Insper

# *Contador com arquitetura Registrador-Memória*

Entrega Intermediaria do Projeto 1 de Design de Computadores

Autores:
Bernardo Cunha Capoferri
Francisco Pinheiro Janela
Guilherme Dantas Rameh

# 1. Introdução:

Neste documento será discutido o funcionamento e a arquitetura de um processador FPGA Cyclone V programado para funcionar como um contador. Serão discutidas as decisões tomadas para a elaboração do projeto assim como suas justificativas.

# 2. Modo de Uso:

O nosso contador possui um botão de incremento instalado no KEY 0, que contabiliza os cliques do usuário nos painéis hexadecimais, com um limite de 999.999, que caso seja alcançado, será aceso LED 9, que indica *overflow* de contagem.

É possível, também, definir um limite ao contador utilizando o botão KEY 1. Assim que pressionado, o contador entrará em modo de definição de limite, indicado pelo LED 8 presente na placa. Os valores nos displays agora indicarão os valores que estão sendo salvos para o limite. Cada valor será salvo individualmente do dígito das unidades até as centenas de milhar, basta indicar o valor de 0~9 nas chaves SW0~SW7 e pressionar KEY1 para seguir para o próximo dígito.

Caso a contagem alcance o valor estipulado, todos os LEDs de 0 a 7 serão acesos, e em caso de mais cliques no botão de incremento (KEY 0), o LED9 de overflow será aceso.

Caso um valor de limite seja configurado menor que aquele atual no contador, o próximo incremento indicará que excedeu o limite e será notificado overflow.

Se o contador tenha alcançado o limite ou indique overflow, basta pressionar o botão FPGA_RESET que o contador será reiniciado para o valor 0.

# 3. Arquitetura:

O processador do projeto é baseado na arquitetura registrador-memória, na qual um bloco de 8 registradores é utilizado em conjunto com um bloco de memória de 64 endereços para realizar as operações.

Foi escolhida esta arquitetura para utilizar a maior rapidez dos registradores para realizar operações comuns como incrementar os valores de unidade, dezena e centena do contador, assim como ter maior robustez do uso dos registradores, com um sendo reservado como retorno de sub-rotina, um para operações lógicas e alguns para guardar constantes de uso frequente como o 1 e o 0.

**Arquitetura Processador Registrador Memória**

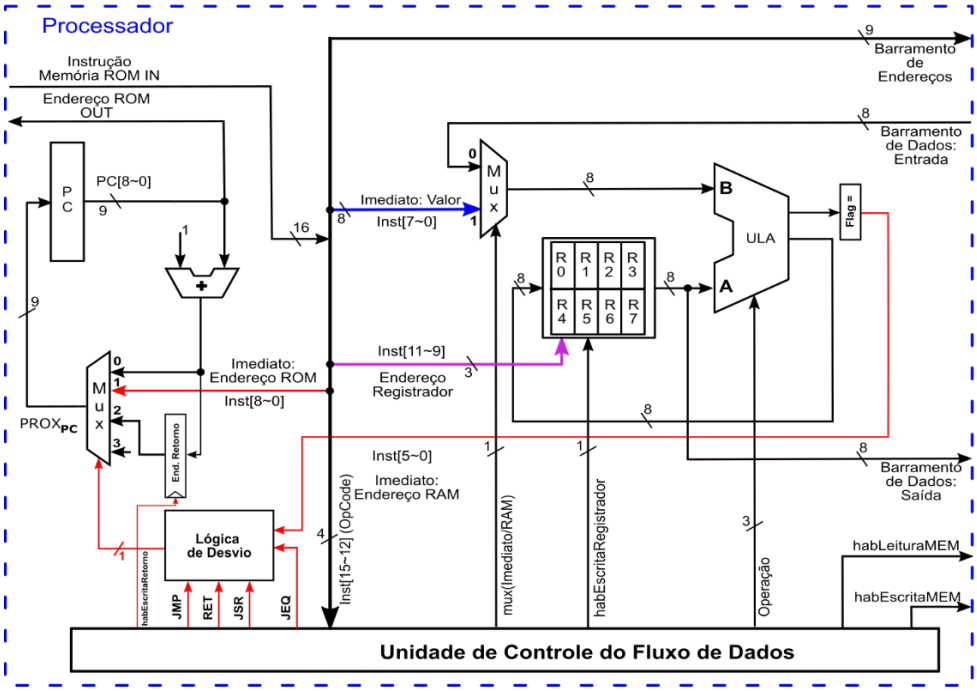



*Figura 1. Arquitetura do processador.*

# 4. Instruções:

As instruções do processador têm o seguinte formato:

