lê uma string recebida através do RX. Devesse enviar um ponteiro (endereço) de string, e a quantidade de bytes (caracteres) máximos a serem lidos (podendo inferior ao valor informado), levando em conta o '\0'. Nó máximo serão lidos 80 caracteres, logo a quantidade de bytes não pode ser maior que 80. O funcionamento dessa chamada de sistema é um pouco diferente, pois ela só coloca na string os caracteres quando o RX receber um <Enter>. Ela fica retornando 0 até chegar um <Enter>, então copiará para a string e retornará a quantidade de bytes lidos. Por esse motivo devesse ficar em loop (while) chamando a 'syscall' até que ela retorne algum valor diferente de 0 e, após isso, tratar a string recebida. Verificar os layers de compatibilidade stdio para encontrar um modo mais prático de realizar essa operação.

### Exemplo de utilização

Primeiramente carrega-se o endereço da string, que será colocado a string lida, no registrador 'a0'. O registrador deve receber a quantidade de bytes a serem lido, e.g. 10 bytes de dados + 1 byte para o '\0', no registrador 'a1'. Move-se para o 'v0' o ID, sendo 3 o da ReadString. Após isso devesse ficar em loop chamando a syscall até ela retornar algo diferente de 0. Somente depois disso é possível processar a string.

```
la      $a0, string  #'a0' deve conter o endereço da string.
li      $a1, 11      #'a1' deve conter a quantidade de bytes a ser lida.
Loop:
li      $v0, 3       #'v0' deve conter o ID da printString.
syscall                  #Executar a chamada de sistema.
beq     $v0, $zero, Loop #Espera até a ReadString copiar os caracteres.
#Processar a string lida.
```

## StringToInteger

Converte uma string que contém valores decimais em inteiros, retornando o valor inteiro. A string deve conter um '\0' indicando o final da mesma. O funcionamento da StringToInteger é o contrário da IntegerToString.

### Exemplo de utilização

Primeiramente carrega-se o endereço da string a ser convertida no registrador 'a0'. Move-se para o 'v0' o ID, sendo 4 o da StringToInteger.

```
la      $a0, string  #'a0' deve conter o endereço da string.
li      $v0, 4       #'v0' deve conter o ID da StringToInteger.
syscall              #Executar a chamada de sistema.
#Agora 'v0' contém o valor decimal que estava na string.
```

# Tutorial

## Estrutura de diretórios

O apêndice A apresenta uma explicação mais detalhada da estrutura dentro do diretório /asm.

- MIPS_uC/
  - bin/
    - Application_code.bin
    - Application_data.bin
  - proto/
    - MIPS_uC.bit
    - MIPS_uC.ucf
  - sim/
    - MIPS_uC_tb.vhd
    - simulate.do
    - wave.wcfg
  - src/
    - asm/
      - boot/
        - boot.asm
        - read_Timer_Boot.asm
      - bootloader/
        - bootloader.asm
      - kernel/
        - kernel.asm
        - ServiceRoutines_Handlers.asm
      - lib/
        - kernel/
          - btnHandler.asm
          - contador_display.asm
          - echoHandler.asm
          - exceptionsHandler.asm
          - macrosKernel.asm
          - print_read_conversion.asm
          - rxHandler.asm
          - timerHandler.asm
        - macrosUsr.asm
        - memoryLib.asm
        - stdio.asm
        - syscallsLib.asm
      - main.asm

- _root.asm
- VHDL/
  - MIPS_uC.vhd
  - DCM/
    - ClockManager.vhd
  - Memory/
    - Application_code.txt
    - Application_data.txt
    - bootloader.txt
    - Memory.vhd
    - Util_pkg.vhd
  - MIPS/
    - ALU.vhd
    - ControlPath.vhd
    - DataPath.vhd
    - MIPS_MultiCycle.vhd
    - MIPS_pkg.vhd
    - Register_n_bits.vhd
    - RegisterFile.vhd
  - PIC/
    - InterruptController.vhd
    - PriorityEncoder.vhd
  - PortIO/
    - BidirectionalPort.vhd
    - ProgrammerPort.vhd
  - Sync/
    - debounce.vhd
    - synchronization.vhd
  - Timer/
    - Timer.vhd
  - UART/
    - UART_RX.vhd
    - UART_TX.vhd

# Reporte da síntese

Área - Device utilization summary

```
Device utilization summary:
---------------------------
Selected Device : 6slx16csg324-3

Slice Logic Utilization:
      Number of Slice Registers:          1615  out of  18224     8%
      Number of Slice LUTs:               5661  out of   9112     62%
      Number used as Logic:               5642  out of   9112     61%
      Number used as Memory:              19    out of   2176     0%
      Number used as SRL:                 19

Slice Logic Distribution:
   Number of LUT Flip Flop pairs used:     6902
   Number with an unused Flip Flop:        5287  out of   6902    76%
   Number with an unused LUT:              1241  out of   6902    17%
   Number of fully used LUT-FF pairs:      374   out of   6902    5%
   Number of unique control sets:           77

IO Utilization:
 Number of IOs:                           22
 Number of bonded IOBs:                   22  out of  232   9%

Specific Feature Utilization:
 Number of Block RAM/FIFO:                5  out of   32    15%
   Number using Block RAM only:           5
 Number of BUFG/BUFGCTRLs:                5  out of   16    31%
 Number of DSP48A1s:                      3  out of   32    9%
 Number of PLL_ADVs:                      1  out of   2     50%
```

