Relatório do projeto de sistemas digitais
Consumo instantâneo de combustível

Bruno Luiz da Silva
Gustavo Fernandes

25 de novembro de 2011

Sumário

Capítulo 1

Introdução

O projeto de consumo instantâneo de combustível, realizado na disciplina de sistemas digitais (EEL 7020) no segundo semestre de 2011, consiste em adquirir a distância percorrida e a injeção de combustível e multiplicar ambos para obter o consumo de combustível. Para tal foi utilizado um FPGA - Field-programmable gate array e a linguagem VHDL para construir o dispositivo que realiza tal operação. O projeto tem como objetivo aprimorar o conhecimento da equipe no área de FPGAs e VHDL assim como dar uma oportunidade para aplicar conceitos teóricos na prática.

Capítulo 2

Desenvolvimento e testes do medidor de consumo de combustível

2.1 Realização de multiplicações com número em ponto flutuante

Os dados que serão dados para o circuito serão todos em ponto flutuante, sendo necessário deixar claro como funcionam as multiplicações com tais números. Os números em ponto flutuante permitem a representação de números reais de forma digital. Para representar um decimal em binário temos algo parecido parecido com a notação científica, sendo que para tal o número será dividido em expoente e mantissa. Exemplo disso é o número 52.125. Ficaria $0.52125 * 10^2$ em notação científica. Em binário acontece algo semelhante. O número fica representado como 110100.001 e após isso é necessário normalizar o número que deixará-o no formato $0.MMMM * 2^E$, sendo MMMM a mantissa e E o expoente. O expoente é o número de deslocamentos que foram necessários para deixá-lo nesse formato. No exemplo do 52.125 ele ficaria $0.110100001 * 2^6$. Seguem mais exemplos.

- $3.5 \rightarrow 0011.1000 \rightarrow 0.11100000 \cdot 2^{0010}$
- $20.125 \rightarrow 10100.001 \rightarrow 0.10100001 \cdot 2^{0101}$
- $15.0625 \rightarrow 1111.0001 \rightarrow 0.11110001 \cdot 2^{0100}$
- $17.0 \rightarrow 10001.0 \rightarrow 0.10001 \cdot 2^{0101}$
- $7.5 \rightarrow 111.1 \rightarrow 0.1111 \cdot 2^{0011}$
- $5.125 \rightarrow 101.001 \rightarrow 0.101001 \cdot 2^{0011}$
- $2.75 \rightarrow 10.11 \rightarrow 0.1011 \cdot 2^{0010}$
- $5.225 \rightarrow 10.001110011001... \rightarrow 0.10001110011001... \cdot 2^{0010}$
- $0.80 \rightarrow 0.11001100110011... \rightarrow 0.11001100110011... \cdot 2^{0000}$

Para realizar a multiplicação de números de ponto flutuante são necessários dois passos.

1 Soma-se os expoentes e obtém-se o expoente

2 Realiza-se a multiplicação binária das mantissas e obtém-se as mantissas

O primeiro passo é realizado como em multiplicações decimais. Quando se tem dois números multiplicados ambos pela mesma base, pode se somar os expoentes das bases. Como no exemplo:

$$(1 \cdot 10^2) \cdot (1 \cdot 10^3) = 1 \cdot 10^{(2+3)}$$

No segundo passo é realizada a multiplicação binária das mantissas. Um exemplo de uma multiplicação binária 0010 (2) e 0011 (3). A multiplicação binária é muito semelhante a decimal. Segue o exemplo:

```
    0010
  x 0011
  ------
    0010
   0010
  0000
+ 0000
  -------
  0000110
```

Após efetuada a multiplicação das mantissas você terá então os expoentes e mantissa, tendo assim o número em ponto flutuante. Nesse projeto é preciso normalizar esse número e para isso é necessário deixar o MSB igual a um. Para tal deslocam-se todos os números da mantissa para a esquerda até atingir tal ponto. Assim, pega-se o número de deslocamentos e decrementa-se do expoente, tendo assim o número em ponto flutuante normalizado.

2.2 Revisão teórica de sistemas digitais

Alguns conceitos das aulas teóricas de sistemas digitais valem ser revistos. O mais básico diz em relação ao que é um binário e o que é o MSB e o LSB do binário. Resumindo, um número binário é representado por 0s e 1s e cada um desses é múltiplicado por $2^{(n-1)}$. N é o número de bits (0s ou 1s) que o binário possui. Exemplo disso é o binário que representa 137. Ele é representado por "10001001", isso é, 8 bits, logo para converter de volta para decimal tem-se $1{\cdot}2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1{\cdot}2^0$ que resulta em 137.

O LSB, *Least Significant Bit*, é o bit mais a direita do número binário e o MSB, *Most Significant Bit*, é o bit mais a esquerda do número binário. O MSB é o mais significante por ele ser o bit de maior valor. Pegue por exemplo o valor binário de 137. O MSB será o '1' da esquerda e ele vale 2^7 e o LSB será o '1' mais a direita, que vale 2^0.