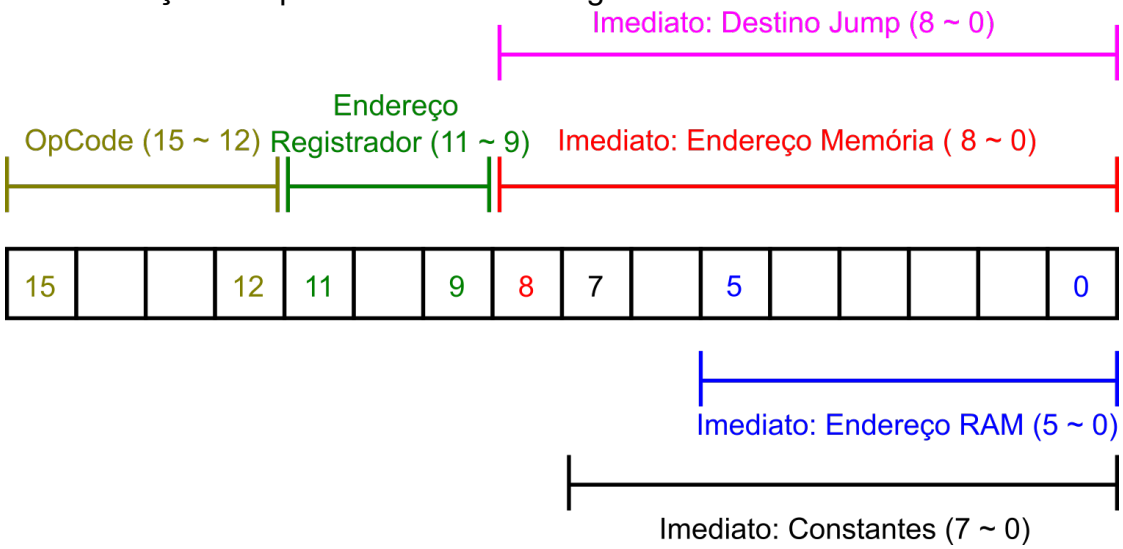



*Figura 2. Formato das instruções do processador.*

Os Mnemônicos das instruções escolhidas para o projeto são as seguintes (utilizam o espaço do OpCode):

| RESUMO MNEMÔNICOS | | |
|---|---|---|
| **Mnemônico** | **Código Binário** | **Função** |
| NOP | 0000 | Faz nada |
| LDA | 0001 | Carrega valor da memória em um registrador |
| SOMA | 0010 | Soma valor de um registrador com memória |
| SUB | 0011 | Subtrai valor de um registrador com memória |
| LDI | 0100 | Carrega constante do imediato em um registrador |
| STA | 0101 | Guarda valor de um registrador na memória |
| JMP | 0110 | Salto incondicional |
| JEQ | 0111 | Salto condicional com condição de igual |
| CEQ | 1000 | Compara se os valores de um registrador e da memória são iguais |
| JSR | 1001 | Salto incondicional de sub-rotina |
| RET | 1010 | retorna da sub-rotina |
| OPAND | 1011 | Operação lógica de AND com o imediato |

*Tabela 1. Resumo dos Mnemônicos das instruções.*

Abaixo, está a sintaxe de cada uma das instruções acima e a explicação de como usá-las.

**Nenhuma** operação:

- NOP

O mnemônico NOP indica No Operation, ou não operação. Não é seguido de argumentos.

Exemplo de uso:

```
(0) NOP
```

Operação para **carregar imediato**:

- LDI

**LDI** carrega um valor arbitrário no registrador por meio do imediato. O código de operação é seguido do registrador, R[X] selecionado e valor da constante, $Y.

Exemplo de uso:

```
(0) LDI R[2] $0
```

Operações de **manejo de memória**:

- LDA
- STA

**LDA** é a operação de carregar um dos registradores com o valor de um endereço de memória. O código de operação é seguido do registrador, R[X] selecionado e do endereço de memória, @Y.

Exemplo de uso:

```
(0) LDA R[1] @0
```

**STA** guarda o valor do registrador indicado em um endereço de memória determinado. O código de operação é seguido do registrador, R[X] selecionado e do endereço de memória, @Y.

Exemplo de uso:

```
(0) STA R[5] @352
```

Operações da **ULA:**

- SOMA
- SUB
- OPAND

**SOMA** e **SUB**, somam e subtraem os valores das entradas A e B da ULA, respectivamente, que podem ser ou o imediato ou um valor da memória na entrada B e um dos 8 registradores na entrada A. O código de operação é seguido do registrador, R[X] selecionado e do endereço de memória, @Y.

Exemplo de uso:

```
(0) SOMA R[4] @2
(1) SUB R[7] @3
```

**OPAND** operação de AND com as entradas A e B da ULA, útil para criar máscaras, como no caso quando estão sendo analisadas as entradas de apenas um bit, que ocorrer quando é verificado se um botão foi pressionado. O código de operação é seguido do registrador, R[X] selecionado e o valor da constante, $Y.

Exemplo de uso:

```
(0) OPAND R[6] $255
```

Operações de **comparação**:

- CEQ

**CEQ** é uma instrução de comparação de igualdade entre os valores A e B das entradas da ULA, definindo um registrador e um endereço de periférico de leitura. Caso sejam iguais, será salvo ‘1’ no registrador de FLAG 0, ou seja, todos os bits na operação de subtração são ‘0’.

Exemplo:

```
(0) CEQ R[0] @0
```

Operações de **desvio**:

- JMP
- JEQ
- JSR
- RET

Para as operações de salto, **JMP**, **JEQ** e **JSR** é recomendado que sejam seguidas de uma *label*, que serve como uma representação da posição de memória de programa onde foi inserida, ou seja ao indicar um salto, no código em binário, este será realizado para a linha imediatamente seguinte da *label*. Vale notar também que no código assembly ao declarar uma *label* ela deve **obrigatoriamente** ser seguida de **":"**.

Exemplo:

```
    LOOP:
(0) LDI R[0] $0
```

Neste caso um salto para "LOOP:" iria para o endereço de memória da operação de LDI (0).

**JMP** é um pulo a um endereço da memória ROM incondicional, recebe um endereço para alterar o *program counter*. No *assembly* o código de operação é seguido por uma *label*.

Exemplo de uso:

```
    INICIO:
(0) NOP
(1) JMP @INICIO
```

Neste exemplo ao chegar no JMP no endereço de memória 1, o program counter "salta" para a posição seguinte a *label* "INICIO", o endereço de memória 0.

**JEQ** é um pulo condicional determinado pelo resultado da comparação feita pela instrução **CEQ**, que, em caso de positiva, é realizado o desvio para o endereço especificado. No assembly o código de operação é seguido por uma *label*.

