Universidade Federal do Rio Grande do Norte
Centro de Tecnologia - CT
Departamento de Engenharia Elétrica

Relatório Técnico: Contador inteligente

ELE2715 - Grupo 03 - Implementação - Problema 03

Alysson Ferreira da Silva
Isaac de Lyra Junior
Lucas Batista da Fonseca
Vinicius Souza Fonsêca
Wesley Brito da Silva

Natal, 07 de março de 2021

RESUMO

O presente relatório tem como objetivo descrever o projeto e implementação de um contador inteligente. O contador proposto pela disciplina, mostrará algumas funções pertinentes às lógicas sequenciais como: contagem crescente e decrescente através da função up/down; faixas de valores controlados pela entrada de máximos e mínimos (max/min); passo de andamento de contagem através da função *step*, função clear (clr) ajustada para contar entre 0 a 999 com passo 1. Para a elaboração da lógica do contador foram utilizados flips flops D e alguns multiplexadores que auxiliam no tratamento dos valores de entradas e de saídas. Também foram utilizados comparadores de magnitudes para ver a relação dos bits da contagem com os bits desejados como limites máximos e mínimos. A utilização de alguns registradores (utilizando flip flops D) para o armazenamento dos valores max/min e dos *steps*, onde este último é, posteriormente, somado ou subtraído para que a contagem ocorra de forma progressiva ou regressiva.

Palavras-chaves: Circuito Digital; Registradores; Comparador de Magnitude; Contador Inteligente; Lógicas Sequenciais; Flip-Flop D; Somador Completo; Subtrator Completo.

SUMÁRIO

1 INTRODUÇÃO

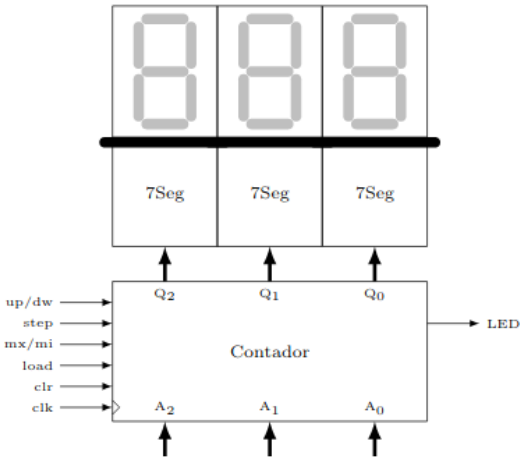
Até o presente momento, acompanhamos as lógicas implementadas em um circuito combinacional, cuja saída pode-se entender como uma função que depende diretamente das entradas desses circuitos. Por não poderem efetivar o armazenamento de bits, os circuitos combinacionais tornam-se limitados em determinados momentos, sendo assim, sugerimos como proposta de melhoria para determinados casos, o circuito sequencial, no qual as saídas dependem não somente das entradas atuais, mas também do seu estado atual, que é o conjunto de todos os bits armazenados no circuito.

Além disso, vale salientar que também deve-se levar em consideração o estado de condição futura, dessa forma, é possível prever as possibilidades que um circuito sequencial poderá descrever, auxiliando assim seu desenvolvimento dentro da função esperada. Dessa maneira, para resolver a referida problemática, serão utilizadas algumas técnicas provenientes da lógica sequencial, adaptando as ferramentas utilizadas da melhor maneira.

2 DESENVOLVIMENTO

O contador inteligente proposto na tarefa terá algumas particularidades que deverão fazer parte do seu desenvolvimento. Primeiramente tem-se 3 entradas: A2, A1 e A0, cada uma contendo 4 bits, para informar valores externos definidos pelo usuário. Além disso, tem-se Q2, Q1 e Q0 como saídas que contém 4 bits cada, representando o valor da contagem em BCD. O contador terá algumas funções que deverão ser implementadas. A função up/dw, realizará a ação de incrementar ou decrementar, neste caso em específico, quando esta função estiver em nível lógico alto o contador estará no modo crescente e quando a função estiver em nível lógico baixo o contador estará no modo decrescente. A função step determinará qual será o passo que o contador incrementa ou decrementa, esse passo será definido pelo usuário através da entrada A0 de 4 bits (essa função funcionará em conjunto com a função load). A função mx/mi definirá qual a faixa de valores que o contador vai operar, sendo esses valores definidos pelo usuário através das entradas A2, A1 e A0, e essa função também funcionará em conjunto com a função load. Para que seja implementada a função mx/mi, é necessário utilizar a lógica de um comparador de magnitude, o qual irá comparar o valor atual do contador com o valor definido do limite (máximo para crescente e mínimo para decrescente). A função clr quando estiver em nível lógico alto, o contador será ajustado para contar de 0 a 999 com passo de 1. Teremos uma saída visual que será representada por um LED que acenderá quando o contador atingir os valores máximos ou mínimos definidos. A função load propriamente dita, será responsável por carregar/definir a característica do contador quando em conjunto com determinada função supracitada. A Figura 1 apresenta o layout desenvolvido do contador inteligente.

Figura 1 - Contador inteligente



Fonte: Disponibilizado pelo orientador.

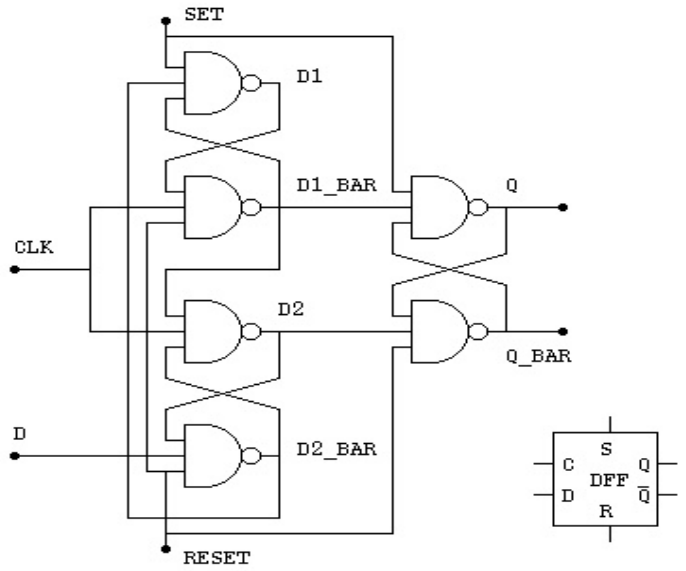
Para a solução da problemática é sugerido trabalhar por blocos, ou seja, pensar em cada função separadamente e fazer a junção desses blocos observando as condições estabelecidas. Dessa forma, o primeiro passo foi entender a lógica por trás de um contador e seus componentes primitivos.

2.1 Flip Flop

De acordo com VAHID, para construir um circuito sequencial, precisa-se de um bloco construtivo que capacite a armazenar um bit, sendo assim, o bloco ligado a essa característica são os flip flops, cada bloco é responsável pelo "manuseio" de um bit. O conjunto de flip flops será a base para o contador em questão, com isso, para a montagem dessa base é sugerido o uso de 10 flip flops do tipo D ligados em cascata.

O flip-flop D ou mais conhecido como flip-flop "data", é um circuito lógico onde apresenta função de armazenar dados de apenas um bit, em conjunto desse bloco de circuito, é possível construir circuitos mais complexos no qual pode armazenar mais dados de bits, como por exemplo os registradores. Onde o valor de entrada "D" é armazenado através da variação do clock, portanto se o nível lógico do clock estiver alto ocorre o armazenamento do bit no bloco, dependendo do valor de entrada a saída Q do circuito é monitorada, sendo que no primeiro ciclo se a entrada D=1 a saída do flip-flop será nível lógico alto no Q e se no próximo ciclo do clock o D=0 a saída será representada por Q/.

Figura 2 - Circuito lógico interno do flip-flop tipo D



Fonte: StackExchange

Na Figura 2 observa-se o esquemático do flip flop D, onde R e S recebem os termos de set e reset. Um ponto importante na configuração D é o caso de entradas com valores iguais, ou seja, sendo R=S=0, quando ativado o clock a saída não se modifica, contudo, quando R=S=1, temos um comportamento não permitido, pois as saídas serão Q e Q/ iguais a 1.

2.2 Função UP/DW

Com a base do contador já definida, pode-se então desenvolver as funções que são pedidas na situação-problema. Sendo assim, começaremos pela função up/dw. Essa função tem como objetivo fazer com que o contador se torne progressivo quando sua entrada estiver em nível lógico alto, em contrapartida fará com que o contador se torne regressivo quando seu sinal estiver em nível lógico baixo, dessa maneira, obtém-se como solução para a problemática, a disposição de como o clock vai para o próximo flip-flop.

Na Figura 3, no qual observa-se 3 flip-flops em cascata, pode-se perceber que a saída Q será a referência de clock para os próximos flip-flops, dessa forma tem-se uma contagem progressiva. Em contrapartida quando há a inversão dessa referência, ou seja, quando é enviado o Q para os clocks dos próximos flip-flops, o contador funciona de forma regressiva.

Sendo assim, é sugerido a utilização de um multiplexador 2x1 em cada saída dos flip flops, possibilitando a seleção do tipo de sinal que será enviado para o flip-flop seguinte. Ou seja, será selecionado 0 ou 1 e em seguida será direcionado ao clock do próximo flip-flop o sinal remetido para cada função.

2.3 Função STEP

Essa função tem como objetivo determinar o passo que o contador deve seguir de acordo com a entrada A0 de 4 bits. Por exemplo: se escolhermos a combinação de 4 bits em $0011_2 = 3_{10}$ e se o contador estiver trabalhando de forma crescente de 0 a 10, a sequência mostrada no display de 7 segmento será de 0, 3, 6, 9, começando novamente do zero até que o usuário mude o passo ou de função. A solução sugerida para implementar essa função foi a utilização da função soma e subtração.

2.4 Função MAX/MIN

Essa função delimita a faixa de operação em que o contador irá trabalhar, para isso será usado as entradas A2, A1, e A0 onde serão definidos tais valores. Para a definição de valores em um contador deve-se levar em consideração alguns fatores. Para esse contador em

específico numa condição de trabalho crescente, o contador segue de 0 a 999 e numa condição decrescente, de 999 a 0. O primeiro ponto é que para termos esse valor em numeração binária, precisamos de 10 bits, como mencionado inicialmente. Porém, teremos uma contagem de 0 a 1023, pois $2^{10}= 1024$, o que nos daria um gap indesejado e que precisaria ser tratado. Para que essa contagem excedente não ocorra, a solução pensada, foi o desenvolvimento de um comparador de magnitude de 10 bits, esse comparador receberia os 10 bits da contagem em tempo real e a compararia bit a bit com o valor definido através das entradas de A2, A1, A0, registradas no registrador de maximo e minimo. Assim, dependendo da função escolhida Up ou Down, o comparador de magnitude de 10 bits, mandaria um sinal para as entradas de set e para as entradas de reset.

Para as entradas de set e reset dos flip flops, vale salientar que elas têm o seguinte funcionamento: quando a função set estiver em nível lógico alto, a saída Q recebe o valor 1, neste caso como se estivéssemos "setando" a saída em 1. Da mesma forma para a função reset, porém, de lógica invertida, quando o sinal em reset for nível lógico alto, a saída Q receberá o valor de 0, como se estivéssemos "resetando" essa saída. Para o funcionamento, foi pensado em atribuir um multiplexador 2x1 em cada entrada set e reset de cada flip flop, que serão controlados pelo sinal que o comparador de magnitude enviar. A ideia é que o comparador envie dois sinais, um sinal que indique quando o valor escolhido for mínimo e outro sinal quando o valor escolhido for máximo. Dessa forma, para os multiplexadores que entram na entrada set teremos dois valores a serem "setados". Na entrada A do multiplexador teremos o valor 0 e na entrada B teremos o valor relacionado à formação do número 1110011111_2 que é 999_{10} invertido, em binário. Dessa forma, quando a contagem estiver regredindo e chegar ao valor mínimo, o sinal de magnitude relativo acionará os multiplicadores, fazendo os flip flops "setarem" o número 999 novamente. Da mesma forma para a entrada reset que receberá os valores relacionados quando a contagem chegar ao valor definido, e retornar para a posição inicial.

2.5 Função CLR

A função CLR tem como objetivo, fazer com que o contador seja ajustado para contar de 0 a 999 com o passo de 1, para que isso aconteça foi sugerido a inserção de multiplexadores, que quando acionado a função CLR, esses multiplexadores desabilitem as outras funções criadas, forçando assim a retirada de qualquer valor remanescente dessas funções. Outro ponto a ser salientado é que será necessário a inserção de multiplexadores que

forcem o valor 1110011111 na entrada do comparador de magnitude para que haja garantia de que 999 seja o valor máximo que o contador poderá acessar.

2.6 Função Load em Conjunto

A função load será responsável por "carregar" valores definidos pelo usuário, essa função trabalhará em conjunto com algumas das funções acima, quando a função load for acionada e STEP também, o valor definido na entrada A0 deverá ser guardado para que quando for escolhida uma função qualquer, esse valor seja usado para compor a função escolhida, como por exemplo: podemos deixar a função up habilitada (forçando a contagem progressiva), habilitamos também a função max/min e através da entrada A0 escolhemos um faixa de valor entre as combinações desses 4 bits, por exemplo 1111, ou seja, igual a 15, quando habilitamos a função load, esse valor será guardado num circuito específico; desabilitamos a função load e habilitamos a função step e definimos o passo de 0011, ou seja, igual a 3, na saída teremos uma contagem progressiva entre 0 e 15 com passo de 3.

2.X Correções do Projeto

Dentre as correções feitas no projeto, podemos citar a lógica do step, pois anteriormente foi projetado para ser necessário a utilização de um multiplicador que tinha como entradas o A0 armazenado no registrador e o número em binário do contador e saída o resultado da multiplicação entre esses dois números, dessa maneira havia um conflito com a função max/min quando o up/dw estava em nível lógico baixo, esse conflito ocorria pois o comparador de 10 bits faz a comparação entre o resultado da multiplicação e o limite inferior(podendo ser o definido pelo usuário ou o padrão 0_2, quando o valor de saída do multiplicador era inferior ao limite do comparador o set é ativado e o contador recebe 999_2, como o multiplicador utiliza o número do contador, a multiplicação logo após a esse set seria então $999_2 \; x \; step$, caso o step não fosse 1_2 essa multiplicação geraria um loop, para corrigir esta inconsistência, trocamos o multiplicador por somadores e subtratores. Outra modificação foi feita no comparador, utilizamos 3 CI's comparadores de 4 bits para a construção do comparador de 10 bits.

3 RESULTADOS

Softwares utilizados para a implementação do projeto:

- Quartus Prime Lite Edition, versão: 20.1.1
- Proteus 8 Professional, versão: 8.9

A relação de componentes utilizados para a implementação do projeto se encontra na tabela 1.

Tabela 1 - Componentes utilizados

OPERAÇÕES LÓGICAS	CI COMERCIAL UTILIZADO
Decodificador Bin-BCD 16 bits	Autoral
Decodificador Para Display 7 Segmentos	4511
Multiplexador 4 bits	74157
AND[2]	7408
OR[2]	7432
OR[3]	74HC4075
OR[4]	4072
NOT	7404
Flip Flop D	4013
Comparador De 4 Bits	74HC85

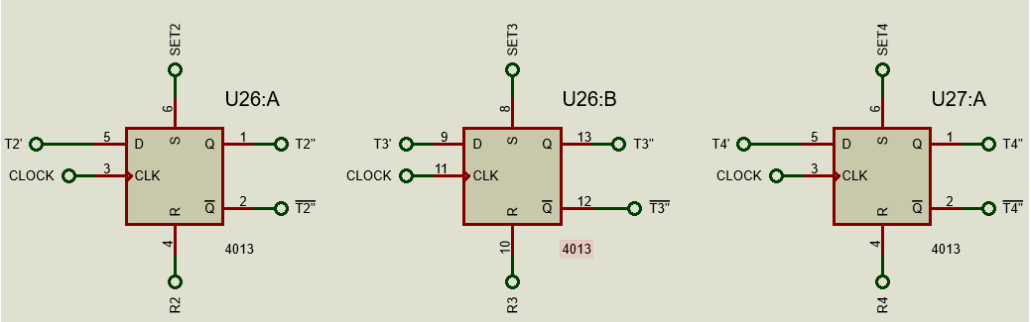
Vale salientar que para a lógica da implementação no VHDL foi levado em consideração exatamente o mesmo circuito combinacional do Proteus. Desta forma, não é explicado como cada entidade foi elaborada, pois seria redundante.

3.1 Flip Flop

Para a base do contador utilizaremos dez blocos de flip flops D ligados em sincronia, tendo então um total na saída de2^{10}, ou seja, podendo gerar números decimais até 1023, essa

configuração é necessária pois precisaremos uma contagem até 999_{10} que na base 2 ficaria, 1111100111_2.

Figura 3 - Flip flops D em síncrono.



Fonte: Autor

Na figura 3 observa-se um esquemático com três flip flops ligados no mesmo clock, vale salientar que a saída mais significativa se dá da direita para esquerda, assim temos T4'' como a mais significativa. No caso como usamos os flip flops do tipo D, cada vez que o clock é mudado, há atualização do sinal na saída, por exemplo: no estado inicial D=1 representa as saídas Q=0 e Q/=0 antes do pulso do clock, teremos o bit 000. Quando se dá pulso do clock o valor de entrada, dessa forma temos $T_{2''}$=1, $T_{3''}$=0 e $T_{4''}$=0, como o sentido de leitura é invertido temos na saída o bit 001. Observe que mesmo que a saída Q do flip flop que gera T2 seja igual a 1, não é o suficiente para que altere o estado do flip flop que gera T3', pois essa troca só dá no próximo impulso do clock pois a descida do clock não altera o valor da memória. Quando o segundo pulso do clock é dado, T3' recebe saída T2'' = 1, nessa mudança de sinal de 1 para 0, o flip flop que gera T3 é recebido, dessa forma teremos as saídas $T_{2''}$=0, $T_{3''}$=1 e $T_{4''}$=0, tendo o bit 010.