Outro conceito importante é o de registradores. Eles nada mais são que um conjunto de flip-flops, sendo que esses são componentes que são utilizados para armazenar dados quando o clock está na borda de subida. Esse banco de flip-flops chamado registrador em alguns casos possui outros recursos como deslocamento de bits, podendo desloca-los tanto para direita quanto para esquerda.

Ainda há o comparador, contador e somador, todos utilizados nesse projeto. O comparador é utilizado para comparar dois dados binários. Pode ser utilizado com um contador criando assim um laço. Enquanto a saída do contador não for a desejada o comparador não acusará nada, porém quando for então ele dará um sinal para tal. O contador a cada operação incrementa, normalmente, um bit a seu registrador interno e este é colocado na saída. Internamente ele possui um somador.

Os somadores possuem normalmente vários blocos para adição (sendo que o número desses blocos é proporcional ao número de bits da saída) os quais se chamam *full-adders* e cada um considera o *carry* anterior. Por exemplo, se você soma '1' + '1' tem-se "10". Só que se você fizer a conta a mão você terá que '1'+'1' será '0' e que o *carry* gerado será '1'. O somador usa esse *carry* para as somas subsequentes até ter a soma total. Para checar se houve ou não overflow cria-se um XOR entre o último e penúltimo *carry*.

Já a máquina de estados (*Finite State Machine - FSM*) é utilizada para controlar os componentes dos projetos. A cada estado ela envia um valor lógico para alguma porta de algum

componente, habilitando ou executando alguma operação nesse, quando algum pré-requisito for executado (exemplo: na subida de clock → mudar de estado). Um exemplo é uma máquina de estados de uma máquina de refrigerantes, que só sairá do estado inicial quando for inserida uma moeda. Assim que for inserida ela mudará de estado e enviará um comando para um registrador avisando o valor que foi colocado na máquina.

Por fim há o decodificador, que possui um funcionamento simples. Quando enviado um número binário, ele a interpretará e então terá somente uma de suas saídas ativadas. O número de saídas é 2^n, sendo n o número de bits do número enviado e a saída a ser ativada será o número binário enviado + 1. Por exemplo, se a entrada for 101 (cinco em decimal) então somente a sexta saída (5 + 1) terá valor lógico alto.

2.3 Fluxo do projeto

O projeto foi realizado em etapas. A primeira foi pensar no funcionamento das multiplicações de números de ponto flutuante. Assim que descoberto como é tal processo a equipe hipotetizou como seria o circuito. Para tal foram checadas também as referências deixadas pelo professor em seu *website* e utilizou-se o conhecimento dado nas aulas teóricas. A primeira modificação da referência deixada pelo professor foi a troca de dois registradores de 8 bits por um de 16 bits, que seria mais facilmente desenvolvido em VHDL. Além disso, a passagem do multiplicador pelo registrador de 8 bits restante se fez desnecessário visto que seria mais fácil construir um bloco de decodificação e ele faria as etapas que passariam pelo registrador de 8 bits.

Após desenvolvida as idéias de como seria o circuito, iniciou-se o desenvolvimento do circuito em VHDL. Essa linguagem, *VHSIC hardware description language*, é utilizada para descrever um hardware e assim "construí-lo". Esse código, no atual projeto, será utilizado para descrever qual será o harware que será implementado na kit de desenvolvimento Altera DE 2 que utiliza a FPGA Cyclone II EP2C35F672C6. A ferramenta utilizada para desenvolver o projeto foi o Altera Quartus II 9.1 SP2, onde criou-se os VHDLs responsáveis por cada componente, além de toda a simulação. Vale constar que o uso da versão 9.1, que não é a mais atual, é devido a ferramenta de simulação não estar mais presente nas versões atuais.

Cada componente foi desenvolvido, comentado e simulado separadamente, possibilitando assim a avaliação de cada um e evitando problemas ao procurar erros no projeto principal. Após finalizados, utilizou-se o recurso "port map" do VHDL para conectar todos os componentes num componente principal, constituindo assim o calculador de consumo. Foram realizadas várias simulações e assim que atestado o funcionamento do projeto, continuou-se a comentar o código e procurar possíveis erros.

Por fim, inicializou-se a criação desse relatório afim de explicar como ocorreu o projeto. Para criá-lo foi utilizado o padrão LaTeX, pretendo aprender mais sobre tal padrão.

2.4 Implementação do circuito

Para criar tal circuito foram utilizados registradores, deslocadores, contadores, somadores, comparadores, decodificadores e por fim uma máquina de estados. Os registradores utilizados foram um de oito bits, muito simples, só utilizado para armazenar valor e outro de dezesseis bits mais um, pois houve a necessidade de armazenar o carry do somador anterior logo foi adicionado um bit a mais. Esse registrador é muito mais complexo que o primeiro, pois possui dois deslocadores e um contador.

Um dos deslocadores desse registrador serve para deslocar os bits a direita, utilizado na multiplicação (soma e deslocamento). Já o segundo deslocador é para deslocar os bits a esquerda, permitindo assim a normalização da mantissa. Vale notar que a cada deslocamento desse é adicionado um num contador que ao final dará o número de deslocamentos que foram necessários para normalizar o número, o que influenciará no resultado do expoente.