Exemplo de uso:

```
    CASOVERDADE:
(0) NOP
(1) CEQ R[0] @0
(2) JEQ @CASOVERDADE
```

Neste exemplo a operação CEQ verifica se os valores no registrador 0 e no endereço de memória 0 são iguais, caso sim, na função JEQ há um salto para a *label* "CASOVERDADE", caso contrário, o programa segue para a instrução seguinte.

**JSR** é um pulo incondicional para uma função de sub-rotina que salva o valor do endereço da próxima instrução em um registrador para ser usado par retornar ao laço original. Recebe uma *label* que indica onde a sub-rotina se encontra.

**RET** retorna da sub-rotina para o valor especificado na instrução JSR. Não recebe argumentos.

Exemplo de uso:

```
(0) JSR @FUNC
(1) NOP
    FUNC:
(2) NOP
(3) RET
```

Neste exemplo a instrução de JSR, leva a um salto para a subrotina “FUNC”, que ao atingir a instrução RET retorna para o endereço seguinte (1) da instrução de salto.

# 5. Fluxo de dados do processador:

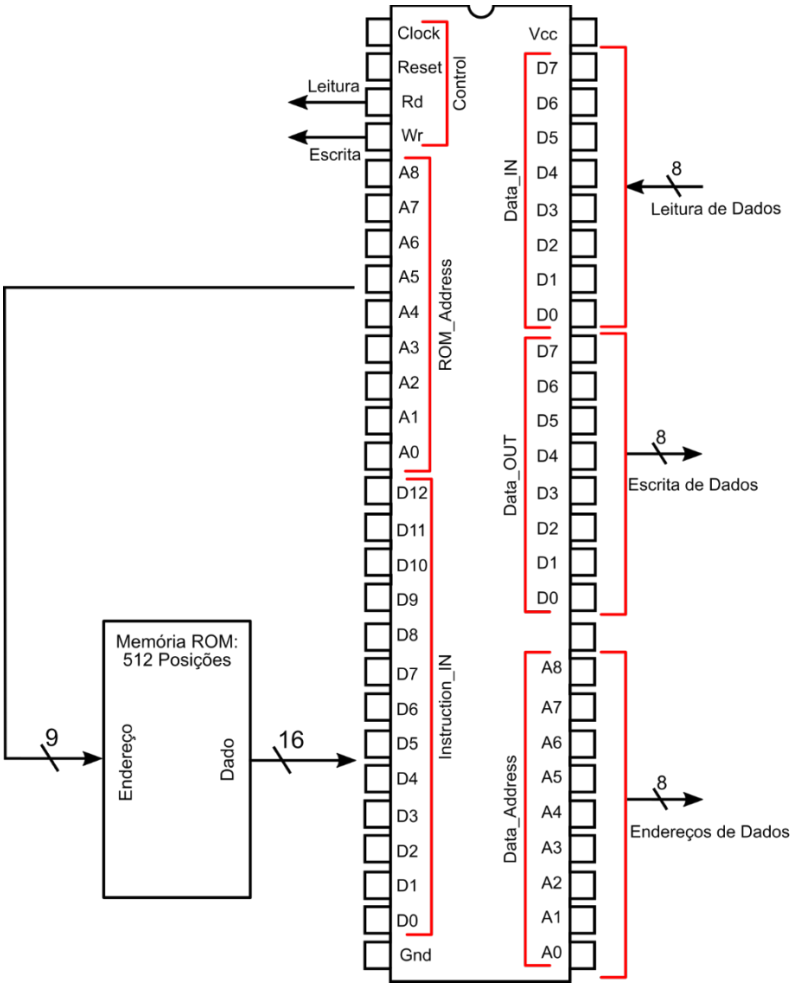



*Figura 3. Esquema do fluxo de dados do processador.*

O fluxo de dados do processador se inicia com a chegada das instruções pela entrada **Instruction_IN**, que está guardada na memória ROM com o endereço sendo determinado pela saída **ROM_Address**. Em seguida as instruções são decodificadas e processadas pelo processador em si. Ao

Caso seja necessário acessar algum endereço de memória para ler seu valor, seja da RAM, seja de algum dos periféricos, é necessário que a CPU indique isso ao habilitar o bit da saída **Leitura**, e indicar qual endereço será lido pela saída **Data_Address** e os dados propriamente ditos serão lidos pela entrada **DATA_IN.**

Quando se deseja escrever algum valor na memória ou nos periféricos, o bit da saída **Wr** será habilitado e novamente o periférico ou endereço de memória devera ser indicado a partir do **Data_Address**, com o valor a ser escrito sendo enviado pela saída **Data_OUT.**

# 6. Pontos de controle:

Nossa unidade de controle de fluxo de dados, está configurada para receber um OpCode de 4 bits, e devolver 12 bits de controle. A ordem dos pontos de controle são, do mais significativo para o menos:

- HabEscritaRetorno – Habilita escrita no registrador que salva o endereço de retorno de uma sub-rotina
- JMP – Indica para a lógica de desvio um salto não condicional sem retorno
- RET – Indica para a lógica de desvio o retorno para o endereço salvo no registrador
- JSR – Indica para a lógica de desvio um salto não condicional com retorno
- JEQ – Indica para a lógica de desvio um salto condicional sem retorno
- SelMux – Seleciona se a entrada B da ULA vem do imediato ou de um periférico
- Hab_R – Habilita Salvar um valor em um registrador no banco de registradores
- Operação ULA – Indica qual operação será realizada pela ULA
- HabFlag – Habilita salvar a FLAG de 0, ou seja, igualdade, no registrador específico
- RD – Habilita Leitura nos periféricos
- WR – Habilita Escrita nos periféricos