Figura 4 - Escrita do flip-flop do tipo D em VHDL

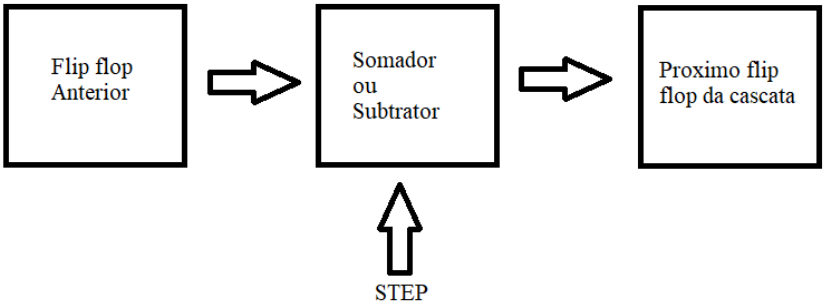
```
--==============--
-- FlipFlop D --
--==============--

ENTITY ffd IS
   port ( clk ,D ,P , C : IN BIT ;
      q : OUT BIT );
END ffd ;

ARCHITECTURE ckt OF ffd IS
   SIGNAL qS : BIT;
   BEGIN
PROCESS ( clk ,P ,C )
   BEGIN
   IF P = '0' THEN qS <= '1';
   ELSIF C = '0' THEN qS <= '0';
   ELSIF clk = '1' AND clk ' EVENT THEN
   qS <= D ;
   END IF;
END PROCESS ;
q <= qS ;
END ckt ;
```

Fonte: Autores.

3.2 Função UP/DW

A implementação do esquemático para esta função foi pensada utilizando circuitos somadores e subtratores. Para cada pulso no clock, o valor recebido pelo próximo flip-flop da cascata será o resultado da soma ou subtração do bit armazenado no flip-flop anterior com o passo definido pelo usuário, o fluxograma da cascata é apresentado na figura 4.

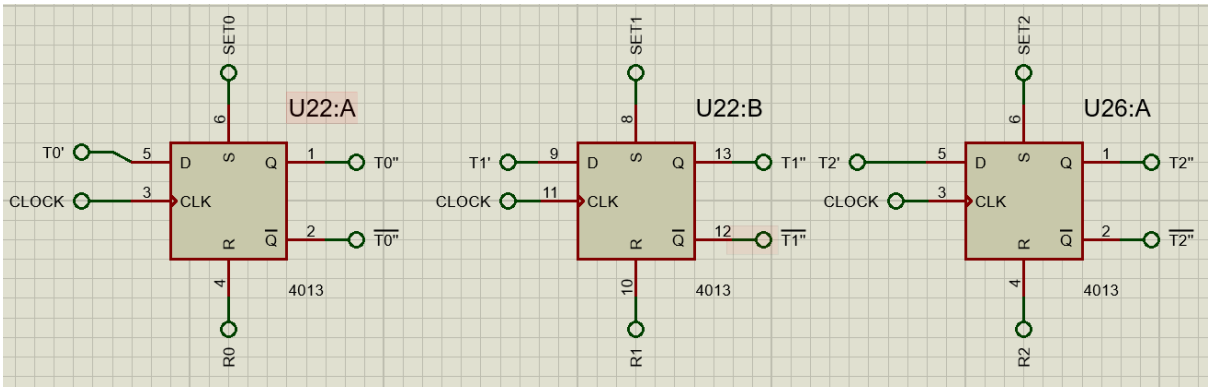
Figura 5 - Fluxograma da lógica da função *UP/DOWN* do contador



Fonte: Autores.

O resultado da implementação está na figura 6, onde os T''s representam o dado que irá ser mandado para o somador ou subtrator e os T's representam o resultado dessa soma ou subtração do bit armazenado no flip flop anterior com o *step* definido pelo usuário.

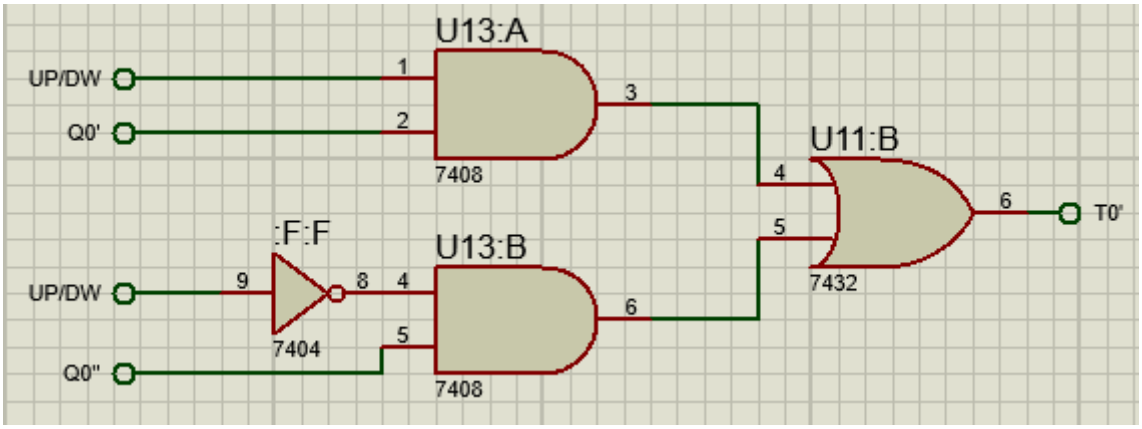
Figura 6 - Implementação no software Proteus da ligação em cascata dos flip flops



Fonte: Autores.

O circuito combinacional da figura 6 representa a lógica de um multiplexador 2x1 e foi utilizada para selecionar se contador iria crescer ou decrescer, as entradas Q's são os resultados provenientes do somador e as entradas Q''s são os resultados do subtrator, a chave seletora nesse caso é o *UP/DOWN*.

Figura 7 - Lógica para crescer ou decrescer da função *UP/DOWN* do contador



Fonte: Autores.

Figura 8 - Escrita da função UP/DOWN em VHDL.

```
entity FLIP10 is
   port(set,reset,soma,sub: in bit_vector (9 downto 0);
        up_down, clk: in bit;
         TLL: out bit_vector(9 downto 0));
end FLIP10;

architecture logic of FLIP10 is

component ffd is
   port ( clk ,D ,P , C : IN BIT;
      q: OUT BIT );
END component;

component MUX21 is
   port(A, B, S: in bit;
      O: out bit);
end component;

signal TL,TLLs:bit_vector(9 downto 0);

begin

mux1: MUX21 port map (sub(0),soma(0),up_down,TL(0));
mux2: MUX21 port map (sub(1),soma(1),up_down,TL(1));
mux3: MUX21 port map (sub(2),soma(2),up_down,TL(2));
mux4: MUX21 port map (sub(3),soma(3),up_down,TL(3));
mux5: MUX21 port map (sub(4),soma(4),up_down,TL(4));
mux6: MUX21 port map (sub(5),soma(5),up_down,TL(5));
mux7: MUX21 port map (sub(6),soma(6),up_down,TL(6));
mux8: MUX21 port map (sub(7),soma(7),up_down,TL(7));
mux9: MUX21 port map (sub(8),soma(8),up_down,TL(8));
mux10: MUX21 port map (sub(9),soma(9),up_down,TL(9));

flip1: ffd port map (clk,TL(0),set(0),reset(0),TLLs(0));
flip2: ffd port map (clk,TL(1),set(1),reset(1),TLLs(1));
flip3: ffd port map (clk,TL(2),set(2),reset(2),TLLs(2));
flip4: ffd port map (clk,TL(3),set(3),reset(3),TLLs(3));
flip5: ffd port map (clk,TL(4),set(4),reset(4),TLLs(4));
flip6: ffd port map (clk,TL(5),set(5),reset(5),TLLs(5));
flip7: ffd port map (clk,TL(6),set(6),reset(6),TLLs(6));
flip8: ffd port map (clk,TL(7),set(7),reset(7),TLLs(7));
flip9: ffd port map (clk,TL(8),set(8),reset(8),TLLs(8));
flip10: ffd port map (clk,TL(9),set(9),reset(9),TLLs(9));

TLL(9 downto 0) <= TLLs(9 downto 0);

end logic;
```

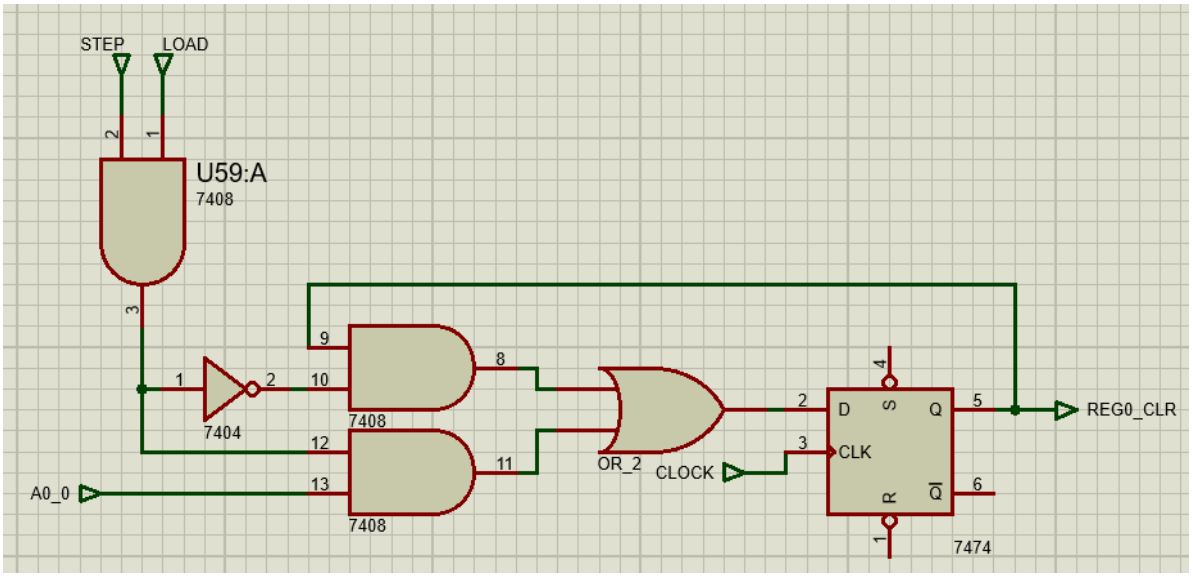
Fonte: Autores.

3.3 Função STEP

Para ser possível a implementação da função step utilizamos 4 registradores de 1 bit para armazenar os 4 bits de A0 que será utilizado como *step* do contador, o registrador registra um bit se, e somente se, houver alteração no clock e a função *step* e *load* estiverem

em nível lógico alto. Vale ressaltar que na configuração de Flip-Flops tipo D com *preset* (P) e *clear* (C), as entradas para essas duas variáveis são negadas. Por causa disso, quando houver chamada do Flip-Flop será atribuído às variáveis P e C nível lógico alto, dessa maneira a entrada para os Flip-Flops será 0. Além disso, é importante lembrar que nestes Flip-Flops apenas a variável de clock está atuando de maneira síncrona, de modo que todos os outros estão atuando sobre o princípio assíncrono. Na figura 9 é demonstrado o circuito combinacional do registrador de 1 bit implementado no Proteus.

Figura 9 - Registrador de 1 bit



Fonte: Autores.

A escrita do registrador de 4 bits em VHDL se encontra na figura 10, foi necessário utilizar o recurso de port map para implementar 4 Flip-Flops tipo D como componentes dentro da entidade. Para poder guardar as variáveis de interação entre os comparadores foi necessário criar 3 sinais extras: load_b0 (vetor de 4 bits), reg (vetor de 4 bits e load_step. O sinal load_b0 fica responsável pela lógica combinacional, isto é, para determinar qual será o valor da entrada D do flip-flop (bit a ser guardado). Já o sinal reg é o vetor de saída responsável pelo armazenamento de bits (saída dos registradores) que culmina na variável utilizada na atribuição de valores para a variável de saída da entidade.

Figura 10 - Escrita do registrador de 4 bits em VHDL

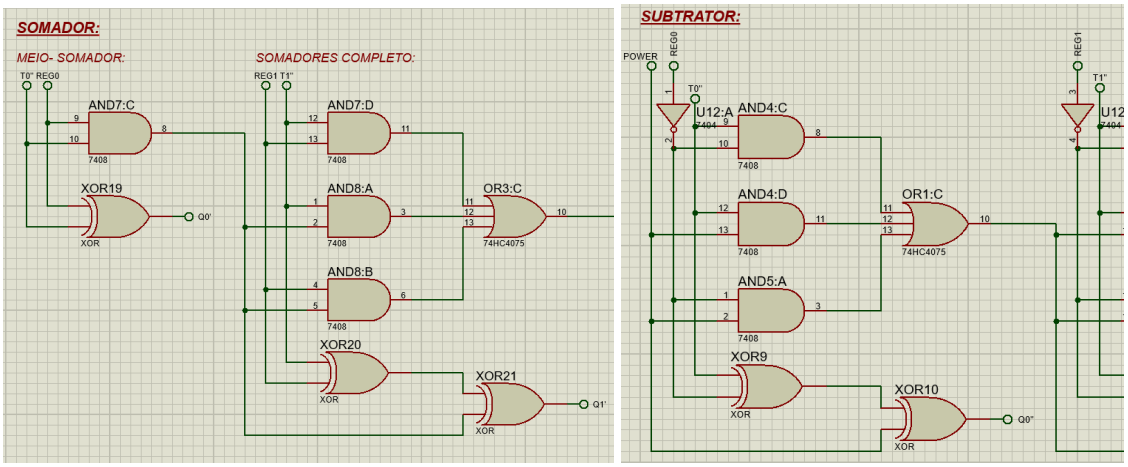
```
entity REG4 is
  port(b: in bit_vector(3 downto 0);
       loadd,clk,step: in bit;
       s_b: out bit_vector(3 downto 0)
      );
end REG4;

architecture logic of REG4 is

  component ffd is
    port ( clk ,D ,P , C : in bit ;
    q : out bit );
  end component;

    signal load_b0, reg:bit_vector(3 downto 0);
    signal load_step:bit;

    begin

    load_step <= loadd and step;

    load_b0(0) <= ((not load_step) and reg(0)) or (load_step and b(0));

    ffd1: ffd port map(clk,load_b0(0),'1','1',reg(0));

    load_b0(1) <= ((not load_step) and reg(1)) or (load_step and b(1));

    ffd2: ffd port map(clk,load_b0(1),'1','1',reg(1));

    load_b0(2) <= ((not load_step) and reg(2)) or (load_step and b(2));

    ffd3: ffd port map(clk,load_b0(2),'1','1',reg(2));

    load_b0(3) <= ((not load_step) and reg(3)) or (load_step and b(3));

    ffd4: ffd port map(clk,load_b0(3),'1','1',reg(3));

    s_b(0) <= reg(0);
    s_b(1) <= reg(1);
    s_b(2) <= reg(2);
    s_b(3) <= reg(3);

end logic;
```

Fonte: Autores.

Os bits armazenados no registrador são utilizados como entrada nos 4 primeiros somadores do somador e subtrator de 10 bits do contador, na figura 9 é possível ver as entradas do somador e subtrator. Por exemplo: caso eu armazene $0010_2 = 2_{10}$ no meu registrador e ative o up/dw, o contador no primeiro pulso receberá $0_2(contador) + 0010_2(step) = 0010_2 = 2_{10}$, no segundo pulso receberá $10_2(contador) + 0010_2(step) = 100_2 = 4_{10}$ e assim sucessivamente, o mesmo vale para o subtrator caso a up/dw esteja em nível lógico baixo.

Figura 11 - Primeiros somadores do somador e subtrator: (a) somador, (b) subtrator.

(a) (b)

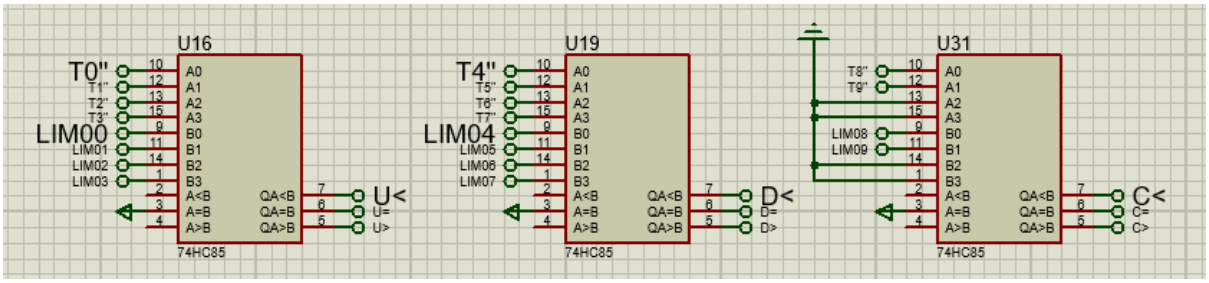


Fonte: Autores.

