Insper

Relatório de Atividade da Disciplina Design de Computadores

Projeto 1: Relógio Utilizando um Processador Personalizado

Relatório Final

Lais Nascimento da Silva

William Augusto Reis da Silva

João Guilherme Cintra de Freitas Almeida

Professor Paulo Carlos Santos

São Paulo
Outubro/2021

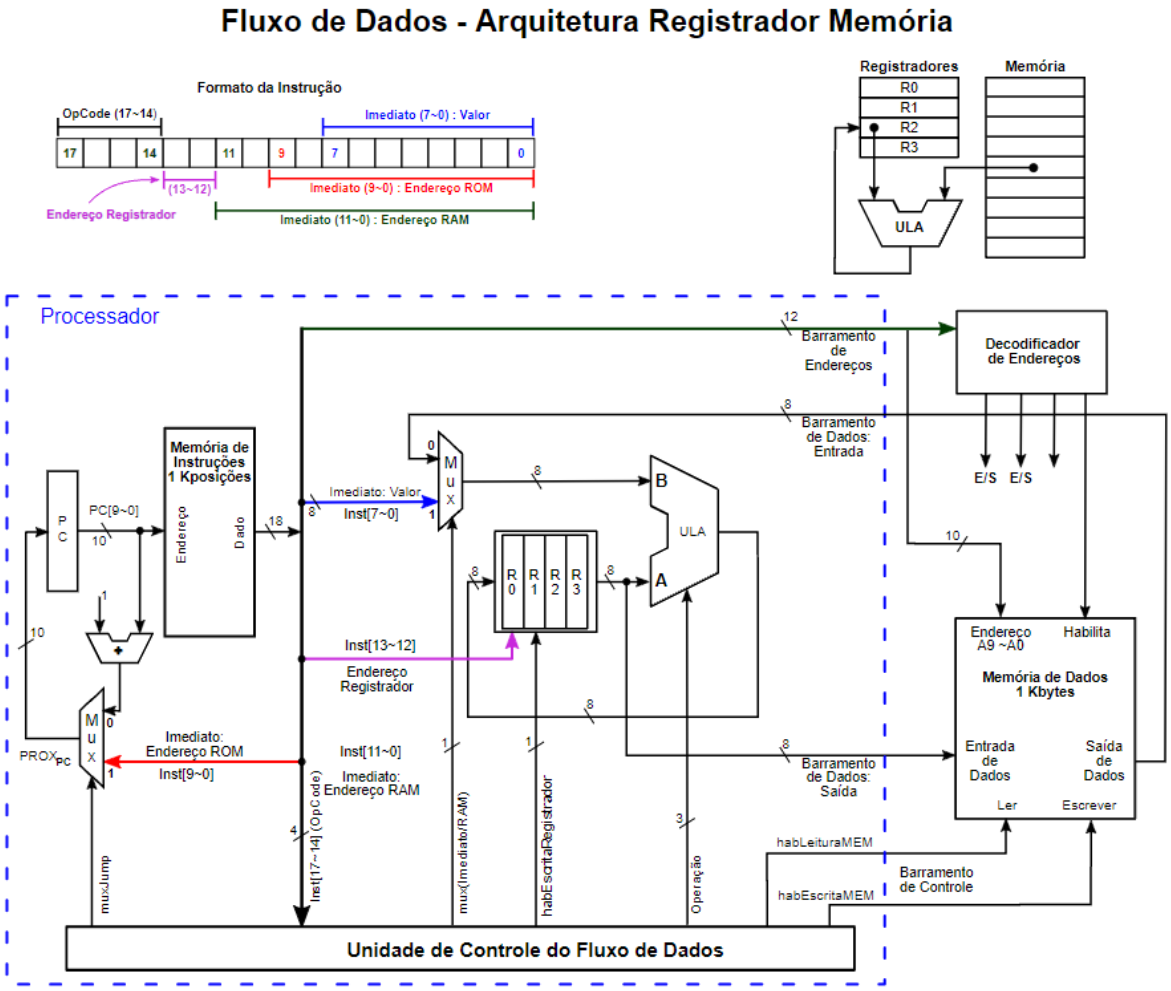
1. Resumo do Projeto

O projeto realizado foi a construção de um relógio a ser utilizado na placa FPGA fornecida pelo Insper. Essa placa possui seis displays de sete segmentos, que representam HH:MM:SS (horas, minutos e segundos), possui chaves (KEY0, KEY1 e KEY2), que são utilizadas para features feitas, como o acerto de horário, a mudança da base de tempo para o tempo correr mais rápido.

Toda a construção é feita por meio do VHDL e os componentes serão descritos na explicação dos próximos tópicos.