Abaixo está o resumo de instruções e os pontos de controle utilizados:

| Instruções e Pontos de Controle | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instrução | Mnemônico | Código Binário | Hab Escrita Retor | JMP | RET | JSR | JEQ | Sel MUX | Hab_A | Operação | habFlag = | RD | WR |
| Sem Operação | NOP | 0000 | 0 | 0 | 0 | 0 | 0 | X | 0 | XX | 0 | 0 | 0 |
| Carrega valor da memória para A | LDA | 0001 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 10 | 0 | 1 | 0 |
| Soma A e B e armazena em A | SOMA | 0010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 01 | 0 | 1 | 0 |
| Subtrai B de A e armazena em A | SUB | 0011 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 00 | 0 | 1 | 0 |
| Carrega valor imediato para A | LDI | 0100 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 10 | 0 | 0 | 0 |
| Salva valor de A para a memória | STA | 0101 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | XX | 0 | 0 | 1 |
| Desvio de execução | JMP | 0110 | 0 | 1 | 0 | 0 | 0 | X | 0 | XX | 0 | 0 | 0 |
| Desvio condicional de execução | JEQ | 0111 | 0 | 0 | 0 | 0 | 1 | X | 0 | XX | 0 | 0 | 0 |
| Comparação | CEQ | 1000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 1 | 0 |
| Chamada de Sub Rotina | JSR | 1001 | 1 | 0 | 0 | 1 | 0 | X | 0 | XX | 0 | 0 | 0 |
| Retorno de Sub Rotina | RET | 1010 | 0 | 0 | 1 | 0 | 0 | X | 0 | XX | 0 | 0 | 0 |

*Tabela 1. Pontos de controle das instruções.*

# 7. Periféricos:

Os periféricos do sistema se encaixam em dois tipos: aqueles em que a CPU realiza uma leitura e obtém um valor, e aqueles em que a CPU escreve um dado. O primeiro exemplo, é o bloco de memória RAM. Esse periférico é responsável pelo armazenamento e leitura de dados em endereços de 0 a 63, no projeto foi utilizada também para realizar operações com os registradores devido a arquitetura do processador.

A categoria de periféricos de escrita foi utilizada principalmente para visualizar os resultados das operações realizadas e verificar o comportamento correto da placa. Os LEDs da placa foram usados como indicadores, como por exemplo: quando o limite de contagem é atingido todos acendem, ao pressionar o botão 8 ou 9 os LEDs com a numeração respectiva acendem. Podem ser acessados um total de 10 LEDs, com sua numeração variando de 0 a 9.

Na categoria de leitura, também há 6 displays de 7 segmentos em hexadecimal, numerados de 0 a 5, estes foram utilizados para demonstrar o valor atual do contador em decimal, e caso esteja configurando um limite, o valor deste em dezena.

Os periféricos de leitura são os Switches (chaves) e Keys (botões). Assim como os LEDs, podem ser lidos 10 chaves numeradas de 0 a 9, estas têm como função inserirem dados em binário, por exemplo, ao gravar o limite cada valor de unidade dezena, centena, em diante. Já os botões, há 4, numerados de 0 a 3, que tem função de mudar o estado da placa, com o botão 0 incrementado o contador e o botão 1 ao ser pressionado inicia o processo de definir o limite. O último botão é especial chamado FPGA_RESET, cuja funcionalidade é reiniciar o programa da placa, resetando todos os registradores.

Os botões 0, 1 e FPGA_RESET tem uma funcionalidade a mais que é que suas leituras passam por um debouncer que salva estes valores em um flip-flop, isso é importante porque apenas o pressionamento é detectado, que será preservado, mesmo que o botão seja solto. Para limpar os valores dos flip-flops, é necessário acessar os valores de memória 511, 510 e 507 para cada botão respectivamente,

Segue abaixo três diagramas, dois mostrando as ligações do processador com os periféricos no qual escreve e lê respectivamente e uma ligação destes nos barramentos do processador:

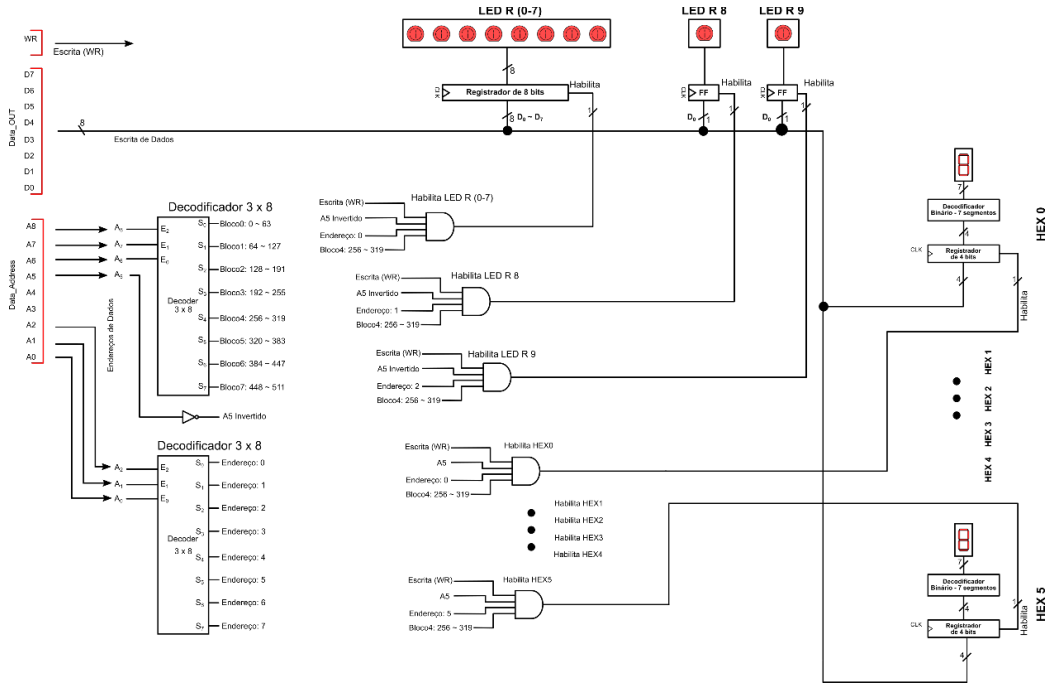



*Figura 4. Periféricos de escrita.*

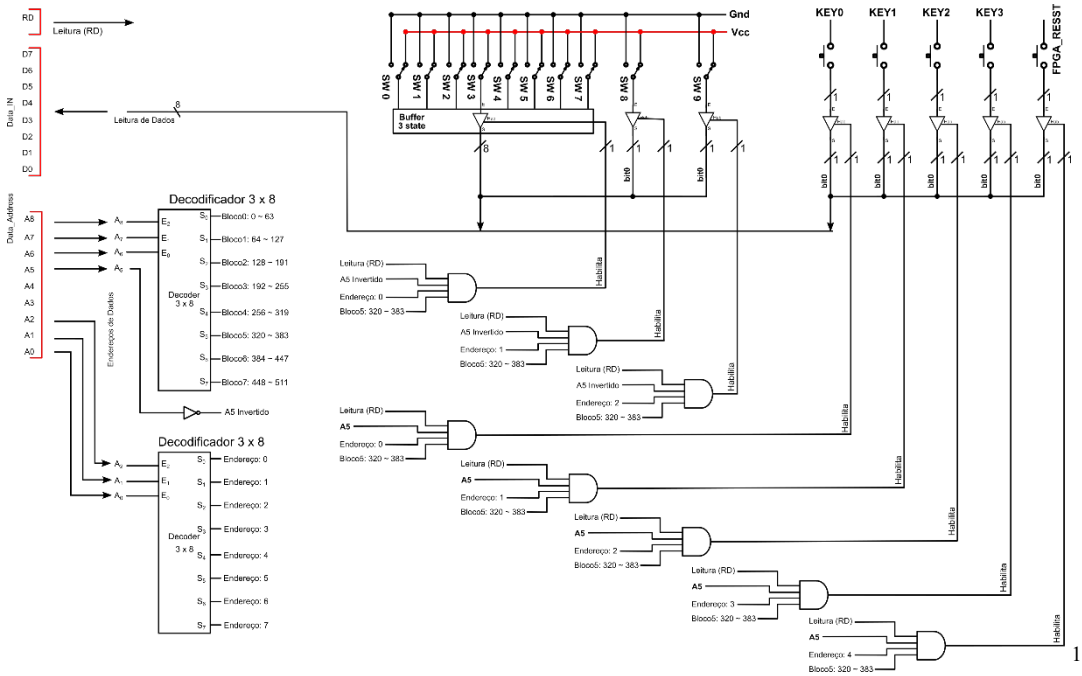
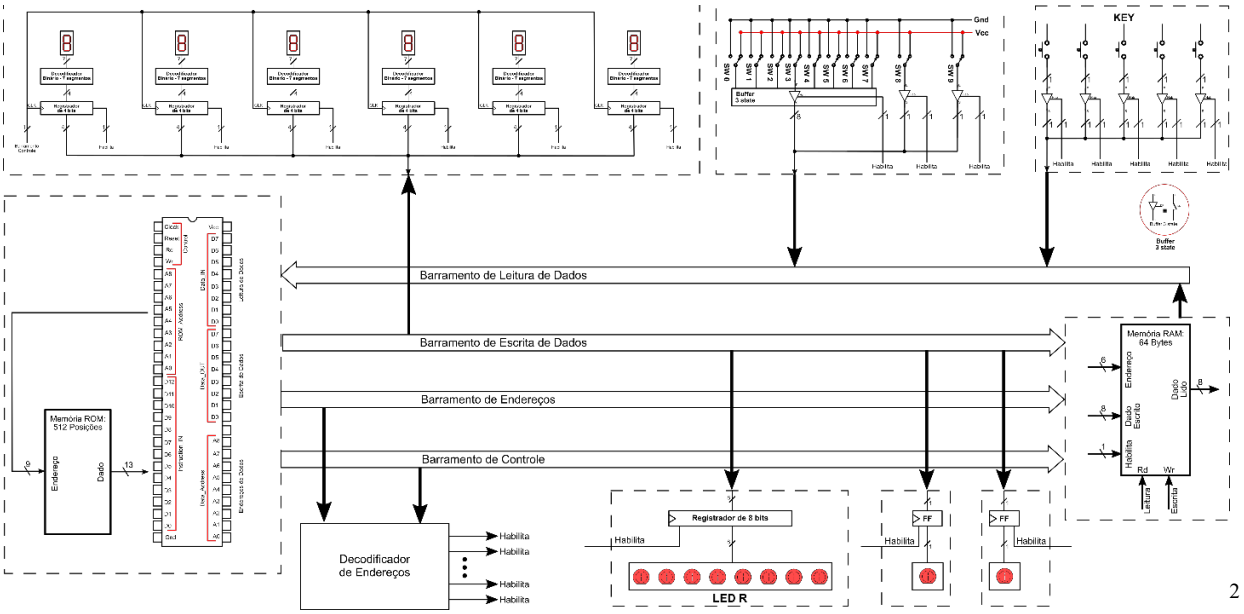