Além disso há um somador anexado a um comparador, que checará se o LSB é '0' ou '1' e a partir disso fará a soma de multiplicando com parte alta do registrador de 16 bits ou não executará nada. Isso é crucial para realizar a multiplicação, afinal o processo todo se dá por soma e deslocamento, sendo que a soma é feita aqui e o deslocamento, como dito anteriormente, no registrador.

Ainda há a parte do expoente. Nele existirá um somador que fará a adição entre os dois expoentes dados e armazenará esses dados até que a mantissa seja normalizada para que o número de deslocamentos possa ser decrementado da soma anterior e assim ter o expoente final.

Segue um lista com mais detalhes sobre cada componente.

Consumo combustível: esse será o bloco "top-entity" onde serão conectadas as chaves (SW), botões (KEY), clock (CLOCK_50) e LEDs (LEDG e LEDR). Também deve-se notar que será aqui que todos os outros componentes serão conectados e assim receberão os dados do usuário. O usuário colocará um valor de oito bits inicialmente (multiplicando) e apertará o botão KEY(1) que amazenará tal valor no registrador de 8 bits. Após o usuário soltar o botão então o componente aguardará a inserção do outro valor (multiplicador) e consequente pressionamento de KEY(1). Após feito isso então o componente aguardará os valores dos expoentes, sendo de 4 bits cada e ambos serão inseridos nas mesmas chaves (SW). Para tal SW(7 downto 4) será um expoente e SW(3 downto 0) será outro. Assim que for apertado KEY(1) então as operações de multiplicação serão realizadas. Somará-se os dois expoentes, realizará-se o processo de multiplicação da mantissa e posterior normalização e por fim haverá a subtração do número de deslocamentos que foram necessários na normalização com o atual valor da soma dos dois expoentes. Tem-se assim o valor final, que terá a mantissa representada em LEDG(7 downto 0) e expoentes em LEDR. Caso houver overflow ele será indicado em LEDG(8).

Os sinais utilizados serão apenas para conectar um componente à outro, com excessão de mantissa_out(15 downto 8) que será ligado a saída LEDG(7 downto 0), exponent_out(3 downto 0) que será ligado a saída LEDR e exponent_out(4) que será ligado a saída LEDG(8). Este último indicará se houve ou não overflow, utilizando a idéia de que o expoente só pode ir de 0 té 15, isso é, de "0000" até "1111". Se esse valor for ultrapassado então terá-se no mínimo '1' no MSB de exponent_out, algo como "10000" (representação binária de 16).

Multiplicador: bloco onde será realizada a multiplicação das duas mantissas. Para tal, o usuário entrará com os dados de 8 bits e para controlar para quais componentes esses dados irão, será utilizado o input_control (logo abaixo haverá uma descrição detalhada dele), sendo que os "arguments" serão enviados por uma FSM e habilitará os componentes corretos no momento certo. Deve-se primeiramente carregar o registrador de 8 bits com esses dados (reg_8 / "001"), depois carregar a parte baixa do registrador de 16 bits (reg16 / "010") e então iniciar a adição e/ou deslocamento.

Para tal o componente de soma deverá ser ativado, sendo que o mesmo argumento ativa o armazenamento de dados para a parte alta registrador de 16 bits ("011"). Essa operação envia a saída da parte alta do registrador de 16 bits para a entrada "b" do somador e na entrada "a" tem-se o valor do registrador de 8 bits. Ainda é enviado o último bit presente no reg_16 pois caso ele for 0 os valores armazenados serão apenas deslocados a direita e caso for 1 então soma-se as duas entradas e desloca-se os valores armazenados. Para realizar o deslocamento usa-se o argumento "100".

O processo de adição e deslocamento deve ser efetuado oito vezes. Quando esse processo terminar tem-se o produto das mantissas, porém para o projeto será necessário normalizar a mantissa. Para tal o primeiro bit do produto (MSB) deve ser 1 e para isso é preciso avaliar esse bit e caso for 0 desloca-se o valor armazenado para a esquerda. Essa operação é realizada pelo registrador de 16 bits (argumento: "101"). Cada vez que houver o deslocamento será incrementado 1 em um contador interno. Após o MSB for 1 então finaliza-se o processo e terá-se o valor de vezes que o número foi deslocado. Esse resultado será utilizado para decrementar o expoente posteriormente.

Os sinais utilizados em sua maioria serão somente para conectar as entradas e saídas dos componentes entre si, com excessão do reg16out onde será ligado na saída q do Multiplicador e exp que será ligado na saída exp_nor.

Input_control: esse componente não é de controle, afinal ele faz um papel de decodificador, mas para facilitar o entendimento foi colocado essa nomenclatura. A partir de um argumento ele habilitará somente os componentes corretos, sendo que esses serão enviados por uma máquina de estados, a qual realmente controla o circuito. Na entrada "a" estarão conectadas as chaves SW e em "reg" estará conectada a parte alta do registrador de 16 bits. Suas saídas serão conectadas diretamente nas entradas que habilitam as operações dos componentes (armazenamento, deslocamento e outras). Segue uma tabela com a especificação de o que cada uma faz.