2. Arquitetura do Processador

Como foi exposto no último relatório, para fazer a escolha da arquitetura do processador para o projeto, o grupo levou em consideração que até o início do projeto, a arquitetura da CPU estava baseada em Acumulador. Após o início do projeto, foi mais conveniente alterar a arquitetura do processador para Registrador-Memoria, devido ao fato de a alteração parecer simples. A imagem abaixo mostra como funciona esse registrador:



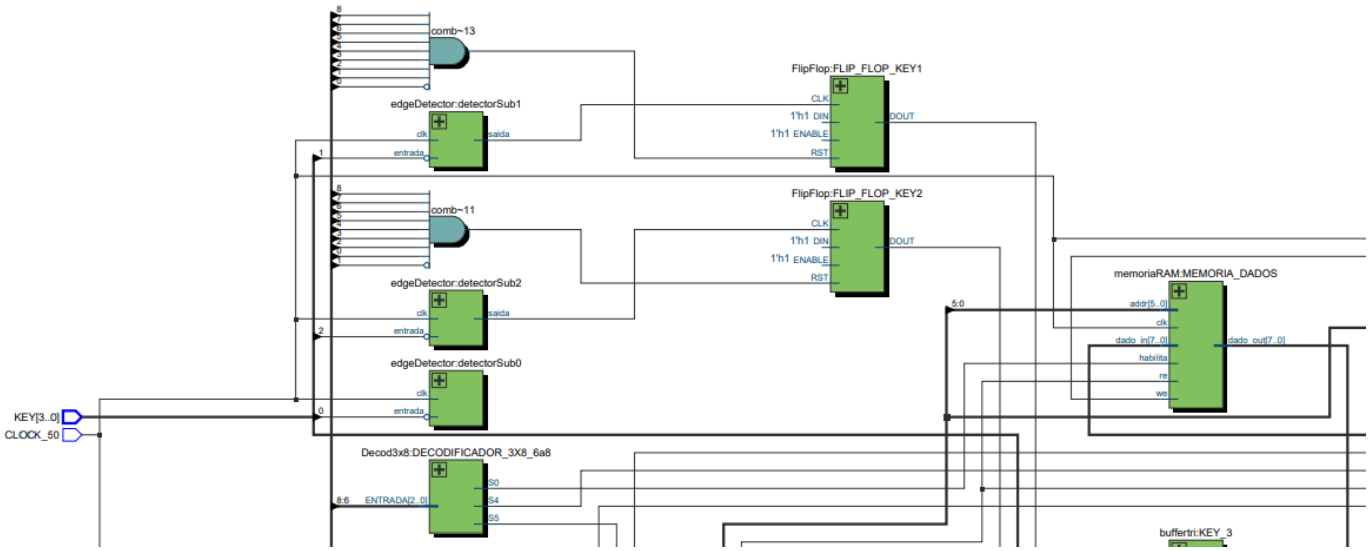
No próximo tópico será demonstrado a implementação no nosso diagrama de blocos, mas a ideia é essa. Temos a memória de instruções, que vai recebendo do PC

o endereço e sai um dado que é o mais importante de tudo, ele mostra o endereço do registrador, tem o valor imediato caso seja um LDI por exemplo, ou o endereço da ROM caso seja um JMP e isso vai determinando todo o fluxo.