*Figura 5. Periféricos de leitura.*

[1] Imagem retirada da aula 8 do site da disciplina de Design de Computadores 2022 2º Semestre




[2]

*Figura 6. Periféricos conectados nos barramentos do processador*

[2] Imagem retirada da aula 8 do site da disciplina de Design de Computadores 2022 2º Semestre

# 8. Mapa de memória:

Segue abaixo o mapa de memória dos periféricos listados na sessão anterior. A memória está dividida em blocos de 64 endereços, e como temos 512 endereços disponíveis (0~511), temos um total de 8 blocos. O bloco 0 é o boco de memória RAM; o bloco 4 é usado como para a escrita dos LEDs (separados no grupo 7~0 para escritas de 8 bits e os LEDR8 e LEDR9 para registros de um bit só, como uma flag) e displays Hex7seg; o bloco 5 lidamos com os periféricos de leitura, no caso os switches (separando do 7~0 para uma leitura de 8 bits e os SW8 e SW9 para leituras individuais), além dos botões KEY0~3 e o botão de Reset FPGA_RESET. Por último, os endereços para limpar os registradores dos botões são 511, 510 e 507, respectivamente para KEY0, KEY1 e FPGA_RESET. O espaço entre eles é para implementar a limpeza para os botões KEY2 e KEY3 caso sejam implementados.

| Mapa de Memória | | | | |
|---|---|---|---|---|
| **Endereço em Decimal** | **Periférico** | **Largura dos Dados** | **Tipo de Acesso** | **Bloco (Página) de Memória** |
| 0-63 | RAM | 8 bits | Leitura/Escrita | 0 |
| 64-127 | Reservado | - | - | 1 |
| 128-191 | Reservado | - | - | 2 |
| 192-255 | Reservado | - | - | 3 |
| 256 | LEDR0-LEDR7 | 8 bits | Escrita | 4 |
| 257 | LEDR8 | 1 bit | Escrita | 4 |
| 258 | LEDR9 | 1 bit | Escrita | 4 |
| 259-287 | Reservado | - | | 4 |
| 288 | HEX0 | 4 bits | Escrita | 4 |
| 289 | HEX1 | 4 bits | Escrita | 4 |
| 290 | HEX2 | 4 bits | Escrita | 4 |
| 291 | HEX3 | 4 bits | Escrita | 4 |
| 292 | HEX4 | 4 bits | Escrita | 4 |
| 293 | HEX5 | 4 bits | Escrita | 4 |
| 294-319 | Reservado | - | - | 4 |
| 320 | SW0-SW7 | 8 bits | Leitura | 5 |
| 321 | SW8 | 1 bit | Leitura | 5 |
| 322 | SW9 | 1 bit | Leitura | 5 |
| 323-351 | Reservado | - | - | 5 |
| 352 | KEY0 | 1 bit | Leitura | 5 |
| 353 | KEY1 | 1 bit | Leitura | 5 |
| 354 | KEY2 | 1 bit | Leitura | 5 |
| 355 | KEY3 | 1 bit | Leitura | 5 |
| 356 | FPGA_RESET | 1 bit | Leitura | 5 |
| 357-383 | Reservado | - | - | 5 |
| 384-447 | Reservado | - | - | 6 |
| 448-506 | Reservado | - | - | 7 |
| 507 | Limpa Leitura FPGA_RESET | - | Escrita | 7 |
| 508-509 | Reservado | - | - | 7 |
| 510 | Limpa Leitura KEY1 | - | Escrita | 7 |
| 511 | Limpa Leitura KEY0 | - | Escrita | 7 |

*Tabela 2. Mapa de memória do Processador.*

# 9. Programa assembly:

Nesta parte do relatório será descrito o funcionamento do programa em *assembly* na integra, explicando a escolha para cada construção do código. O código completo será enviado junto com este relatório, dentro da pasta assembler, com o nome **ASM.txt**.

```
SETUP:
LDI R[0] $0
STA R[0] @0
LDI R[2] $0         # Limpando o registrador das unidades
LDI R[3] $0         # Limpando o registrador das dezenas
LDI R[4] $0         # Limpando o registrador das centenas
STA R[0] @10        # Espaço na memória dedicado ao milhar
STA R[0] @11        # Espaço na memória dedicado à dezena de milhar
STA R[0] @12        # Espaço na memória dedicado à centena de milhar
LDI R[0] $1
STA R[0] @1
STA R[0] @3         # Utilizado para máscara de Bit menos significativo
LDI R[0] $9
STA R[0] @2         # Comparação para definir o aumento do próximo HEX
STA R[0] @20        # Redefinindo todos os HEX para 0
STA R[0] @21
STA R[0] @22
STA R[0] @23
STA R[0] @24
STA R[0] @25
LDI R[0] $128
STA R[0] @4         # Utilizado para máscara de Bit mais significativo
STA R[0] @511       # Limpando o botão de Incremento
STA R[0] @510       # Limpando o botão de Alterar Limite
STA R[0] @507       # Limpando o botão de RESET
```