Argumento	**Descrição**	**l1**	**l2**	**s1**	**shift**	**n1**	**q**
000	nada	0	0	0	0	0	
001	carrega reg. de 8 bits	1	0	0	0	0	
010	carrega parte baixa do reg. de 16 bits	0	0	1	0	0	
011	ativa somador e parte alta do registrador de 16 bits	0	1	0	0	0	"reg"
100	habilitará deslocamento para esquerda do registrador de 16 bits	0	0	0	1	0	
101	normaliza os dados do reg. de 16 bits	0	0	0	0	1	

Registrador de 8 bits: guardará o valor de 8 bits que for dado para a entrada "d", nesse caso o multiplicando, quando for dado um valor lógico alto para a entrada "enable". A saída ("q") será o valor armazenado quando o enable foi dado como alto.

Somador: este realizará a soma entre o valor do registrador de 8 bits (entrada a) com a parte alta do registrador de 16 bits (entrada b) quando seu "enable" estiver com valor lógico alto e o MSB do registrador de 16 bits for '1', o qual será dado na entrada "control". Caso contrário, quando "enable" for alto então o componente só copiará o valor da parte alta para o registrador direto para sua saída, não executando nenhuma operação. A saída será o resultado gerado por uma dessas duas condições.

Registrador de 16 bits (+ 1): ao final será ele que guardará o produto das mantissas. Inicialmente serão dados os valores do multiplicador na entrada "chaves" e estes serão guardados na parte baixa do registrador (7 downto 0). Para tal será necessário dar um valor lógico alto para "start". Para carregar o valor que vier do somador terá-se que conectar a saída do contador na entrada "soma" e colocar um valor lógico alto em "load" para assim o valor será guardado na parte alta do registrador (15 downto 8).

Como esse bloco será usado para um multiplicador então será necessário que o valor seja deslocado após a operação do componente "somador". Para realizar isto será necessário dar um valor lógico alto para "shift" a cada deslocamento desejado.

A normalização fará-se necessária neste projeto, logo "normalize" deve receber um valor lógico alto para que se inicie a normalização. Nesse processo o signal "works" tornará-se 1 e enquanto o MSB não for 1, "works" permanecerá em 1, desabilitando qualquer outra

operação do componente. Quando a operação estiver concluída terá-se o número normalizado e "exp" armazenará o número de deslocamentos que foram necessários.

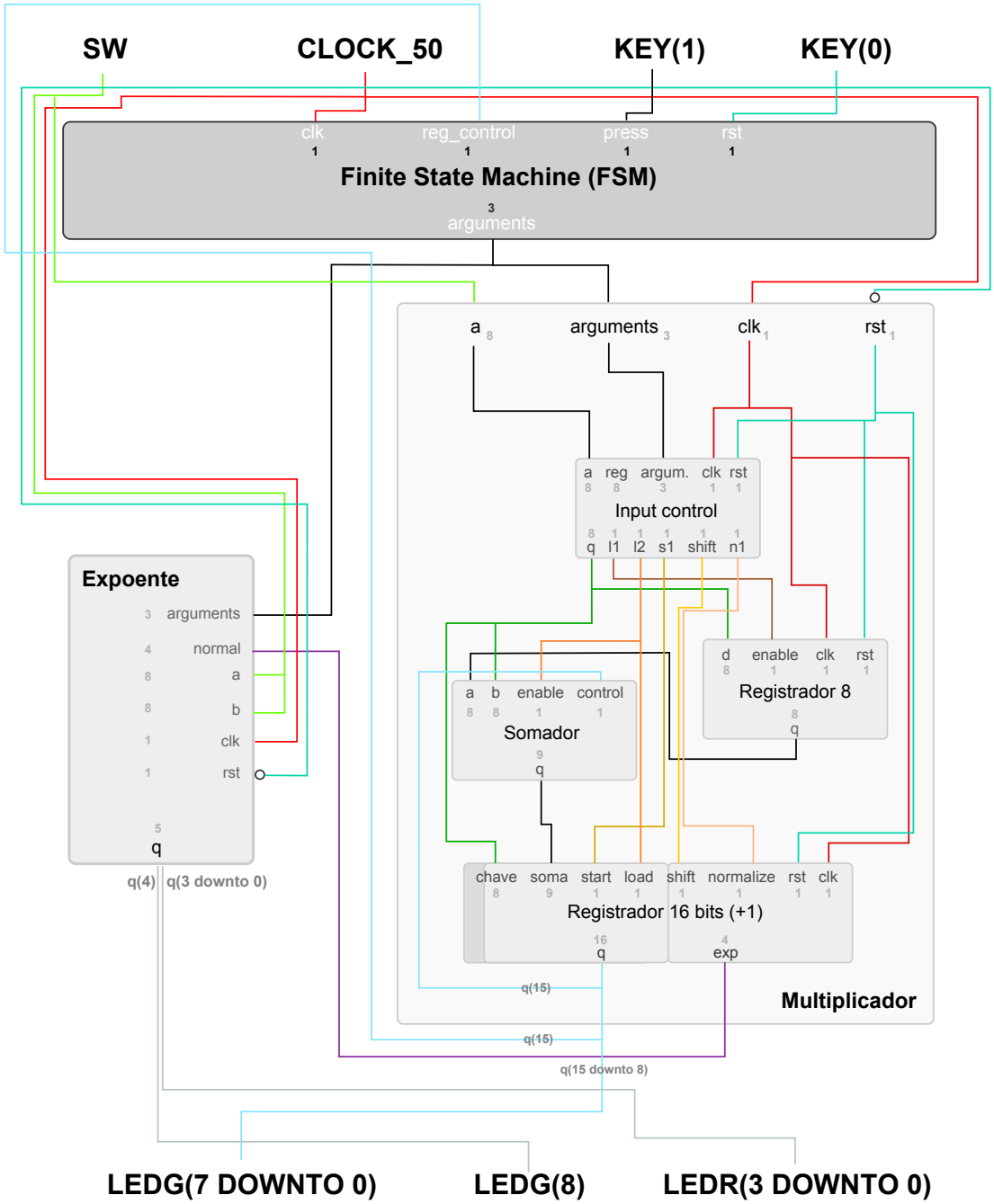
Vale notar que o *+1* no título do componente deve-se ao bit extra colocado no registrador para não acontecerem problemas com o carry da soma entre multiplicando e parte alta do registrador.