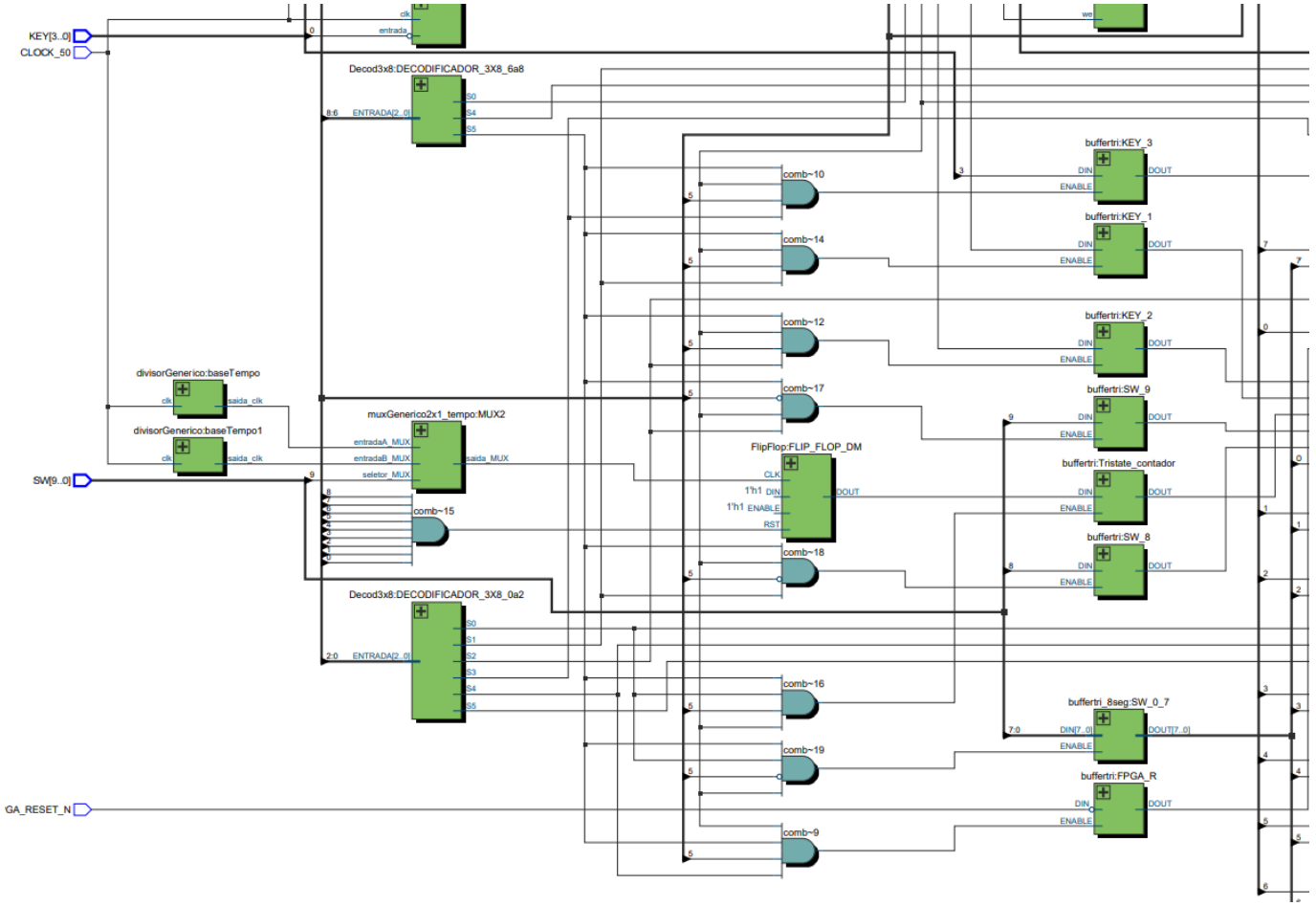
3. Diagrama de Blocos

O diagrama de blocos completo pode ser visto por meio desse link: https://bit.ly/3ndBYCj, onde abrindo/baixando o PDF, ao se dar zoom, consegue-se ver corretamente e nitidamente cada ponto do nosso projeto. Como uma imagem dessa não ficaria de qualidade neste arquivo, a explicação se dará com imagens de partes do diagrama, correlacionando com os que estão em aula, explicando assim cada tópico.

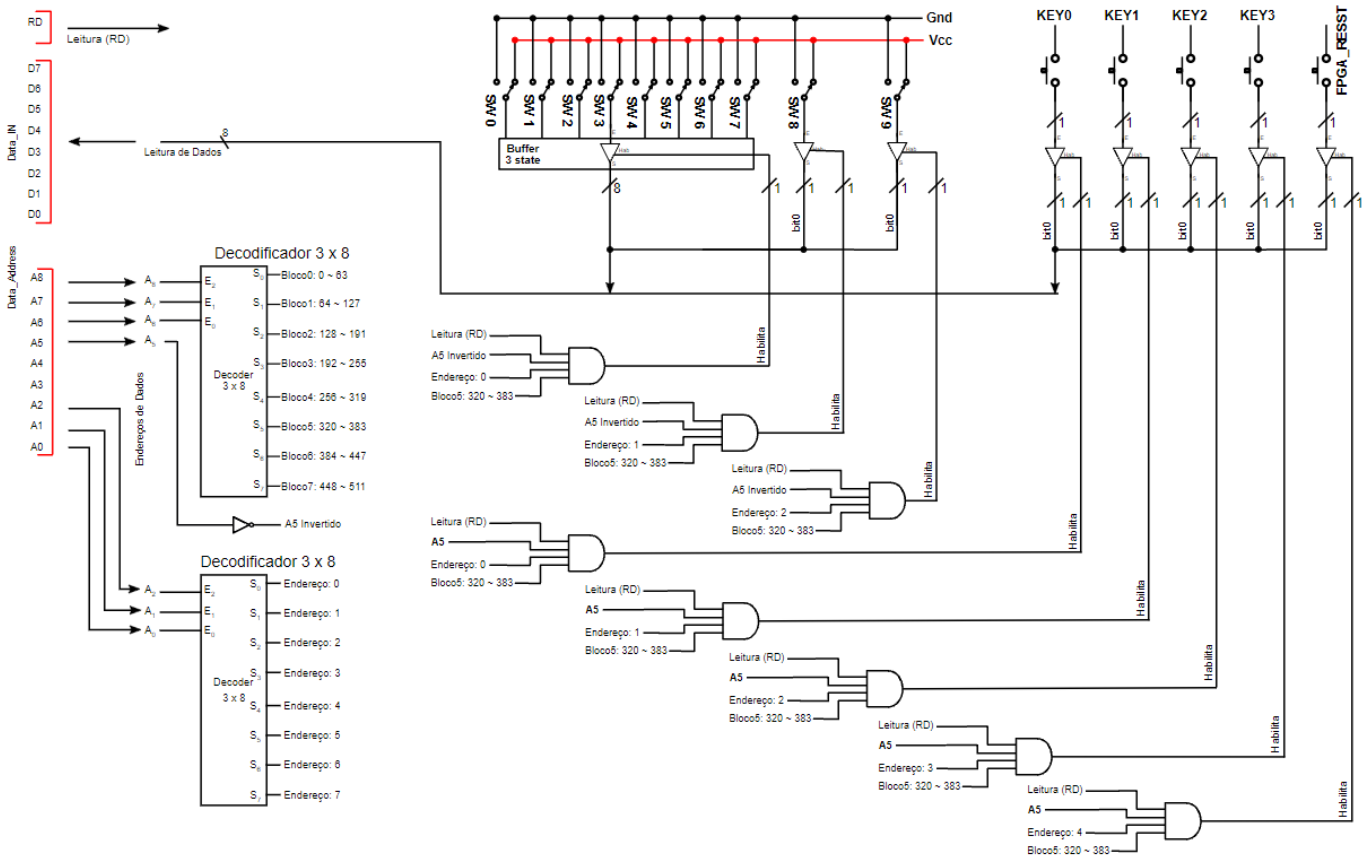
De início, tem-se os blocos abaixo:



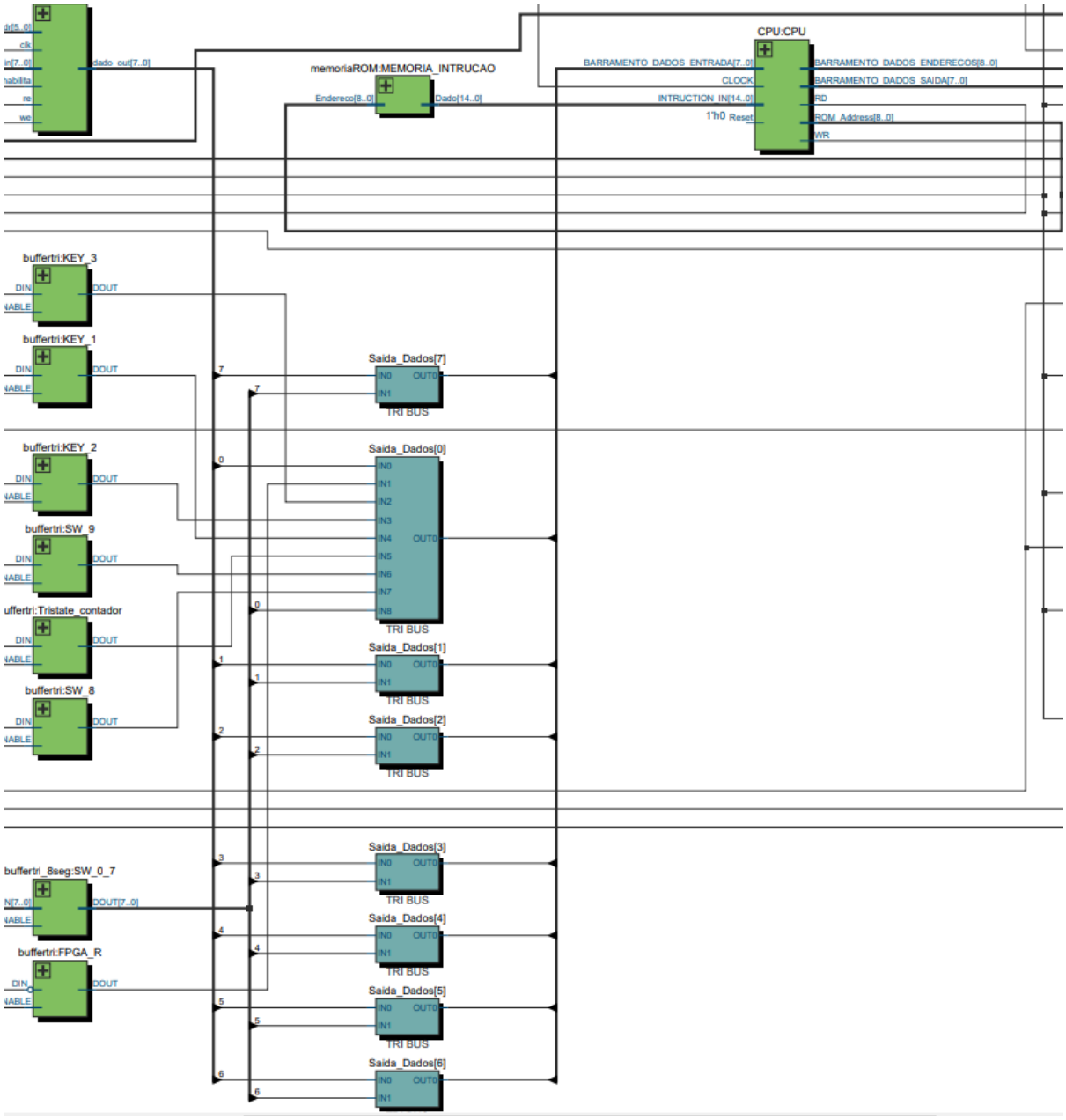
Como é possível ver, o que se tem nessa parte são os detectores de borda para os tratamentos especiais das KEYs, evitando os múltimos incrementos. Ali, implementamos nos FLIP-FLOPs o circuito de debounce. Essa parte faz a chamada limpeza das chaves. Para emendar juntando essa parte com outra do circuito, pode-se observar o decodificador 3x8, que está também na imagem abaixo.