Frequência - Timing summary

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 108.444ns (Maximum Frequency: 9.221MHz)
   Minimum input arrival time before clock: 2.408ns
   Maximum output required time after clock: 5.123ns
   Maximum combinational path delay: No path found
```

## Configurando o terminal serial

1. Utilizando um programa emulador de terminal, configure a porta serial de acordo com as seguintes especificações (*9600*/*8-N-1)*:
    a. Speed: 9600
    b. Bits de dados: 8
    c. Bit de paridade: N (none)
    d. Bit de parada: 1
2. Com o microcontrolador gravado na placa *Nexys 3*, siga os passos abaixo para programá-lo.

## Programando o MIPS

1. Posicione o *switch* da direita (SW0) para a baixo (memória de instruções);
2. Posicione o *switch* da esquerda (SW1) para cima, entrando no modo de programação;
3. Selecione o arquivo binário da memória de instruções, no diretório /bin, para ser enviado à placa;
4. Aguarde a conclusão do envio. Ao finalizar, o LED de transferência se apagará.
5. Posicione o *switch* da direita (SW0) para cima (memória de dados);
6. Selecione o arquivo binário da memória de dados, no diretório /bin, para ser enviado à placa;
7. Aguarde a conclusão do envio.
8. Mova o *switch* da esquerda (SW1) para baixo, entrando no modo de execução. Não é necessário pressionar o *reset* e veja a seguinte tela aparecer. Se a mesma não aparecer pressione *reset* (botão central - *BTNS*);

```
----------------------------------------
        Kernel start
        Trabalho 7
----------------------------------------
Waiting...
```

## Executar uma aplicação

1. Pronto, a aplicação está rodando no microcontrolador MIPS_uC.
2. A aplicação do trabalho 7, no terminal, aparecerá como mostra a tela abaixo:
3. Ao pressionar o *count* (botão inferior- *BTNS*) o par de displays 7-seg da esquerda irão incrementar 1 unidade. Já o par de displays 7-seg da direita irão incrementar 1 unidade a cada 1 segundo.

```
Digite uma string: hello world
String: dlrow olleh

Bubble sort
Digite o tamanho do array: 2
a[0] = 2
a[1] = 1
Digite a ordem (0 crescente; !0 decrescente): 0

Array desordenado: 2 1
Array ordenado: 1 2
```

Configuração padrão:

- Frequência padrão 9600 bps
- Slide switch prog_mem (SW0)
  **0** memória de instruções
  **1** memória de dados
- Slide switch prog_mode (SW1)
  **0** execução
  **1** programa

---

# APÊNDICE A - Arquivos e funções

Para poder montar o código é necessário alterar uma opção no MARS. Para isso deve-se manter a opção de  montar todos os arquivos no diretório ( Settings->Assemble all files in directory). Tendo esta opção selecionado basta abrir o arquivo _root.asm e montá-lo, assim todo o sistema irá ser montado junto.

## boot/

| | |
|---|---|
| boot.asm | Boot principal do sistema. É um boot padrão que desativa tudo, sendo necessário o usuário incluir o boot dele no meio do código do boot principal. |
| read_Timer_Boot.asm | Para a aplicação era necessário configurar algumas coisas, por isso foi feito um boot que configurava somente o necessário e foi incluído no boot principal. |

## /bootloader

| | |
|---|---|
| bootloader.asm | Responsável por carregar as memórias de dados e instruções, ficando na memória ROM do bootloader, não sendo possível alterá-lo após programar a placa. |

## /kernel

| | |
|---|---|
| kernel.asm | Kernel do sistema, local onde o kernel é montado. Juntando as rotinas de tratamento de interrupção com alguns módulos necessários para o kernel, como as funções localizadas na print_read_conversion. |
| ServiceRoutines_Hanlders.asm | Junta as duas rotinas de tratamento, a de interrupção e a de exceção. Além disso inclui os arquivo de handlers que incluem as funções usadas para tratar as interrupções. |

## /lib/kernel

| | |
|---|---|
| btnHandler.asm | Rotina de tratamento para a interrupção gerada pela portIO, no caso o botão. |

| | |
|---|---|
| contador_display.asm | Contém funções utilizadas para implementar o contador no display 7-seg. Contém função para converter decimal para 7-seg, escrever nos display, entre outras. |
| exceptionHandler.asm | Handlers para as exceções geradas pelo processador. |
| macrosKernel.asm | Macros usadas no kernel, assim dando um nível de abstração para o kernel. |
| print_read_conversion.asm | Local onde estão implementadas as rotinas da Chamadas de sistema. |
| rxHandler.asm | Handler para o RX, tratando o recebimento de dados e os armazenando na memória. |
| timerHandler.asm | Handler para o Timer, implementando os múltiplos de tempo, controlando os displays e incrementando o contador de 1 segundo. |

/lib

| | |
|---|---|
| macrosUsr.asm | Macros usadas no usuário, assim aumentando ainda mais o nível de abstração. |
| memoryLib.asm | Biblioteca que contém rotinas com a memória, implementando por software algumas instruções que não são feitas por hardware. Exemplos são a loadByte e a storeByte. |
| stdio.asm | Layer de abstração transformando as chamadas de sistema que realizam comunicação externa em funções, logo não sendo necessário ficar em loop na syscal ReadString. |
| syscallsLib.asm | Layer de abstração, transformando chamadas de sistemas em funções, não sendo necessário lembrar os IDs delas. |

/

| | |
|---|---|
| _root.asm | Arquivo principal que une o boot com o kernel. Deve-se selecionar este arquivo ao montar o sistema. |

| | |
|---|---|
| main.asm | Programa principal, sendo obrigado conter uma função com nome main. |

## Modelo estratificado

A seguinte imagem ilustra como é feita uma chamada de sistema utilizando o kernel desenvolvido.