3.4 Função MAX/MIN

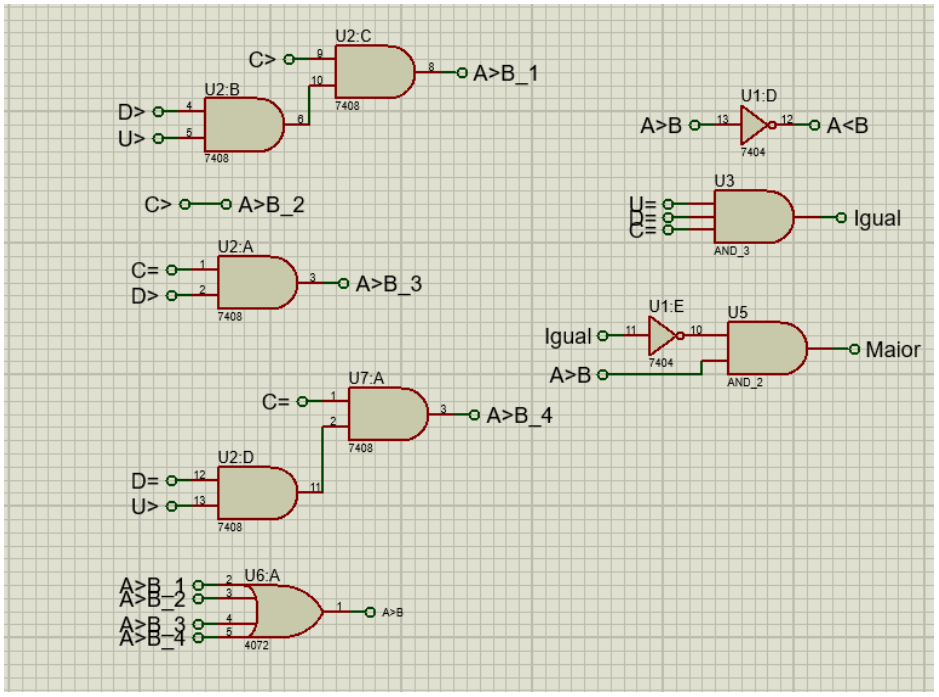
Para implementar a lógica do comparador de magnitude, foi utilizado três comparadores 74HC85 em que cada um realiza a comparação de 4 bits da saída do flip-flop e o valor de máximo e mínimo determinado, dessa comparação fizemos uma lógica associando as saídas dos comparadores 74HC85 para determinar se os bits dos flip-flop's são maior, igual ou menor do que os limites. Essa lógica de comparação pode ser vista na figura a seguir.

Figura 12 - Comparador de Magnitude de bit a bit



Fonte: Autores.

Figura 13 - Lógica de comparação dos 3 comparadores

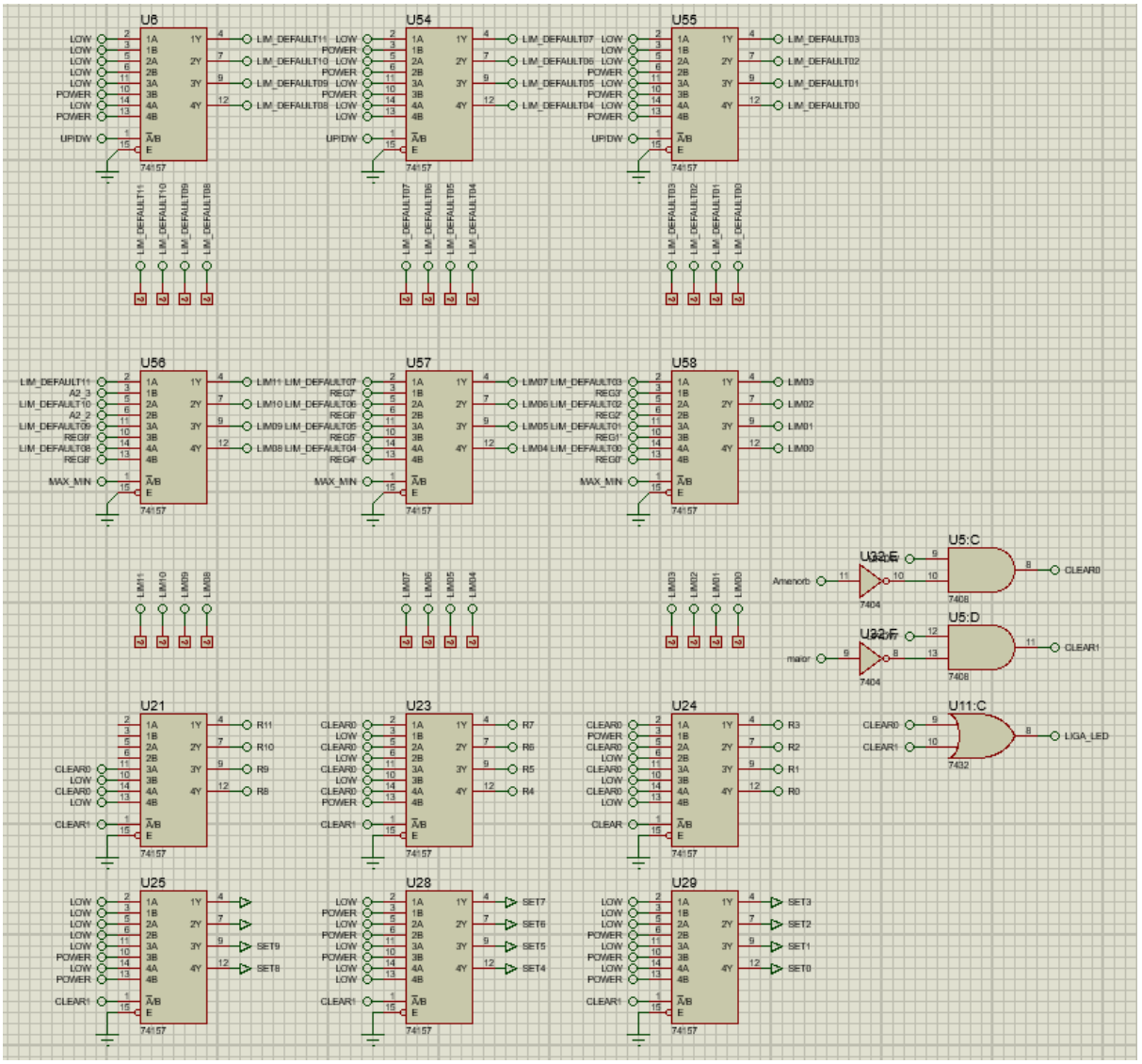


Fonte: Autores.

Note que os comparadores de magnitude de 4 bit (figura 13) possuem 8 valores de 1 bit que são os bits , T'' e LIM. Resultado em 3 comparações: U, D e C, a figura 6 mostra como essas saídas foram tratadas para que no final tenha uma 3 saídas, T''=LIM ,T''>LIM e T''<LIM, em que denominamos Maior para T>LIM, A<B para T<LIM e Igual para T=LIM.

Os valores de LIM são obtidos através de multiplexadores de 2x1, primeiramente foi utilizado 10 mux 2x1 em que a chave seletora é UP/DW, esses mux definem o máximo e mínimo global, ou seja, se UP/DW estiver com valor 1 lógico, então o limite global é 999, e o inverso é 000.

Figura 14 - Comparador MX/MI



Fonte: Autores

Figura 15 - Comparador VHDL

```
entity COMPARADOR is
  port(a,b: in bit_vector(11 downto 0);
       igual,maior, menor: out bit);
end COMPARADOR;

architecture logic of COMPARADOR is

component COMP4 is
  port(a,b: in bit_vector(3 downto 0);
       Sa_Igual_b, Sa_maior_b, Sa_menor_b: out bit);
end component;

signal a_vetor_centena, a_vetor_dezena, a_vetor_unidade,b_vetor_centena, b_vetor_dezena, b_vetor_unidade:bit_vector(3 downto 0);
signal saida_centena, saida_dezena, saida_unidade: bit_vector (2 downto 0);
signal AmaiorB_1, AmaiorB_2, AmaiorB_3, AmaiorB_4:bit;
signal maiors1, iguals:bit;

begin
  a_vetor_centena(3 downto 0) <= a(11 downto 8);
  b_vetor_centena(3 downto 0) <= b(11 downto 8);
  a_vetor_dezena(3 downto 0) <= a(7 downto 4);
  b_vetor_dezena(3 downto 0) <= b(7 downto 4);
  a_vetor_unidade(3 downto 0) <= a(3 downto 0);
  b_vetor_unidade(3 downto 0) <= b(3 downto 0);

   COMP1: COMP4 port map( a_vetor_centena,b_vetor_centena,saida_centena(2),saida_centena(1),saida_centena(0));
   COMP2: COMP4 port map( a_vetor_dezena,b_vetor_dezena,saida_dezena(2),saida_dezena(1),saida_dezena(0));
   COMP3: COMP4 port map( a_vetor_unidade,b_vetor_unidade,saida_unidade(2),saida_unidade(1),saida_unidade(0));

  AmaiorB_1 <= ((not saida_centena(0)) and ((not saida_dezena(0)) and (not saida_unidade(0))));
  AmaiorB_2 <= saida_centena(1);
  AmaiorB_3 <= saida_centena(2) and saida_dezena(1);
  AmaiorB_4 <= (saida_centena(2) and (saida_dezena(2) and saida_unidade(1)));

  igual <= saida_centena(2) and saida_dezena(2) and saida_unidade(2);
  iguals <= saida_centena(2) and saida_dezena(2) and saida_unidade(2);
  maiors1 <= (AmaiorB_1 or AmaiorB_2 or AmaiorB_3 or AmaiorB_4);
  maior <= maiors1 and (not iguals);
  menor <= not maiors1;

end logic;
```

Fonte : Autores

Segundamente foi realizado mais 10 mux que possui como chave seletora MI/MX, e as entradas são: limites global e os valores guardados no load do MI/MX. quando MI/MX estiver com valor lógico 1, o LIM vai receber o valor que está armazenado no load do MI/MX. Se o MI/MX estiver com valor 0 lógico, LIM vai receber o valor dos limites globais.

Após esses passos, é necessário definir os valores de SET e RESET dos flip-flop's para que a contagem inicie de 000 ou 999 e que eles devem resetar a contagem quando passar dos limites. Para realizar a lógica dos limites, foi utilizado 10 mux 2x1, tendo como chave seletora a saída de uma AND que verifica se a contagem é crescente e se o número passou do limite, e outros 10 mux 2x1 que tem como chave seletora a saída de outra porta AND que verifica se a contagem é decrescente e se o número é menor que o limite. Se a contagem for crescente e o número não é maior que o limite o set e reset dos flip-flop's recebem valor 0 lógico, quando o número passa do limite, é enviado um pulso de 1 lógico para os resets dos flip-flop's assim zerando a contagem. De forma similar, quando a contagem é decrescente e o número é maior que o limite, os sets e os resets recebem o lógico, entretanto quando o número de contador fica menor que o limite, set recebe um pulso de 1 lógico fazer que set 0,1,2,5,6,7,8,9 recebem 1 lógico e set3e4 recebam 0 lógico, e que os resets recebam os valor dos set negados.

Por fim a lógica para que o valor inicie em 999 quando a contagem seja decrescente é similar a lógica anteriormente explicitada, quando UP/DW for 0 set irá receber o valor 999 em binário e reset irá receber $\overline{999}$, dessa forma iniciando a contagem em 999. O código no VHDL para a função de máximo e mínimo está na figura 15.

Figura 16 - Máximo e Mínimo VHDL

```
entity MAX_MIN is
   port( reg: in bit_vector(9 downto 0);
   up_down, mxmi: in bit;
   lim: out bit_vector (9 downto 0));
end MAX_MIN;

architecture logic of MAX_MIN is

component MUX21 is
   port(A, B, S: in bit;
      O: out bit);
end component;

signal lim_default: bit_vector(9 downto 0);

begin
   mux1: MUX21 port map ('0','1',up_down,lim_default(9));
   mux2: MUX21 port map ('0','1',up_down,lim_default(8));
   mux3: MUX21 port map ('0','1',up_down,lim_default(7));
   mux4: MUX21 port map ('0','1',up_down,lim_default(6));
   mux5: MUX21 port map ('0','1',up_down,lim_default(5));
   mux6: MUX21 port map ('0','0',up_down,lim_default(4));
   mux7: MUX21 port map ('0','0',up_down,lim_default(3));
   mux8: MUX21 port map ('0','1',up_down,lim_default(2));
   mux9: MUX21 port map ('0','1',up_down,lim_default(1));
   mux10: MUX21 port map ('0','1',up_down,lim_default(0));

   mux1: MUX21 port map (lim_default(9),reg(9),mxmi,lim(9));
   mux2: MUX21 port map (lim_default(8),reg(8),mxmi,lim(8));
   mux3: MUX21 port map (lim_default(7),reg(7),mxmi,lim(7));
   mux4: MUX21 port map (lim_default(6),reg(6),mxmi,lim(6));
   mux5: MUX21 port map (lim_default(5),reg(5),mxmi,lim(5));
   mux6: MUX21 port map (lim_default(4),reg(4),mxmi,lim(4));
   mux7: MUX21 port map (lim_default(3),reg(3),mxmi,lim(3));
   mux8: MUX21 port map (lim_default(2),reg(2),mxmi,lim(2));
   mux9: MUX21 port map (lim_default(1),reg(1),mxmi,lim(1));
   mux10: MUX21 port map (lim_default(0),reg(0),mxmi,lim(0));


end logic;
```

Fonte: Autores.

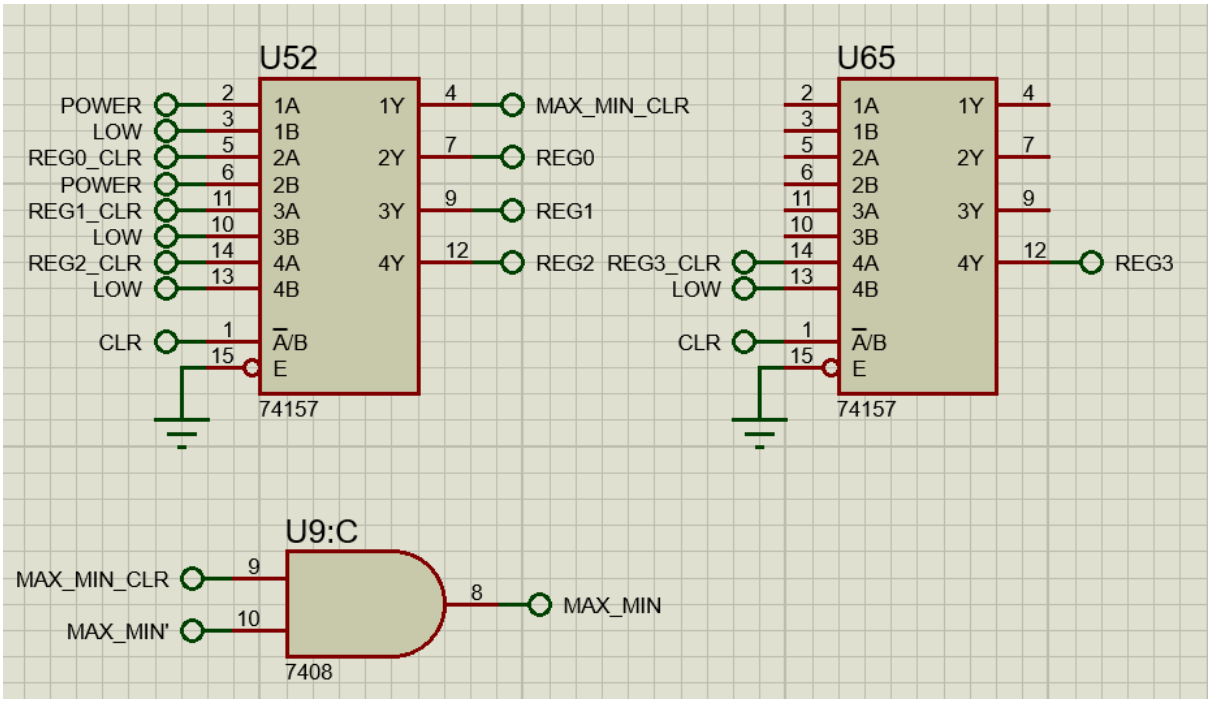
3.5 Função CLR

Na implementação do CLR foi utilizado 1 mux 2x1 que possui CLR como chave, se a chave CLR receber 1, MI/MX_CLR recebê 0 e o oposto se CLR receber 0, esse valor de MI/MX_CLR é usando em uma porta AND junto com o valor de MI/MX, dessa forma se MI/MX_CLR for 0, os limites serão os globais, se CLR tiver desligado e MI/MX estiver ligado, os limites serão os que estão guardados no load do MI/MX.

Para definir o passo sempre sendo 1 sempre que o CLR foi acionado, foi utilizado 4 mux 2x1 que possui como chave seletora o CLR em que, se o CLR estiver desligado o valor do passo é o que está guardado no load do STEP, entretanto, se CLR estiver ligado o valor do passo vai ser 1.

A função CLR foi implementada com o CI 74157 que possui 4 mux 2x1 internamente.

Figura 17 - Mux da Função CLR.



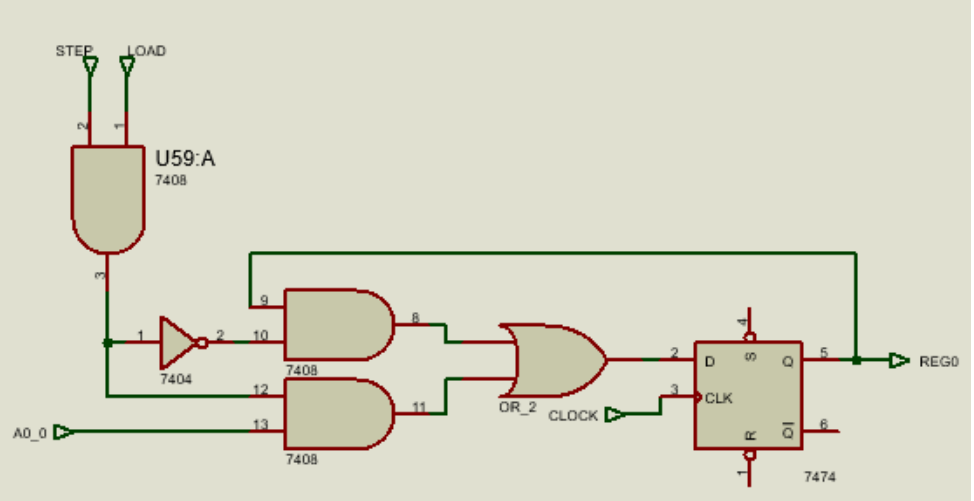
Função: Autores.