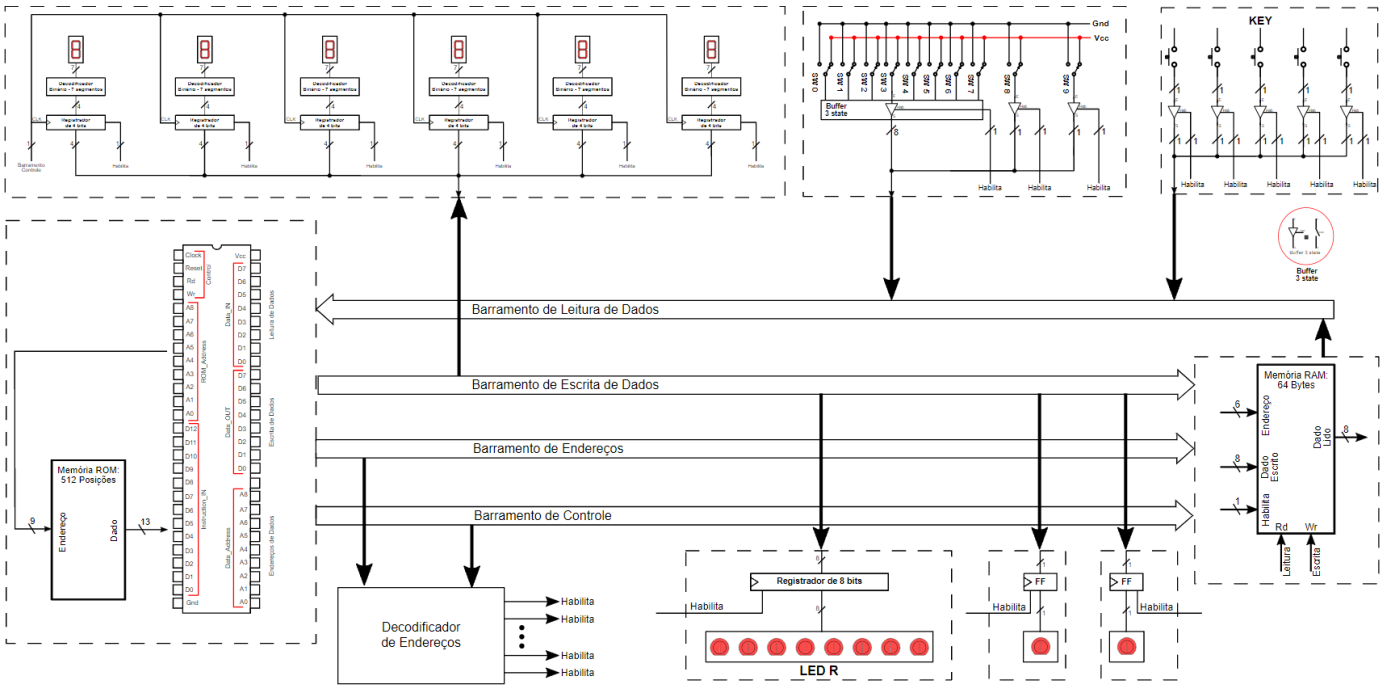
Acima, vemos o decodificador, onde suas saídas são utilizadas em outros locais, como na RAM, por exemplo, que será explorada nas próximas imagens. Abaixo dele, tem-se os divisores genéricos, entrando num MUX, que é da funcionalidade do relógio “andar” mais rápido. Ou seja, cada um tem sua base de tempo e o MUX, a partir do seu seletor, que vem de um dos SWs. E todos os outros buffertri do lado podem ser melhor explicados pelo diagrama de blocos apresentado na sala de aula:



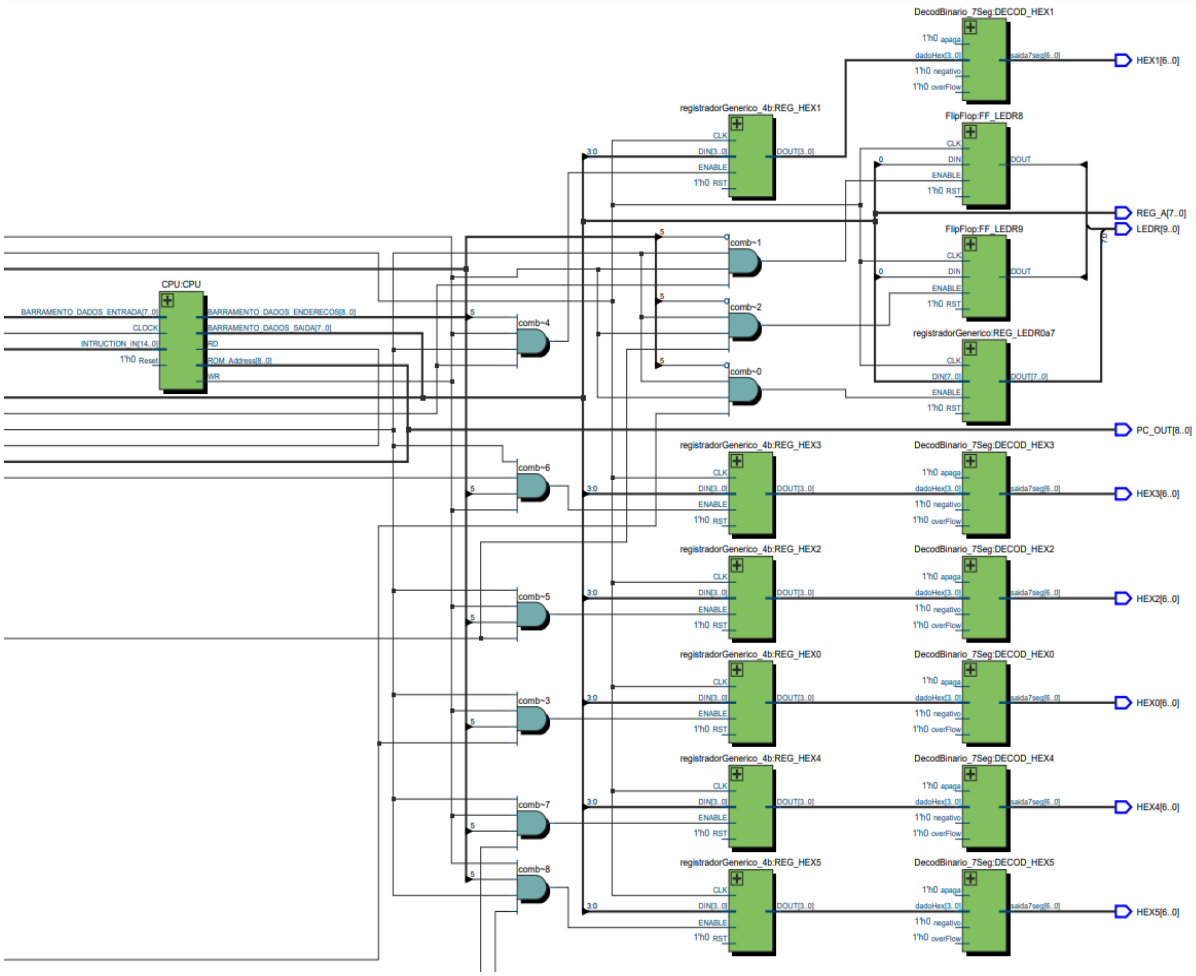
Como é possível ver, tem-se os buffertri em cada SW e KEY, além do FPGA_RESET, e também as portas AND, determinando o Habilita desses buffers. Como podemos ver, as saídas desses buffers são conectadas com a leitura de dados da CPU, a entrada dela e na imagem abaixo tem-se exatamente tal descrição:



A imagem da aula (abaixo) demonstra de forma mais geral como funciona o fluxo desses dados, dividido pelos componentes. As saídas vão para os barramentos, que vão para a CPU. E existem outros, que fazer a escrita de dados, as que pegam os endereços e faz toda a lógica necessária para o funcionamento.

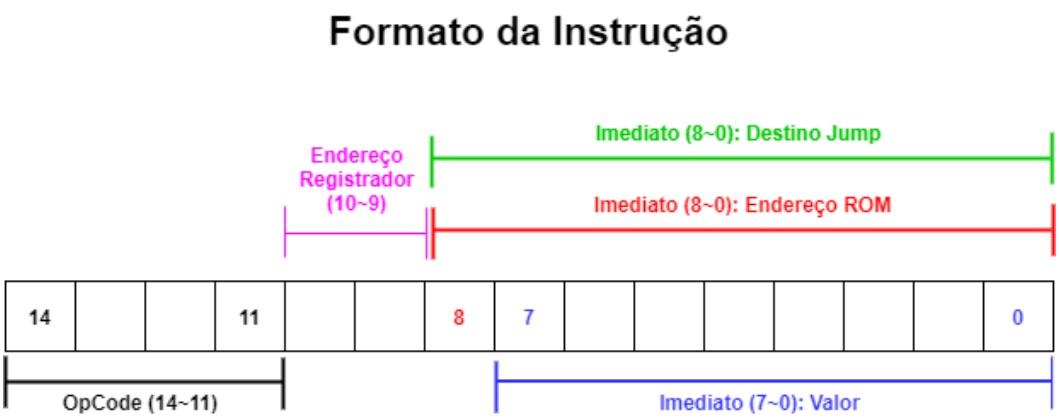


Por fim, as duas últimas imagens representam um pouco mais dos displays, onde se tem os registradores de cada um, ligado diretamente ao decodificador de cada um, que faz o trabalho de transformar o número em forma visual.



4. Formato das Instruções

Como foi proferido no último relatório, as instruções são do tipo:



Essa é uma lembrança para compreender o próximo tópico, no qual realizamos o montador.

5. Opcionais

a. Assembler

Analisando a instrução, consegue-se entender como foi feito o principal opcional, que foi o Assembler para o construtor. O arquivo se encontra no repositório do grupo caso seja necessária a consulta, porém a ideia foi ler um arquivo em assembly, e transformando, a partir das instruções, em uma instrução apenas com bits, para passar para um arquivo *.mif*.

- Sintaxe do Assembly: utilizamos uma sintaxe como:

OPCODE, REGISTRADOR, IMEDIATO;

Porém, o imediato pode alterar sua forma dependendo de qual o comando estamos almejando.

Alguns exemplos para a construção do assembly:

LDI, R1, $00; - Para a leitura de um valor imediato, usou-se $ antes do número, com o número em hexadecimal;

STA, R1, DSP0; - Para os displays, usamos essa sintaxe de DSP e seu número;

STA, R1, MEM[00]; - Ao se referir à memória*, utilizou-se a posição por meio de parênteses e com dois dígitos.

JMP, R1, .LABEL; - Para se referir à ida para uma outra linha, foi usado um . antes da referência;

LABEL: JMP, R1, .LABEL2; - Para saber qual é a referência, a linha é construída dessa forma;

*Uma observação importante são as posições de memória e para que elas são utilizadas. A lista abaixo demonstra exatamente:

Limite de HEX0: MEM[46]

Limite de HEX1: MEM[47]

Limite de HEX2: MEM[48]

Limite de HEX3: MEM[49]

Limite de HEX4: MEM[50]

Limite de HEX5: MEM[51]

HEX0: MEM[52]

HEX1: MEM[53]

HEX2: MEM[54]

HEX3: MEM[55]

HEX4: MEM[56]

HEX5: MEM[57]

MEM[06]: SALVA O NOVO LIMITE DO HEX4. Pode ser 4 ou 3. Para não substituir o valor direto no MEM[56]

MEM[05]: Flag de sitema AM/PM ou 24h, 0 se está no AM/PM, 1 se está no 24hrs

- Programa em Python

Nosso programa foi baseado em transformar esse Assembly para o arquivo *.mif*. É importante lembrar que tínhamos três partes da linha do Assembly: OPCODE, REGISTRADOR e INSTRUÇÃO. Por conta disso, fizemos uma função para fazer a transformação do que estava escrito para a sua palavra de bits.