Os sinais utilizados, como *aux*, *exp_aux* e *works*, servirão, repectivamente, para armazenar os resultados, armazenar quantas vezes foi deslocado o produto e para ativar a normalização.

Expoente: esse será o componente responável para dar o expoente da multiplicação, que se faz necessário por ser uma multiplicação de números de ponto flutuante. Para tal o usuário entrará com os expoentes nas entradas "a" e "b" além de dar o valor "110" para o "argument" para assim realizar a soma de ambos. Para previnir casos onde houver uma soma que extrapole o valor "1111" (15 em decimal) então foi adicionado um bit extra na saída para armazenar o possível valor extra. Em algum momento o "argument" será "111" e então será realizada a subtração entre o valor da soma dos expoentes com o número de deslocamentos realizados para normalizar a mantissa, que será dado na entrada "normal". Após isso então tem-se o expoente final, porém ele deve estar na faixa de "0000" a "1111" (0 a 15 em decimal), pois caso extrapole essa faixa ele não poderá apresentar o valor correto nos LEDs designados sendo assim um caso de *overflow*, que ativará o LED(8) por meio do bit extra que existe no componente (q(4)).

O sinal utilizado (aux) será apenas para armazenar os dados e em alguns momentos modificá-los.

Essa foi uma descrição de quase todos os componentes. Ainda há a maquina de estados, que será tratada na próxima seção. Segue abaixo um diagrama do circuito.


SW
CLOCK_50
KEY(1)
KEY(0)
clk
reg_control
press
rst
Finite State Machine (FSM)
arguments
a
arguments
clk
rst
a reg argum. clk rst
Input control
q l1 l2 s1 shift n1
Expoente
arguments
normal
a
b
clk
rst
q
q(4)
q(3 downto 0)
a b enable control
Somador
q
d enable clk rst
Registrador 8
q
chave soma start load shift normalize rst clk
Registrador 16 bits (+1)
q
exp
q(15)
Multiplicador
q(15)
q(15 downto 8)
LEDG(7 DOWNTO 0)
LEDG(8)
LEDR(3 DOWNTO 0)

2.5 Máquina de estados (Finite State Machine)

Para o projeto foi criada uma máquina de estados para controlar os componentes e registradores. No caso os sinais de saída de FSM serão "argumentos" que dentro dos blocos de multiplicação e expoente serão interpretados e habilitarão alguns componentes ou darão algum comando para o componente funcionar como desejado (exemplo: deslocar bits para direita ou para esquerda).

No diagrama é possível checar quais são os sinais de saída dele (arg) e pode-se verificar o que cada um faz na tabela A tabela que segue representa a transição dos estados e vê-se exatamente o comportamento da FSM a partir de certos parâmetros como do *enter*, *reset*, *count* e *reg_control*.

Estado atual	**Próximo estado**	**K(1)**	**K(0)**	**COUNT**	**REG_CONTROL**
Inicial	Carrega_reg8	1	0	X	X
Carrega_reg8	Dummy1	0	1	X	X
Carrega_reg8	Carrega_reg8	1	1	X	X
Dummy1	Carrega_reg16	1	1	X	X
Dummy1	Dummy1	0	1	X	X
Carrega_reg16	Dummy2	0	1	X	X
Carrega_reg16	Carrega_reg16	1	1	X	X
Dummy2	Soma_expoentes	1	1	X	X
Dummy2	Dummy2	0	1	X	X
Soma_expoentes	Soma_mantissa	0	1	X	X
Soma_expoentes	Soma_expoentes	1	1	X	X
Soma_mantissa	Desloca_mantissa	X	1	count != 0000	X
Soma_mantissa	Normaliza	X	1	count = 0000	X
Desloca_mantissa	Soma_mantissa	X	1	X	X
Normaliza	Expoente	X	1	X	1
Normaliza	Normaliza	X	1	X	0
Expoente	Final	X	1	X	X
Final	Final	X	1	X	X
Final	Inicial	X	0	X	X

Observação: ENTER e K(1) referem-se a KEY(1) e RESET e K(0) a KEY(0)

Segue uma descrição detalhada do que ocorre em cada estado:

Inicial: esse será o estado inicial, onde não haverá nenhuma saída. Só vira para esse estado se o *RESET* for pressionado.