3.6 Função Load em conjunto

Para o armazenamento desses valores foi usado como principal componente o flip flop do tipo D, utilizamos 4 blocos de uma lógica combinacional.

Cada bloco receberá o valor dos bits da entrada A0 que quando habilitado a função load, esse valor se encaminha para a saída Q do flip flop, e lá fica armazenada e essa saída Q é definida como sinal REG, nesse load é definido os passos. Na figura abaixo podemos ver o bloco para o armazenamento de 1 dos passos.

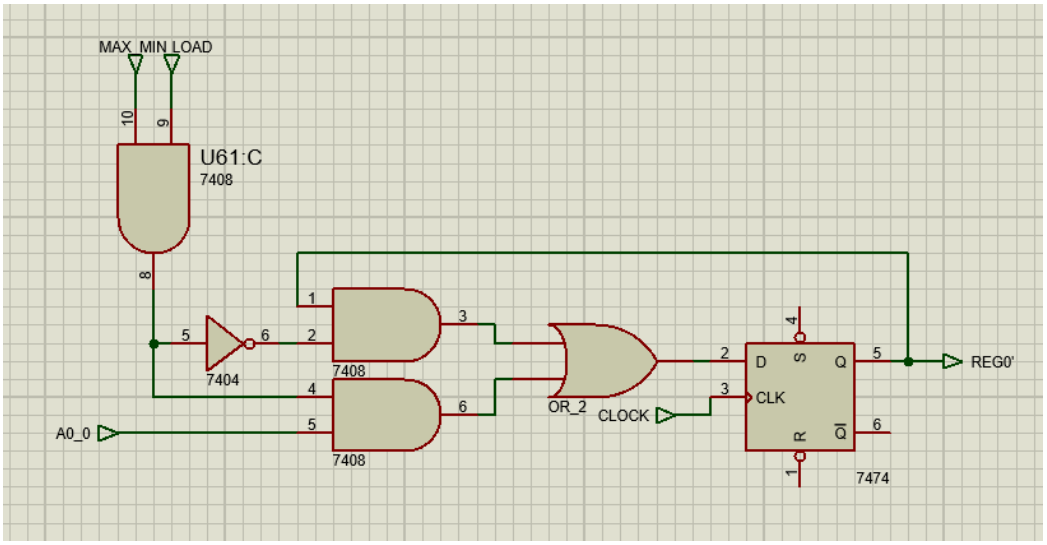
Figura 18 - Bloco de armazenamento de um bit para os passos.



Fonte: Autores

Para o load dos limites é feito a mesma lógica, entretanto usando os valores de A0, A1 e A2, e esses sinais serão encaminhados às entradas do comparador de magnitude para compor o valor máximo ou mínimo, dependendo da escolha.

Figura 19 - Bloco de armazenamento de um bit para os limites.



Fonte: Autores

4 CONCLUSÃO

O presente trabalho propôs projetar e a implementar um Contador Inteligente de entrada 12 bits que tem capacidade para realizar incremento e decremento do valor dado ao usuário, sendo estas controladas por 3 chaves seletoras (número binário) de 4 bits. Os Flip flops foram capazes de realizar as contagens de forma correta, entretanto o módulo responsável pela multiplicação teve problema no momento da ativação do step.

As unidades “somador” e “subtrator” foram sendo substituídas no lugar da multiplicação, o que resultou em simplificar a resposta ser demonstrada no display.

Neste trabalho foi utilizado o componente base da lógica sequencial, o flip flop. Dentre os modelos pesquisados, no início foi utilizado o flip flop tipo JK, mas depois decidimos ir ao tipo D para facilitar o desempenho da base do contador, e ligados de forma síncrona, o que resultou na mudança de estado mais rápida. Uma das grandes dificuldades deste projeto foi a integração de todas as funções em um único contador, as ideias sofreram modificações nas ideias iniciais para melhor adaptação dos componentes, mas o pensamento foi tentar agregar os conhecimentos obtidos com a lógica combinacional para implementação do projeto demonstrado.

REFERÊNCIAS

VAHID, F. **Sistemas digitais:** projeto, otimização e HDLs. Porto Alegre: Artmed, 2008. 560p.

http://www.mecaweb.com.br/eletronica/content/e_flip_flop. Acesso em 23 de fevereiro de 2021.

http://www.univasf.edu.br/~romulo.camara/novo/wp-content/uploads/2013/07/Aula9_10_Circuitos_sequenciais.pdf.pdf. Acesso em 23 de fevereiro de 2021.

ANEXOS

ANEXO A - RELATÓRIO SEMANAL

Líder: Alysson Ferreira da Silva

A.1 Equipe

Tabela 1 - Identificação da equipe

Funções	Componentes
Redator	Isaac de Lyra
Debatedor	Lucas Batista
Videomaker	Wesley Brito
Auxiliar	Vinicius Souza

Fonte: Elaborado pelos autores.

A.2 Defina o problema

Para o terceiro projeto da disciplina de Circuitos Digitais (Teoria), a ideia é projetar um circuito capaz de implementar um contador inteligente. Esse contador deve possuir as seguinte funcionalidades: contagem crescente ou decrescente; deve ser possível estabelecer um limite máximo ou mínimo durante a contagem; deve ser possível estabelecer um passo para a contagem; os limites de operação do contador vai de 0 a 999.

São estabelecidas as portas de entrada e saída que esse componente contador deve possuir. São elas:

- Entradas: up/dw;mx/mi; step; clr;load;clk; A2;A1 e A0;
- Saídas: Q0;Q1;Q2 e um led.

As saídas do contador irão ser direcionadas para três displays de 7 segmentos, que serão responsáveis por mostrar os valores das unidades, dezenas e centenas do seu atual estado.

O presente trabalho trata da implementação da proposta sugerida pelo grupo anterior.

A.3 Registro do brainstorming

A rotina de reuniões do grupo seguiu conforme exposto pela tabela quadro 1. Porém a elaboração do esquemático levou mais tempo do que o programado se estendendo até o domingo. Por causa disso, para não haver disparidade entre VHDL e esquemático, os esforços foram remanejados para que todos ajudassem prioritariamente na elaboração do esquemático.

Quadro 1 - Rotina de reuniões.

	Segunda	Terça	Quarta	Quinta	Sexta
M3					
M4					
M5					
M6					
T1					
T2					
T3					
T4					
T5			Implementação em VHDL	Implementação do Esquemático	Implementação do Esquemático
T6					
N1		Reunião de Alinhamento e divisão de tarefas	Implementação em VHDL e Esquemático	Implementação do Esquemático	Implementação do Esquemático
N2					
N3					
N4					

Fonte: Elaborado pelos autores.

Até o sábado, o grupo insistiu na implementação do contador inteligente com a abordagem proposta no relatório da semana. Mas, a multiplicação demonstrou alguns erros que tornaram a implementação mais difícil. A exemplo disso, o reset e o set quando setados, estes valores eram comparados com o display para saber se o contador deveria resetar ou não. Porém, o valor disponível para a comparação era sempre o valor da contagem anterior vezes o passo e por causa disso a exibição em caso decrescente sempre estava errada.

Após desistir da implementação, o grupo decidiu implementar um contador com somas e subtrações, que demonstrou ser uma maneira mais fácil e ágil de implementação.

A.4 Pontos-chaves

O ponto chave deste relatório foi compreender o efeito de circuitos sequenciais na implementação de projetos e uso dos FlipFlop do tipo D para a memorização de dados. Dito isso, trabalhar o delay de informações e uso de entradas e saídas que eram destinadas a lugares diferentes ao mesmo tempo sem dar conflito foi crucial para a excelência do grupo na implementação do projeto do contador inteligente.

A.5 Questões de pesquisa

- Flip Flops JK;
- Flip Flops D;

A.6 Planejamento da pesquisa

A pesquisa foi realizada através da leitura do livro “Sistemas digitais: projeto, otimização e HDLS” de Frank Vahid, bem como vídeos no YouTube.