O código acima diz respeito ao comportamento de SETUP do nosso contador, onde guardamos o primeiro valor de comparação no endereço da memória 0 e limpamos todos os registradores e espaços de memória para os valores do contador. Em seguida guardamos o segundo valor de comparação para o endereço de memória 1 e guardamos no endereço de memória 3 para a realização das máscaras com o bit menos significativo. Guardamos, então, o valor 9 para comparação no endereço de memória 2 e predefinimos o limite máximo de contagem para 999.999, um valor em cada endereço de memória do menos ao mais significativo. Depois o valor 128 é salvo no endereço de memória 4 para máscara do bit mais significativo. Por último, limpamos todos os registradores dos botões alocados para o contador.

```
# ========================================================= #
#                      LOOP PRINCIPAL                       #
# ========================================================= #
LOOP:
LDA R[1] @352
OPAND R[1] @3
CEQ R[1] @0          # O botão de incremento não foi clicado
JEQ @TROCALIMITE
JSR @INCREMENTA
JSR @VERIFICA
TROCALIMITE:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0          # O botão de troca de limite não foi clicado
JEQ @LOOP
JSR @SETLIMITE
JMP @LOOP
```

É no loop principal que ocorre a leitura dos botões para entrada nas sub-rotinas. Primeiro é verificado o botão de incremento. Caso ele seja pressionado, o código entrará, primeiramente, na sub-rotina de incremento do valor do display, e depois entrará na sub-rotina de verificação do limite definido para o contador. Caso não seja pressionado ou terminem as sub-rotinas acima, será verificado o botão para troca do valor de limite. Caso seja pressionado entrará em tal sub-rotina. Caso nenhum botão seja clicado ou tenham passado por todas as sub-rotinas que foram iniciadas, voltará ao início para mais uma verificação.

```
# ========================================================= #
#                     INCREMENTA VALOR                      #
# ========================================================= #
INCREMENTA:
STA @511
CEQ R[2] @2
JEQ @ATDEZ
SOMA R[2] @1
STA R[2] @288
RET
ATDEZ:              # Atualiza o dígito das dezenas
LDI R[2] $0
STA R[2] @288
CEQ R[3] @2
JEQ @ATCEN
SOMA R[3] @1
STA R[3] @289
```

```
RET
ATCEN:              # Atualiza o dígito das centenas
LDI R[3] $0
STA R[3] @289
CEQ R[4] @2
JEQ @ATMIL
SOMA R[4] @1
STA R[4] @290
RET
ATMIL:              # Atualiza o dígito dos milhares
LDI R[4] $0
STA R[4] @290
LDA R[5] @10
CEQ R[5] @2
JEQ @ATDMIL
SOMA R[5] @1
STA R[5] @291
STA R[5] @10
RET
ATDMIL:             # Atualiza o dígito das dezenas de milhar
LDI R[5] $0
STA R[5] @291
STA R[5] @10
LDA R[5] @11
CEQ R[5] @2
JEQ @ATCMIL
SOMA R[5] @1
STA R[5] @292
STA R[5] @11
RET
ATCMIL:             # Atualiza o dígito das centenas de milhar
LDI R[5] $0
STA R[5] @292
STA R[5] @11
LDA R[5] @12
CEQ R[5] @2
JEQ @MAXOVERFLOW    # Caso exceda o limite máximo do contador
SOMA R[5] @1
STA R[5] @293
STA R[5] @12
RET
```

Acima está descrita a sub-rotina de incremento do contador. Ela é responsável por aumentar um valor nos dígitos do contador, garantindo que não ultrapassem o valor 9. Caso o dígito esteja em 9, será adicionado um no dígito acima, zerando o dígito atual. Caso isso aconteça para o display das centenas de milhar, o código irá para a função que irá apresentar ao utilizador que o contador alcançou o overflow máximo de 999.999.

```
# ========================================================== #
#                     VERIFICA LIMITE                        #
# ========================================================== #
## A verificação começa com a centena de milhar
VERIFICA:
LDA R[7] @25
SUB R[7] @12
OPAND R[7] @4
CEQ R[7] @4
JEQ @OVERLIMIT
LDA R[7] @12
CEQ R[7] @25
JEQ @VERDMIL
RET
VERDMIL:
LDA R[7] @24
SUB R[7] @11
OPAND R[7] @4
CEQ R[7] @4
JEQ @OVERLIMIT
LDA R[7] @11
CEQ R[7] @24
JEQ @VERMIL
RET
VERMIL:
LDA R[7] @23
SUB R[7] @10
OPAND R[7] @4
CEQ R[7] @4
JEQ @OVERLIMIT
LDA R[7] @10
CEQ R[7] @23
JEQ @VERCEN
RET
VERCEN:
STA R[4] @30
LDA R[7] @22
```

```
SUB R[7] @30
OPAND R[7] @4
CEQ R[7] @4
JEQ @OVERLIMIT
CEQ R[4] @22
JEQ @VERDEZ
RET
VERDEZ:
STA R[3] @30
LDA R[7] @21
SUB R[7] @30
OPAND R[7] @4
CEQ R[7] @4
JEQ @OVERLIMIT
CEQ R[3] @21
JEQ @VERUNI
RET
VERUNI:
STA R[2] @30
LDA R[7] @20
SUB R[7] @30
OPAND R[7] @4
CEQ R[7] @4
JEQ @OVERLIMIT
CEQ R[2] @20
JEQ @LIMITE
RET
# ================== Chegou no limite ==================== #
LIMITE:
LDI R[0] $255
STA R[0] @256
JMP @END
```