Carrega_reg8: responsável por carregar o registrador de 8 bits quando *ENTER* for pressionado pela primeira vez, guardando o valor do multiplicando, que deverá estar configurado nas chaves. Para tal sua saída será "001".

Dummy1: será encarregado de aguardar que *ENTER* seja despressionada, pois caso não existisse esse estado então em apenas alguns segundos todos os registradores estariam carregados com o mesmo dado.

Carrega_reg16: responsável por carregar o registrador de 16 bits quando *ENTER* for pressionado pela segunda vez, guardando o valor do multiplicador na parte baixa do registrador (7 downto 0). Para isso sua saída será "010".

Dummy2: será encarregado de aguardar que *ENTER* seja despressionada, novamente.

Soma_expoentes: quando *ENTER* for pressionada pela terceira vez então os valores que estiverem nas chaves serão enviados para um somador que somará os primeiros 4 bits com os 4 últimos (7 downto 4 e 3 downto 0) para que se tenha o expoente. Para tal a saída da FSM será "101" e além dos processo descritos acima ela inicializará um contador (*count*) em oito ("1000").

Soma_mantissa: na multiplicação devem ser executadas somas e deslocamentos. Esse estado será responsável por fazer a possível soma entre valores da parte alta do registrador de

16 bits (15 downto 8) e registrador de 8 bits (multiplicando), porém somente enquanto o contador, antes inicializado, for diferente de zero ("0000"). Quando o contador for zero então o próximo estado não será mais "Desloca_mantissa", mas sim "Normaliza". Para realizar a soma a saída deverá ser "011".

Desloca_mantissa: essa parte será responsável por deslocar os dados do registrador de 16 bits, fazendo parte de mais um passo da operação de multiplicação. Ainda nesse estado o contador é decrementado em um, pois só assim em algum momento o contador *count* será zero. A saída desse estado será "100".

Normaliza: após a multiplicação ter sido efetuada e os expoentes somados temos o número em ponto flutuante, porém é necessário ainda normalizar o número, isso é, o MSB deverá ser igual à '1'. Para tal a saída será "101" que fará com que o registrador de 16 bits inicialize a normalização. A saída desse estado para "Expoente" só ocorrerá quando reg_control for igual à '1' (ele é o MSB citado).

Expoente: depois da normalização é necessário subtrair o número de deslocamentos com a soma dos expoentes efetuadas anteriormente. Para tal a saída da FSM será "111".

Final: chegou ao final das operações e o resultado final é disponibilizado. Só sairá desse estado caso for pressionado o *RESET*.

A tabela a seguir resume quais serão as saídas da máquina de estados, facilitando o encontro de possíveis erros.

Estado	**Saída (arg)**	**COUNT**
Inicial	000	X
Carrega_reg8	001	X
Dummy1	000	X
Carrega_reg16	010	X
Dummy2	000	X
Soma_expoentes	101	count = "1000"
Soma_mantissa	000	count = "0000"
Soma_mantissa	011	count != "0000"
Desloca_mantissa	100	count - 1
Normaliza	101	X
Expoente	111	X
Final	000	X

Abaixo segue um diagrama de estados que representa graficamente os estados denotados acima.

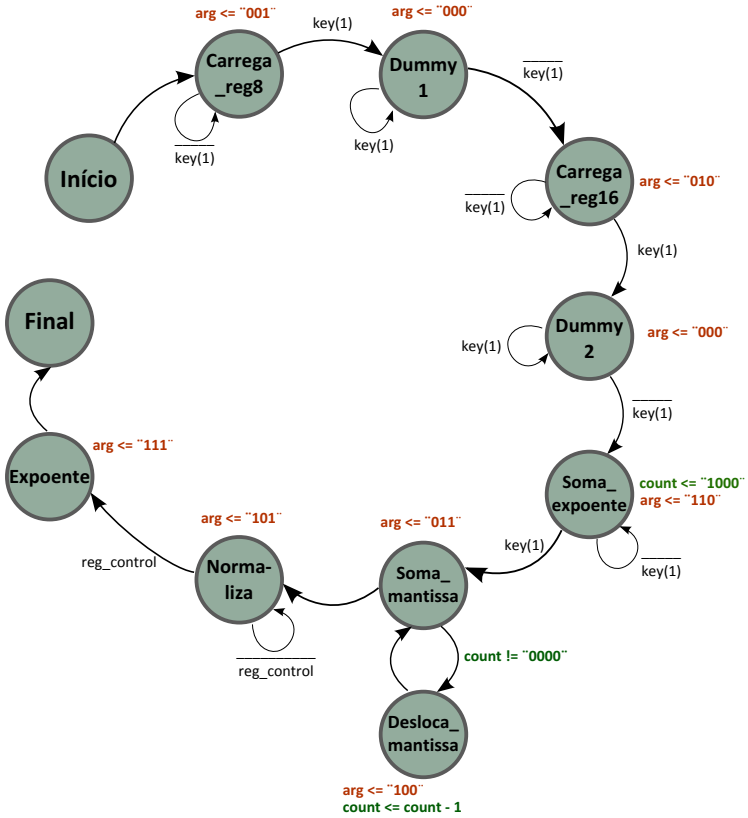


Figura 2.1: Diagrama da máquina de estados

2.6 Resultados e simulações

Após toda a implementação do circuito em VHDL foram realizadas simulações. Para tal foi usado o próprio Quartus II 9.1, que ainda inclui a ferramentas para simulações. No Quartus foi utilizada a ferramenta de simulação, disponível em *Processing -¿ Simulator Tool*, para realizar multiplas simulações de uma só vez.

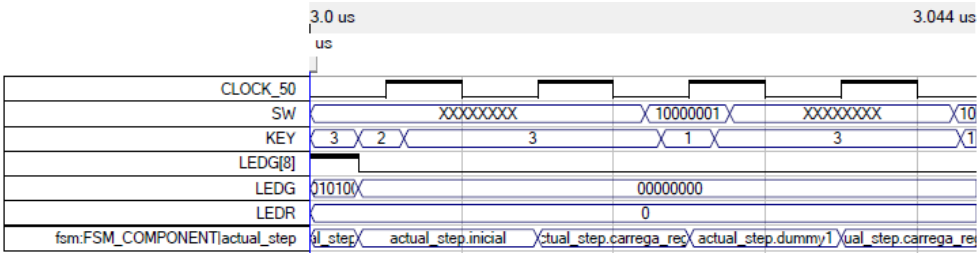
Uma observação a ser colocada é que pela proposta do projeto permitir apenas o uso de 8 LEDs para indicar o produto haverá perda de precisão nas multiplicações. Além disso a FPGA utilizada usa sinal alto para KEY desativada e sinal baixo para KEY ativada (pressionada). Segue uma tabela de algumas simulações realizadas.

Multiplicação	Mantis. 1	Exp. 1	Mantis. 2	Exp. 2	Resultado	Decimal	Esperado
8.71 ·128	10001011	0100	10000000	1000	$0.10001011 \cdot 2^{11}$	1112	1114.88
64.5 ·30.6	10000001	0111	11110100	0101	$0.11110101 \cdot 2^{11}$	1960	1973.7
671.125 ·64.5	10100111	1010	10000001	0111	Overflow	Overflow	43287.56
0 ·0	00000000	0000	00000000	0000	00000000	0	0
555.125 ·118.03125	10001010	1010	11101100	0111	Overflow	Overlow	65522.098
88.8 ·175.63	10110001	0111	10101111	1000	$11110001 \cdot 2^{14}$	15424	15595.94

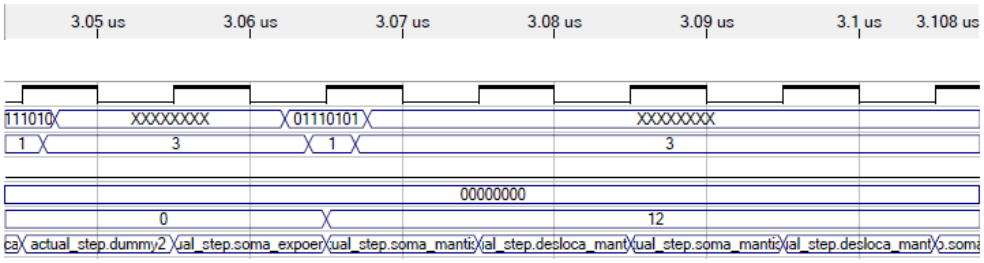
Obs.: "Mantis." significa Mantissa e "Exp." significa expoente.

Na tabela é possível ver a perda de precissão que é gerada pelo circuito e como ele não suporta números muito grandes. No caso de números grandes é indicado overflow, dado no projeto pelo LEDG(8).

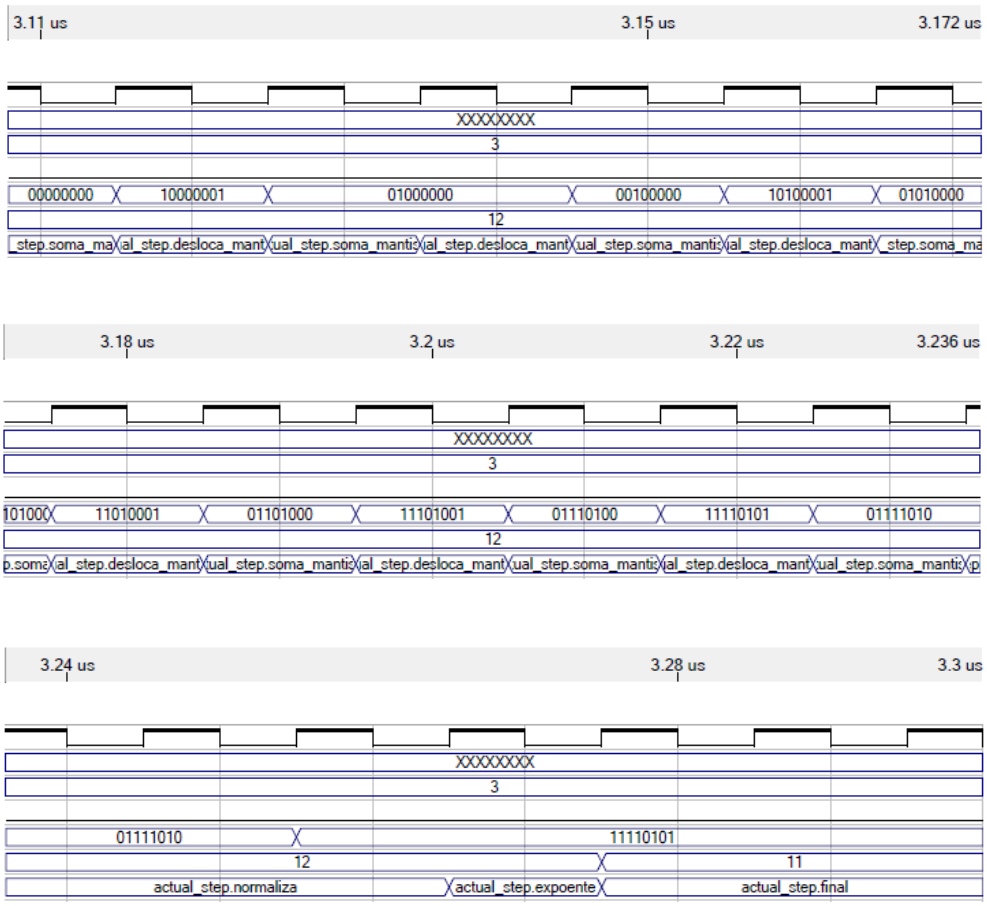
Os dados para simulação são colocados como sinais (alto ou baixo) que forma números binários. Para isso é usado um arquivo chamado *vector waveform* e após a simulação ter sido realizada tem-se o relatório. Segue o exemplo do relatório da multiplicação 64.5·30.6.



Quando KEY = 2, então tem-se que KEY(1) é 1 e KEY(0) é 0. Isso quer dizer que KEY(0) está sendo pressionado, fazendo com que o sistema seja reiniciado, voltando assim para o estado "inicial". Alguns pulsos de clocks e então tem-se o armazenamento de dados no registrador de 8 bits, quando KEY = 1 (tem-se KEY(1) igual a 0 e KEY(0) igual a 1). Segue-se o estado vazio, para esperar o KEY ser despressionado então tem-se o carregamento da parte baixa do registrador de 16 bits (KEY = 1 novamente). A forma de onda ficou cortada, porém é perceptível que ele guardou o dado pois passou do estado "carrega_reg16" para "dummy2".



Insere-se então os valores dos expoentes e pressiona-se o KEY(1), tornando KEY = 1 novamente e realizando a adição de ambos expoentes (checados em LEDR, penúltima linha da simulação). A partir disso não será mais necessário pressionar nenhum botão. Entra-se então no processo de soma e deslocamento, isso é, multiplicação das mantissas, a qual pode ser avaliada em LEDG (anti-penúltima linha da simulação).



Continua-se o processo de soma e deslocamento, até que se realize as oito vezes propostas. Após a última interação de soma e desloca (presente na penúltima imagem) tem-se o processo de normalização, que roda até que MSB do registrador de 16 bits seja '1'. Após isso entra-se no "expoente" onde realiza-se o decremento da soma dos expoentes com o número de deslocamentos da normalização e então tem-se o resultado final, representado nos LEDG e LEDR, sendo eles mantissa e expoente respectivamente. Não há overflow, como pode ser visto, levando em conta que a proposta do projeto relacionava o LEDG(8) para indicar a ocorrência de overflow.

O resultado, $0.11110101 \cdot 2^{11}$ pode ser traduzido para decimal e tem-se então 1960, muito próximo do produto original, que seria $64.5 \cdot 30.6 = 1973.7$. Comprova-se que o circuito para calcular o consumo de combustível se mostra bem eficaz, embora a perda de precisão.

Capítulo 3

Conclusão

Sobre o circuito nota-se que seu funcionamento é total, porém há muita perda de precisão pelo fato de ter-se poucos LEDs para exibição do resultado. Com dezesseis LEDs para a mantissa e talvez somente mais um para o expoente haveria uma maior precisão.

Relativo ao conhecimento é certo que após todo o desenvolvimento do projeto várias dúvidas relativas a VHDL foram sanadas. Aprendeu-se uma das várias aplicações do VHDL na indústria, embora é de conhecimento da equipe que não é bem assim que os sistemas industriais são efetuados pois esse projeto, por exemplo, não foi bem otimizado, porém já é possível ter-se uma idéia de como projetos desse tipo são elaborados.

Além da prática, a parte teórica da disciplina foi bem aprendida pois com o projeto foi possível aprender melhor o funcionamento de registradores, máquinas de estados, circuitos sequenciais e outros.