ANEXO B - Código em VHDL

```
--======================--
-- DISPLAY 7 SEGMENTOS --
--======================--

entity D7SEG is
  port (I: in bit_vector (3 downto 0);
        O: out bit_vector (6 downto 0));      -- Vetor de saidas, com 7 posicoes, que representa os 7 segmentos do display que serão acesos
end D7SEG;

architecture logic of D7SEG is
  signal A, B, C, D, NA, NB, NC, ND: bit;
begin
  A <= I(3);
  B <= I(2);
  C <= I(1);
  D <= I(0);

  NA <= not A;
  NB <= not B;
  NC <= not C;
  ND <= not D;

  O(0) <= C or A or (NB and ND) or (B and D);

  O(1) <= NB or (NC and ND) or (C and D);

  O(2) <= NC or D or B;

  O(3) <= (NB and ND) or (NB and C) or (C and ND) or (B and NC and D);

  O(4) <= (NB and ND) or (C and ND);
```

```
   O(5) <= A or (NC and ND) or (B and NC) or (B and ND);

   O(6) <= A or (NB and C) or (C and ND) or (B and NC);

end logic;


--================--
-- BLOCO Bin-BCD --
--================--

entity bloco_bin_bcd is
   port (A: in bit_vector (3 downto 0);
                  S: out bit_vector(3 downto 0));
end bloco_bin_bcd;

architecture logic of bloco_bin_bcd is
begin

    S(0) <= (A(3) and (not A(0))) or ((not A(3)) and (not A(2)) and A(0)) or (A(2) and A(1)
and (not A(0)));

   S(1) <= ((not A(2)) and A(1)) or (A(1) and A(0))  or (A(3) and (not A(0)));

   S(2) <= (A(3) and A(0)) or (A(2) and (not A(1))  and (not A(0)));

   S(3) <= A(3) or (A(2) and A(0)) or (A(2) and A(1));

end logic;


--====================================--
-- DECODIFICADOR Bin-BCD (16 bits) --
```

```
--==================================--

entity decod_bcd_16bits is
	port(bin: in bit_vector(9 downto 0);
		bcd: out bit_vector (15 downto 0));
end decod_bcd_16bits;

architecture logic of decod_bcd_16bits is

component bloco_bin_bcd
	port(A: in bit_vector(3 downto 0);
			S: out bit_vector(3 downto 0));
end component;

	signal B1_A, B1_S, B2_A, B2_S, B3_A, B3_S, B4_A, B4_S,
		B5_A, B5_S, B6_A, B6_S, B7_A, B7_S, B8_A, B8_S, B9_A,
		B9_S,B10_A, B10_S, B11_A, B11_S, B12_A, B12_S, B13_A,
		B13_S, B14_A, B14_S, B15_A, B15_S, B16_A, B16_S, B17_A,
		B17_S, B18_A, B18_S, B19_A, B19_S, B20_A, B20_S: bit_vector(3 downto
0);

begin
	B1_A(3) <= '0';
	B1_A(2) <= '0';
	B1_A(1) <= '0';
	B1_A(0) <= '0';

	B1: bloco_bin_bcd port map(B1_A, B1_S);

	B2_A(3) <= B1_S(2);
	B2_A(2) <= B1_S(1);
	B2_A(1) <= B1_S(0);
	B2_A(0) <= bin(9);
```

```
B2: bloco_bin_bcd port map(B2_A, B2_S);

B3_A(3) <= B2_S(2);
B3_A(2) <= B2_S(1);
B3_A(1) <= B2_S(0);
B3_A(0) <= bin(8);

B3: bloco_bin_bcd port map(B3_A, B3_S);

B4_A(3) <= B3_S(2);
B4_A(2) <= B3_S(1);
B4_A(1) <= B3_S(0);
B4_A(0) <= bin(7);

B4: bloco_bin_bcd port map(B4_A, B4_S);

B5_A(3) <= '0';
B5_A(2) <= B1_S(3);
B5_A(1) <= B2_S(3);
B5_A(0) <= B3_S(3);

B5: bloco_bin_bcd port map(B5_A, B5_S);

B6_A(3) <= B4_S(2);
B6_A(2) <= B4_S(1);
B6_A(1) <= B4_S(0);
B6_A(0) <= bin(6);

B6: bloco_bin_bcd port map(B6_A, B6_S);

B7_A(3) <= B5_S(2);
B7_A(2) <= B5_S(1);
B7_A(1) <= B5_S(0);
B7_A(0) <= B4_S(3);
```

```
B7: bloco_bin_bcd port map(B7_A, B7_S);

B8_A(3) <= B6_S(2);
B8_A(2) <= B6_S(1);
B8_A(1) <= B6_S(0);
B8_A(0) <= bin(5);

B8: bloco_bin_bcd port map(B8_A, B8_S);

B9_A(3) <= B7_S(2);
B9_A(2) <= B7_S(1);
B9_A(1) <= B7_S(0);
B9_A(0) <= B6_S(3);

B9: bloco_bin_bcd port map(B9_A, B9_S);

B10_A(3) <= B8_S(2);
B10_A(2) <= B8_S(1);
B10_A(1) <= B8_S(0);
B10_A(0) <= bin(4);

B10: bloco_bin_bcd port map(B10_A, B10_S);

B11_A(3) <= B9_S(2);
B11_A(2) <= B9_S(1);
B11_A(1) <= B9_S(0);
B11_A(0) <= B8_S(3);

B11: bloco_bin_bcd port map(B11_A, B11_S);

B12_A(3) <= B10_S(2);
B12_A(2) <= B10_S(1);
B12_A(1) <= B10_S(0);
```

```
B12_A(0) <= bin(3);

B12: bloco_bin_bcd port map(B12_A, B12_S);

B13_A(3) <= B11_S(2);
B13_A(2) <= B11_S(1);
B13_A(1) <= B11_S(0);
B13_A(0) <= B10_S(3);

B13: bloco_bin_bcd port map(B13_A, B13_S);

B14_A(3) <= B5_S(3);
B14_A(2) <= B7_S(3);
B14_A(1) <= B9_S(3);
B14_A(0) <= B11_S(3);

B14: bloco_bin_bcd port map(B14_A, B14_S);

B15_A(3) <= B12_S(2);
B15_A(2) <= B12_S(1);
B15_A(1) <= B12_S(0);
B15_A(0) <= bin(2);

B15: bloco_bin_bcd port map(B15_A, B15_S);

B16_A(3) <= B13_S(2);
B16_A(2) <= B13_S(1);
B16_A(1) <= B13_S(0);
B16_A(0) <= B12_S(3);

B16: bloco_bin_bcd port map(B16_A, B16_S);

B17_A(3) <= B14_S(2);
B17_A(2) <= B14_S(1);
```

```
B17_A(1) <= B14_S(0);
B17_A(0) <= B13_S(3);

B17: bloco_bin_bcd port map(B17_A, B17_S);

B18_A(3) <= B15_S(2);
B18_A(2) <= B15_S(1);
B18_A(1) <= B15_S(0);
B18_A(0) <= bin(1);

B18: bloco_bin_bcd port map(B18_A, B18_S);

B19_A(3) <= B16_S(2);
B19_A(2) <= B16_S(1);
B19_A(1) <= B16_S(0);
B19_A(0) <= B15_S(3);

B19: bloco_bin_bcd port map(B19_A, B19_S);

B20_A(3) <= B17_S(2);
B20_A(2) <= B17_S(1);
B20_A(1) <= B17_S(0);
B20_A(0) <= B16_S(3);

B20: bloco_bin_bcd port map(B20_A, B20_S);

bcd(15) <= '0';
bcd(14) <= B14_S(3);
bcd(13) <= B17_S(3);
bcd(12) <= B20_S(3);
bcd(11) <= B20_S(2);
bcd(10) <= B20_S(1);
bcd(9) <= B20_S(0);
bcd(8) <= B19_S(3);
```

```
        bcd(7) <= B19_S(2);
        bcd(6) <= B19_S(1);
        bcd(5) <= B19_S(0);
        bcd(4) <= B18_S(3);
        bcd(3) <= B18_S(2);
        bcd(2) <= B18_S(1);
        bcd(1) <= B18_S(0);
        bcd(0) <= bin(0);

end logic;



--===============--
-- MEIO SOMADOR --
--===============--

entity HALF_ADD is
  port(A, B: in bit;
                S, CO: out bit);
end HALF_ADD;

architecture logic of HALF_ADD is

begin
  S <= A xor B;
  CO <= A and B;
end logic;



--====================--
-- SOMADOR COMPLETO --
--====================--

entity COMP_ADD is
```

```
  port(A, B, CI: in bit;
                  S, CO: out bit);
end COMP_ADD;

architecture logic of COMP_ADD is

begin
   S <= A xor B xor CI;
   CO <= (B and CI) or (A and CI) or (A and B);
end logic;



--====================--
-- SOMADOR (10 bits) --
--====================--

entity ADD10 is
  port(A, B: in bit_vector(9 downto 0);
                  O: out bit_vector(9 downto 0);
     CO: out bit);
end ADD10;

architecture logic of ADD10 is

component HALF_ADD is
  port(A, B: in bit;
                  S, CO: out bit);
end component;

component COMP_ADD
  port(A, B, CI: in bit;
                  S, CO: out bit);
end component;
```

```
signal VAI_UM: bit_vector(8 downto 0);

begin
   S0: HALF_ADD port map(A(0), B(0), O(0), VAI_UM(0));
   S1: COMP_ADD port map(A(1), B(1), VAI_UM(0), O(1), VAI_UM(1));
   S2: COMP_ADD port map(A(2), B(2), VAI_UM(1), O(2), VAI_UM(2));
   S3: COMP_ADD port map(A(3), B(3), VAI_UM(2), O(3), VAI_UM(3));
   S4: COMP_ADD port map(A(4), B(4), VAI_UM(3), O(4), VAI_UM(4));
   S5: COMP_ADD port map(A(5), B(5), VAI_UM(4), O(5), VAI_UM(5));
   S6: COMP_ADD port map(A(6), B(6), VAI_UM(5), O(6), VAI_UM(6));
   S7: COMP_ADD port map(A(7), B(7), VAI_UM(6), O(7), VAI_UM(7));
         S8: COMP_ADD port map(A(8), B(8), VAI_UM(7), O(8), VAI_UM(8));
         S9: COMP_ADD port map(A(9), B(9), VAI_UM(8), O(9), CO);

end logic;



--=================--
-- Meio Subtrator --
--=================--

entity MSUB is
  port(a, b : in bit;
       bo, s: out bit
     );
end MSUB;

architecture logic_MSUB of MSUB is

  begin
    bo <= (not(a) and b);
    s <= (a xor b);

end logic_MSUB;
```

```
--====================--
-- Subtrator Completo --
--====================--

entity SUBC is
  port(a, b, ci: in bit;
       bo, s: out bit
     );
end SUBC;

architecture logic_SUBC of SUBC is

  begin
    bo <= ((not(a) and ci) or (b and ci) or (not(a) and b));
    s <= (a xor b xor ci);

end logic_SUBC;

--====================--
-- Subtrator (10 bits) --
--====================--

entity SUB is
  port(A, B: in bit_vector(9 downto 0);
       O: out bit_vector(9 downto 0);
       CO: out bit);
end SUB;

architecture hardware of SUB is

component COMP_ADD
    port(A, B, CI: in bit;
           S, CO: out bit);
```

```
end component;

signal VAI_UM: bit_vector(8 downto 0);
signal AUX_B: bit_vector(9 downto 0);

begin
    -- Como  subtrator, ent o inverte B e soma 1 (VAI_UM='1'). (Complemento de 2: A-B=A+B'+1)

   AUX_B(0) <= not B(0);
   AUX_B(1) <= not B(1);
   AUX_B(2) <= not B(2);
   AUX_B(3) <= not B(3);
   AUX_B(4) <= not B(4);
   AUX_B(5) <= not B(5);
   AUX_B(6) <= not B(6);
   AUX_B(7) <= not B(7);
   AUX_B(8) <= not B(8);
   AUX_B(9) <= not B(9);

   S0: COMP_ADD port map(A(0), AUX_B(0), '1', O(0), VAI_UM(0));
   S1: COMP_ADD port map(A(1), AUX_B(1), VAI_UM(0), O(1), VAI_UM(1));
   S2: COMP_ADD port map(A(2), AUX_B(2), VAI_UM(1), O(2), VAI_UM(2));
   S3: COMP_ADD port map(A(3), AUX_B(3), VAI_UM(2), O(3), VAI_UM(3));
   S4: COMP_ADD port map(A(4), AUX_B(4), VAI_UM(3), O(4), VAI_UM(4));
   S5: COMP_ADD port map(A(5), AUX_B(5), VAI_UM(4), O(5), VAI_UM(5));
   S6: COMP_ADD port map(A(6), AUX_B(6), VAI_UM(5), O(6), VAI_UM(6));
   S7: COMP_ADD port map(A(7), AUX_B(7), VAI_UM(6), O(7), VAI_UM(7));
   S8: COMP_ADD port map(A(8), AUX_B(8), VAI_UM(7), O(8), VAI_UM(8));
   S9: COMP_ADD port map(A(9), AUX_B(9), VAI_UM(8), O(9), CO);
end hardware ;

--====================--
-- MULTIPLEXADOR 2x1 --
```

```
--====================--

entity MUX21 is
   port(A, B, S: in bit;
      O: out bit);
end MUX21;

architecture hardware of MUX21 is

begin
    O <= (B and S) or (A and (not S));
end hardware;

--==============--
-- FlipFlop JK --
--==============--

entity ffjk is
        port ( clk ,J ,K ,P ,C : IN bit;
                q : OUT bit );
END ffjk ;

ARCHITECTURE ckt OF ffjk IS
        SIGNAL qS : bit;
        BEGIN
PROCESS ( clk ,P ,C )
        BEGIN
        IF P = '0' THEN qS <= '1';
        ELSIF C = '0' THEN qS <= '0';
        ELSIF clk = '1' AND clk ' EVENT THEN
        IF J = '1' AND K = '1' THEN qS <= NOT qS ;
        ELSIF J = '1' AND K = '0' THEN qS <= '1';
        ELSIF J = '0' AND K = '1' THEN qS <= '0';
        END IF;
```

```
        END IF;
END PROCESS ;
q <= qS ;
END ckt ;

--=============--
-- FlipFlop D --
--=============--

ENTITY ffd IS
        port ( clk ,D ,P , C : IN BIT ;
                q : OUT BIT );
END ffd ;

ARCHITECTURE ckt OF ffd IS
        SIGNAL qS : BIT;
        BEGIN
PROCESS ( clk ,P ,C )
        BEGIN
        IF P = '0' THEN qS <= '1';
        ELSIF C = '0' THEN qS <= '0';
        ELSIF clk = '1' AND clk ' EVENT THEN
        qS <= D ;
        END IF;
END PROCESS ;
q <= qS ;
END ckt ;



--=====================--
-- REGISTRADOR 4 bits --
--=====================--

entity REG4 is
```

```
  port(b: in bit_vector(3 downto 0);
       loadd,clk,step: in bit;
       s_b: out bit_vector(3 downto 0)
     );
end REG4;

architecture logic of REG4 is

  component ffd is
    port ( clk ,D ,P , C : in bit ;
    q : out bit );
  end component;

     signal load_b0, reg:bit_vector(3 downto 0);
     signal load_step:bit;

    begin

    load_step <= loadd and step;

    load_b0(0) <= ((not load_step) and reg(0)) or (load_step and b(0));

    ffd1: ffd port map(clk,load_b0(0),'1','1',reg(0));

    load_b0(1) <= ((not load_step) and reg(1)) or (load_step and b(1));

    ffd2: ffd port map(clk,load_b0(1),'1','1',reg(1));

    load_b0(2) <= ((not load_step) and reg(2)) or (load_step and b(2));

    ffd3: ffd port map(clk,load_b0(2),'1','1',reg(2));

    load_b0(3) <= ((not load_step) and reg(3)) or (load_step and b(3));
```

```
    ffd4: ffd port map(clk,load_b0(3),'1','1',reg(3));

    s_b(0) <= reg(0);
    s_b(1) <= reg(1);
    s_b(2) <= reg(2);
    s_b(3) <= reg(3);

end logic;



--======================--
-- REGISTRADOR 10 bits --
--======================--

entity REG10 is
  port(b: in bit_vector(9 downto 0);
       loadd,clk,mx_mi: in bit;
       s_b: out bit_vector(9 downto 0)
     );
end REG10;

architecture logic of REG10 is

  component ffd is
    port ( clk ,D ,P , C : in bit ;
    q : out bit );
  end component;

   signal load_b0, reg:bit_vector(9 downto 0);
   signal load_step:bit;

  begin

  load_step <= loadd and mx_mi;
```

```
load_b0(0) <= ((not load_step) and reg(0)) or (load_step and b(0));

ffd1: ffd port map(clk,load_b0(0),'1','1',reg(0));


load_b0(1) <= ((not load_step) and reg(1)) or (load_step and b(1));

ffd2: ffd port map(clk,load_b0(1),'1','1',reg(1));


load_b0(2) <= ((not load_step) and reg(2)) or (load_step and b(2));

ffd3: ffd port map(clk,load_b0(2),'1','1',reg(2));


load_b0(3) <= ((not load_step) and reg(3)) or (load_step and b(3));

ffd4: ffd port map(clk,load_b0(3),'1','1',reg(3));


        load_b0(4) <= ((not load_step) and reg(4)) or (load_step and b(4));

ffd5: ffd port map(clk,load_b0(4),'1','1',reg(4));


        load_b0(5) <= ((not load_step) and reg(5)) or (load_step and b(5));

ffd6: ffd port map(clk,load_b0(5),'1','1',reg(5));


        load_b0(6) <= ((not load_step) and reg(6)) or (load_step and b(6));

ffd7: ffd port map(clk,load_b0(6),'1','1',reg(6));
```

```
        load_b0(7) <= ((not load_step) and reg(7)) or (load_step and b(7));

  ffd8: ffd port map(clk,load_b0(7),'1','1',reg(7));


        load_b0(8) <= ((not load_step) and reg(8)) or (load_step and b(8));

  ffd9: ffd port map(clk,load_b0(8),'1','1',reg(8));


        load_b0(9) <= ((not load_step) and reg(9)) or (load_step and b(9));

  ffd10: ffd port map(clk,load_b0(9),'1','1',reg(9));

  s_b <= reg;

end logic;


--====================================--
-- COMPARADOR DE MAGNITUDE (1 bits) --
--====================================--

entity COMP_BIT is
  port(a, b, a_Igual_b,a_maior_b,a_menor_b: in bit;
                        Sa_Igual_b, Sa_maior_b, Sa_menor_b: out bit);
end COMP_BIT;

architecture logic of COMP_BIT is

begin
   Sa_Igual_b <= ((a xnor b) and a_igual_b);
```

```
   Sa_maior_b <= (((not b) and a) and a_igual_b) or a_maior_b;
   Sa_menor_b <= (((not a) and b) and a_igual_b) or a_menor_b;

end logic;

--====================================--
-- COMPARADOR DE MAGNITUDE (4 bits) --
--====================================--

entity COMP4 is
  port(a,b: in bit_vector(3 downto 0);
                          Sa_Igual_b, Sa_maior_b, Sa_menor_b: out bit);
end COMP4;

architecture logic of COMP4 is

  component COMP_BIT is
   port(a, b, a_Igual_b,a_maior_b,a_menor_b: in bit;
        Sa_Igual_b, Sa_maior_b, Sa_menor_b: out bit
     );
  end component;

signal V_Sa_Igual_b, V_Sa_maior_b, V_Sa_menor_b: bit_vector (3 downto 0);

begin
        COMP1:          COMP_BIT          port          map(
a(3),b(3),'1','0','0',V_Sa_Igual_b(0),V_Sa_maior_b(0),V_Sa_menor_b(0));
        COMP2:          COMP_BIT          port          map(
a(2),b(2),V_Sa_Igual_b(0),V_Sa_maior_b(0),V_Sa_menor_b(0),V_Sa_Igual_b(1),V_Sa_mai
or_b(1),V_Sa_menor_b(1));
        COMP3:          COMP_BIT          port          map(
a(1),b(1),V_Sa_Igual_b(1),V_Sa_maior_b(1),V_Sa_menor_b(1),V_Sa_Igual_b(2),V_Sa_mai
or_b(2),V_Sa_menor_b(2));
```

```
COMP4: COMP_BIT port map( a(0),b(0),V_Sa_Igual_b(2),V_Sa_maior_b(2),V_Sa_menor_b(2),V_Sa_Igual_b(3),V_Sa_maior_b(3),V_Sa_menor_b(3));

  Sa_Igual_b <= V_Sa_Igual_b(3);
  Sa_maior_b <= V_Sa_maior_b(3);
  Sa_menor_b <= V_Sa_menor_b(3);

end logic;

--=============--
-- COMPARADOR --
--=============--

entity COMPARADOR is
  port(a,b: in bit_vector(11 downto 0);
                        igual,maior, menor: out bit);
end COMPARADOR;

architecture logic of COMPARADOR is

component COMP4 is
  port(a,b: in bit_vector(3 downto 0);
                        Sa_Igual_b, Sa_maior_b, Sa_menor_b: out bit);
end component;

signal a_vetor_centena, a_vetor_dezena, a_vetor_unidade,b_vetor_centena, b_vetor_dezena, b_vetor_unidade:bit_vector(3 downto 0);
signal saida_centena, saida_dezena, saida_unidade: bit_vector (2 downto 0);
signal AmaiorB_1, AmaiorB_2, AmaiorB_3, AmaiorB_4:bit;
signal maiors1, iguals:bit;

begin
  a_vetor_centena(3 downto 0) <= a(11 downto 8);
```

```
  b_vetor_centena(3 downto 0) <= b(11 downto 8);
  a_vetor_dezena(3 downto 0) <= a(7 downto 4);
  b_vetor_dezena(3 downto 0) <= b(7 downto 4);
  a_vetor_unidade(3 downto 0) <= a(3 downto 0);
  b_vetor_unidade(3 downto 0) <= b(3 downto 0);

        COMP1:          COMP4          port          map(
a_vetor_centena,b_vetor_centena,saida_centena(2),saida_centena(1),saida_centena(0));
        COMP2:          COMP4          port          map(
a_vetor_dezena,b_vetor_dezena,saida_dezena(2),saida_dezena(1),saida_dezena(0));
        COMP3:          COMP4          port          map(
a_vetor_unidade,b_vetor_unidade,saida_unidade(2),saida_unidade(1),saida_unidade(0));

     AmaiorB_1 <= ((not saida_centena(0)) and ((not saida_dezena(0)) and (not
saida_unidade(0))));
  AmaiorB_2 <= saida_centena(1);
  AmaiorB_3 <= saida_centena(2) and saida_dezena(1);
  AmaiorB_4 <= (saida_centena(2) and (saida_dezena(2) and saida_unidade(1)));

  igual <= saida_centena(2) and saida_dezena(2) and saida_unidade(2);
  iguals <= saida_centena(2) and saida_dezena(2) and saida_unidade(2);
  maiors1 <= (AmaiorB_1 or AmaiorB_2 or AmaiorB_3 or AmaiorB_4);
  maior <= maiors1 and (not iguals);
  menor <= not maiors1;


end logic;

--======--
-- STEP --
--======--

entity STEP is
  port(a: in bit_vector(3 downto 0);
```

```
                        step,load,clk: in bit;
                        b: out bit_vector(3 downto 0));
end STEP;

architecture logic of STEP is

component ffd is
        port ( clk ,D ,P , C : IN BIT;
                q: OUT BIT );
END component;

  signal D1,D2,D3,D4:bit;
  signal Q:bit_vector (3 downto 0);

  begin

  D1 <= ((not (step and load)) and Q(0)) or (a(0) and (step and load));
  D2 <= ((not (step and load)) and Q(1)) or (a(1) and (step and load));
  D3 <= ((not (step and load)) and Q(2)) or (a(2) and (step and load));
  D4 <= ((not (step and load)) and Q(3)) or (a(3) and (step and load));

  flip1: ffd port map (clk,D1,'1','1',Q(0));
  flip2: ffd port map (clk,D2,'1','1',Q(1));
  flip3: ffd port map (clk,D3,'1','1',Q(2));
  flip4: ffd port map (clk,D4,'1','1',Q(3));

  b(3 downto 0) <= Q(3 downto 0);

end logic;

--=========--
-- MAX/MIN --
--=========--
entity MAX_MIN is
```

```
        port( reg: in bit_vector(9 downto 0);
        up_down, mxmi: in bit;
        lim: out bit_vector (9 downto 0));
end MAX_MIN;

architecture logic of MAX_MIN is

component MUX21 is
  port(A, B, S: in bit;
    O: out bit);
end component;

signal lim_default: bit_vector(9 downto 0);

begin
        mux1: MUX21 port map ('0','1',up_down,lim_default(9));
        mux2: MUX21 port map ('0','1',up_down,lim_default(8));
        mux3: MUX21 port map ('0','1',up_down,lim_default(7));
        mux4: MUX21 port map ('0','1',up_down,lim_default(6));
        mux5: MUX21 port map ('0','1',up_down,lim_default(5));
        mux6: MUX21 port map ('0','0',up_down,lim_default(4));
        mux7: MUX21 port map ('0','0',up_down,lim_default(3));
        mux8: MUX21 port map ('0','1',up_down,lim_default(2));
        mux9: MUX21 port map ('0','1',up_down,lim_default(1));
        mux10: MUX21 port map ('0','1',up_down,lim_default(0));

        mux1: MUX21 port map (lim_default(9),reg(9),mxmi,lim(9));
        mux2: MUX21 port map (lim_default(8),reg(8),mxmi,lim(8));
        mux3: MUX21 port map (lim_default(7),reg(7),mxmi,lim(7));
        mux4: MUX21 port map (lim_default(6),reg(6),mxmi,lim(6));
        mux5: MUX21 port map (lim_default(5),reg(5),mxmi,lim(5));
        mux6: MUX21 port map (lim_default(4),reg(4),mxmi,lim(4));
        mux7: MUX21 port map (lim_default(3),reg(3),mxmi,lim(3));
        mux8: MUX21 port map (lim_default(2),reg(2),mxmi,lim(2));
```

```
        mux9: MUX21 port map (lim_default(1),reg(1),mxmi,lim(1));
        mux10: MUX21 port map (lim_default(0),reg(0),mxmi,lim(0));



end logic;
--=====--
-- CLR --
--=====--



--======--
-- LOAD --
--======--

entity LOAD is
  port(a: in bit_vector(12 downto 0);
                          mxmi,load,clk: in bit;
                          b: out bit_vector(9 downto 0));
end LOAD;

architecture logic of LOAD is

component ffd is
        port ( clk ,D ,P , C : IN BIT;
                q: OUT BIT );
END component;

  signal D1,D2,D3,D4,D5,D6,D7,D8,D9,D10:bit;
  signal Q:bit_vector (9 downto 0);

  begin

  D1 <= ((not (mxmi and load)) and Q(0)) or (a(0) and (mxmi and load));
  D2 <= ((not (mxmi and load)) and Q(1)) or (a(1) and (mxmi and load));
```

```
  D3 <= ((not (mxmi and load)) and Q(2)) or (a(2) and (mxmi and load));
  D4 <= ((not (mxmi and load)) and Q(3)) or (a(3) and (mxmi and load));
  D5 <= ((not (mxmi and load)) and Q(4)) or (a(4) and (mxmi and load));
  D6 <= ((not (mxmi and load)) and Q(5)) or (a(5) and (mxmi and load));
  D7 <= ((not (mxmi and load)) and Q(6)) or (a(6) and (mxmi and load));
  D8 <= ((not (mxmi and load)) and Q(7)) or (a(7) and (mxmi and load));
  D9 <= ((not (mxmi and load)) and Q(8)) or (a(8) and (mxmi and load));
  D10 <= ((not (mxmi and load)) and Q(9)) or (a(9) and (mxmi and load));

  flip1: ffd port map (clk,D1,'1','1',Q(0));
  flip2: ffd port map (clk,D2,'1','1',Q(1));
  flip3: ffd port map (clk,D3,'1','1',Q(2));
  flip4: ffd port map (clk,D4,'1','1',Q(3));
  flip5: ffd port map (clk,D5,'1','1',Q(4));
  flip6: ffd port map (clk,D6,'1','1',Q(5));
  flip7: ffd port map (clk,D7,'1','1',Q(6));
  flip8: ffd port map (clk,D8,'1','1',Q(7));
  flip9: ffd port map (clk,D9,'1','1',Q(8));
  flip10: ffd port map (clk,D4,'1','1',Q(9));

  b(9 downto 0) <= Q(9 downto 0);

end logic;

--==========================--
-- FLIP-FLOPS DO CONTADOR --
--==========================--

entity FLIP10 is
        port(set,reset,soma,sub: in bit_vector (9 downto 0);
            up_down, clk: in bit;
                    TLL: out bit_vector(9 downto 0));
end FLIP10;
```

```
architecture logic of FLIP10 is

component ffd is
        port ( clk ,D ,P , C : IN BIT;
                q: OUT BIT );
END component;

component MUX21 is
  port(A, B, S: in bit;
    O: out bit);
end component;

signal TL,TLLs:bit_vector(9 downto 0);

begin

mux1: MUX21 port map (sub(0),soma(0),up_down,TL(0));
mux2: MUX21 port map (sub(1),soma(1),up_down,TL(1));
mux3: MUX21 port map (sub(2),soma(2),up_down,TL(2));
mux4: MUX21 port map (sub(3),soma(3),up_down,TL(3));
mux5: MUX21 port map (sub(4),soma(4),up_down,TL(4));
mux6: MUX21 port map (sub(5),soma(5),up_down,TL(5));
mux7: MUX21 port map (sub(6),soma(6),up_down,TL(6));
mux8: MUX21 port map (sub(7),soma(7),up_down,TL(7));
mux9: MUX21 port map (sub(8),soma(8),up_down,TL(8));
mux10: MUX21 port map (sub(9),soma(9),up_down,TL(9));

flip1: ffd port map (clk,TL(0),set(0),reset(0),TLLs(0));
flip2: ffd port map (clk,TL(1),set(1),reset(1),TLLs(1));
flip3: ffd port map (clk,TL(2),set(2),reset(2),TLLs(2));
flip4: ffd port map (clk,TL(3),set(3),reset(3),TLLs(3));
flip5: ffd port map (clk,TL(4),set(4),reset(4),TLLs(4));
flip6: ffd port map (clk,TL(5),set(5),reset(5),TLLs(5));
flip7: ffd port map (clk,TL(6),set(6),reset(6),TLLs(6));
```