A sub-rotina acima é utilizada para a verificação do limite no contador. A verificação começa pelo dígito das centenas de milhar para facilitar a verificação de um valor maior, que pode ser alcançado ao configurar um limite inferior àquele já alcançado pelo contador. Essa verificação é feita ao subtrair o valor atual do contador do valor definido como limite. Caso essa subtração resulte em um valor negativo, será feita uma máscara com o bit mais significativo, que caso seja ‘1’, em complemento de dois, indica um valor negativo. Caso a verificação seja verdadeira é enviado direto para a função mostrará ao usuário que o valor indicado está acima do limite (*OVERLIMIT*). Caso não seja verdade, a verificação será feita para o caso de igualdade com o dígito do limite. Se igual, passará

para a verificação do dígito inferior. Caso menor, ou seja, nem maior nem igual, voltará para o loop principal. Essa operação será feita para todos os dígitos em ordem decrescente de significância. Caso todos os dígitos sejam iguais àqueles salvos como limite, o contador chegou a seu limite, entrando na função LIMITE, que acenderá todos os LEDs de 0 a 7 e por fim será enviado para o loop final (END).

```
# ========================================================== #
#                         SET LIMITE                         #
# ========================================================== #
SETLIMITE:
STA @510
LDI R[0] $0
STA R[0] @288
STA R[0] @289
STA R[0] @290
STA R[0] @291
STA R[0] @292
STA R[0] @293
LDI R[6] @1
STA R[6] @257
WAITUNI:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0
JEQ @WAITUNI
STA @510
LDA R[6] @320
STA R[6] @20
STA R[6] @288
WAITDEZ:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0
JEQ @WAITDEZ
STA @510
LDA R[6] @320
STA R[6] @21
STA R[6] @289
WAITCEN:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0
JEQ @WAITCEN
```

```
STA @510
LDA R[6] @320
STA R[6] @22
STA R[6] @290
WAITMIL:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0
JEQ @WAITMIL
STA @510
LDA R[6] @320
STA R[6] @23
STA R[6] @291
WAITDMIL:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0
JEQ @WAITDMIL
STA @510
LDA R[6] @320
STA R[6] @24
STA R[6] @292
WAITCMIL:
LDA R[1] @353
OPAND R[1] @3
CEQ R[1] @0
JEQ @WAITCMIL
STA @510
LDA R[6] @320
STA R[6] @25
STA R[6] @293
# ============ Retornando o valor do contador ============= #
STA R[2] @288
STA R[3] @289
STA R[4] @290
LDA R[5] @10
STA R[5] @291
LDA R[5] @11
STA R[5] @292
LDA R[5] @12
STA R[5] @293
LDI R[6] @0
STA R[6] @257
```

```
RET
```

Para a sub-rotina de troca do valor de limite do contador, será redefinido os valores dos displays para zero e acenderá o LED8 para indicar que entrou nessa sub-rotina. A partir desta configuração, o programa a cada clique do KEY1 lerá o valor das chaves de 0 a 7 e utilizará como limite do dígito, crescendo de unidades para centenas de milhar. A cada clique, o valor definido será apresentado para o usuário no display do dígito definido. Assim que forem definidos todos os valores, o contador continuará a contagem nos displays, redefinindo os valores salvos antes da entrada nesta subrotina.

```
# ========================================================== #
#                          OVERFLOW                          #
# ========================================================== #
MAXOVERFLOW:
LDI R[2] $9
LDI R[3] $9
LDI R[4] $9
STA R[2] @10
STA R[2] @11
STA R[2] @12
STA R[2] @288
STA R[2] @289
STA R[2] @290
STA R[2] @291
STA R[2] @292
STA R[2] @293
JMP @OVERFLOW
OVERLIMIT:
LDI R[0] $255
STA R[0] @256
JMP @OVERFLOW
OVERFLOW:
STA @511
LDI R[0] $1
STA R[0] @258
JMP @END
```

Acima, estão descritas as funções de *overflow*. MAXOVERFLOW é a função que apresentará ao usuário que este chegou ao valor máximo de contagem do sistema (999.999), redefinindo todos os valores como 9, para garantir que nenhum foi alterado e apresentando isso nos displays e finalizará pulando para a função de OVERFLOW. A Função de OVERLIMIT ligará todos os LEDs de 0 a 7 para indicar que chegou no limite

definido e pulará para a função de OVERFLOW. Por fim, a função de OVERFLOW limpará o botão de incremento, ligará o LED9, indicador de *overflow,* e pulará para o loop final (END).

```
# ========================================================= #
#                          LOOP END                         #
# ========================================================= #
END:
LDA R[1] @356
OPAND R[1] @3
CEQ R[1] @1
JEQ @CLEAR
LDA R[1] @352
OPAND R[1] @3
CEQ R[1] @1
JEQ @OVERFLOW
JMP @END
```

No loop final (END), o sistema está aguardando o clique em dois botões. O primeiro deles, o botão FPGA_RESET, é para limpar os valores do contador e reiniciar o sistema, pulando para a função de CLEAR. Já o segundo é para acender o LED9 na função de OVERFLOW, caso o valor de limite seja alcançado e haja uma tentativa de incremento do valor, garantindo essa indicação ao usuário e impedindo uma variação do valor atual do contador. Caso nada disso tenha acontecido ocorrerá um pulo para o início desta função, fechando o loop final.

```
# ========================================================= #
#                           CLEAR                           #
# ========================================================= #
CLEAR:
LDI R[0] $0
STA R[0] @288
STA R[0] @289
STA R[0] @290
STA R[0] @291
STA R[0] @292
STA R[0] @293
STA R[0] @256
STA R[0] @257
STA R[0] @258
JMP @SETUP
```

Por último, a função de CLEAR irá limpar todos os valores dos displays e dos LEDs, definindo todos como zero e voltará para o início do código na função de SETUP.

## Referências:

S. Carlos, Paulo. Design de Computadores. BlackBoard, 2022. Disponível em: https://insper.blackboard.com/bbcswebdav/pid-1027548-dt-content-rid-10756798_2/courses/202262.GRENGCOM_201561_0004.DESIGNCOMP_6ENGCOMPA/Atividades/desComp.html . Acesso em: 23 out. 2022.