Antes de tudo isso, foi definido dicionários para os três tipos, demonstrando o que eram e qual sua palavra de bits correspondente, como mostra abaixo:

```
# DICIONÁRIO: "OPCODE": "CÓDIGO BIN"
Dic_OpCode = {"NOP":  "0000",
              "LDA":  "0001",
              "SOMA": "0010",
              "SUB":  "0011",
              "LDI":  "0100",
              "STA":  "0101",
              "JMP":  "0110",
              "JEQ":  "0111",
              "CEQ":  "1000",
              "JSR":  "1001",
              "RET":  "1010"}
```

```
# DICIONÁRIO: "REGISTRADOR": "CÓDIGO DO REG"
Dic_Regs = {"R0":"00",
            "R1":"01",
            "R2":"10",
            "R3":"11"}
```

Assim também foi com os endereços, referenciando os displays, LEDs, SWs, KEYs, FPGA_RESET.

Após isso, o grupo criou as funções de pegar do Assembly cada um dos tipos, além de uma que pega a linha referente aos LABEL, pois é necessária para saber qual linha referencia. O código abaixo com comentários especifica muito bem como funciona essa tradução do LABEL e sua linha correspondente.

```
'''
Função get_lines_label lê quais são as linhas onde tem LABEL

EXEMPLO DE FORMA DO LABEL:
SUPONHA QUE TEMOS AS LINHAS 1, 2 E 3 COM LABEL EM ALGUMA

1| JMP, R2, .EXAMPLE;
2| LDA, R2, MEM[00];
3| EXAMPLE: STA, R2, MEM[54];

A funcao vai retornar o label e sua linha
{"EXAMPLE": 3}

'''
def get_lines_label(name, outfile):
    jumps = {}
    contador_linhas = 0
    novo_arquivo = open(outfile, 'w')
    arquivo = open(name, 'r')
    # Percorre todas as linhas do arquivo
    for l in arquivo:
        contador_caracs = 0
        apagou = False
        for e in l:
            if e == ":":
                 # Coloca no dicionário
                jumps[l[:contador_caracs]] = contador_linhas
                # Apaga o LABEL p/ arquivo .mif
                line = l.replace(l[:contador_caracs+2], "")
                novo_arquivo.write(line)
                apagou = True
            contador_caracs += 1
        contador_linhas += 1
        if not apagou:
            novo_arquivo.write(l)
    return jumps, arquivo
```

Após isso, é feita a função de transformar o OpCode em sua palavra de bit correspondente. É importante ressaltar que assim como nela, todas as outras são adicionadas em uma lista e posteriormente é conectado os três por meio dessas listas. Assim como na de cima, os comentários explicam como é feito.

```
'''
Funcao get_opcode faz a traducao do que tem de opcode a partir do dicionario
e devolve o seu codigo

EXEMPLO
se tem NOP, ele vai no dicionario de OPCODE, encontrar a chave NOP e ver qual
eh seu valor.

'''
def get_opcode(novo_arquivo, dic_op):
    arquivo = open(novo_arquivo, 'r')
    lista_opcodes = []
    for l in arquivo:
        palavra_binaria = ""
        conta_virgulas = 0
        conta_carac = 0
        # Lê todos os caracteres de cada linha, para pegar o OpCode
        for e in l:
            if e==","  and conta_virgulas == 0: # Se estiver até primeira
vírgula (que é onde está o OpCode)
                conta_virgulas += 1
                print(l[:conta_carac])
                valor = dic_op[l[:conta_carac]] # Pega o valor binario
                palavra_binaria += valor
                lista_opcodes.append(palavra_binaria)
            conta_carac += 1
    return lista_opcodes
```

Semelhante no de registradores:

```
'''
FUNÇÃO get_reg vai pegar a parte do meio do código em Assembly
e transformar os R0, R1, R2, e R3 para seus códigos (suas palavras binárias)

EXEMPLO
NOP, R1, $00;

A função conta até a segunda vírgula e pega os dois caracteres de antes, que é
o R(number)

'''
def get_reg(novo_arquivo, dic_reg):
    arquivo = open(novo_arquivo, 'r')
    lista_regs=[]
```

```
    for l in arquivo:
        palavra_binaria = ""
        conta_virgulas = 0
        conta_carac = 0
        for e in l:
            if e==",":
                conta_virgulas += 1
                if conta_virgulas == 2:
                    # Pega qual é o registrador e já pega o binário no
dicionário
                    valor = dic_reg[l[conta_carac-2:conta_carac]]
                    palavra_binaria += valor
                    lista_regs.append(palavra_binaria)
            conta_carac += 1
    return lista_regs
```

Já agora no de instruções, fica um pouco mais complicado, tendo em vista que existem 4 tipos de instrução. O código fica muito maior, porém tudo explicado nos comentários.

```
'''
Assim como as de cima, pega no Assembly e transforma para bits, porém
a última parte agora, da instrução
Porém, há quatro tipos, exemplos de cada:

Se começa com ponto, é de JMP
JEQ, R1, .UM_QUATRO;

Se tem $, é um número em hexa, utilizado para limites ou algo do tipo
NOP, R1, $00;

Se tem M, é de local da memória
CEQ, R3, MEM[03];

Se não é nenhum de cima, ele pega no de endereço, pois é algum deles
STA, R1, LIMPA0;

'''
def get_inst(novo_arquivo, dic_end, dic_jump):
    arquivo = open(novo_arquivo, 'r')

    lista_inst=[]
    #ler cada linha
    for l in arquivo:
        conta_virgulas = 0
        conta_carac = 0
        for e in l:
            if e == ",":
```

```
                conta_virgulas += 1
                if conta_virgulas == 2:
                    pos1 = conta_carac + 2
            if e == ";":
                pos2 = conta_carac
            conta_carac += 1
        inst = l[pos1:pos2]
        print(l)

        # Exemplo: JEQ, R1, .UM_QUATRO;
        if inst[0] == ".":
            valor = dic_jump[inst[1:]]
            number_bin = bin(int(valor))[2:].zfill(9)
            lista_inst.append(str(number_bin))

        # Exemplo: NOP, R1, $00;
        elif inst[0] == "$": #binario de 9 bits
            number_hex = inst[1:3]
            number_bin = bin(int(number_hex, 16))[2:].zfill(9)
            lista_inst.append(str(number_bin))

        # Exemplo: CEQ, R3, MEM[03];
        elif inst[0] == "M":
            number_dec = int(inst[4:6])
            number_bin = bin(number_dec)[2:].zfill(9)
            lista_inst.append(str(number_bin))

        # Exemplo: STA, R1, LIMPA0;
        else:
            valor = dic_end[inst]
            lista_inst.append(valor)

    return lista_inst
```

Após essas funções, o importante é criar uma que faz o arquivo mif e vai completando a partir do que é realizado nessas funções de cima. Assim, nosso python consegue cumprir corretamente seu papel, criando um arquivo *.mif* corretamente.

b. Horário AM/PM

Como a ideia geral era fazer com 0h até 23h59min, o grupo fez isso de início e para complementar foi realizado também a troca para o modelo de 12h. A implementação foi feito no software em Assembly. Para ser feito, é necessário adicionar uma chave, que foi a KEY3, na qual ao ser apertada, altera o modelo de 24h para o modelo AM/PM.

A lógica básica no Assembly é de ficar conferindo se a KEY3 foi apertada e, caso foi, muda-se os limites dos HEX para que não vá mais para 23 e sim 12 (HEX5 e HEX4).

- Como testar o AM/PM?

O projeto, ao ser rodado pelo Quartus, vai conectar o relógio com o modo 24h, ou seja, indo de 00:00:00 (HEXs 54:32:10) à 23:59:59. Para alterar para o modo de AM/PM, que vai de 00:00:00 à 12:59:59, é só apertar a KEY3, botão da placa FPGA.

c. Despertador

O despertador é possível de ser feito ao apertar a KEY2. Ao apertar a KEY2, abre-se a configuração da hora a ser despertada. Consegue-se ir alterando as horas e para alterar os outros valores é só ir apertando a KEY2 novamente. O número é configurado pelas SW de 0 a 7, com o valor em binário.

Logo, se quer colocar 1 em algum dos HEX, é necessário mudar a SW 0, por exemplo.

Quando se chega na hora do despertador, o LED9 acende, demonstrando que funciona corretamente.

Para auxiliar, foram usadas outras posições da memória:

MEM[09]: Flag se tem despertador ou não

MEM[10]: Guarda número do despertável no hex2

MEM[11]: Guarda número do despertável no hex3

MEM[12]: Guarda número do despertável no hex4

MEM[13]: Guarda número do despertável no hex5

- Como testar o despertador?

De início, aperta-se a KEY2, botão da placa FPGA. Após isso, irá abrir a configuração dos minutos, ou seja, coloca-se o valor que os minutos deverão despertar. Para colocar esse valor, usa-se as SWs de 0 a 7 em forma de um número em bits.

São 8 chaves, representando 8 bits 00000000. Caso ligue por exemplo a SW0, a SW1 e a SW3, ficará na forma 00000111, ou seja, o número decimal 7.

Para confirmar que é 7 nos minutos, clica-se novamente na KEY2, indo para a configuração das horas, que funciona de forma semelhante. Ao finalizar, novamente aperta-se KEY2 e agora é só esperar o horário do despertador que, ao chegar, irá acender o LED9.

6. Mudança de Base do Tempo

Para especificar uma outra feature que pode ser utilizada, como uma forma de “Como testar” assim como foi feito para os extras, esse tópico é apenas para deixar registrado o funcionamento e de que forma alterar no relógio pela FPGA.

Essa mudança é feita por duas bases de tempo que foram implementadas. Um faz a divisão 250 milhões, representando os segundos normalmente. E 250 mil para dar uma velocidade maior no relógio.

Para realizar a mudança, é necessário mudar a SW9 de posição, para ligada, fazendo com que o relógio fique mais rápido.