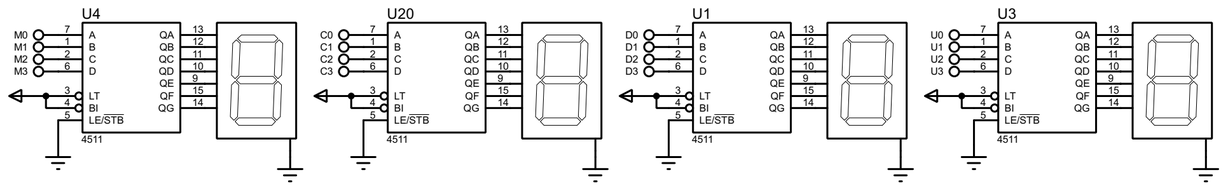
```
flip8: ffd port map (clk,TL(7),set(7),reset(7),TLLs(7));
flip9: ffd port map (clk,TL(8),set(8),reset(8),TLLs(8));
flip10: ffd port map (clk,TL(9),set(9),reset(9),TLLs(9));

TLL(9 downto 0) <= TLLs(9 downto 0);

end logic;

--==========--
-- CONTADOR --
--==========--
entity contador is
        port(A2,A1,A0: in bit_vector(3 downto 0);
                up_down, step, mx_mi, load, clr, clk, led: in bit;
                steps: in bit_vector (3 downto 0);
                Q2,Q1,Q0: out bit_vector(3 downto 0);
                s_temp: out bit_vector(9 downto 0));
end contador;

architecture logic of contador is

component FLIP10 is
  port(up_down,clk: in bit;
     S: out bit_vector(9 downto 0));
end component;

component MUX_STEP is
  port(a,b: in bit_vector(9 downto 0);
      step: in bit;
      s_b: out bit_vector(9 downto 0)
    );
end component;

component MUL is
```

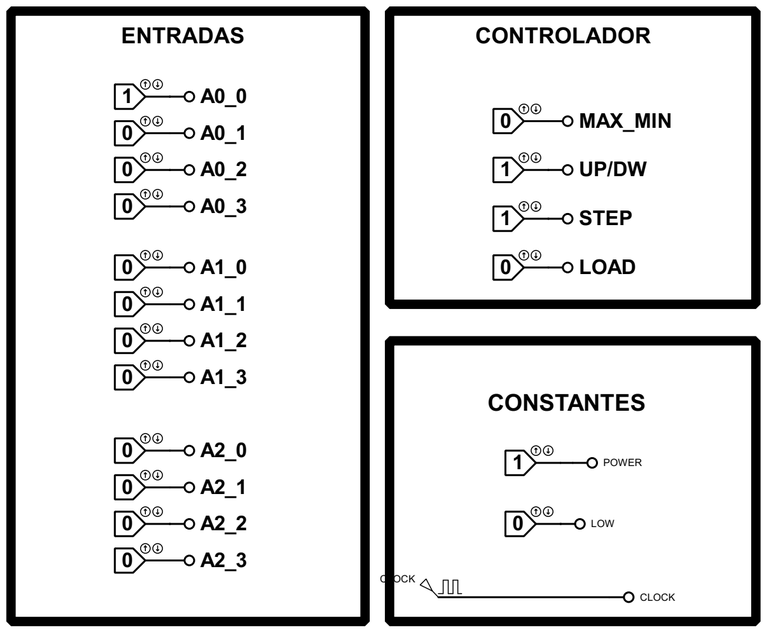
```
    port(A: in bit_vector(9 downto 0);
                                    B: in bit_vector(3 downto 0);
          O: out bit_vector(9 downto 0);
          CO: out bit);
end component;

signal saida_ffjk,O:bit_vector(9 downto 0);
signal CO:bit;

begin

ffjk: FLIP10 port map(up_down,clk,saida_ffjk);

multiplicador: MUL port map(saida_ffjk,steps,O,CO);

muxstep: MUX_STEP port map(saida_ffjk,O,step,s_temp);

end logic;

 entity CLR is
  port(step: in bit_vector(3 downto 0);
          mxmi,clr: in bit;
          b: out bit_vector(3 downto 0);
          mxmi_clr: out bit);
end CLR;

architecture logic of CLR is

component MUX21 is
   port(A, B, S: in bit;
      O: out bit);
end component;

signal mx_mi_clrs:bit;
```

```
signal reg:bit_vector(3 downto 0);

begin

mux1: MUX21 port map('0','1',clr,mx_mi_clrs);
mux2: MUX21 port map('1',step(0),clr,reg(0));
mux3: MUX21 port map('0',step(1),clr,reg(1));
mux4: MUX21 port map('0',step(2),clr,reg(2));
mux5: MUX21 port map('0',step(3),clr,reg(3));



b(3 downto 0) <= reg(3 downto 0);
mxmi_clr <= mx_mi_clrs and mxmi;

end logic;
```

ANEXO C - Esquemáticos

Display

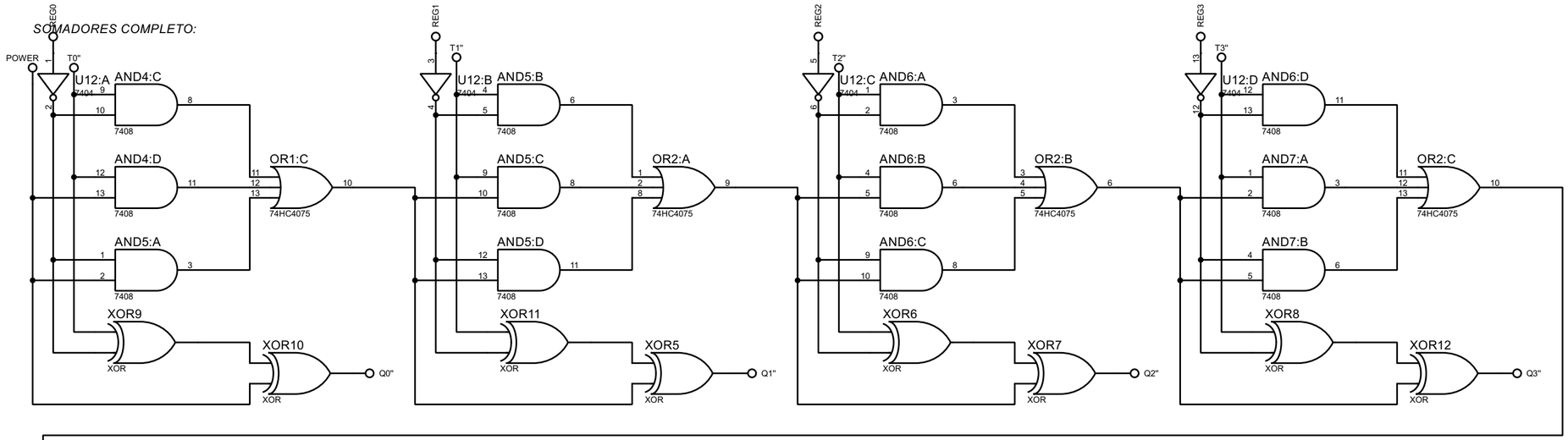


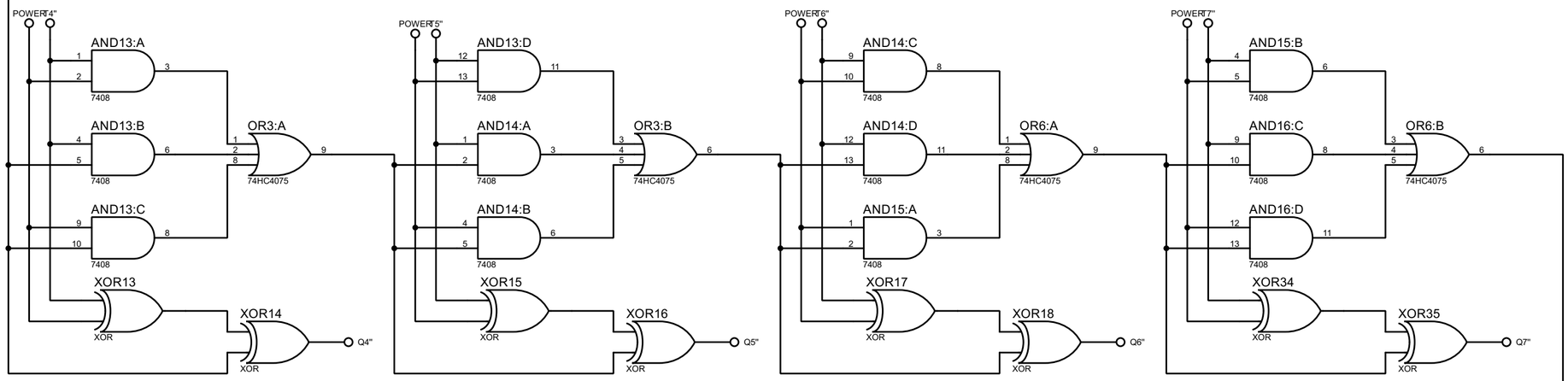


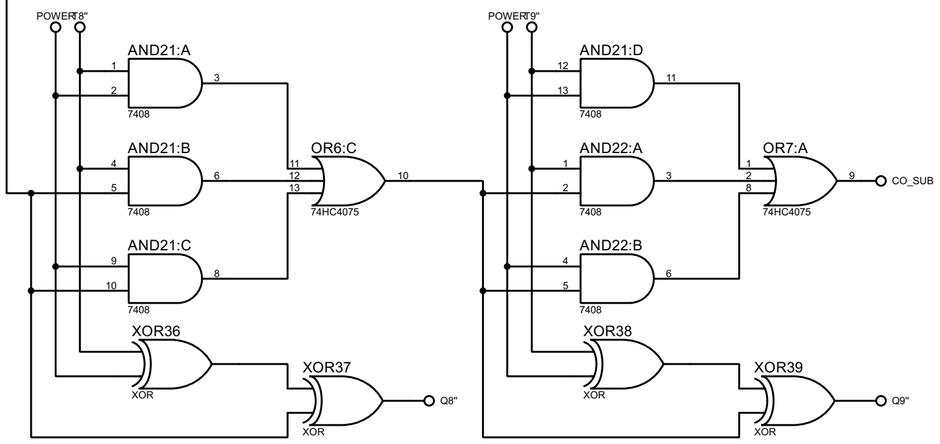
FILE NAME: Comparador			DATE: 07/03/2021
DESIGN TITLE: Comparador			PAGE: 1 of 1
PATH:			TIME:
BY:		REV:	

SUBTRATOR:

SOMADORES COMPLETO:

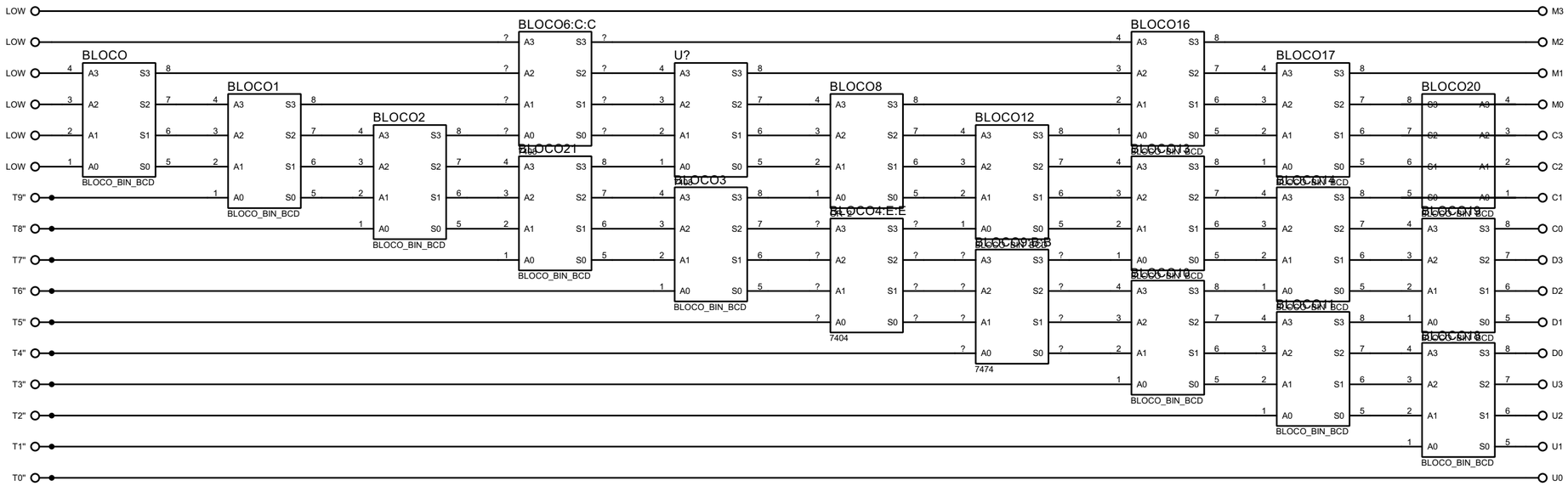






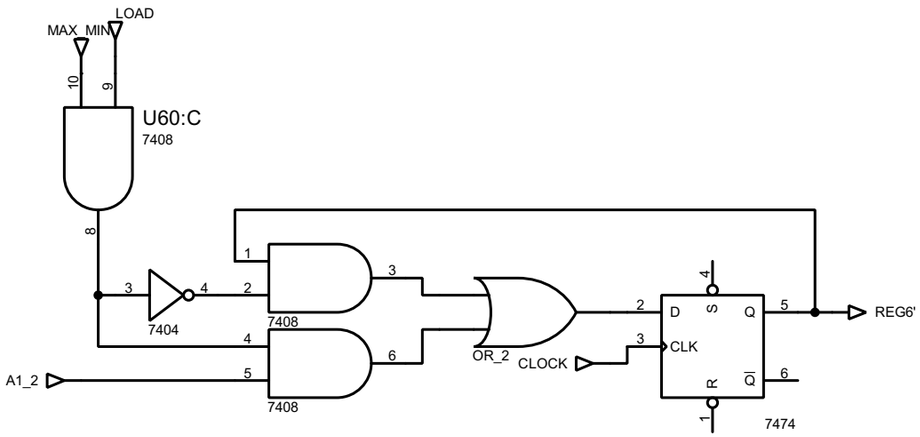
FILE NAME: **Subtrator**			DATE: **07/03/2021**
DESIGN TITLE: **Subtrator**			PAGE: 1 of 1
PATH:			
BY:	REV:		TIME:

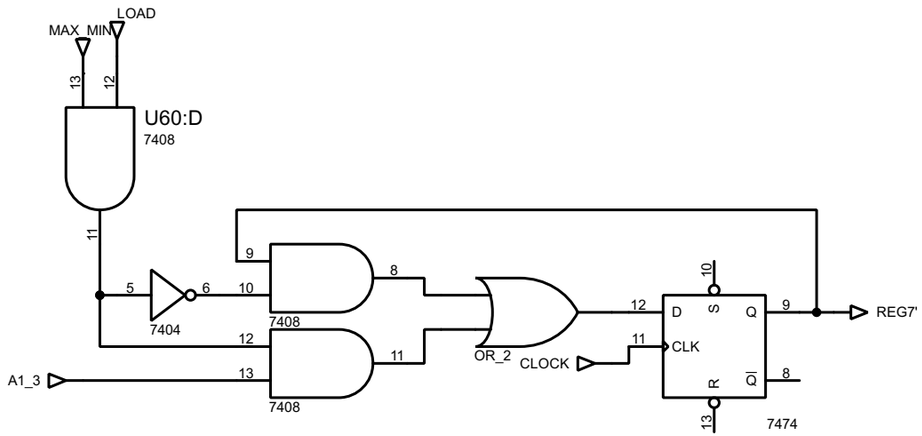
Conversor Bin-BCD 16-bits

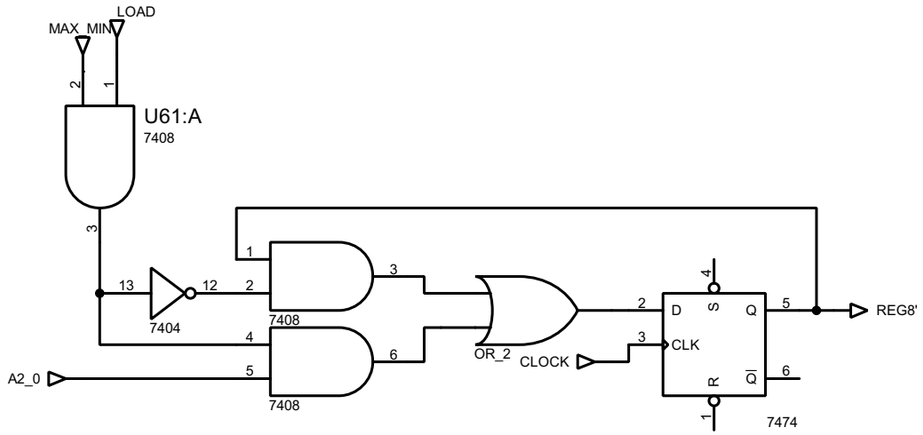


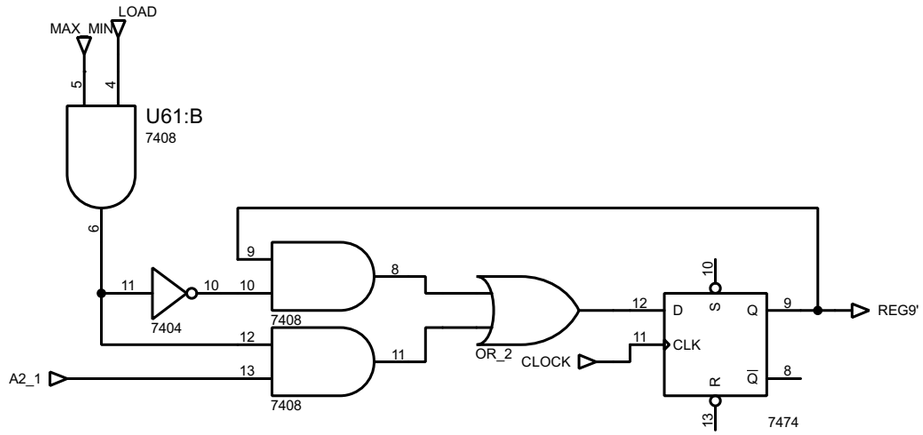
FILE NAME: BCD 16 bits								DATE: 07/03/2021	
DESIGN TITLE: BCD 16 bits								PAGE: 1 of 1	
PATH:									
BY:						REV:		TIME:	

REGISTRADOR



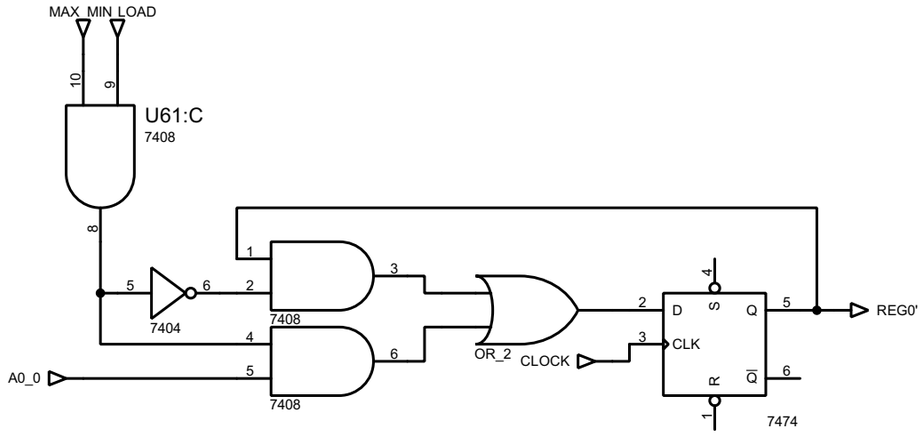


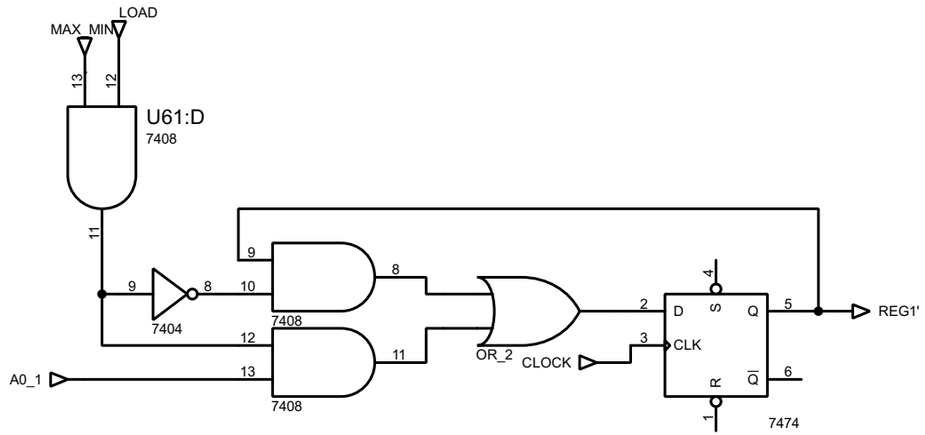


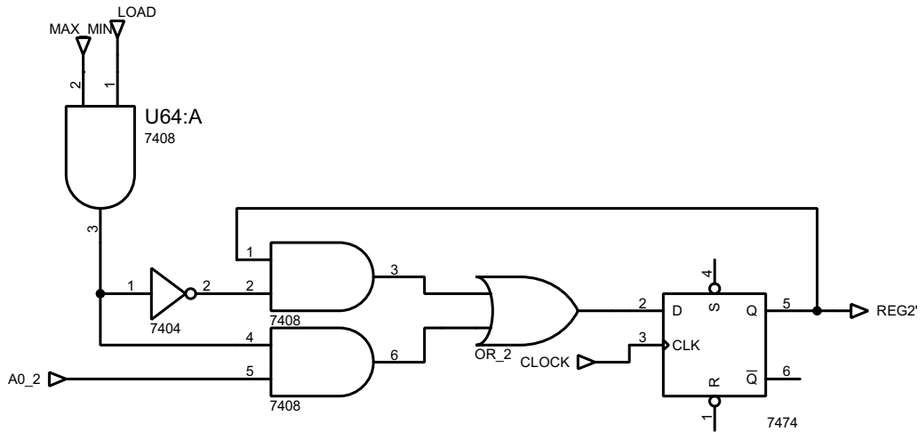


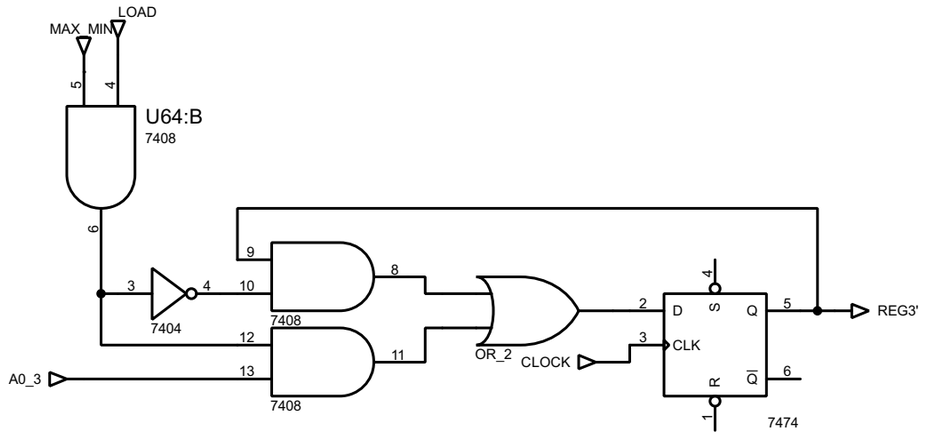
FILE NAME:	Registradores
DESIGN TITLE:	Registradores
PATH:	
BY:	REV:
DATE:	07/03/2021
PAGE:	3 of 3
TIME:	

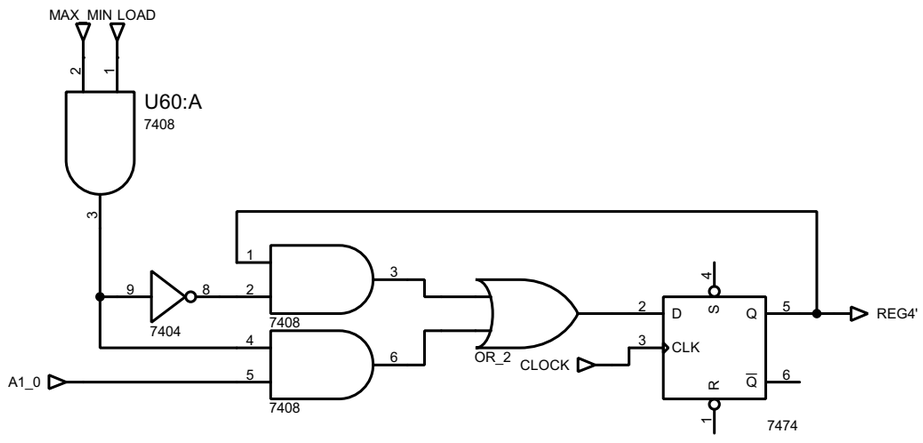
REGISTRADOR

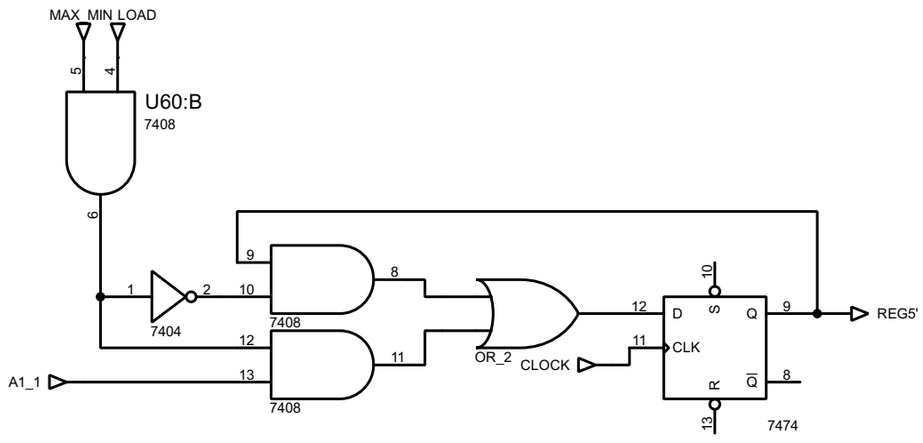






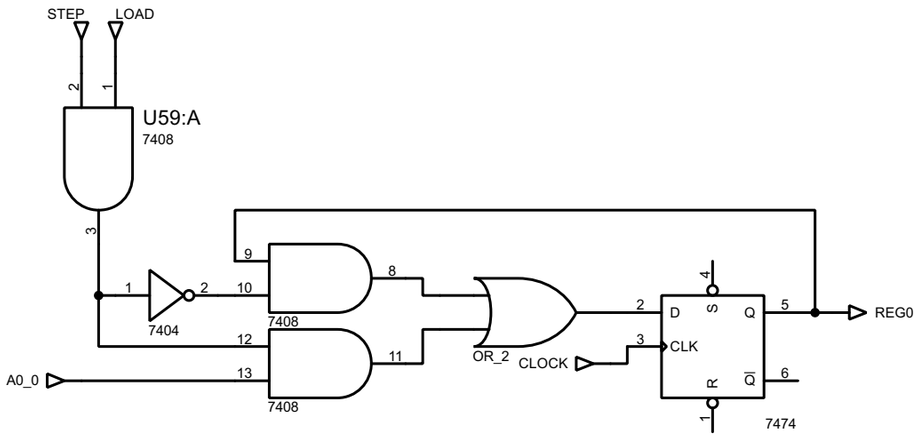


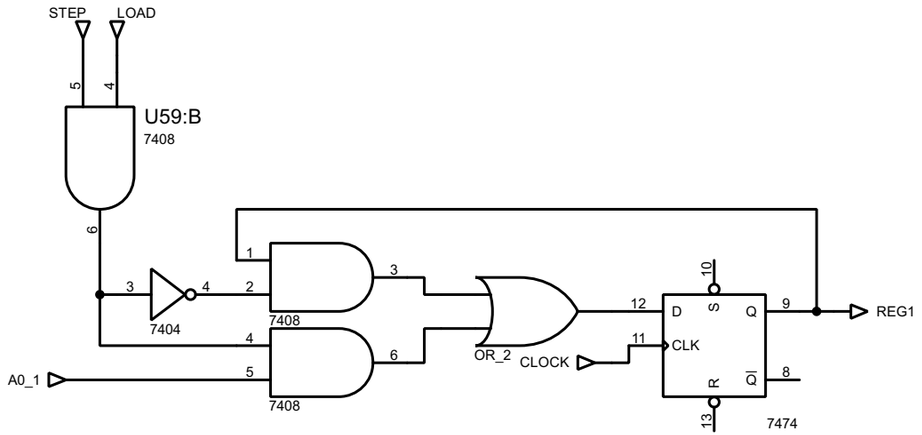


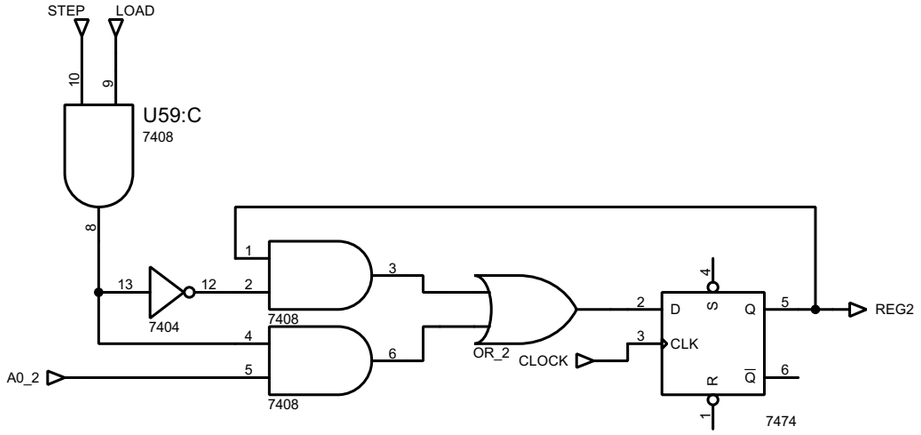


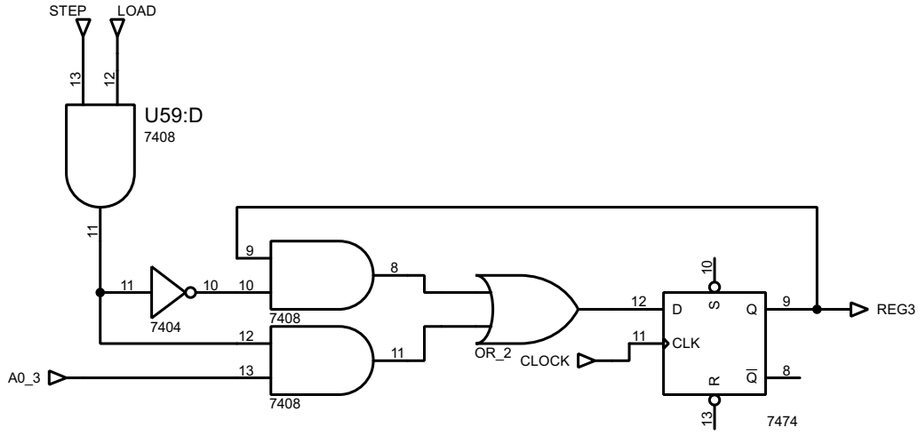
FILE NAME:	Registradores
DESIGN TITLE:	Registradores
PATH:	
BY:	REV:
DATE:	07/03/2021
PAGE:	2 of 3
TIME:	

REGISTRADOR



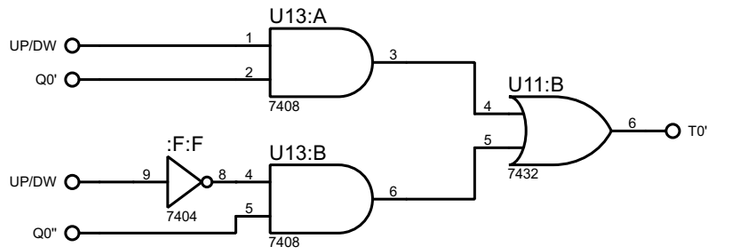


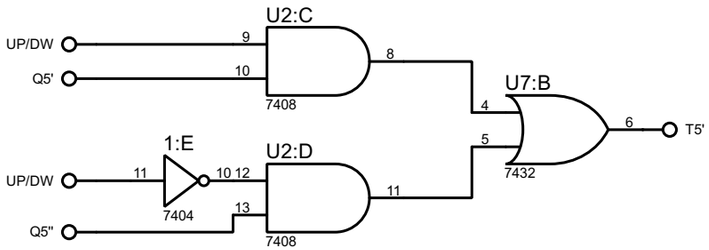


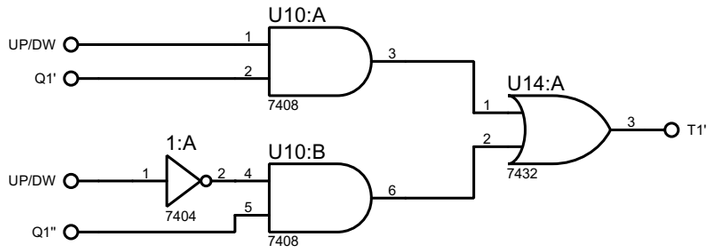


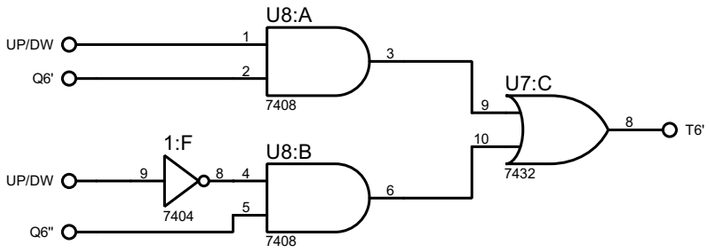
FILE NAME:	Registradores
DESIGN TITLE:	Registradores
PATH:	
BY:	REV:
DATE:	07/03/2021
PAGE:	1 of 3
TIME:	

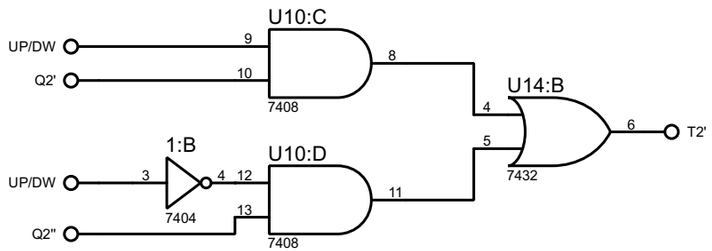
LÓGICA DO UP/DW

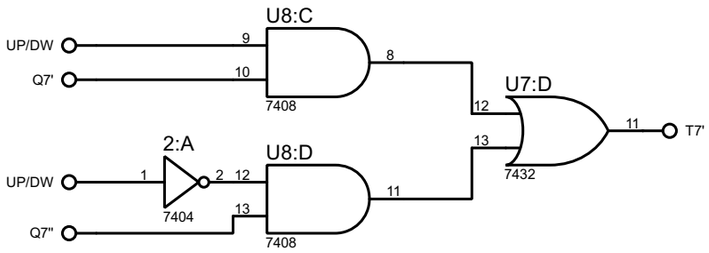


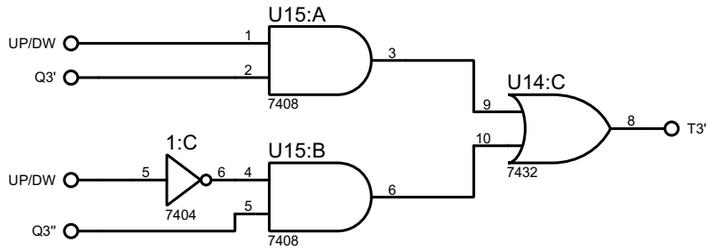


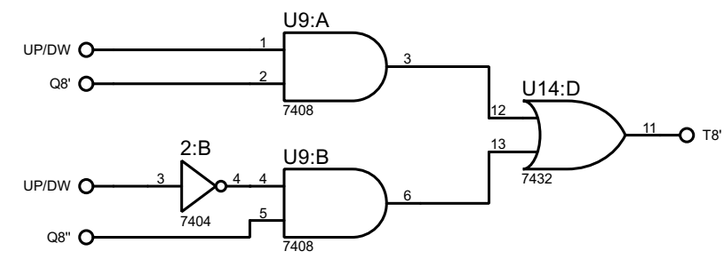


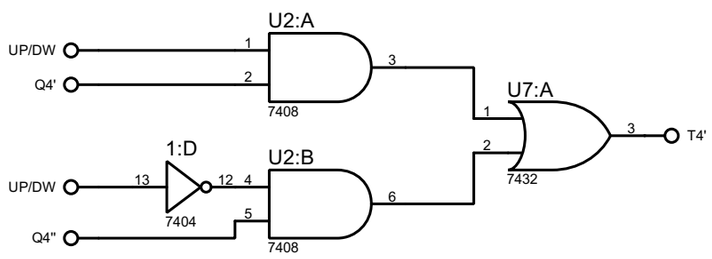


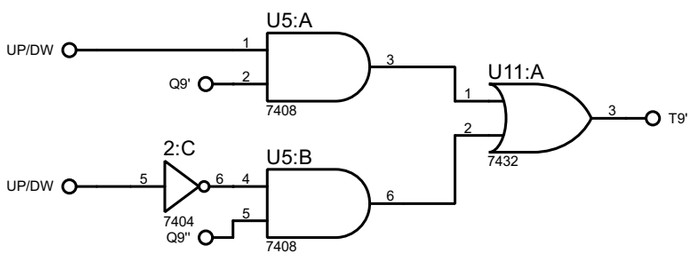






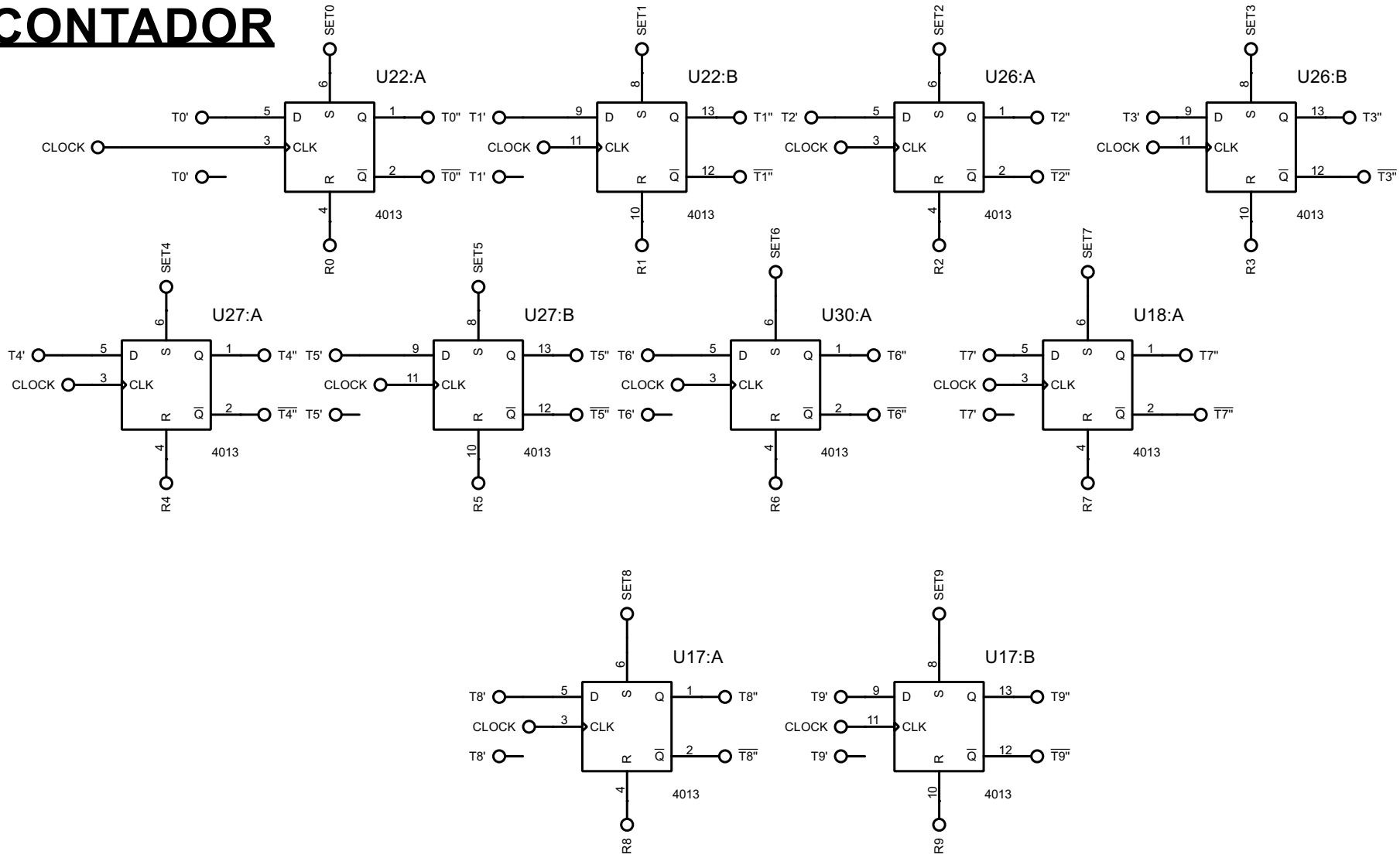






FILE NAME:	Lógica UP and Down
DESIGN TITLE:	Lógica UP and Down
PATH:	
BY:	REV:
DATE:	07/03/2021
PAGE:	1 of 1
TIME:	

CONTADOR

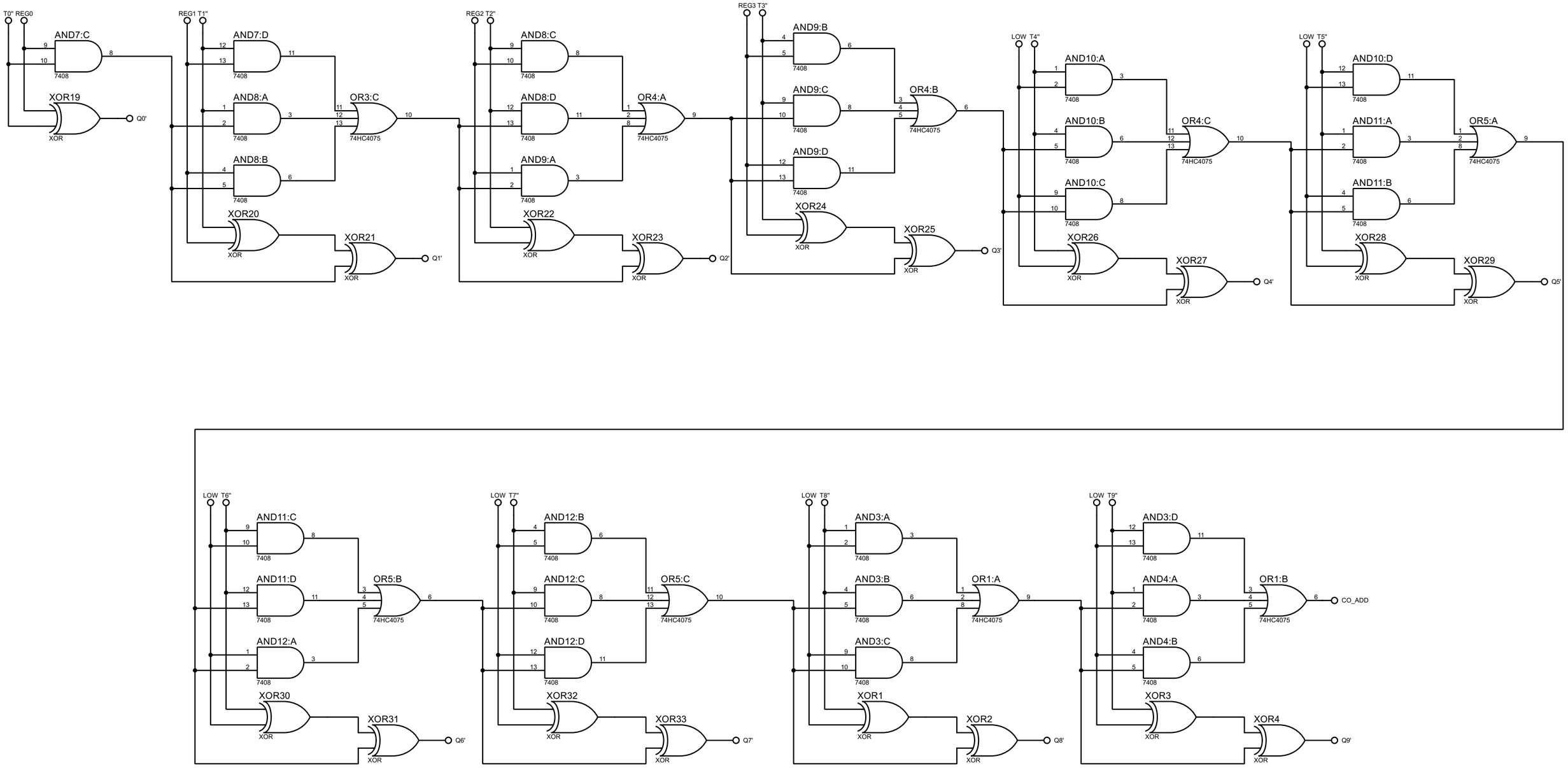


FILE NAME: **Contador**	DATE: **07/03/2021**
DESIGN TITLE: **Contador**	PAGE: **1 of 1**
PATH:	
BY: **Grupo - 03** REV:	TIME: **22:12:58**

Somador

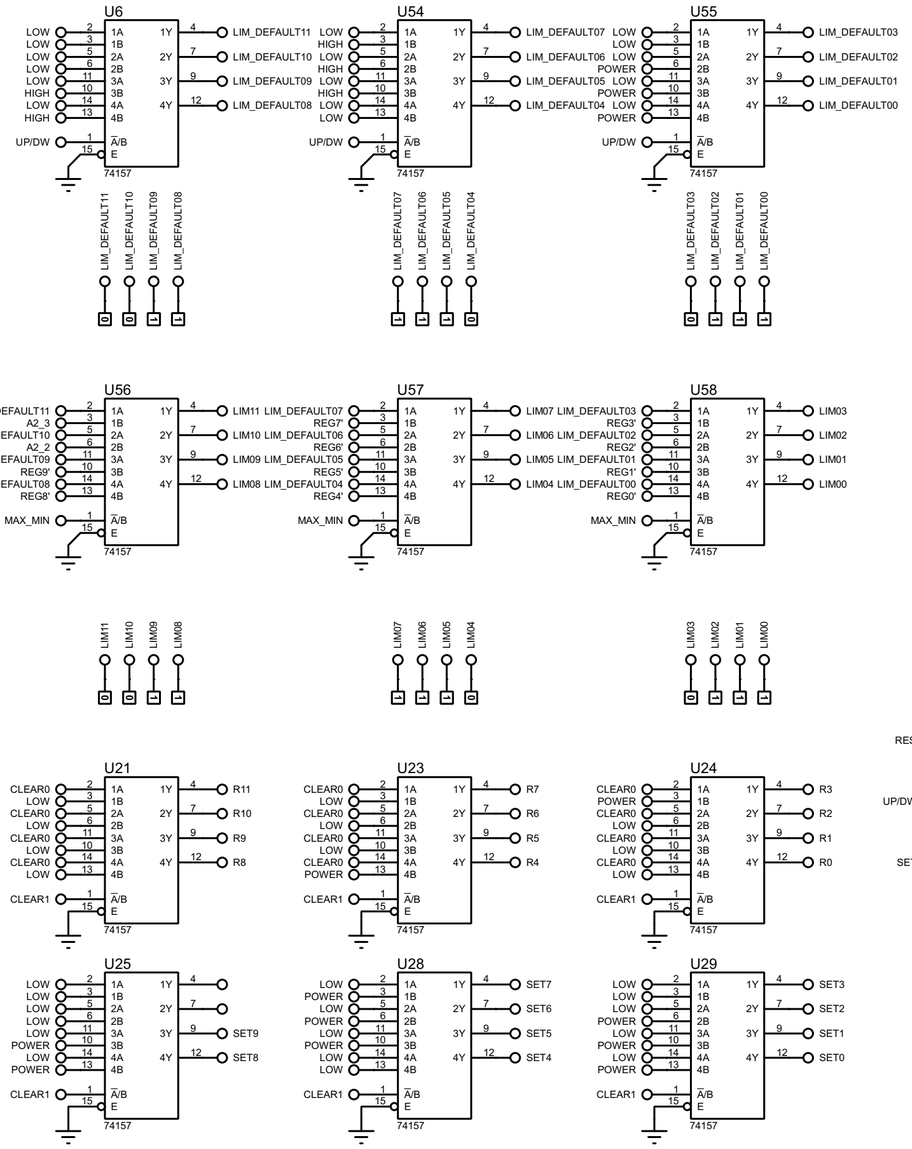
MEIO- SOMADOR:

SOMADORES COMPLETO:



FILE NAME: Somador	DATE: 07/03/2021
DESIGN TITLE: Somador	PAGE: 1 of 1
PATH:	
BY: REV:	TIME:

Comparador



FILE NAME: Comparador	DATE: 07/03/2021
DESIGN TITLE: Comparador	PAGE: 1 of 1
PATH:	
BY: REV:	TIME: