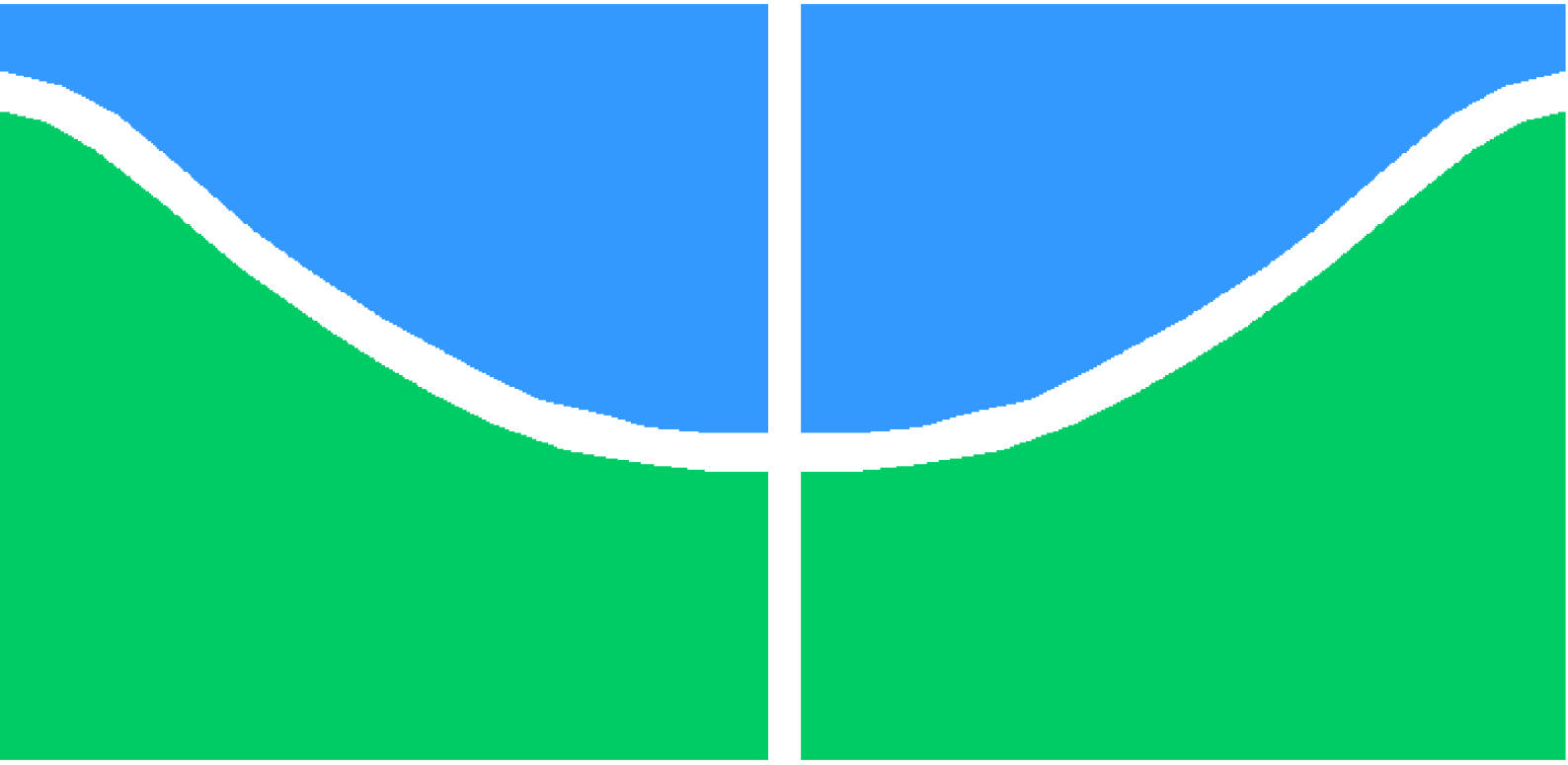

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia Eletrônica

Implementação de um Módulo de Leitura de ECG Abdominal em Gestantes para Estimativa da Frequência Cardíaca Fetal Usando FPGA

Autores: Helton Isamu Carvalho Tutida

Orientador: Prof. Gilmar Silva Beserra

Brasília, DF

2017

Helton Isamu Carvalho Tutida

Implementação de um Módulo de Leitura de ECG Abdominal em Gestantes para Estimativa da Frequência Cardíaca Fetal Usando FPGA

Monografia submetida ao curso de graduação em Engenharia Eletrônicada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Gilmar Silva Beserra

Brasília, DF

2017

Tutida, Helton I. C.

Implementação de um Módulo de Leitura de ECG Abdominal em Gestantes para Estimativa da Frequência Cardíaca Fetal Usando FPGA/ Helton Isamu Carvalho Tutida. Helton Isamu Carvalho Tutida – Brasília, DF, 2017-

71 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Gilmar Silva Beserra

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2017.

1. ECG abdominal. 2. FHR. I. Prof. Gilmar Silva Beserra . II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação de um Módulo de Leitura de ECG Abdominal em Gestantes para Estimativa da Frequência Cardíaca Fetal Usando FPGA

CDU 02:141:005.6

Helton Isamu Carvalho Tutida

Implementação de um Módulo de Leitura de ECG Abdominal em Gestantes para Estimativa da Frequência Cardíaca Fetal Usando FPGA

Monografia submetida ao curso de graduação em Engenharia Eletrônicada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 30 de Agosto de 2017:

Prof. Gilmar Silva Beserra
Orientador

Prof. Marcelino Monteiro de Andrade
Convidado 1

Prof. Daniel Mauricio Muñoz Arboleda
Convidado 2

Brasília, DF
2017

Agradecimentos

A Deus, fonte de toda minha força.

À minha família, os responsáveis por tudo que sou hoje, pelo apoio incondicional e por todos os valores a mim ensinados.

Ao meu pai, Marcelo Tutida, o homem que me fez ser quem eu sou, e ao meu irmão, Victor Tutida, que esteve sempre ao meu lado e que por sempre cuidar de mim se tornou a personificação do amor. São a minha fonte de inspiração diária.

A todos os meus queridos amigos que fiz ao longo da minha caminhada durante a graduação. Em especial ao Johnson Andrade, que me auxiliou com os seus conhecimentos e tornou possível a realização deste projeto.

Ao Brenddon Gontijo e Luan Talles, meus parceiros e irmãos de coração, que partilharam comigo todos os momentos dessa jornada e me motivaram a ser cada vez melhor.

A minha namorada, Beatriz Rodrigues de Alencar, a pessoa que me ensinou o significado do amor, que sempre me proporcionou alegria e nunca me abandonou nos momentos difíceis.

Ao meu orientador, Gilmar Beserra, pela paciência e compreensão, por todo o auxílio, incentivo e correções, tornando possível a conclusão deste trabalho.

Por último, dedico minhas palavras a mulher responsável por tudo que sou e tenho, Maria da Luz Carvalho Tutida, mãe. Por ser o motivo de todo meu esforço e dedicação. Todos os meus objetivos alcançados e que alcançarei serão dedicados à ela, minha heroína, que mesmo não estando conosco neste plano, sempre estará em meu coração, sendo a minha base aonde eu estiver.

Resumo

Na área de biomédica, o monitoramento da frequência cardíaca fetal (do inglês, FHR - *Fetal Heart Rate*) tem sido determinante para a obtenção de informações significativas acerca das reais condições do bebê dentro da barriga da mãe. Uma das maneiras não-invasivas de se estimar a FHR é através do eletrocardiograma fetal (FECG). Eletrodos são posicionados no abdômen materno e o sinal resultante é o ECG abdominal (AECG), que é composto pelo ECG materno (MECG), pelo FECG e por ruído. A partir do processamento do AECG, pode-se extrair o FECG e aplicar algoritmos de estimação para se obter a FHR. Considerando esse contexto, duas propostas foram desenvolvidas como temas de trabalho de conclusão de curso em Engenharia Eletrônica na Faculdade do Gama da Universidade de Brasília. A primeira consistiu na implementação de um módulo estimador da FHR baseado em FPGA, e a segunda, na realização da comunicação sem fio entre o FPGA e um dispositivo móvel com Android, que recebe o valor estimado e emite alarmes quando o mesmo ultrapassa os limites predefinidos. O objetivo deste trabalho é complementar a primeira proposta citada, implementando um módulo de leitura de maneira que vários sinais de AECG fiquem disponíveis para que o FPGA possa realizar o processamento e, posteriormente, enviar os valores da FHR estimada para o dispositivo móvel.

Palavras-chaves: FHR. ECG Abdominal. FPGA.

Abstract

In the biomedical field, monitoring the fetal heart rate (FHR) has been proven to provide meaningful information about the actual conditions of the baby inside the womb. One of the non-invasive methods to estimate the FHR is using the fetal electrocardiogram (FECG). Electrodes are placed on the maternal abdomen and the resulting signal is the abdominal ECG (AECG), which is composed by the maternal ECG (MECG), by the FECG and also by noise. The FECG can be extracted by processing the AECG, and the FHR can be obtained by applying estimation algorithms on the FECG. Considering this context, two final course projects were developed at the Faculty of Gama, University of Brasilia. One of them was an FPGA-based FHR estimator module, and the other one was the implementation of a wireless communication between the FPGA and a mobile device with Android operating system, which receives the estimated value and issues an alarm when it exceeds the predefined limits. The objective of this work is to complement the first proposal through the implementation of a reading module in order to provide previously collected AECG signals to the FPGA, so that it can carry out the processing and send the estimated FHR to the mobile device.

Key-words: Abdominal ECG. FHR. FPGA.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

FPGA	*Field Programmable Gate Array*
FHR	*Frequency Heart Rate*
ECG	Eletrocardiograma
FECG	Eletrocardiograma Fetal
AECG	Eletrocardiograma abdominal
MECG	Eletrocardiograma materno
SVD	*Singular Value Decomposition*
BSS	*Blind Source Separation*
RAM	*Random Access Memory*
PCI	Placa de Circuito Impresso
ICA	*Independent Component Analysis*
IOS	*Iphone or Ipad Operacional System*
FC	Frequência Cardíaca
bpm	Batimentos por Minuto
TCP	Protocolo de Controle de Transmissão
IP	Protocolo de Internet
CWT	Transformada em *wavelet* contínua
DWT	Transformada em *wavelet* discreta
PCA	Análise de Componentes Principais
ANN	Redes Neurais Artificiais
ASIC	*Aplication-Specific Integrated Circuit*
EDA	*Eletronic Design Automation*
PLD	Dispositivos de Lógica Programável
LUT	*Look-up Table*

Sumário

1 Introdução

1.1 Contextualização e Problematização

No final do século XVIII e início do século XIX, a área de cardiologia desenvolveu-se aceleradamente devido à invenção da eletrocardiografia. Essa tecnologia permitiu o aprofundamento nos conhecimentos relacionados aos mecanismos cardíacos, um melhor entendimento sobre as arritmias do coração e, o mais importante, estudar e desenvolver formas de tratamentos para a correção deste tipo de patologia. O termo ECG (eletrocardiograma) foi dado em 1887 pelo fisiologista inglês *Augustus Desiré Waller*, da *St Mary's Medical School* de Londres, que testou o primeiro ECG de superfície em um ser humano. O registro do traçado foi conseguido por meio da colocação de dois eletrodos: um na parte anterior do tórax e o outro nas costas. O primeiro era conectado a uma coluna de mercúrio do eletrômetro de Lippman e o segundo, ao ácido sulfúrico, a qual formava uma interface com a parte superior da coluna de mercúrio do eletrômetro (CARNEIRO, 1991).

Essa tecnologia ofereceu mais um recurso para as mulheres grávidas, visto que o monitoramento da frequência cardíaca do feto (FHR - *Fetal Heart Rate*) é um procedimento muito importante durante a gestação. Com esses dados, consegue-se basicamente obter o estado da saúde e detectar possíveis doenças cardíacas presentes no feto. A partir da realização de um ECG no abdômen materno (AECG), é possível extrair o eletrocardiograma fetal (FECG) e estimar a FHR. Além da obtenção de dados de muita importância para a análise clínica, trata-se de um método totalmente não invasivo. Entretanto, como os sinais vindos do AECG são compostos do ECG materno (MECG), do FECG e de muitos tipos de ruídos causados pelos instrumentos e pela atividade muscular da mãe, um dos grandes desafios desse método é como extrair o FECG. O FECG foi primeiramente observado por Cremer (1906), sendo que os primeiros trabalhos nesta área foram realizados com a utilização de aparelhos de galvanômetro da época, e foram limitados devido à baixa a amplitude dos sinais fetais.

Sendo assim, desde 1960 várias técnicas de processamento de sinais foram introduzidas para melhorar a qualidade da detecção do FECG. Dentre elas, pode-se destacar o uso de filtros adaptativos, a decomposição em valores singulares (SVD - do inglês *Singular-Value Decomposition*), a transformada *Wavelet*, redes neurais e separação "cega"de fontes (BSS - do inglês *Blind Source Separation*).

Encontra-se em desenvolvimento na Faculdade do Gama da Universidade de Brasília um protótipo para estimar a FHR a partir do AECG. Este protótipo utiliza uma placa com FPGA para realizar o cálculo da FHR. A FPGA foi escolhida pois será feita

uma aceleração com base em um co-projeto Hardware-Software, o qual permite explorar o uso de funções mais complexas na sua parte de software e o paralelismo das linguagens de descrição de hardware. (BARBOSA, 2016)

O protótipo é composto dos seguintes módulos:

- **Aquisição**: as funções deste módulo são obter os sinais de ECG do abdômen materno e disponibilizá-los para a parte de processamento;
- **Processamento**: este módulo extrairá o FECG, estimará a *FHR* e a enviará para o módulo de comunicação. Será utilizado um FPGA (*Field Programmable Gate Array*) para realizar a aceleração de algoritmos de processamento;
- **Comunicação**: este módulo deve enviar a FHR estimada para um dispositivo móvel;
- **Aplicativo**: o dispositivo móvel recebe as informações, mostra a FHR estimada e emite um alarme caso a mesma ultrapasse os limites predefinidos.

1.2 Objetivos

1.2.1 Objetivo Geral

Realizar a implementação de um módulo de leitura de sinais de AECG, de maneira que sinais previamente obtidos e gravados possam ser lidos e processados pelo FPGA.

1.2.2 Objetivos Específicos

Com o intuito de se alcançar o objetivo principal do trabalho, foram definidos os seguintes passos:

- Testar o leitor de cartão da placa DE1-SoC e/ou desenvolver uma solução para permitir a comunicação entre outro leitor de cartão e a mesma;
- Acrescentar uma interface para permitir a comunicação entre o processador MIPSfpga e o leitor de cartão;
- Alterar o algoritmo de processamento realizado em Barbosa (2016), de maneira que a placa realize o processamento de vários sinais lidos do cartão micro SD, sem a necessidade de reprogramar o FPGA.

1.3 Trabalhos Correlatos

Na dissertação de mestrado de Huseby (2013), foram descritos a concepção, o desenvolvimento e a análise de uma unidade de medição independente de ECG utilizando FPGA. O sistema completo é composto de um *front-end* analógico para aquisição de ECG, um cartão de aquisição de dados, um módulo baseado em FPGA para a execução dos algoritmos e para a detecção em tempo real dos picos QRS. Foi desenvolvida também uma aplicação para a plataforma iOS, sendo que os dados do módulo ECG-FPGA são enviados para um iPad através de uma conexão TCP/IP. As aquisições dos dados do ECG foram obtidas com os eletrodos posicionados no peitoral, um em cada lado (lado esquerdo e direito do tórax), e para o aterramento colocou-se um eletrodo na perna direita.

No Departamento de Eletrônica e Telecomunicação, localizado no Instituto Fr. C. Rodrigues de tecnologia Navi em Mumbai, Índia, realizou-se um estudo de extração do ECG fetal usando algoritmos diferenciais e análise através do LabVIEW (GADAKARI et al.,). Para análise de dados, foram efetuadas medições em 7 pacientes grávidas, sendo que os eletrodos foram posicionados de forma aleatória no abdômen e no toráx. Foram obtidos resultados próximos do desejado, pois, utilizando a ferramenta LabVIEW e algoritmos de separação do AECG, conseguiu-se extrair o sinal do FECG. A proposta foi utilizá-lo como um meio para as grávidas monitorarem a frequência cardíaca do feto sem precisarem sair de casa, visto que o equipamento é portátil e tem baixo custo.

Na Universidade de Brasília, Faculdade do Gama, encontra-se em desenvolvimento um protótipo para realizar a extração do FECG, realizar a estimativa da FHR através do uso de FPGA. A placa utilizada foi a DE1-SoC, da Altera, uma das propostas deste trabalho é que os códigos usados no desenvolvimento do mesmo possam ser utilizados em outras plataformas. Para que este objetivo possa ser alcançado, um MIPS 32-bits é utilizado como plataforma de software, o que permite a portabilidade para outros dispositivos (BARBOSA, 2016). Neste mesmo protótipo foi incluído como periférico, por (RODRIGUES,), um módulo de comunicação com objetivo de realizar a transmissão via Bluetooth a FHR estimada, através de algoritmos implementados em um FPGA para um dispositivo móvel com sistema Android, bem como desenvolver um aplicativo que analise a FHR e emita alarmes, caso a mesma ultrapasse limites predefinidos.

1.4 Contribuições

A contribuição principal deste trabalho para o projeto foi a não necessidade de ressintetizar o MIPSfpga de novo para que se faça a estimativa de uma nova FHR. O tempo de sintetização médio do projeto era entre 30 a 40min, fazendo com que o processador realize os cálculos utilizando os valores no cartão SD é necessário que se faça a sintetização e programação na placa apenas uma vez, pois toda vez que precisar ler um novo valor,

basta mudar os valores no cartão SD, inserir no módulo e pressionar o botão para que mostre a nova FHR.

1.5 Organização do Trabalho

Este documento está dividido da seguinte forma: o capítulo 2 contém a fundamentação teórica necessária para a elaboração do trabalho, abrangendo conhecimentos na área da eletrônica, e também uma breve explicação sobre a fisiologia cardíaca; o capítulo 3 contém os passos e ferramentas utilizados para a realização do trabalho; no capítulo 4, são descritas a arquitetura e a implementação do projeto, bem como são apresentados os resultados obtidos e as discussões; finalmente, no capítulo 5 são resumidas as conclusões e sugestões de trabalhos futuros.

2 Fundamentação Teórica

2.1 Fisiologia Cardíaca e Eletrocardiograma (ECG)

Inicialmente, é de suma importância a compreensão de algumas características fisiológicas do coração, para que posteriormente se possa entender o como se dá o processo de aquisição de sinais a partir do mesmo. O coração é formado em cerca de três semanas após a concepção do embrião e é o primeiro órgão que se torna funcional.

O sistema circulatório é o sistema de transporte de nutrientes para todo o corpo. Pode-se considerar que um embrião possui pouca reserva de nutrientes. Logo, é necessário que ocorra uma união entre ele e a mãe, através de uma circulação compartilhada, para que dessa forma o feto receba os suprimentos necessários.

O sistema circulatório de um adulto normal consiste basicamente em três partes:

- **Coração**: é o órgão que realiza a função de bombeamento para que todo o suprimento sanguíneo chegue aos tecidos que compõem o corpo humano;
- **Vasos sanguíneos**: são estruturas que direcionam o sangue tanto para os tecidos, como de volta ao coração. As veias retornam o sangue dos tecidos para os átrios, enquanto que as artérias levam o sangue dos ventrículos para os tecidos;
- **Sangue**: é o fluido que reúne diversos componentes, tais como oxigênio, gás carbônico, nutrientes, resíduos, eletrólitos e hormônios.

Anatomicamente, o coração é um órgão, porém os seus lados direito e esquerdo funcionam como duas bombas distintas. O coração é dividido nas metades direita e esquerda e possui quatro câmaras, conforme mostra a figura 1.

As câmaras superiores são os átrios, que tem as funções de receber o sangue que retorna ao coração e transferir paras os ventrículos, que são as câmaras inferiores. Os ventrículos, então, bombeiam o sangue para fora do coração.

As duas metades do coração são separadas pelo septo, que evita que o sangue se misture, visto que um é rico em oxigênio e o outro não.

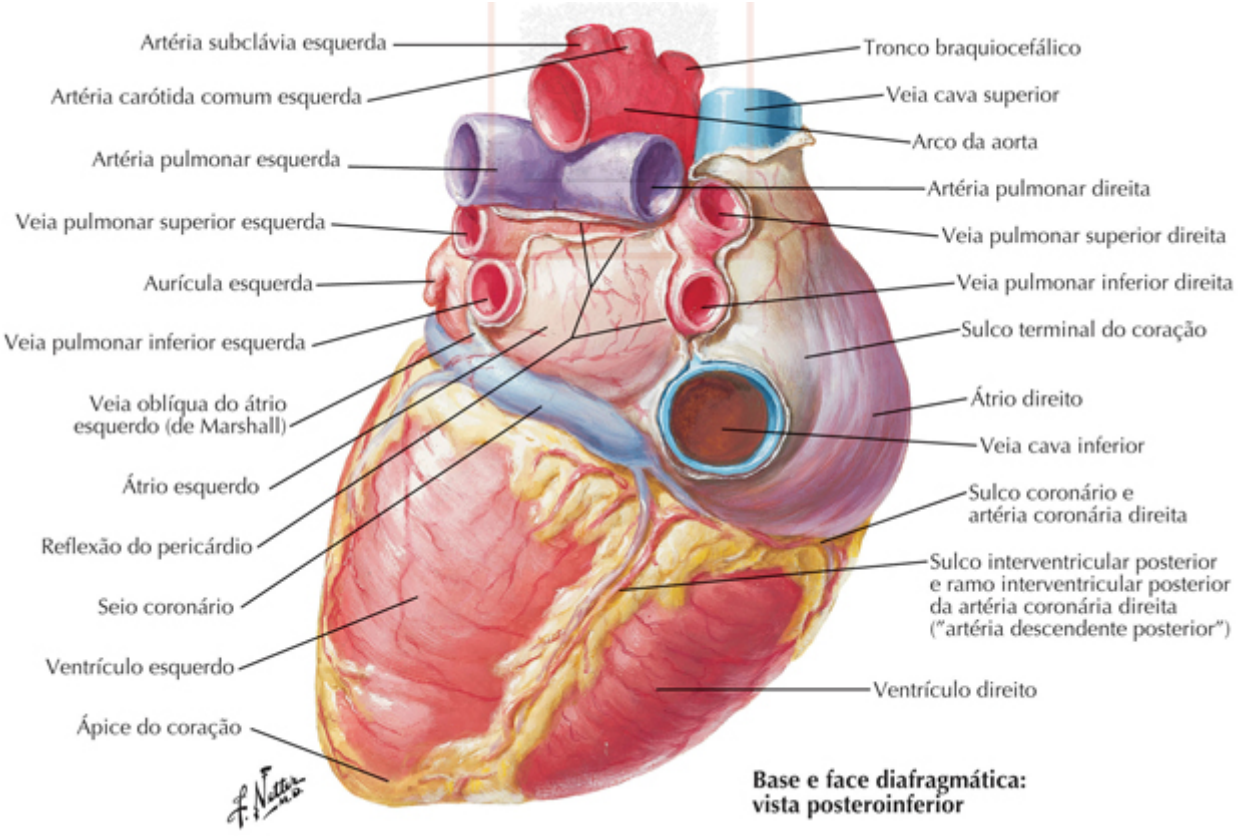

Figura 1 – Anatomia geral do coração. Fonte: (NETTER, 2008)

2.1.1 Ciclo Cardíaco

O conjunto dos eventos cardíacos que ocorre entre o início de um batimento e o início do próximo é denominado ciclo cardíaco (2). A contração do músculo cardíaco é fundamental para que o coração realize sua função de levar suprimento sanguíneo para todo o corpo, através do bombeamento. Ela depende da despolarização das células cardíacas, pois, para que a fibra muscular se contraia, é necessário que haja a despolarização da mesma fibra.

A ativação elétrica ordenada do coração se dá a partir da propagação dos potenciais de ação despolarizantes de forma sequencial. Cada ciclo se inicia no nodo sinusial, localizado na parede lateral superior do átrio direito. O início se dá com um potencial de ação espontâneo, o qual se dissipa por todo o miocárdio atrial direito e chega ao miocárdio atrial esquerdo, levando à contração dos átrios.

Essa ativação irá convergir para uma única conexão elétrica, no nodo atrioventricular, localizado entre o miocárdio atrial e ventricular. Após passar pelo nodo atrioventricular, a onda de ativação irá alcançar o feixe de *His*, localizado no miocárdio ventricular. Por fim, a onda de despolarização, também conhecida como impulso cardíaco, é propagada aos ventrículos direito e esquerdo. Esse fenômeno é conhecido como contração ventricular.

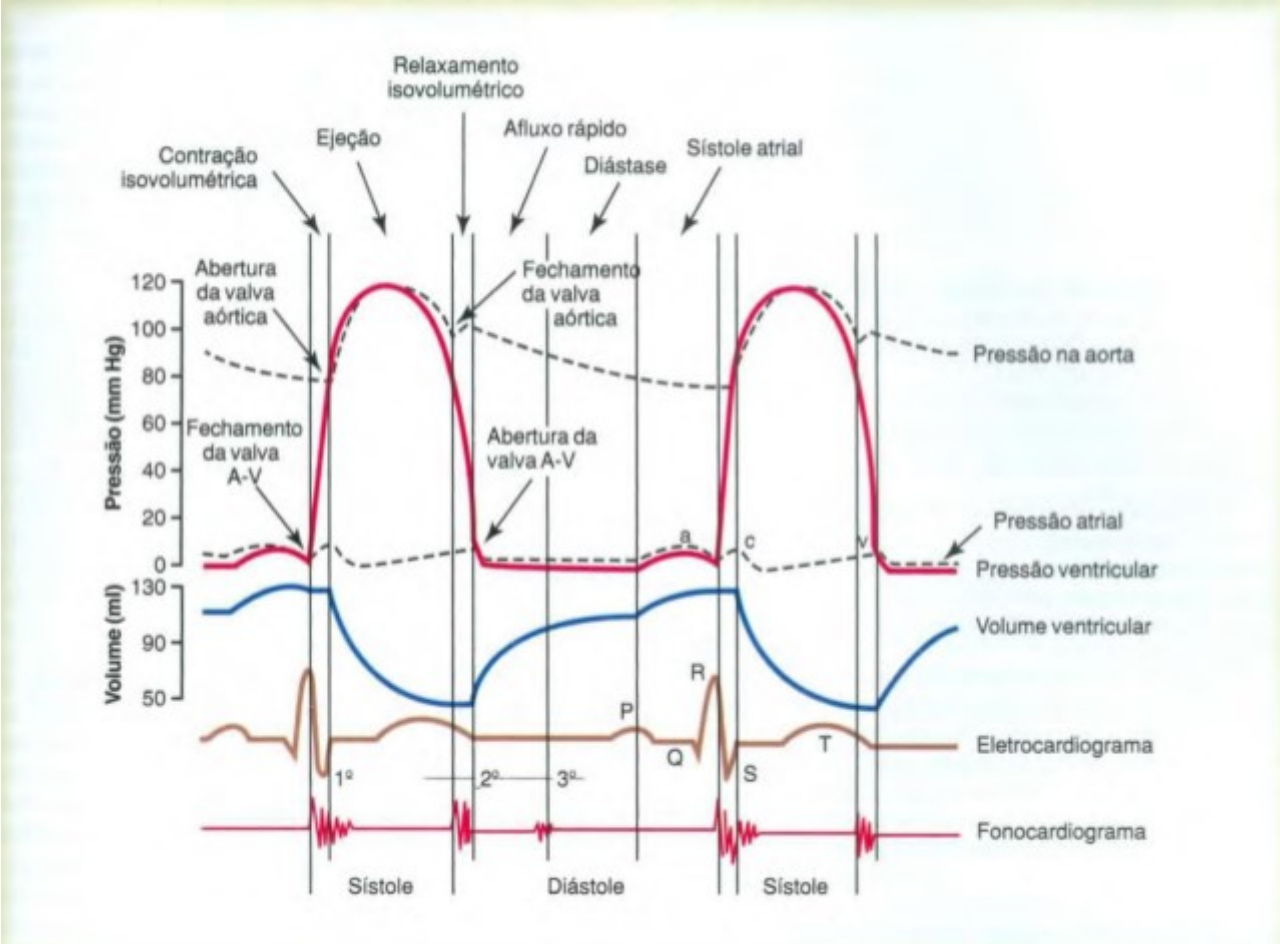

Figura 2 – Eventos do ciclo cardíaco para o funcionamento do ventrículo esquerdo, mostrando as variações na pressão do átrio esquerdo, na pressão da aorta, no volume ventricular, no eletrocardiograma e no fonocardiograma. Fonte: Hall (2015)

2.1.2 A Relação Entre o Ciclo Cardíaco e o ECG

A extração dos pulsos cardíacos adquiridos pelo ECG é feita através de um sensor chamado eletrocardiógrafo. Trata-se de um sensor de alta sensibilidade que registra a diferença de voltagens na superfície de contato.

O ECG apresenta as ondas P, QRS e T (figura 3), que são as ondas com voltagens elétricas geradas pelo coração e registradas posteriormente pelo eletrocardiógrafo presente na superfície do corpo. A onda P é causada pela disseminação da despolarização pelos átrios, e isso é seguido pela contração atrial, que gera um aumento discreto na curva de pressão imediatamente após a onda P eletrocardiográfica. Cerca de 0,16 segundos após o início da onda P, as ondas QRS surgem como resultado da despolarização elétrica dos ventrículos, que é responsável pela iniciação da contração ventricular, fazendo com que a pressão ventricular comece a aumentar.

Portanto, o complexo QRS começa pouco antes do início da sístole ventricular.

Finalmente, vê-se a onda T ventricular no eletrocardiograma, que representa o estágio de repolarização dos ventrículos, quando suas fibras musculares começam a relaxar. Sendo assim, a onda T surge pouco antes do final da contração ventricular.

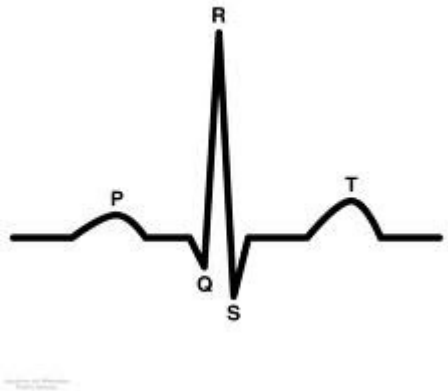

Figura 3 – Representação gráfica das ondas P,QRS,T de um ECG.

Numa análise eletrocardiográfica, deve-se examinar o intervalo entre essas ondas, como o intervalo P-R, o intervalo P-Q, o intervalo S-T, o intervalo Q-S, dentre outros. Estes intervalos possuem valores pré-definidos pela Sociedade Brasileira de Cardiologia (SBC), sendo que valores que estejam fora dos mesmos podem significar alterações estruturais e funcionais do coração. No ritmo normal do coração, denominado ritmo sinusal, o intervalo P-R possui uma duração entre 0,12 a 0,2 segundos; o intervalo QRS, entre 0,04 a 0,12 segundos; o intervalo R-R, entre 0,6 a 1,2 segundos, e a frequência cardíaca (FC) é de 60 a 100 batimentos por minuto (bpm) (CARDIOLOGIA, 2016).

As ondas mais importantes para determinar a eficiência do sistema de análise do ECG são as Q, R e S, que formam o complexo QRS (HASAN et al., 2009).

2.1.3 Arritmias Cardíacas

A arritmia cardíaca é o nome dado a uma desordem fisiológica que altera o comportamento do coração, podendo ter sua frequência ou ritmo alterado. A maioria dos tipos de arritmias cardíacas é inofensiva, não influenciando muito na vida da pessoa, podendo levar uma vida normal.

Porém, existem alguns tipos de arritmias que são consideradas mais graves, e se este problema for tratado a tempo pode ocorrer em sérias complicações, podendo até levar o paciente a óbito.

Dos diversos tipos de arritmias cardíacas, as mais comuns são a taquicardia, cuja frequência cardíaca é superior a 100 bpm, a bradicardia, no qual a frequência cardíaca é inferior a 60 bpm, a parada sinusal e a arritmia sinusal (CARDIOLOGIA, 2016).

2.1.4 ECG Abdominal

O ECG abdominal (AECG) de uma gestante é formado pelo ECG fetal (FECG) e pelo ECG materno (MECG), além de ruídos indesejados. Os eletrodos são colocados de forma aleatória no abdômen da mãe, não possuindo uma organização padrão para o posicionamento dos eletrodos.

A dificuldade da extração do FECG é devido ao fato do sinal do AECG vir com o ECG da mãe, que possui uma onda QRS com amplitude consideravelmente maior que a do feto, dificultando a leitura do FECG, pois as ondas da mãe podem encobrir a onda do feto. A figura 4 mostra um exemplo de AECG, onde o M indica o MECG, e o F, o FECG.

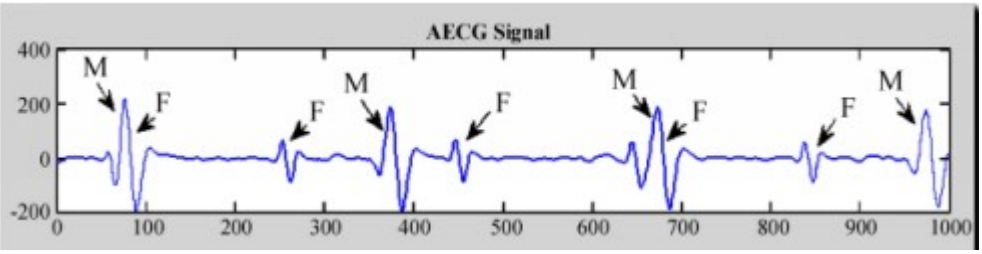

Figura 4 – Eletrocardiograma abdominal (AECG). Fonte: (HASAN et al., 2009)

2.1.5 ECG Fetal

A busca de técnicas não invasivas de análise de dados biomédicos é uma tendência crescente em todas as áreas da medicina, inclusive com relação à monitoração fetal e toda a captação de informações na fase pré-natal. Dentro desse contexto, surgiram várias técnicas para extração do FECG como: filtros adaptativos, a decomposição em valores singulares (*singular-value decomposition* ou SVD), a transformada *wavelet*, redes neurais e separação "cega"de fontes (*blind source separation* ou *BSS*).

2.2 FPGA (*Field-Programmable Gate Array*)

O desenvolvimento de projeto de circuitos digitais tem evoluído rapidamente nas últimas décadas. Segundo Compton e Hauck (2002), um dos métodos na computação tradicional para a execução de algoritmos é o que utiliza um circuito integrado específico para determinada aplicação, chamado de ASIC (*Aplication-Specific Integrated Circuit*), que é implementado em pastilha de silício. ASICs são altamente eficientes e rápidos para executar a tarefa ao qual foram determinadas a fazer. No entanto, após a fabricação deste circuito, ele não pode ser mais modificado. Esses circuitos são programados no ato da fabricação do dispositivo, tornando o custo fixo de produção extremamente alto. Outra desvantagem de seu uso é que se houver algum erro durante o projeto de programação, não será possível ser corrigido, sendo necessário descartar toda a produção.

Outro método que tem simplificado e acelerado exponencialmente esse processo é a utilização de ferramentas de *software* denominadas EDA (*Eletronic Design Automation*), juntamente com dispositivos de lógica programáveis PLDs (*Programmable Logic Devices*), que tem como vantagem a possibilidade de serem configurados pelo próprio usuário. Essa característica de ter a capacidade de programar as funções lógicas elimina todo o processo de fabricação do circuito integrado, o que torna mais fácil realizar as prováveis mudanças que ocorrerão no projeto. Sendo assim, em comparação com outras tecnologias de circuitos integrados digitais, os PLDs apresentam um ciclo de projeto menor e custos reduzidos.

Em 1985, uma empresa americana chamada *Xilinx Inc.*, apresentou um novo modelo de PLD chamado de FPGA (*Field-Programmable Gate Array*). A diferença entre um FPGA e um microcontrolador é a implementação de uma arquitetura eficiente, baseada em lógica programável estruturada para execução desse algoritmo de controle. Essa tarefa que traduz um algoritmo para uma arquitetura de *hardware* eficiente é chamada de síntese. A síntese elabora uma arquitetura com blocos lógicos que executam as operações do algoritmo implementado, sem a necessidade de se gerar e decodificar instruções.

A utilização de PLDs com essa nova arquitetura proposta é a possibilidade de se definir vários blocos de hardware, operando em paralelo, aumentando a capacidade computacional do sistema e ganhando tempo para a execução do processo. Logo, o FPGA pode ser chamado de *hardware* reconfigurável, que é um dispositivo constituído por um conjunto de blocos lógicos e de roteamento configurados por uma memória. Basicamente, a lógica e as conexões estabelecidas pelo *hardware* são determinadas pelos valores que estão armazenados nesta memória (GARCIA et al., 2006).

Os blocos lógicos são formados por um conjunto de blocos de granularidade fina (*fine-granularity*) que executam operações com largura de 1 *bit* e implementam a lógica de aplicação desejada. Também são organizados por uma matriz bidimensional, com os fios de interconexão sendo organizados como canais de roteamento horizontais e verticais entre os blocos lógicos. Esses canais de roteamento possuem fios e *switches* programáveis, permitindo que os blocos lógicos se conectem de diferentes formas.

Observa-se na figura 5 a arquitetura básica de um FPGA, que é composta pelos blocos de entrada e saída *(I/O)*, pelo bloco de configuração de controle, que é responsável por receber a lógica do usuário e transmitir para os blocos lógicos, e pelo bloco de controle de clock, que sincroniza e transmite o sinal de relógio para todos os outros componentes do circuito.

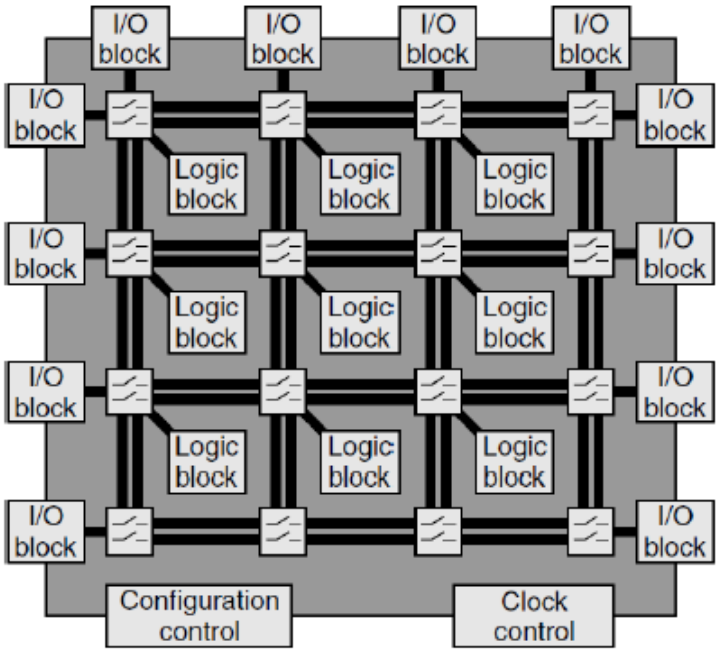

Figura 5 – Arquitetura Básica de um FPGA. Fonte: Bailey (2011)

2.2.1 Blocos Lógicos

Dentro de cada bloco lógico do FPGA, existem uma variedade de modos possíveis para implementação de funções lógicas. Um dos mais utilizados pelos fabricantes deste tipo de placa como, por exemplo, a empresa americana *Altera Corp.*, é o bloco de memória LUT (*Look-Up Table*). As LUTs possuem células de armazenamento que são utilizadas para implementar funções lógicas, sendo possível armazenar, em cada célula, apenas um valor lógico, zero (0) ou um (1). Geralmente, os blocos lógicos LUTs possuem quatro a cinco entradas (*I/O*), permitindo endereçar 16 ou 32 células de armazenamento.

Geralmente, é utilizada uma pequena memória *FLASH* ou *EEPROM* (*Electrically-Erasable Programmable Read-Only Memory*), cuja função é carregar automaticamente as células de armazenamento toda vez que o FPGA for energizado (COSTA, 2006). Sendo assim, no caso de falta de suprimento de energia elétrica, haverá a perda do conteúdo armazenado nas células LUTs do FPGA, pois elas são voláteis. Dessa maneira, o FPGA deve ser programado toda vez que for energizado.

Quando um circuito lógico é sintetizado em um FPGA, os blocos lógicos são programados para funcionar de acordo com as funções necessárias, ao mesmo tempo que os canais de roteamento são estruturados para realizar as interconexões entre esses blocos lógicos.

2.2.2 Arquitetura de roteamento

O roteamento de um FPGA é realizado através de seus barramentos e chaves de comutação (*switches*),que são organizados para permitir a interconexão entre os blocos lógicos. A Fig. 6 mostra essa arquitetura, e a descrição de cada estrutura é feita a seguir.

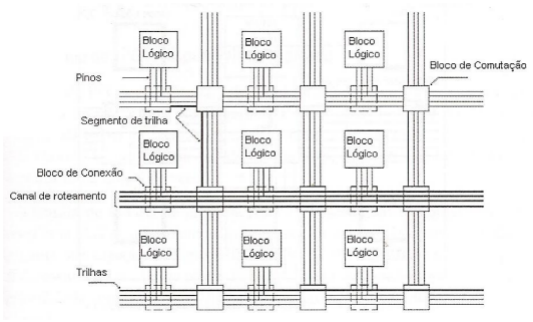

Figura 6 – Arquitetura básica de roteamento de um FPGA. Fonte: Costa (2006)

1. **Pinos**: entrada e Saída de blocos lógicos;
2. **Conexão**: ligação elétrica de um par de pinos;
3. **Rede**: conjunto de pinos que estão conectados;
4. **Bloco de Comutação**: utilizado para conectar os segmentos da trilha;
5. **Segmentos da Trilha**: segmento não interrompido por chaves programáveis;
6. **Canal de Roteamento**: grupo de 2 ou mais 3 trilhas paralelas;
7. **Bloco de Conexão**: permite a conectividade das entradas e saídas de um bloco lógico com os segmentos de trilhas nos canais.

2.2.3 Granularidade

Granularidade é uma característica dos FPGAs relacionada com o grão, que é a menor unidade configurável que compõe um FPGA. A fim de classificar os FPGAs quanto ao bloco lógico, foram criadas algumas categorias.

- **Grão Grande**: os FPGAs dessa categoria podem possuir como grão unidades lógicas aritméticas, pequenos microprocessadores e memórias.
- **Grão Médio**: os FPGAs de grão médio frequentemente contêm duas ou mais LUTs e dois ou mais flip-flops. A maioria das arquiteturas de FPGAs implementam a lógica em LUTs de quatro entradas.
- **Grão Pequeno**: os FPGAs de grão pequeno contêm um grande número de blocos lógicos simples. Os blocos lógicos normalmente contêm uma função lógica de duas entradas ou multiplexadores 4x1 e um flip-flop.

2.3 Processadores Embarcados

Os processadores embarcados geralmente são *chips* que executam tarefas que foram determinadas por meio de instruções e processam esses dados através de operações. Existem dois tipos de arquitetura para os processadores embarcados: RISC (*Reduced Instruction Set Computer*) ou CISC(*Complex Instruction Set Computer*). Os processadores CISC são capazes de executar operações complexas em uma única instrução, o que reduz consideravelmente a sua velocidade. O propósito dos processadores RISC é oferecer maior desempenho através da elaboração de um conjunto de instruções simples, que não gaste muito tempo para ser executada. Atualmente, a arquitetura RISC é a mais utilizadas pelos fabricantes.

Existem três formas que o processador embarcado pode ser implementado no FPGA: *soft*, *hard* ou *firm*.

- **Soft**: o processador é implementado em uma linguagem de descrição de hardware (HDL), sendo elaborada usando a lógica de propósito geral do FPGA.
- **Firm**: implementação feita em HDL igual à do *soft*, no entanto, é utilizada para uma arquitetura específica de FPGA.
- **Hard**: processador construído de silício, sendo fixo dentro do FPGA.

A Fig. 7 mostra modelos de processadores *hard* e *soft* em um FPGA.

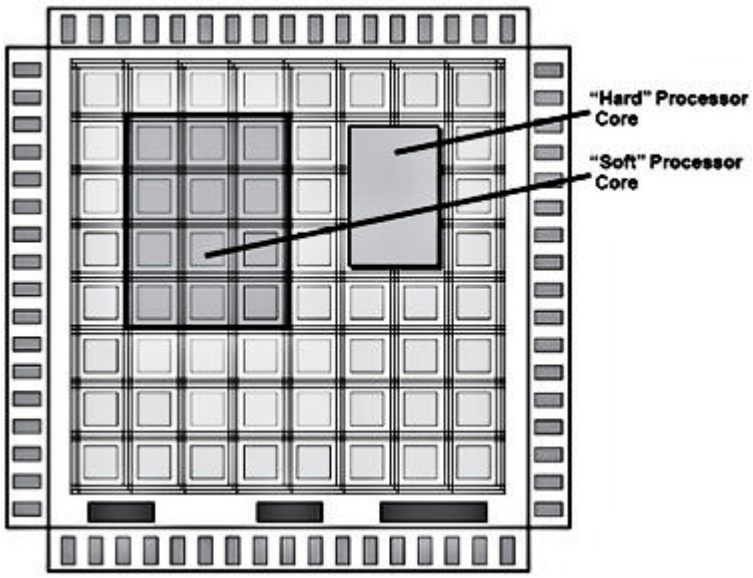

Figura 7 – Processadores *Hard* e *Soft* em um FPGA. Fonte: Abdelfattah e Betz (2012)

Algumas das vantagens de se utilizar processadores embarcados em FPGA são citadas a seguir.

- **Customização**: o processador tem a liberdade de escolher qualquer combinação dos controladores e periféricos da FPGA, e também a quantidade de memória e tipo que o usuário desejar, além da possibilidade da criação de um novo periférico que pode ser conectado diretamente no barramento do processador.
- **Redução dos componentes e custo**: como o FPGA possui diversos periféricos e controladores embarcados, não precisa de diversos componentes para a realização do projeto, tornando-se uma alternativa mais barata e com menor custo quando comparado com outros sistemas.
- **Aceleração de *Hardware***: a possibilidade de realizar co-projeto entre *hardware* e *software* no FPGA, torna-o mais eficiente e com um desempenho de sistema otimizado.
- **Capacidade de reprogramação tardia no projeto**: o processador embarcado no FPGA pode ser reprogramado mesmo quando o projeto já está em andamento.

2.4 Processador MIPS

No sistema para estimativa da FHR, será utilizado um processador embarcado com base na arquitetura MIPS devido às suas características e às vantagens em utilizá-lo implementado em um FPGA.

O MIPS (*Microprocessor with Interlocked Pipeline Stages*) é um processador RISC de 32 bits do modelo *Harvard*, ou seja, com memória de instruções e memória de dados separados. Ele possui uma quantidade menor de instruções no seu ISA (*Instruction Set Architecture*) que um processador CISC.

As instruções no MIPS são acessadas de quatro em quatro endereços. Isso ocorre porque a memória é endereçada por *byte* e, uma vez que uma palavra (*word*) contem 32 bits, é necessário buscar 4 bytes para obter uma instrução ou dado. O MIPS usa a semântica *Big Endian*, logo, os primeiros *bytes* buscados são os mais significativos da palavra.

Existem três tipos de instruções no MIPS. As instruções do tipo registrador (tipo R), do tipo Imediato (tipo I), e de desvio (Tipo J), como é apresentado na Fig. 8.

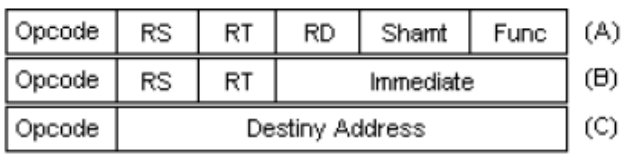

Figura 8 – Formato das instruções tipo R (A), tipo I (B) tipo J (C) Fonte: Hauser e Wawrzynek (1997)

- **Instruções tipo R**: realizam operações aritméticas e lógicas com dados buscados do banco de registradores, salvando no mesmo a resposta de operação. Este tipo de instrução possui seis campos. O campo *Opcode* (*Operation Code*) de 6 bits, responsável por informar que a instrução é do tipo R, e o campo Funct, também de 6 *bits*, determina qual é a operação a ser realizada. O campo RS (*Register Source*) informa qual é o registrador a ser lido para se obter o primeiro operando da instrução. Da mesma forma, o campo RT (*Register Target*) informa o registrador a ser lido também, porém, para o segundo operando. RD (*Register Destiny*), é responsável por informar o registrador que armazenará o resultado da operação de RS e RT. E, por último, o campo *Shamt* (*Shift Amount*), informa quantos *bits* deverão ser deslocados, em suas instruções de deslocamentos, tanto para esquerda quanto para direita.

- **Instruções tipo I**: são utilizadas de várias formas. As operações lógicas e aritméticas são semelhantes às do tipo R. A diferença é que o segundo operando vem no campo IMM (*Immediate*), possuindo 16 *bits*, junto da instrução buscada na memória, e o RT é o responsável por armazenar o resultado da operação. Na instrução tipo I, o campo *Opcode* é o responsável por informar a comparação feita entre RS e RT.

- **Instruções tipo J**: são instruções de desvio incondicional. O campo *Opcode* informa o código da operação a ser tomado, e o campo *Destiny Adress* de 26 bits contém o endereço de destino para o qual o programa será desviado. Desvio e retorno de subrotinas, assim como o as instruções de salto, enquadram-se neste tipo.

2.5 Comunicação e Transmissão de Dados

2.5.1 UART - *Universal Asynchronous Receiver/Transmitter*

A UART é um protocolo utilizado para implementar uma comunicação serial, possuindo duas linhas de transmissão de dados, sendo elas, TX(envio) e RX(recebimento). No envio dos dados, a UART cria um pacote de dados (conjunto de bits pareados e sincronizados), e os envia através da linha TX com um tempo preciso, de acordo com a taxa de transmissão(*boud rate*). No recebimento dos dados, a UART amostra a linha RX em taxas de acordo com a taxa de transmissão estabelecida, pega os bits sincronizados e obtém o dado. (INSTRUMENTS, 2010)

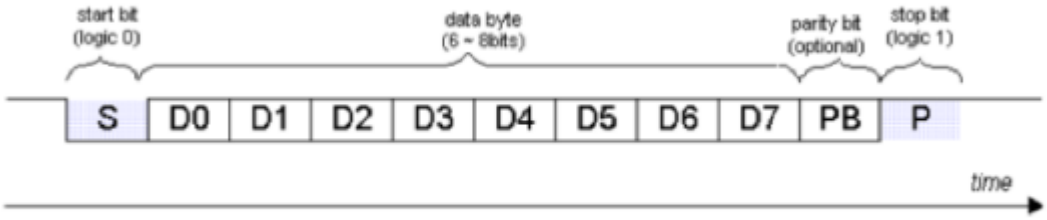

Figura 9 – Formato básico de uma UART. Fonte: SUMAN (2016)

A Figura 10 mostra o formato básico de uma UART contendo 8 bits de dado, 1 bit de início, 1 bit de paridade (opcional) e um bit de parada. Os bits são dispostos de maneira a serem enviados e recebidos serialmente, onde através dos bits recebidos é possível sua leitura de início e fim. O bit de início (*start bit*) indica ao receptor que o dado está se preparando para ser transmitido, e o bit de parada (*stop bit*) indica que a transmissão chegou ao fim. O bit de paridade (*parity bit*) é opcional e é usado para detecção e correção de erros.

3 Aspectos Metodológicos e Ferramentas

3.1 Aspectos Metodológicos

Esta seção descreve os passos utilizados na realização dos trabalhos de conclusão de curso 1 e 2, incluindo as alterações que foram necessárias para atingir os objetivos propostos.

3.1.1 Trabalho de Conclusão de Curso 1

A proposta inicial do trabalho era implementar uma interface para a comunicação entre o módulo de processamento implementado no processador MIPSfpga, embarcado no kit DE1-SoC (item 3.3.6), e um módulo de aquisição de ECG abdominal. Para essa última função, os materiais aos quais foi possível ter acesso foram uma placa de *front-end* analógico para ECG, disponibilizada juntamente com o *TMS320C5515 DSP Medical Development Kit* da Texas Instruments (item 3.3.1), e um cabo ECG, disponibilizado com o kit CSN 808 (item 3.3.4). O objetivo era efetuar medidas de ECG abdominal em 5 gestantes, a partir de uma ou mais configurações de eletrodos, e realizar o envio das mesmas para o módulo de processamento. Sendo assim, inicialmente foi realizada uma revisão bibliográfica para familiarização com os kits disponíveis e com as metodologias existentes para aquisição de ECG abdominal em gestantes. Posteriormente, foram sugeridas duas etapas para a realização da proposta: implementação do *hardware* de aquisição e coleta de dados.

Os resultados parciais apresentados no Trabalho de Conclusão de Curso 1 consistiram no teste dos kits através da medição de sinais de ECG torácicos efetuados com o kit CSN 808 em voluntários no próprio laboratório. Esses testes foram necessários para facilitar o entendimento de como seria feita a extração dos sinais que seriam posteriormente utilizados para processamento. Entretanto, não foi possível efetuar medidas com a placa de *front-end* analógico para ECG devido à incompatibilidade entre o conector da mesma (DB-15) e o do cabo ECG disponível (circular). Com a impossibilidade de aquisição de um cabo adequado à placa de *front-end*, foi desenvolvido um adaptador, mas o nível de ruído adicionado tornou a sua utilização inviável. Além disso, como o cabo disponível só possui 5 vias, a configuração de eletrodos para aquisição do ECG abdominal ficaria bastante limitada. Por fim, como a etapa de coleta de dados depende da aprovação de um comitê de ética, cujo processo via de regra é bastante demorado, foi recomendada pela banca uma alteração da proposta de maneira que o trabalho de bancada fosse priorizado e não dependesse desse processo. Com isso, no Trabalho de Conclusão 2, a proposta foi

modificada de maneira a excluir a etapa de coleta e os objetivos geral e específicos foram alterados para os expostos no item 1.2.

3.1.2 Trabalho de Conclusão de Curso 2

Uma vez realizada a alteração da proposta, o próximo passo consistiu na revisão das implementações já realizadas em Barbosa (2016) e na execução de seus primeiros tutoriais, a fim de se obter familiaridade com as ferramentas. Sendo assim, foram implementados projetos simples, como o de ligar e desligar leds, e o de um somador binário, com as chaves representando cada bit e os resultados sendo mostrados no display de 7 segmentos. Essas execuções mostraram como utilizar os periféricos da placa.

Em seguida, foram realizados tutoriais mais avançados para aprender a utilizar o processador MIPSfpga, executar algoritmos em C no mesmo e adicionar periféricos via barramento. Com isso, o objetivo era determinar como adicionar o leitor de cartão de memória como fonte de dados a serem processados pelo MIPSfpga.

Para obter os dados a serem gravados no cartão micro SD, foi utilizada uma função do MatLab que gera ondas de ECG. Neste caso, os dados foram salvos em dois arquivos no formato ".txt", sendo um para o abdômen e outro para o tórax. Esses dados foram inicialmente copiados no arquivo de memória do MIPSfpga e, após a compilação do módulo de processamento, a frequência cardíaca fetal da amostra foi calculada. Assim, foi possível também testar o módulo de processamento implementado em Barbosa (2016) com ondas de frequências diferentes, verificando-se se a FHR extraída e apresentada no display de sete segmentos da placa após o processamento seria igual à obtida no MatLab.

O próximo passo foi a realização dos testes com o cartão SD. Inicialmente, tentou-se utilizar o leitor da própria placa DE1-SoC. Porém, como o seu controlador está implementado no processador HPS (*Hard Processor System*) da Altera e a proposta é uma solução que possa ser facilmente implementada em plataformas de fabricantes diferentes, optou-se por utilizar um módulo externo de cartão SD, que utiliza uma interface SPI. O protocolo SPI foi testado efetuando-se uma comunicação entre o FPGA e o leitor externo de cartão SD. Porém, devido a dificuldades na implementação da interface, um Arduino foi utilizado como módulo intermediário entre o leitor de cartão e o FPGA. Sendo assim, a comunicação SPI foi implementada utilizando-se o Arduino como *master* e o leitor externo de cartão como *slave*, de maneira que primeiro efetua a leitura dos dados do cartão. Para que os dados possam ser enviados para o FPGA, a comunicação entre o mesmo e o Arduino é feita utilizando-se o protocolo UART.

Com o intuito de facilitar a manipulação dos dados de ECG a serem gravados no cartão micro SD, foi criado um código em C com o objetivo de ler ambos os arquivos de ECG gerados pelo MatLab e realizar a conversão dos mesmos para ponto fixo, no

formato binário de 32 bits. Esse algoritmo foi desenvolvido para agilizar o processo, que inicialmente era feito de forma manual (copiando e colando). Para que a conversão seja realizada de forma correta, é necessário que o código esteja no mesmo local que os arquivos com dados de ECG.

Por fim, a última etapa consistiu em configurar o projeto implementado em Barbosa (2016) de maneira que o processador compilasse as várias amostras disponíveis no cartão, ao invés de uma amostra por vez, sem a necessidade de reprogramar a placa para cada nova amostra, o que consistia em uma das maiores limitações do módulo de processamento.

3.2 Proposta Metodológica

Serão feitos testes com o kit da Altera DE1-SoC, com o arduíno e cartão SD, com o intuito de analisar o funcionamento de cada um para posteriormente serem implementados juntos. Sendo assim, esse trabalho foi desenvolvido nas seguintes etapas:

3.2.1 Etapa 1 - Fundamentação

- Estudos dos sinais de ECG;
- Estudo de projetos que incluem intefaces de comunicação com cartão SD;
- Aprendizagem do software para simulação e desenvolvimento de códigos em Verilog;
- Revisão e análise do estado da arte.

3.2.2 Etapa 2 - Implementação

- Funcionamento dos módulos trabalhando separadamente;
- Analisar tempo de cada processo;
- Verificar se é os métodos utilizados são confiáveis.

3.2.3 Etapa 3 - Verificação e Testes

- Adição de periféricos no MIPSfpga;
- Testes de comunicação da FPGA com o cartão SD via SPI;
- Aplicação do módulo de comunicação Uart no processador MIPSfpga;
- Adição do arduíno para ler os dados do cartão SD;

- Aplcação do módulo de comunicação como periférico do processador;
- Revisão dos recursos utilizados.

3.2.4 Etapa 4 - Documentação

- Elaboração de relatório de TCC.

3.3 Materiais e Ferramentas

Neste tópico, serão brevemente descritas as características e especificações dos materiais e ferramentas utilizados na execução deste trabalho.

3.3.1 *TMS320C5515 DSP Medical Development Kit* (MDK)

Trata-se de uma plataforma de desenvolvimento da *Texas Instruments* que inclui:

- Placas-filhas de *front-end* analógico (AFE) específicas para aplicações médicas (ECG, estetoscópio digital e oxímetro de pulso). Apesar de fazerem parte do MDK, essas placas podem ser usadas com outras plataformas.
- **C5515 EVM**: placa-mãe com C5515, um DSP baseado na plataforma DSP de ponto fixo C5000 da TI.
- *Software* de aplicações médicas, incluindo exemplos como demonstrações.

A fig. 10 mostra uma visão geral do *hardware* MDK, que consiste em placas de *front-end* analógico individuais para ECG, estetoscópio digital e oxímetro de pulso, além da C5515 EVM. Qualquer uma das placas de *front-end* pode ser ligada, uma de cada vez, no C5515 EVM, utilizando conectores universais.

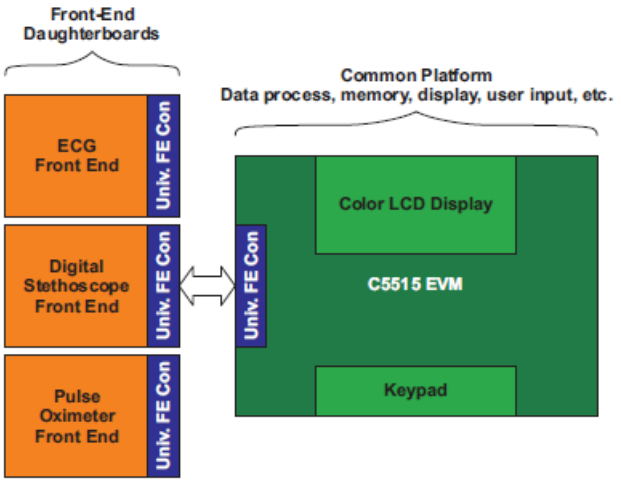

Figura 10 – Diagrama de blocos do *hardware* MDK Fonte: Instruments (1991)

As placas de *front-end* analógico utilizam sensores apropriados e realizam a conversão do sinal analógico para o digital (A/D). Depois, o sinal digital é enviado para a placa C5515 EVM, onde o C5515 DSP executa algoritmos de processamento de sinais para a aplicação. O DSP também é responsável pela gestão do controle e interação com o usuário, incluindo a visualização gráfica dos resultados, que podem ser transferidos para um PC, para análise posterior, e armazenados através de um *software* fornecido junto com o MDK.

3.3.2 Sistema MDK ECG

Para este trabalho, foi considerada apenas a placa de *front-end* analógico para ECG (figura 11), cujas características estão listadas a seguir:

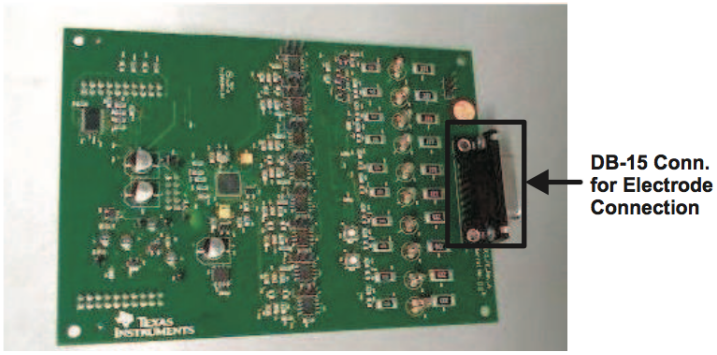

Figura 11 – Placa de *front-end* analógico para ECG Fonte: Instruments (1991)

- 12 derivações de ECG, através de 10 eletrodos
- Circuito de proteção para uso de desfibrilador

- Diagnóstico de qualidade de ECG com largura de banda de 0,05 Hz a 150 Hz
- Permite exibir a frequência cardíaca
- Detecção de derivações desconectadas
- Permite exibir as 12 derivações do ECG, em tempo real, sendo uma de cada vez na tela LCD do EMV, e três de cada vez na aplicação para PC
- Opção de zoom para o eixo Y (amplitude) no LCD
- Opção de zoom para os eixos X e Y na aplicação para PC
- Opção de congelar a tela na aplicação para PC
- Permite armazenar os dados e mostrá-los offline na aplicação para PC

Os componentes principais e interfaces do EVM que são usados em conjunto com o *front-end* analógico de ECG são:

- DSP C5515, operando em 100 MHz
- Porta USB
- EEPROM com interface I2C/SPI
- Interface para memória externa (EMIF) e interfaces I2C, UART e SPI
- SAR
- Interface JTAG
- Controlador JTAG embarcado
- Display LCD colorido
- Chaves

A placa de *front-end* analógico para ECG é conectada à EVM através de um conector universal usando interfaces I2C e I2S. Ela possui um conversor ADC com 16 canais, configurado para uma taxa de amostragem de 500 Hz, com resolução de 24 bits. O ADC se comunica com o DSP C5515 através de um barramento SPI.

3.3.3 Cabo EGC

O cabo de ECG possui 6 eletrodos para o peito e 4 para braços/pernas, e é conectado à placa através de um conector DB15. A figura 12 mostra um cabo com conectores banana.

Figura 12 – Cabo de ECG com 10 eletrodos do tipo banana

Conforme explicado no item 3.1.1, este cabo não se encontra disponível com o kit e não foi possível sua aquisição. Sendo assim, o cabo disponibilizado foi o do kit CSN 808, descrito no item a seguir.

3.3.4 Kit CSN 808

Trata-se de um kit multiparamétrico (ECG+SpO2+NIBP) que inclui o módulo de funções (figura 13), um cabo ECG com 5 eletrodos (figura 14), e outros cabos/conectores correspondentes que fogem ao escopo deste trabalho.

Figura 13 – Módulo de funções do kit multiparamétrico CSN 808

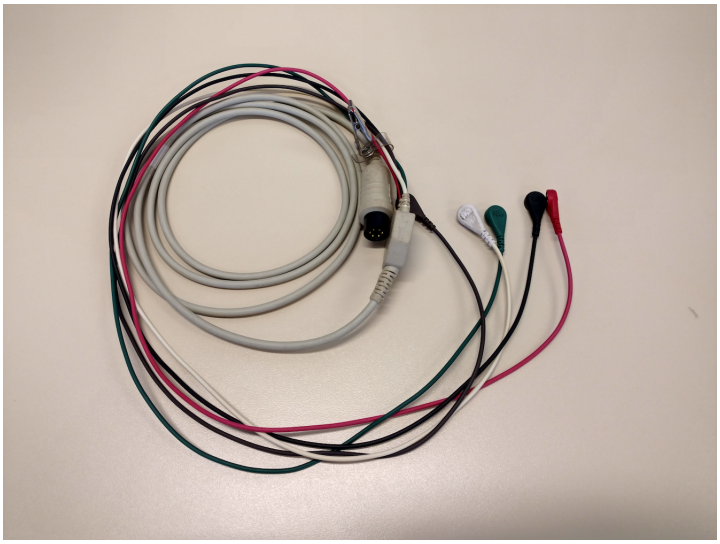

Figura 14 – Cabo de ECG com 5 eletrodos do kit CSN 808

3.3.5 Eletrodos

O sensor pode ser um eletrodo do tipo pulseira, que capta a variação dos sinais cardíacos nas extremidades do corpo, ou o eletrodo mostrado na figura 15.

Figura 15 – Eletrodo, Sensor de ECG. Fonte: Ramos e Sousa (2007)

Este tipo de sensor da figura 15 deve ser utilizado junto com um gel que adere à pele e conduz o sinal elétrico, permitindo assim a medição. Para a sua utilização, deve-se lembrar de limpar a pele, retirando ao máximo todo tipo de sujeira para evitar o mau contato e possíveis ruídos.

3.3.6 Kit DE1-SoC

A placa utilizada para realizar a leitura do sinal do AECG será a DE1-SoC Development Kit, da TERASIC, que é um *hardware* robusto, que possui o FPGA Altera System-on-Chip, que combina os núcleos dual-core Cortex-A9 com lógica programável para permitir flexibilidade, reconfigurabilidade e baixo consumo de potência. A integração entre o HPS (*Hard Processor System* baseado no ARM e que consiste no processador, periféricos e interfaces para memória, com a arquitetura do FPGA usa uma interconexão rápida, com grande largura de banda. A figura 16 mostra a divisão entre as duas partes (FPGA e HPS) através de um diagrama de blocos.

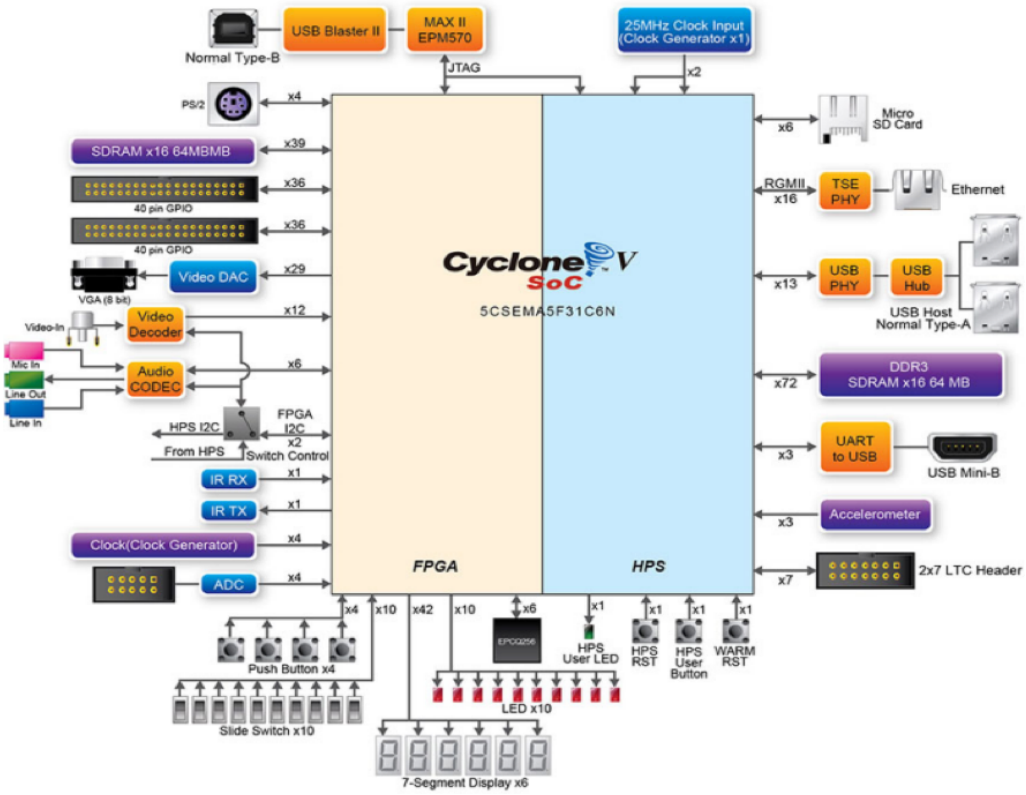

Figura 16 – Placa DE1-SoC representada por diagrama de blocos. Fonte: Terasic e Program (2015)

Algumas outras características da DE1-SoC estão listadas a seguir:

- SoC Cyclone V 5CSEMA5F31C6, da Altera;
- Dual-core ARM Cortex-A9 (HPS);
- 85K elementos de lógica programável;
- 2 tipos de RAMs, um DDR3-RAM de 1 gigabyte para a parte HPS do sistema, e uma memória SDRAM de 64MB na FPGA;
- Soquete para micro SD card no HPS;
- Um conector LTC (uma interface SPI Master, uma I2C e uma GPIO)
- Conversores A/D com taxa de amostragem de 500 KSPS, resolução de 12 bits e taxa de entrada analógica de 0 4.096V.
- Chaves, LEDs, displays de 7 segmentos

As saídas e entradas da placa estão localizadas nas suas bordas, para permitir um fácil e rápido acesso. Ela possui muitos recursos que servem para fazer desde projetos simples até os mais complexos. A figura 17 representa a placa com seus recursos disponíveis.

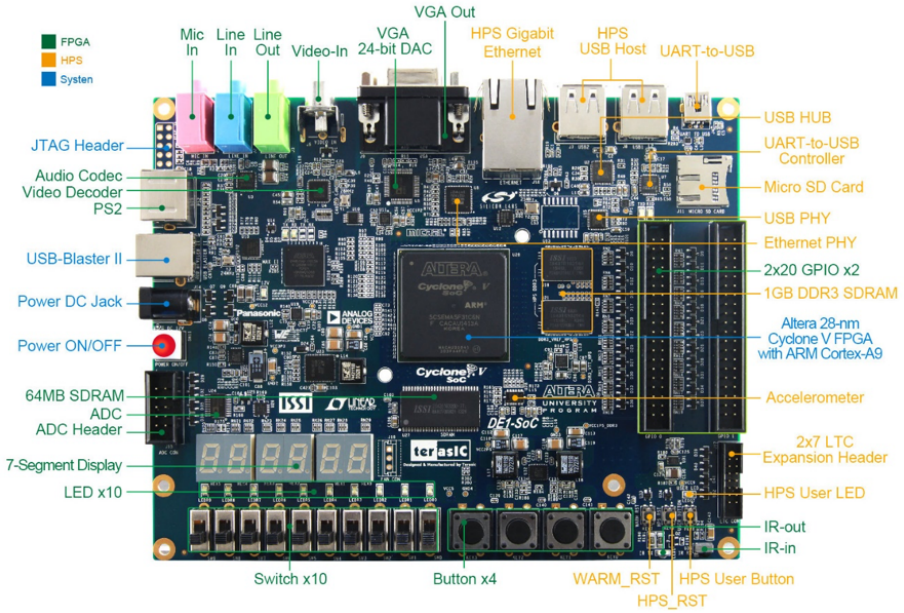

Figura 17 – Placa da DE1-SoC da TerasIC. Fonte: Terasic (2016)

3.3.7 Cartão Micro SD

O cartão de memória SDC (*SD Card* ou *Secure Digital Card*) é um dispositivo removível de armazenamento em memória *flash* usado em diversos dispositivos eletrônicos portáteis. O padrão SD foi projetado e licenciado pela *SD Card Association* (Panasonic, SanDisk Corporation e Toshiba Corporation) como uma evolução do padrão *MultiMediaCard* (MMC), com especificações sobre a forma do cartão, camada física, performance de escrita, entre outras. Os SD Cards possuem a característica de baixo consumo - não mais que 100 mA para escrita ou leitura - operando na frequência padrão até 25 MHz (SANDISK, 2004), são relativamente simples, de longa duração, baixo custo, e são pequenos, havendo 3 variações de tamanho: SD padrão, mini SD e micro SD. Os tipos de cartões são mostrados na figura 18.

Figura 18 – Tipos de cartão SD Fonte: Sabia (2017)

O SDMC da SanDisk apresenta dois protocolos básicos de comunicação. Um deles é o próprio de cartões SD, com um pino de comando (CMD), um pino de clock (CLK)

e quatro de entrada e saída de dados (DAT0-3). Já o protocolo utilizado neste projeto é do tipo SPI com 3 pinos, sendo um de seleção de chip (CS), um de clock (CLK), um de entrada de dados (MOSI) e um de saída de dados (MISO). A Tabela 1 apresenta os pinos do SD e suas respectivas funções em ambos os protocolos de comunicação (SANDISK, 2004).

Tabela 1 – Descrição dos pinos SD

Pino	Protocolo SD		Protocolo SPI	
	Função	Tipo	Função	Tipo
1	DAT3	Entrada/Saída	CS	Entrada
2	CMD	Entrada/Saída	MOSI	Entrada
3	GND	Alimentação	GND	Entrada
4	VCC	Alimentação	VCC	Entrada
5	CLK	Entrada	CLK	Entrada
6	GND	Alimentação	GND	Entrada
7	DAT0	Entrada/Saída	MISO	Saída
8	DAT1	Entrada/Saída	Reservado	Entrada
9	DAT2	Entrada/Saída	Reservado	Entrada
MOSI: Master Output Slave Input – Saída do mestre, entrada do escravo				
MISO: Master Input Slave Output – Entrada do mestre, saída do escravo				

A tabela de alocação de arquivo (FAT) é o nome de uma arquitetura de sistema de arquivos. Com a evolução dos drives de disco, diferentes versões do formato FAT foram criadas, as quais foram nomeadas com o número de bits para endereçamento dos dados: FAT12, FAT16 e FAT32. A maioria dos SD Cards são pré-formatados com uma ou mais partições MBR (*Master Boot Record*), onde a primeira ou única partição contém o sistema de arquivos, podendo ser qualquer variação da formatação FAT (R., 2012). A unidade básica de alocação do FAT é denominada *cluster* e corresponde, geralmente, a um conjunto de setores de 512 bytes. No FAT32, o tamanho padrão de um *cluster* em um volume entre 256 MB e 8GB é 4 KB (MICROSOFT, 2015). Cada arquivo aloca a quantidade de *clusters* necessária para seu armazenamento. Espaços vazios em *clusters* já alocados por arquivos não podem ser compartilhados com arquivos diferentes.

O número de dispositivos portáteis para armazenamento de dados digitais aumentou bastante nos últimos anos. Exemplos desses dispositivos são os pendrives e os cartões de memória. No primeiro caso, há uma maior utilização no transporte de dados entre dispositivos (JUNIOR, 2013). Já os cartões de memória são muito utilizados em dispositivos portáteis, como uma fonte extra de memória. Tais dispositivos são baseados em memória *flash*, um tipo específico de memória EEPROM que possui estrutura de acesso em blocos, o que aumenta sua velocidade em comparação às demais EEPROMs(SABIA, 2017).

O cartão escolhido para utilização no projeto foi o compatível com o kit DE1-SoC, que é do tipo micro SD. Além disso, o mesmo pode ser facilmente adaptado para

leitores de cartões SD de tamanho normal, que são comumente encontrados como fonte de armazenamento extra em eletrocardiógrafos.

3.3.8 Módulo externo de cartão SD

O módulo externo usado, mostrado na figura 19, foi para cartão SD de tamanho normal.

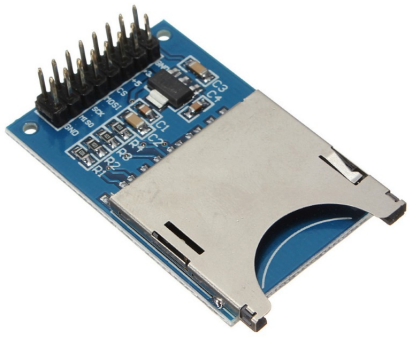

Figura 19 – Módulo de cartão SD Fonte: INFINITO (2017)

Esse módulo possui as saídas de cada pino do cartão SD, sendo eles: dois de GND, dois de alimentação, sendo um de 3,3 V e outro de 5 V, CS (*chip select*), MOSI(*Master Out Slave In*), MISO (*Master In Slave Out*), e sCLK (*serial clock*). A função de cada pino é descrita a seguir:

- **Ground (Gnd)**: aterrar o circuito;
- **Alimentação (3,3V e 5 V**): alimentar o circuito;
- **Chip Select**: utilizado para indicar quando o cartão vai começar a receber os dados vindos do Mestre;
- **MOSI**: pino utilizado para o cartão receber os dados do Mestre;
- **MISO**: pino utilizado para o Mestre receber dados do escravo (no caso, o módulo com o cartão SD);
- **sCLK**: sincroniza o cartão com o Mestre, para que funcione corretamente.

3.3.9 Software

3.3.9.1 Quartus II (Altera)

O Quartus II é o software de desenvolvimento de projetos da Altera para FPGAs que possibilita a análise e síntese utilizando VHDL ou Verilog, análises de desempenho

do sistema, dos diagramas RTL e da reação de diferentes estímulos a uma entrada. Para este projeto, utilizou-se a versão 16.0.1 da Quartus II Web Edition, disponível para Linux e Windows.

3.3.9.2 MatLab

O MatLab é uma linguagem de alto nível que disponibiliza de um ambiente interativo para computação numérica, simulação, visualização e programação, também podendo ser usado para análise de dados, criação de algoritmos, dentre inúmeras outras aplicações das mais simples até as mais complexas (MATHWORKS, 2005). Algumas de suas características estão listadas abaixo:

- Ambiente interativo para exploração, projetos e solução de problemas;
- Funções matemáticas para álgebra linear, estatística, análise de Fourier, filtragem, otimização, integração numérica e solução de equações diferenciais ordinárias;
- Ferramentas para criação de gráficos;
- Ferramentas de desenvolvimento para melhoria da qualidade de códigos e maximização de performance;
- Ferramentas para criação de aplicações com interface gráfica customizável.

  Para este projeto, utilizou-se a versão 2016.

3.3.9.3 Ardiono IDE

Trata-se de uma ferramenta baseada em linha de comando para se desenvolver projetos para o Arduino. Com ele, é possível compilar *firmware* a partir de diversos arquivos-fonte e bibliotecas, bem como fazer o *upload* do mesmo para um determinado dispositivo e realizar a comunicação serial (INOTOOL, 2017).

4 Implementação e Resultados

Considerando-se os passos e limitações descritos no capítulo 3, este capítulo contém os detalhes das implementações realizadas para permitir a leitura dos dados de sinais ECG gravados em um cartão micro SD por parte do processador MIPSfpga, no qual o módulo de processamento dos sinais que estima a FHR é executado.

4.1 Limitações do módulo de processamento

A maioria das propostas que usam filtros adaptativos para estimar a frequência cardíaca fetal baseia-se em algoritmos como o de Mínimos Quadrados (vide (HATAI; CHAKRABARTI; BANERJEE, 2013) e (PRASANTH; PAUL; BALAKRISHNAN, 2013)) e apresenta uma arquitetura semelhante. Sendo assim, ela também foi utilizada como base para o módulo de processamento desenvolvido em Barbosa (2016), cujo diagrama de blocos está representado na figura 20.

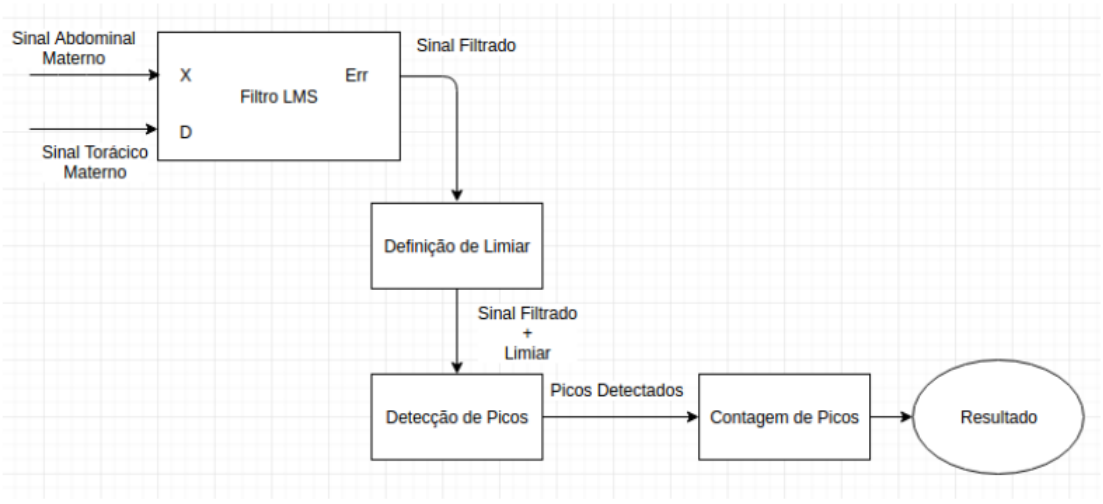

Figura 20 – Diagrama de blocos do projeto desenvolvido em Barbosa (2016)

Essa arquitetura foi implementada como *software* embarcado no processador MIPSfpga. Após passar pelo filtro LMS, o sinal já apresenta picos bem destacados. Após o limiar ser definido, os picos que ficam acima do mesmo são detectados e contados. Assim, para se obter a quantidade de batimentos por unidade de tempo, é necessário dividir o resultado da contagem pelo intervalo de tempo em que a mesma foi feita. Esse resultado é convertido para BPM (batimentos por minuto) e apresentado como resultado final (BARBOSA, 2016).

Entretanto, essa arquitetura funciona apenas com sinais simulados, gerados no MatLab. Para que os sinais simulados se aproximassem ao máximo dos sinais reais de ECG, foram adicionados ruídos gaussianos, bem como ruídos com coeficiente aleatório a fim de simular a propagação dessas ondas pelos tecidos do corpo. Cada sinal gerado representa uma amostra de ECG, a partir da qual obtém-se a frequência fetal em BPM. Outra limitação é que o filtro realiza a operação em apenas uma amostra, sendo necessário recompilar a arquitetura e reprogramar a placa para extrair a FHR cada vez que uma nova amostra é obtida.

4.2 Etapa inicial

A proposta deste trabalho é a implementação da comunicação entre o processador MIPSfpga e um leitor de cartão SD para realizar a estimativa da frequência cardíaca fetal a partir de várias amostras de ondas de AECG (ECG abdominal), sem a necessidade de recompilar e reprogramar a placa para cada amostra, resolvendo assim uma das limitações descritas no item anterior. Com isso, o comportamento do protótipo se aproximará do esperado, que é realizar o monitoramento de pacientes e mostrar o resultado para diferentes ondas de AECG, à medida em que as mesmas vão sendo amostradas.

A etapa inicial consistiu na realização de tutoriais para se obter familiaridade com as ferramentas e com o processador MIPSfpga. Os resultados dessa etapa estão mostrados no Anexo A, que mostra como implementar o processador no kit DE1-SoC. O Anexo B mostra como executar códigos em C através de exemplos simples, como acionamento de chaves, leds e displays e implementação de um somador binário.

4.3 Teste do módulo de processamento

Para efetuar o teste, o projeto apresentado em Barbosa (2016) foi compilado e a placa foi programada. O resultado foi a exibição de uma frequência fetal extraída da amostra que estava sendo utilizada naquele momento, que é igual a 142 bpm.

Para testar o módulo de processamento, os valores contidos no arquivo de memória do MIPSfpga (chamado "$mipsfpga\_ahb\_memory.v$"), que são utilizados no processo de filtragem, foram alterados. Para isso, foram geradas novas ondas no MatLab com valores de FHR distintos (por exemplo, 147 bpm). Nesta etapa, foram utilizados os algoritmos implementados com base na função ecg() do MatLab, conforme os critérios explicados em Barbosa (2016). Executando-se o projeto novamente, foi possível verificar que os valores estimados e exibidos nos displays correspondiam aos do MatLab, conforme mostram as figuras 21 e 22.

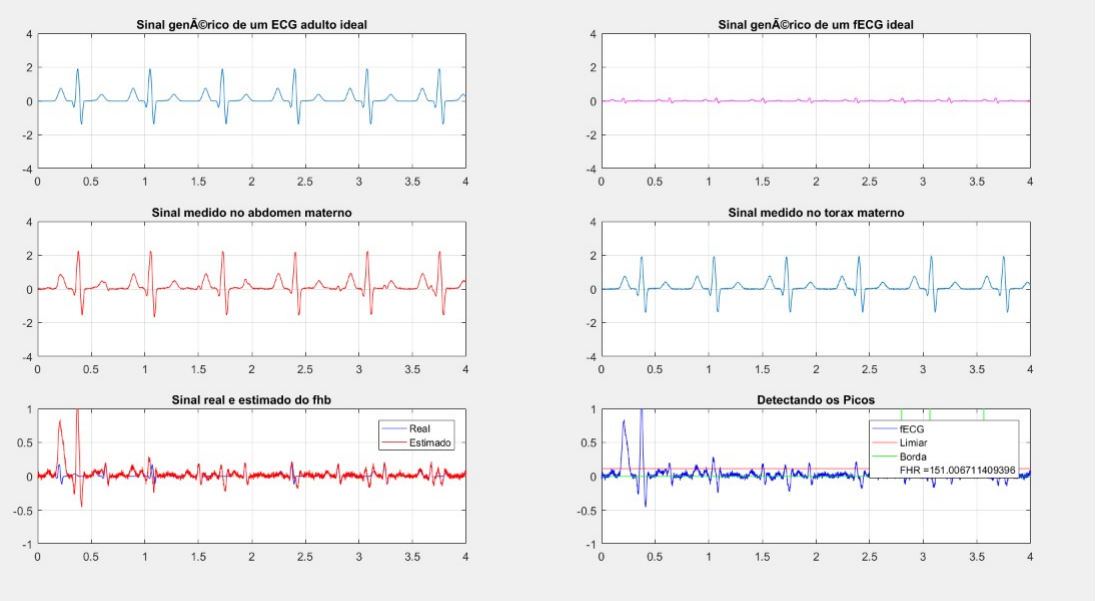

Figura 21 – Nova amostra de ECG gerada no MatLab, com FHR aproximadamente igual a 151 bpm

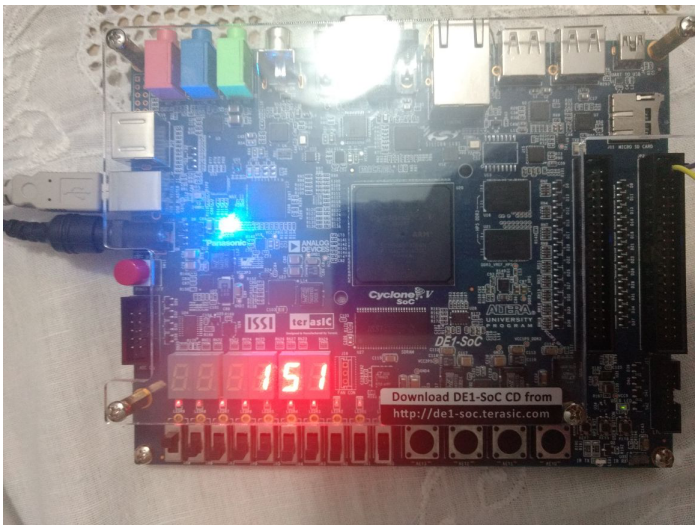

Figura 22 – FHR estimada e mostrada nos displays de 7 segmentos

Com o objetivo de facilitar o processamento de novas amostras, foi criado um código em C que converte os valores dos arquivos gerados no MatLab (abdomen.txt e torax.txt) para ponto fixo, representados no formato binário de 32 bits. Este código encontra-se no Anexo C.

A próxima etapa foi a implementação da interface para leitura do cartão micro SD. Para isso, foi necessário utilizar um adaptador de cartão de micro SD para SD comum, pois o módulo externo utilizado possui a entrada de cartão de tamanho tradicional.

4.4 Implementação do módulo de leitura

O kit DE1-SoC possui uma entrada para leitura de cartão micro SD. Entretanto, seu controlador está implementado no processador HPS (*Hard Processor System*). Apesar de haver a possibilidade de usar o processador MIPSfpga para acessar esse controlador via barramento, optou-se por utilizar outra solução, tendo em vista facilitar a portabilidade do protótipo, ou seja, a possibilidade de implementá-lo em kits de diferentes fabricantes.

Sendo assim, a opção escolhida foi utilizar um leitor externo de cartão SD e incluí-lo como um periférico do MIPSfpga. Para efetuar a comunicação, foi criada uma interface para implementar o protocolo SPI entre o FPGA e o leitor. A interface consistiu em uma máquina de estados, cujo diagrama está mostrado na figura 23. Porém, ocorreram erros durante os testes de comunicação. Os resultados mostraram que a mesma não conseguia chegar sempre ao estado desejado (vide transição em vermelho na figura 23), havendo falhas de comunicação quando o *reset* da placa era acionado.

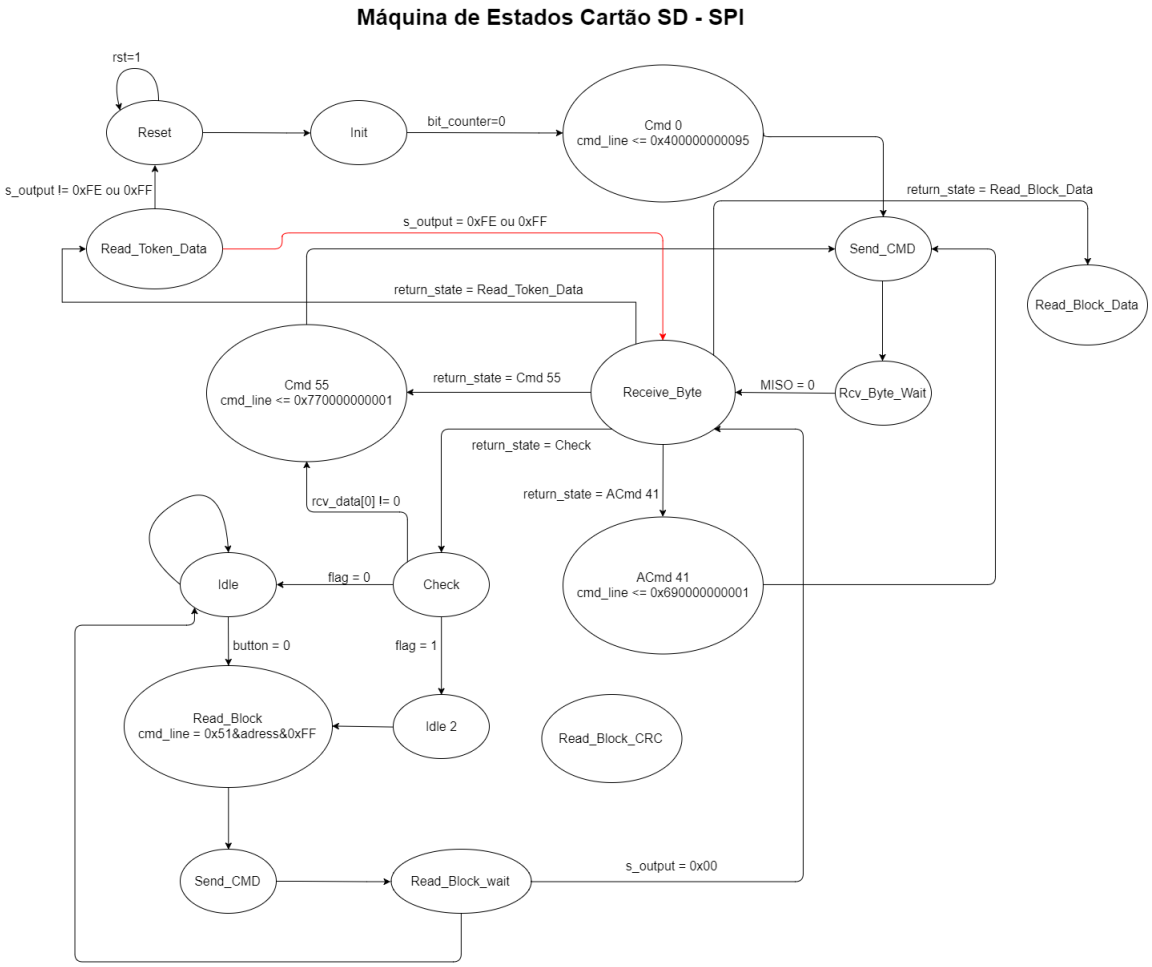

Figura 23 – Diagrama da máquina de estados da interface SPI para comunicação entre o FPGA e o leitor de cartão

Como este procedimento não estava confiável, a solução encontrada foi utilizar o

protocolo UART, descrito em 2.5.1. Em termos de *hardware*, a utilização da UART implica na adição de um novo componente e alguns fios(*jumpers*). Tendo em vista a simplicidade de implementação, o Arduino foi escolhido, pois ele já possui uma biblioteca de comunicação com o cartão SD. Com o leitor externo de cartão SD conectado ao Arduino, sendo este conectado ao FPGA, foi possível fazer a comunicação entre eles. A figura 24 mostra o sistema completo.

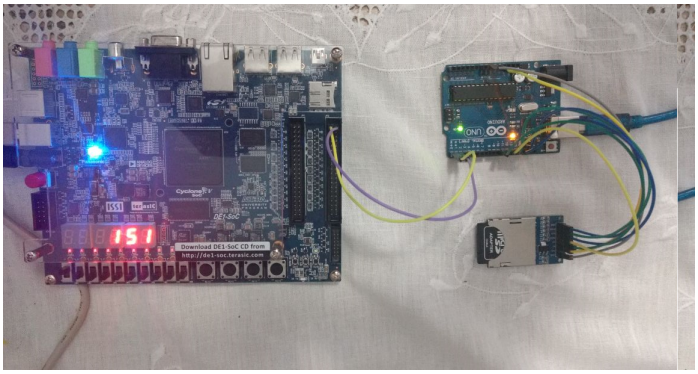

Figura 24 – Sistema completo, composto pelo FPGA, Arduino e leitor de cartão

Os pinos utilizados no módulo SD foram o MOSI, MISO, CLK, CS (*chipselect*), e os de alimentação, Vcc (5V) e terra (GND). Os pinos são conectados às entradas disponíveis no Arduino, que, por sua vez, são definidas no código (figura 25).

Figura 25 – Pinos do Arduino utilizados pelo leitor de cartão

Foi desenvolvido um algoritmo em linguagem C responsável para ler os dados do cartão SD. O código foi elaborado a partir de uma biblioteca existente no próprio Arduino, denominada $< sd.h >$, que serve para fazer a leitura dos dados de ECG contidos no cartão SD através da comunicação SPI. A estrutura do algoritmo se divide em três funções básicas. A primeira verifica se há algum cartão SD válido inserido no módulo. A segunda verifica se o arquivo *.txt* contido no cartão está com o nome correto para que a leitura possa ser efetuada. A terceira e última parte lê as amostras dentro do arquivo e as envia, bit a bit, para o canal TX do Arduino. O código completo se encontra no Anexo D.

Para que o FPGA possa receber esses valores, ou seja, para que ocorra a comunicação do FPGA com o Arduino, foi elaborado um módulo em Verilog descrevendo a transmissão RX. Vale observar que é necessário que o MIPSfpga apenas receba os valores que são lidos pelo Arduino.

No protocolo UART, a transmissão é realizada em pacotes de 8 bits, enviados serialmente. Sendo assim, foi criado um vetor de 32 bits para armazenar cada dado da amostra, de maneira que o mesmo será preenchido com os valores do cartão SD, lidos pelo Arduino. Quando o vetor é preenchido, o FPGA indicará que está pronto para enviar esse vetor para que o processador MIPSfpga realize o cálculo.

O módulo UART foi implementado em Verilog e adicionado como periférico do processador MIPSfpga, sendo instanciando através do arquivo *mips_ahb_gpios*. O Anexo E mostra ambos os arquivos.

4.5 Alteração no algoritmo de processamento

A última etapa consistiu em alterar o código C principal do módulo de processamento, que é responsável por fazer os cálculos da FHR. O código original lê os valores de um outro módulo presente no processador, chamado *mips_ahb_memory.v*. Este arquivo contém uma amostra de ECG (que consiste em vinte mil números binários), que era manualmente inserida antes de se sintetizar o processador na placa. Com a alteração proposta neste trabalho, será possível executar o processador na placa sem a necessidade de reprogramá-la a cada nova amostra, bastando apertar o botão designado para iniciar a estimativa da FHR. O código pode ser visto em Anexos F.

O tempo para sintetizar o MIPSfpga demora entre 30 a 40 minutos, e toda vez que precisava mostrar outro valor de FHR era necessário ressintetizar o projeto completo. Após a implementação do módulo de leitura no processador, é necessário sintetizar apenas uma vez o projeto, e após programar na placa, basta trocar os valores no cartão SD, inserir no módulo e pressionar o botão de *start*, para que mostre a nova FHR.

Os componentes presentes no projeto são acessados através do endereço físico declarado em um módulo que contém todos os endereços de todos os componentes do processador, denominado *mipsfpga_ahb_const.v*, representado na Fig. 26

```
1  //
2  // mipsfpga_ahb_const.vh
3  // Verilog include file with AHB definitions
4
5  //-------------------------------------------
6  // Memory-mapped I/O addresses
7  //-------------------------------------------
8  `define H_LEDR_ADDR      (32'h1f800000)
9  `define H_LEDG_ADDR      (32'h1f800004)
10 `define H_SW_ADDR        (32'h1f800008)
11 `define H_PB_ADDR        (32'h1f80000c)
12 `define H_7SEG_ADDR      (32'h1f800010)
13 `define H_TX           (32'h1f800014)
14 `define H_RX           (32'h1f800018)
15 `define H_RXCONTROL      (32'h1f80001c)
16 `define H_CONTROLE       (32'h1f800020)
17 `define H_MEMDATA     (32'h1f800024)
18
19 `define H_LEDR_IONUM         (4'h0)
20 `define H_SW_IONUM           (4'h2)
21 `define H_PB_IONUM           (4'h3)
22 `define H_7SEG_IONUM            (4'h4)
23 `define H_TX_IONUM           (4'h5)
24 `define H_RX_IONUM           (4'h6)
25 `define H_RXCONTROL_IONUM      (4'h7)
26 `define H_CONTROLE_IONUM       (4'h8)
27 `define H_MEMDATA_IONUM     (4'h9)
28
29
30 //-------------------------------------------
31 // RAM addresses
32 //-------------------------------------------
33 `define H_RAM_RESET_ADDR       (32'h1fc?????)
34 `define H_RAM_ADDR          (32'h0????????)
35 `define H_RAM_RESET_ADDR_WIDTH       (14)
36 `define H_RAM_ADDR_WIDTH       (16)
37
38 `define H_RAM_RESET_ADDR_Match       (7'h7f)
39 `define H_RAM_ADDR_Match       (1'b0)
40 `define H_LEDR_ADDR_Match      (7'h7e)
```

Figura 26 – Código com os endereços dos componentes

Após a alteração, a Fig. 27 apresenta os resultados da síntese.

Flow Summary

<<Filter>>

Flow Status	Successful - Thu Aug 31 00:05:25 2017
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	SD_FPGA
Top-level Entity Name	SD_FPGA
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	145 / 32,070 (< 1 %)
Total registers	154
Total pins	27 / 457 (6 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)

Figura 27 – Resultados da síntese no software Quartus 2.

5 Conclusões

Um dos parâmetros mais significativos para avaliar as reais condições de um feto é a frequência cardíaca fetal (FHR). Sendo assim, é fundamental efetuar o monitoramento das pacientes, sobretudo as que possuem gravidez de risco. Dentre as várias maneiras de se estimar a FHR a partir do ECG abdominal (AECG), pode-se destacar a implementação de arquiteturas que fazem uso de filtros adaptativos. Em Barbosa (2016), foi proposta e implementada em FPGA uma arquitetura para efetuar essa estimativa usando filtro LMS. Entretanto, uma das limitações foi a necessidade de recompilar e reprogramar a placa para cada amostra de AECG. Neste trabalho, foi realizada a implementação da comunicação entre o processador MIPSfpga e um leitor de cartão SD para realizar a estimativa da FHR a partir de várias amostras de ondas de AECG, sem a necessidade de recompilar e reprogramar a placa para cada amostra. Com isso, o comportamento do protótipo se aproxima do esperado, que é realizar o monitoramento de pacientes e mostrar o resultado para diferentes ondas de AECG, à medida em que as mesmas vão sendo amostradas.

Foram geradas novas amostras de sinais simulados de AECG utilizando o MatLab e as mesmas foram gravadas em um cartão micro SD, então desenvolveu-se uma solução que permite a comunicação entre um leitor externo de cartão e o FPGA, utilizando o Arduino como módulo intermediário, que recebe os valores das amostras do cartão usando o protocolo SPI.

O próximo passo foi desenvolver uma interface UART para realizar a comunicação entre o processador MIPSfpga e o Arduíno. O módulo de processamento realizado em Barbosa (2016) foi alterado, de maneira que a placa realiza o processamento dos sinais lidos do cartão micro SD, sem a necessidade de reprogramar o FPGA.

Entretanto, após as alterações no módulo de processamento, observou-se que o mesmo está processando apenas uma amostra do cartão, ou seja, para processar uma nova amostra, é necessário apagar a anterior e escrever a nova no cartão. Apesar dessa desvantagem, não é necessário ressintetizar e reprogramar a placa, como acontecia anteriormente.

Considerando todos esses aspectos, algumas sugestões de trabalhos futuros são:

- Alteração do módulo de leitura para permitir o processamento sequencial de várias amostras;
- Teste do sistema integrado (módulo de leitura, módulo de processamento e módulo de comunicação) para envio da FHR para o aplicativo em um dispositivo com Android;

- Utilização de uma placa de front-end para realizar as medidas de AECG em gestantes, e implementação da comunicação entre essa placa e o MIPSfpga;
- Alteração do módulo de processamento para ler sinais de bases de dados, que podem ser lidos a partir do cartão SD.

Referências

ABDELFATTAH, M. S.; BETZ, V. Design tradeoffs for hard and soft fpga-based networks-on-chip. In: IEEE. *Field-Programmable Technology (FPT), 2012 International Conference on*. [S.l.], 2012. p. 95–103. Citado 2 vezes nas páginas 7 e 25.

BAILEY, D. G. *Design for embedded image processing on FPGAs*. [S.l.]: John Wiley & Sons, 2011. Citado 2 vezes nas páginas 7 e 23.

BARBOSA, I. J. T. Aceleração de algoritmos para estimativa da frequência cardíaca fetal utilizando fpga. 2016. Citado 8 vezes nas páginas 7, 14, 15, 30, 31, 42, 43 e 49.

CARDIOLOGIA, S. B. de. 2016. Disponível em: <http://www.cardiol.br/>. Citado na página 20.

CARNEIRO, E. F. O eletrocardiograma. In: *O eletrocardiograma*. [S.l.]: Enéas Ferreira Carneiro, 1991. Citado na página 13.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, ACM, v. 34, n. 2, p. 171–210, 2002. Citado na página 21.

COSTA, C. da. Projetando controladores digitais com fpga. *Editora Novatec, primeira ediçao, Maio de*, v. 9, 2006. Citado 3 vezes nas páginas 7, 23 e 24.

CREMER, M. *Über die direkte ableitung der aktionsströme des menschlichen herzens vom oesophagus und über das elektrokardogramm des fötus*. [S.l.]: Lehmann, 1906. Citado na página 13.

GADAKARI, R. M. et al. Non-invasive extraction of fetal ecg using differential algorithm and analysis using labview. Citado na página 15.

GARCIA, P. et al. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, Hindawi Publishing Corp., v. 2006, n. 1, p. 13–13, 2006. Citado na página 22.

HALL, J. E. *Guyton and Hall textbook of medical physiology*. [S.l.]: Elsevier Health Sciences, 2015. Citado 2 vezes nas páginas 7 e 19.

HASAN, M. A. et al. Detection and processing techniques of fecg signal for fetal monitoring. *Biological procedures online*, Springer-Verlag, v. 11, n. 1, p. 263, 2009. Citado 3 vezes nas páginas 7, 20 e 21.

HATAI, I.; CHAKRABARTI, I.; BANERJEE, S. Fpga implementation of a fetal heart rate measuring system. In: IEEE. *Advances in Electrical Engineering (ICAEE), 2013 International Conference on*. [S.l.], 2013. p. 160–164. Citado na página 42.

HAUSER, J. R.; WAWRZYNEK, J. Garp: A mips processor with a reconfigurable coprocessor. In: IEEE. *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*. [S.l.], 1997. p. 12–21. Citado 2 vezes nas páginas 7 e 26.

HUSEBY, M. K. Fpga based development platform for biomedical measurements: Ecg module. 2013. Citado na página 15.

INFINITO, H. 2017. Disponível em: <http://www.huinfinito.com.br/modulos/487-modulo-cartao-sd-micro-compativel-5v-33v-.html?search_query=sd&results=13>. Citado 2 vezes nas páginas 7 e 40.

INOTOOL. 2017. Disponível em: <http://inotool.org/>. Citado na página 41.

INSTRUMENTS, T. *TMS320c5x user's guide*. [S.l.]: Texas Instruments, 1991. Citado 2 vezes nas páginas 7 e 33.

INSTRUMENTS, T. Keystone architecture universal asynchronous receiver/transmitter (uart) user guide. *Texas Instruments User Guide*, 2010. Citado na página 27.

JUNIOR, J. L. W. O. AplicaÇÃo de registro de dados de um acelerometro digital de 3 eixos baseado em fpga. *UNIVERSIDADE FEDERAL DO CEARÁ CENTRO DE TECNOLOGIA DEPARTAMENTO DE ENGENHARIA ELÉTRICA CURSO DE ENGENHARIA ELÉTRICA*, 2013. Citado na página 39.

MATHWORKS, I. *MATLAB: the language of technical computing. Desktop tools and development environment, version 7*. [S.l.]: MathWorks, 2005. v. 9. Citado na página 41.

MICROSOFT. 2015. Disponível em: <https://support.microsoft.com/en-us/kb/140365/>. Citado na página 39.

NETTER, F. H. *Netter Atlas de Anatomia Humana*. [S.l.]: Elsevier Brasil, 2008. Citado 2 vezes nas páginas 7 e 18.

PRASANTH, K.; PAUL, B.; BALAKRISHNAN, A. A. Fetal ecg extraction using adaptive filters. *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, v. 2, n. 4, p. 1483–1487, 2013. Citado na página 42.

R., W. L. Interfacing a msp430 with an sd card: Fat32 format. In: *FAT32*. [S.l.]: University of Florida, 2012. Citado na página 39.

RAMOS, Â. P.; SOUSA, B. S. Eletrocardiograma: princípios, conceitos e aplicações. *Centro de Estudos de Fisiologia do Exercício*, 2007. Citado 2 vezes nas páginas 7 e 36.

RODRIGUES, J. A. Implementação da comunicação sem fio de um módulo estimador da frequência cardíaca fetal baseado em fpga. Citado na página 15.

SABIA, V. 2017. Disponível em: <http://www.vocesabia.net/ciencia/cartao-de-memoria/>. Citado 3 vezes nas páginas 7, 38 e 39.

SANDISK, S. *Card Product Manual*. [S.l.]: Version, 2004. Citado 2 vezes nas páginas 38 e 39.

SUMAN, R. P. *UART: Universal Asynchronous Receiver Transmitter*. 2016. Disponível em: <http://students.iitk.ac.in/eclub/assets/lectures/summer12/uart.pdf>. Citado 2 vezes nas páginas 7 e 28.

TERASIC; PROGRAM, A. U. *DE1-SoC: User Manual*. [S.l.], 2015. Accessed: 2016-06-07. Citado 4 vezes nas páginas 7, 37, 55 e 56.

TERASIC, T. *D5M 5 Mega Pixel Digital Camera Development Kit vol.* [S.l.]: Version, 2016. Citado 2 vezes nas páginas 7 e 38.

Apêndices

APÊNDICE A – Implementação do MIPSfpga

A implementação do MIPSfpga foi feita de acordo com o tutorial disponibilizado pela (TERASIC; PROGRAM, 2015). Porém ele é um tutorial para ser executado nos kits de desenvolvimento Nexys4 DDR e DE2-115. Como o kit utilizado foi o DE1-SoC, algumas alterações nos códigos fornecidos precisaram ser alteradas, sendo elas:

- Criar um arquivo Top-Level para conectar o núcleo do MIPSfpga à placa FPGA. A imagem 28 mostra o arquivo criado, podendo ver que o MIPSfpga poderá utilizar 10 *switches*, 4 *pushbuttons*, 10 LEDs e 7 pinos de entrada e saída da placa, além de dois pinos instanciados RX e TX, que trabalham como os canais do protocolo de comunicação UART.

```
1  module mipsfpga_de1_soc(input         CLOCK_50,
2                          input  [ 9:0] SW,
3                          input  [ 3:0] KEY,
4                          output [ 9:0] LEDR,
5                          inout  [ 6:0] EXT_IO,
6                          output [ 6:0] HEX0,
7                          output [ 6:0] HEX1,
8                          output [ 6:0] HEX2,
9                          output [ 6:0] HEX3,
10                         output [ 6:0] HEX4,
11                         output [ 6:0] HEX5,
12                         output        TX,
13                         input         RX);
14
15    // Use KEY[0] (push button switch 0) for SI_Reset_N.
16    // Note: KEY[0] must be pushed down (held low) to reset the processor.
17
18    wire clk_out;
19
20    altpll0 altpll0 (.inclk0(CLOCK_50), .c0(clk_out));
21
22
23
24    mipsfpga_sys mipsfpga_sys(.SI_Reset_N(KEY[0]),
25                              .SI_ClkIn(clk_out),
26                              .HADDR(),
27                              .HRDATA(),
28                              .HWDATA(),
29                              .HWRITE(),
30                              .EJ_TRST_N_probe(EXT_IO[6]),
31                              .EJ_TDI(EXT_IO[5]),
32                              .EJ_TDO(EXT_IO[4]),
33                              .EJ_TMS(EXT_IO[3]),
34                              .EJ_TCK(EXT_IO[2]),
35                              .SI_ColdReset_N(EXT_IO[1]),
36                              .EJ_DINT(EXT_IO[0]),
37                              .IO_Switch(SW),
38                              .IO_PB({1'b0, ~KEY}),
39                              .IO_LEDR(LEDR),
40                              .IO_LEDG(),
41                              .IO_HEX0(HEX0),
42                              .IO_HEX1(HEX1),
43                              .IO_HEX2(HEX2),
44                              .IO_HEX3(HEX3),
45                              .IO_HEX4(HEX4),
46                              .IO_HEX5(HEX5),
47                              .IO_TX(TX),
48                              .IO_RX(RX));
49
50 endmodule
```

Figura 28 – Arquivo Top-Level do MIPSfpga.

- O arquivo $mipsfpga\_ahb\_const.vh$ presente nas linhas 24 e 25, mostram o tamanho da memória RAM de reset e da usada nos programas do MIPSfpga, precisam ser diminuídas. O parâmetro da memória de reset foi diminuído de 15 para 14, resultando em 2^{14} palavras, ou 64KB. O parâmetro da memória usada nos programas foi diminuído de 16 para 15, resultando em 2^{15} palavras, ou 128KB.

- Após o projeto ser criado com as determinadas alterações, é gerado um arquivo *.qsf, responsável por mapear os sinais do programa que roda na FPGA aos seus pinos. O arquivo foi editado e os pinos corretamente endereçados utilizando os endereços dos pinos disponíveis no manual do kit (TERASIC; PROGRAM, 2015).

APÊNDICE B – Execução de códigos em C no MIPSfpga

Os arquivos que ditam o programa que será executado no MIPSfpga são *ram_reset_init.txt* e *ram_program_init.txt*, que no caso do exemplo dos switches e LEDS foram fornecidos pela *Imagination Technologies*. Para a execução de arquivos escritos em C no processador, precisamos gerar estes dois arquivos. Para cumprir esta tarefa, é necessário um sistema operacional Windows com os programas OpenOCD e Codescape instalados.

A fim de se compilar o código em C usando o software Codescape é necessário que se nomeie o arquivo com o código como main.c e o coloque em uma pasta com os arquivos $boot.h, boot.S, boot-uhi32.ld, fpga.h, FPGA\_Ram.ld, init\_caches.S, init\_cp0.S, init\_gpr.S, init\_t$ que são fornecidos. Em seguida abrir um terminal (cmd.exe no menu Iniciar), navegar até a pasta com estes arquivos e dar um comando make. Diversos arquivos serão criados. Em seguida o script createMemfiles, também fornecido, deve ser executado e receber como parâmetro a pasta onde se encontra o arquivo main.c. Este script irá gerar uma pasta chamada MemoryFiles contendo os arquivos *ram_reset_init.txt* e *ram_program_init.txt* que farão com que o MIPSfpga execute o programa em C desejado.

Na implementação dos códigos em C para testes na placa, utilizou os programas fornecidos pela *Imagination Tecnologies*, para programar o primeiros testes no MIPSfpga. O primeiro foi o de assimilar cada *switch* presente na placa ao LED que estão acima deles. Quando o switch for acionado o Led deverá ser aceso. A Fig. 29 mostra o programa funcionando corretamente.

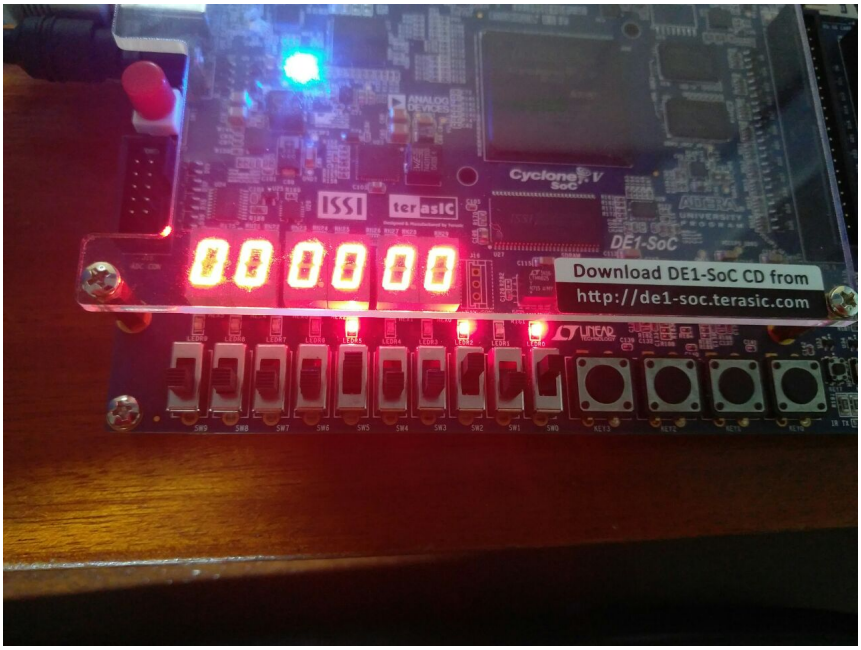

Figura 29 – Switches [0], [2] e [5] acionados e seus respectivos Leds acesos.

O segundo teste foi o contador binário, que basicamente tem os switches como os números binários sendo os pinos da esquerda os menos significativos e os da direita os mais significativos. As Fig.30, 31 e 32 mostram os resultados.

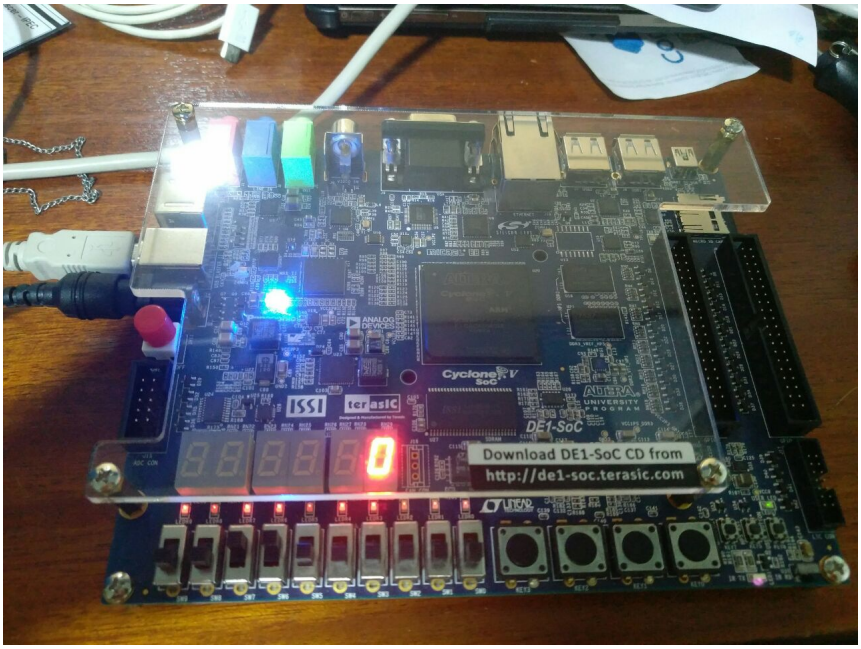

Figura 30 – Nenhum switch acionado, representando o numero 0 em decimal.

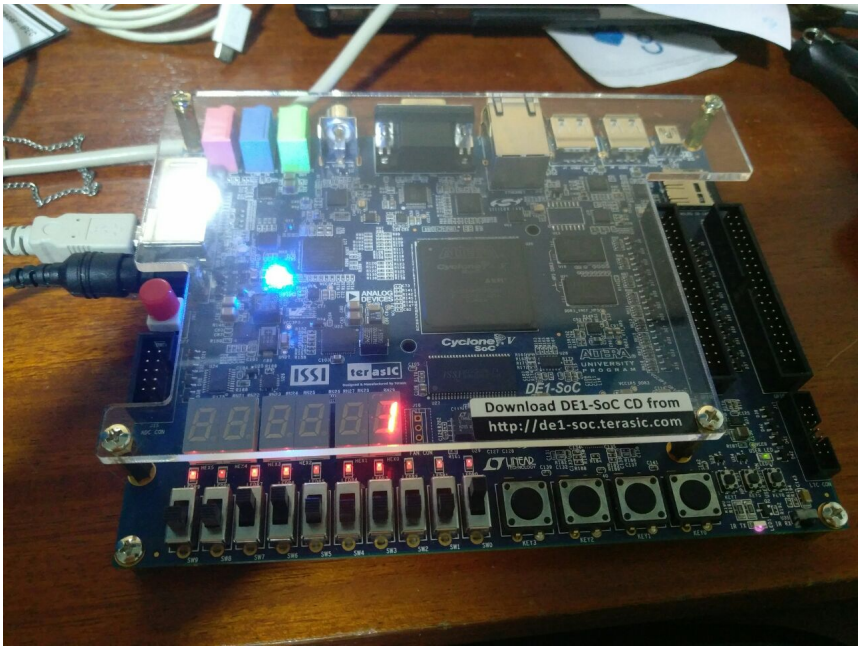

Figura 31 – Switch [0] acionado representando o numero decimal 1

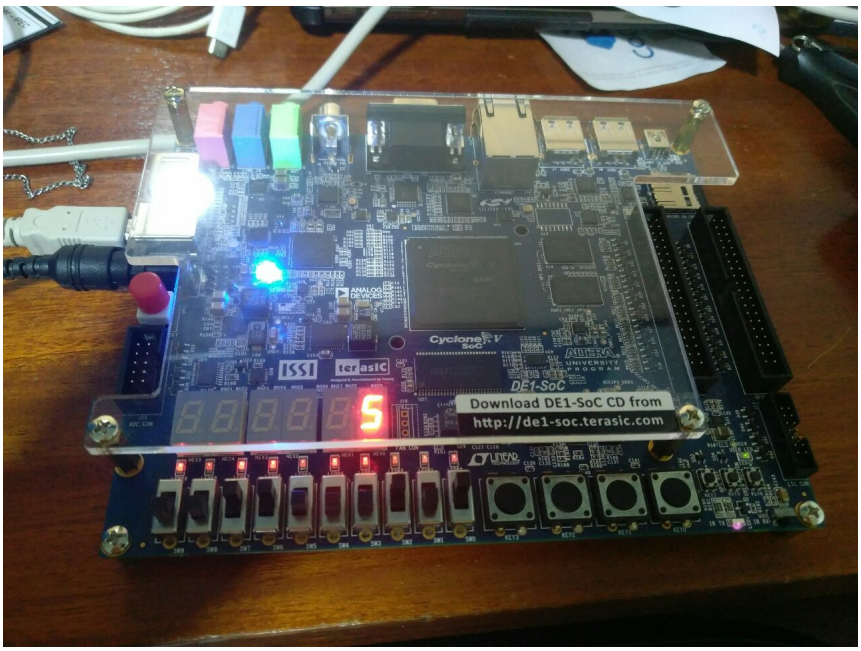

Figura 32 – Switches [0] e [2] acionados resultando no número 5.

APÊNDICE C – Algoritmo para conversão para ponto fixo no formato binário

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <string.h>
5
6 float * x; // ponteiro x para variável do tipo float
7 float * d; // ponteiro d para variável do tipo float
8
9 /* Função que recebe o nome de um arquivo via ponteiro de char
10 (correspondente de uma string em c) e retorna um ponteiro de
11 float (vetor de floats alocado dinamicamente).
12 *  É necessario chamar a função na main*/
13 float * values (char * string)
14 {
15         FILE * pFile; // ponteiro pFile para variável do tipo FILE
16         pFile = fopen (string,"r"); // abre arquivo para leitura,
17                                     // cujo nome está na váriavel string,
18                                     // e aponta o ponteiro pFile para o
19                                     // inicio desse arquivo
20         if (pFile==NULL) // verifica se o arquivo foi aberto corretamente
21                          // e pFile aponta para o começo desse arquivo
22                 perror ("Erro ao abrir arquivo");
23
24         float * valores; // ponteiro valores para variável do tipo float
25         int i = 0; // iterador
26         char virgula; // caracter que será usado para receber as vírgulas
27                       // dos arquivos, que contém floats, vírgulas e espaços
28
29         valores = (float*) malloc (sizeof(float)); //aloca e atribui para
30                                                    // valores um espaço de
31                                                    // memória sizeof(float) (4 bytes)
32
```

Figura 33 – Algoritmo para conversão - parte 1

```
33        /*
34        * laço cuja variavel de controle é chegar ao fim do arquivo. Fscanf
35        * tem a função de capturar dados do arquivo para o qual aponta (pFile).
36        * Neste caso, é capturado um float e uma virgula em cada iteração do laço.
37        */
38        while(fscanf(pFile, "%f%c", &valores[i], &virgula) != EOF)
39        {
40                i++; //incremento da variável de iteração i
41
42                // realocação do espaço de memória para o ponteiro valores.
43                // Note que é atribuido mais um espaço de memória do tamanho
44                // de um float a cada iteração do comando while, devido a
45                // variável de iteração i.
46                valores = (float*) realloc (valores, (i+1) * sizeof(float));
47        }
48
49
50        fclose (pFile); //fecha o arquivo apontado por pFile
51
52        return valores; // retorna o vetor de floats alocado dinamicamente
53 }
54
55
56
57 char * decToBin (float decimal)
58 {
59        char *binary;
60        char zero[] = "0", zeros[] = "00", one[] = "1", ones[] = "01", two[] = "10";
61        int j, parteInteira, neg = 0, inteiro, carry = 0;
62        int *complemento;
63        float parteFracionaria, k, temp;
64        float binFracionario;
```

Figura 34 – Algoritmo para conversão - parte 2

```
65
66        binary = (char *) calloc (33, sizeof(char));
67
68        binary[32] = '\0';
69
70        k = 0.1;
71        temp = 0;
72
73        parteInteira = (int)decimal;
74
75        if(decimal < 0)
76        {
77                neg = 1;
78                parteFracionaria = -(decimal + (int)decimal);
79        }
80        else
81                parteFracionaria = decimal - (int)decimal;
82
83        strcpy(binary, zero);
84
85        if(parteInteira == 0)
86                strcat(binary, zeros);
87        if(parteInteira == 1)
88                strcat(binary, ones);
89        if(parteInteira == 2)
90                strcat(binary, two);
91
92        while(k > 0.00000000000000000000000000000001)
93        {
94                binFracionario = parteFracionaria * 2;
95
```

Figura 35 – Algoritmo para conversão - parte 3

```
96              if((int)binFracionario == 1)
97                      strcat(binary, one);
98              if((int)binFracionario == 0)
99                      strcat(binary, zero);
100
101             temp = temp + ((int)binFracionario)*k;
102             parteFracionaria = binFracionario - (int)binFracionario;
103             k = k / 10;
104     }
105
106     //Condicao if para avaliacao se o numero era negativo,
107     // caso for fazer complemento de 2
108     if(neg == 1)
109     {
110             neg = 0;
111
112             complemento = (int*) calloc (33, sizeof(int));
113
114             for(j=0; j<32; j++)
115             {
116                     inteiro = (int) binary[j];
117
118                     if(inteiro == 49) // 49 == '1'
119                             inteiro = 0;
120                     else              // 48 == '0'
121                             inteiro = 1;
122
123                     complemento[j] = inteiro;
124             }
```

Figura 36 – Algoritmo para conversão - parte 4

```
125
126             for(j=31; j>=0; j--)
127             {
128                     if(j == 31)
129                     {
130                             if(complemento[j] == 1)
131                             {
132                                     complemento[j] = 0;
133                                     carry = 1;
134                             }
135                             else
136                             {
137                                     complemento[j] = 1;
138                                     carry = 0;
139                             }
140                     }
141                     else
142                     {
143                             if(carry == 1)
144                             {
145                                     if(complemento[j] == 0)
146                                     {
147                                             complemento[j] = 1;
148                                             carry = 0;
149                                     }
150                                     else
151                                     {
152                                             complemento[j] = 0;
153                                             carry = 1;
154                                     }
155                             }
156                     }
```

Figura 37 – Algoritmo para conversão - parte 5

```
157
158                 }
159
160                 for(j=0; j<32; j++)
161                 {
162                         if(complemento[j] == 0)
163                                 complemento[j] = 48;
164                         else
165                                 complemento[j] = 49;
166
167                         binary[j] = (char) complemento[j];
168                 }
169
170                 free(complemento);
171         }
172
173         binary[32] = '\0';
174
175         return binary;
176 }
177
178 int main () {
179
180     FILE *abdomen;
181     FILE *torax;
182     char toraxFILE[] = "torax_string.txt";
183     char abdomenFILE[] = "abdomen_string.txt";
184     char *binary;
185     float valor;
186     int i;
```

Figura 38 – Algoritmo para conversão - parte 6

```
187
188     d = (float*) malloc (sizeof(float)); //aloca e atribui para d,
189                                          // declarado como variável global
190                                          // no começo do programa, um espaço de
191                                          // memória  sizeof(float) (4 bytes)
192     x = (float*) malloc (sizeof(float)); //aloca e atribui para x, declarado como
193                                          // variável global no começo do programa,
194                                          // um espaço de memória  sizeof(float) (4 bytes)
195
196         // chamada da função que abre um arquivo com nome torax_string.txt e retorna
197         // um vetor de floats alocado dinamicamente, e atribuição desse vetor de
198         // floats ao ponteiro x
199     x = values(toraxFILE);
200
201     // chamada da função que abre um arquivo com nome abdomem_string.txt e
202     // retorna um vetor de floats alocado dinamicamente, e atribuição desse
203     // vetor de floats ao ponteiro d
204     d = values(abdomenFILE);
205
206     abdomen = fopen ("abdomen.txt", "w");
207     fclose (abdomen);
208     abdomen = fopen ("abdomen.txt", "a");
209
210         /*For loop para transformacao dos valores do abdomen em binario*/
211     for (i = 0; i<10000; i++)
212         {
213                 valor = d[i];
214                 binary = decToBin(valor);
215
216                 fprintf(abdomen, "%s\n", binary);
217         }
```

Figura 39 – Algoritmo para conversão - parte 7

```
217         }
218
219         fclose(abdomen);
220
221         torax = fopen ("torax.txt", "w");
222     fclose (torax);
223     torax = fopen ("torax.txt", "a");
224
225     /*For loop para transformacao dos valores do torax em binario*/
226     for (i=0; i<10000; i++)
227         {
228         valor = x[i];
229                 binary = decToBin(valor);
230
231                 fprintf(torax, "%s\n", binary);
232         }
233
234     fclose(torax);
235
236     free(x);
237     free(d);
238     free(binary);
239 }
240
```

Figura 40 – Algoritmo para conversão - parte 8

APÊNDICE D – Código do Arduino para leitura do cartão SD

MIPS_ARDUINO_HELTON

```
#include <SPI.h>
#include <SD.h>

const int chipSelect = 10;

int led = 13;
int val = 1;
volatile int cont = 1;
volatile int incomingBytes;

void setFile();

void setup() {

  pinMode(led, OUTPUT);
  digitalWrite(led, LOW);
  Serial.begin(9600);

}

void loop()
{
    if(Serial.available())
    {
      incomingBytes = Serial.read();
      //Serial.println(incomingBytes);
      if(incomingBytes == 1)
      {
        setFile();
      }
    }
}

void setFile()
{
  // see if the card is present and can be initialized:
  if (!SD.begin(chipSelect))
  {
    Serial.println("Card failed, or not present");
    // don't do anything more:
    return;
  }

  File dataFile = SD.open("teste.txt");

  }
  // if the file is available, write to it:
  if (dataFile)
  {
    while (dataFile.available())
    {
      Serial.write(dataFile.read());
    }
    dataFile.close();
  }
  // if the file isn't open, pop up an error:
  else
  {
    Serial.println("error opening datalog.txt");
  }

}
```

Figura 41 – Algoritmo para comunicação com leitor de cartão

APÊNDICE E – Módulo UART

A figura 42 mostra o algoritmo desenvolvido para o canal RX da Uart. É um código em verilog que possui duas entradas, uma delas é HCLK que representa o clock do sistema, a outra RX_LINE, que é a informação que será recebida. Para as saídas temos o data sendo o vetor que será preenchido com 8 bits com as informações recebidas pelo RX_LINE, o busy e controle são *flags* para indicar o estado, e $fhr\_samples$ é um registrador de 32 bits que será puxado pelo processador para realizar os cálculos para converter em decimal.

Foi definido uma taxa de transmissão de 9600 *bits* por segundo para operar neste módulo. Esta taxa é definida pelo *prescaler*, fazendo com que os bits sejam transmitidos de acordo com o valor desejado.

```
module RXVerilog(

        input HCLK,
        input RX_LINE,
        output [ 7:0] data,
        output busy,
        output reg [31:0] fhr_samples,
        output controle
);

    integer prscl = 0;
    reg rx_flag = 1'b0;
    integer index = 0;
    reg [9:0] s_data;
    reg [7:0] data_RX = 7'b0;
    reg s_busy = 1'b0;

    reg [31:0] dataBuffer = 32'b0;
    reg        s_read = 1'b0;
    integer    aux = 31;
    integer    aux2 = 0;

    reg s_controle = 1'b0;

    assign data = data_RX;
    assign busy = s_busy;
    assign controle = s_controle;

    always @(posedge HCLK) begin

        if(rx_flag == 1'b0 && RX_LINE == 1'b0) begin
            index = 0;
            prscl = 0;
            s_busy = 1'b1;
            rx_flag = 1'b1;
            s_controle = 1'b0;
        end

        if(rx_flag == 1'b1) begin

            s_data[index] = RX_LINE;

            if(prscl < 5207) begin
                prscl = prscl + 1;
            end
            else begin
                prscl = 0;
            end

            if(prscl == 2604) begin
                if(index < 9) begin
                    index = index + 1;
                end
                else begin
                    if(s_data[0] == 1'b0 && s_data[9] == 1'b1) begin
                        data_RX = s_data[8:1];
                    end
                    else begin
                        data_RX = 7'b0;
                    end

                    rx_flag = 1'b0;
                    s_busy = 1'b0;
                    s_read = 1'b1;

                end
            end
        end

        if(s_read == 1'b1) begin

            if(s_data[8:1] == 8'b00110001) begin
                dataBuffer[aux] = 1'b1;
            end
            else if(s_data[8:1] == 8'b00110000) begin
                dataBuffer[aux] = 1'b0;
            end

            if(aux > 0 && (s_data[8:1] == 8'b00110001 || s_data[8:
                aux = aux - 1;
            end
            else if(aux == 0) begin
                aux = 31;
```

Figura 42 – Uart-RX em Verilog

A figura 43 mostra o arquivo *mips_ahb_gpios* alterado para instanciar o módulo UART como periférico do MIPSfpga.

```
42          IO_LEDG <= 9'b0;  // Green LEDs
43          SEGMENTOS = 19'b0;
44          TXDATA = 8'b0;
45        end else if (HWRITE & HSEL)
46          case (HADDR)
47            `H_LEDR_IONUM: IO_LEDR <= HWDATA[17:0];
48            `H_7SEG_IONUM: SEGMENTOS <= HWDATA[18:0];
49            `H_TX_IONUM: TXDATA <= HWDATA[7:0];
50          endcase
51
52    mipsfpga_ahb_sevensegtimer mipsfpga_ahb_sevensegtimer(.SEGMENTOS(SEGMENTOS), .IO_HEX0(IO_HEX0), .IO_HEX1(IO_HEX1), .IO_HEX2(IO_HEX2), .IO_HEX3(IO_HEX3), .IO_HE
53    //mipsfpga_ahb_uart mipsfpga_ahb_uart(.HCLK(HCLK), .DATA_TX(TXDATA), .RX_LINE(IO_RX), .DATA_RX(RXDATA), .FHR_SAMPLES(MEMDATA), .CONTROLE(RXCONTROLE));
54
55    TXVerilog TXVerilog (.UART_TX(TXDATA), .HCLK(HCLK), .IO_TX(IO_TX));
56    RXVerilog RXVerilog (.HCLK(HCLK), .RX_LINE(IO_RX), .data(RXDATA), .busy(CONTROLE), .fhr_samples(MEMDATA), .controle(RXCONTROL));
57
58     always @(*)
59       case (HADDR)
60         `H_SW_IONUM: HRDATA = {14'b0, IO_Switch};
61         `H_PB_IONUM: HRDATA = {27'b0, IO_PB};
62         `H_RX_IONUM: HRDATA = {25'b0, RXDATA[6:0]};
63         `H_RXCONTROL_IONUM: HRDATA = {31'b0, RXCONTROL};
64         `H_CONTROLE_IONUM: HRDATA = {31'b0, CONTROLE};
65         `H_MEMDATA_IONUM: HRDATA = MEMDATA;
66         default:     HRDATA = 32'h00000000;
67       endcase
68
69  endmodule
70
71
```

Figura 43 – Arquivo *mips_ahb_gpios*

APÊNDICE F – Código C principal alterado

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "rt_nonfinite.h"
#include "setup.h"
#include "SystemCore.h"
#include "LMSFilter.h"

#define inline_assembly()  asm("ori $0, $0, 0x1234")

int Fs = 1000, Time = 10;                       //Frequência e tempo de amostragem
float NumSamp;                                  //Numero de amostras
volatile int i;
float d[10000];                                 //ECG abdominal materno
float x[10000];                                 //ECG toracico materno
float fhr;

volatile int *IO_LEDR = (int*)0xbf800000;
volatile int *IO_PUSHBUTTONS = (int*)0xbf80000c;
volatile int *IO_DISPLAY = (int *)0xbf800010;
volatile int *IO_TXDATA = (int *) 0xbf800014;
volatile int *IO_RXDATA = (int *) 0xbf800018;
volatile int *IO_RCVDCONTROL = (int *) 0xbf80001c;
volatile int *IO_CONTROLE = (int *) 0xbf800020;
volatile int *IO_MEMDATA = (int *) 0xbf800024;

volatile unsigned int pushbutton, control;


void delay();
void data_read();
void lms(const float d[10000], const float x[10000]);
void show();

int main () {

    NumSamp = Fs*Time;

    while(1)
    {
        pushbutton = *IO_PUSHBUTTONS;

        switch(pushbutton)
        {
            case 0x04:
                data_read();
                lms(d, x);
                show();
                break;

            default:
                break;
        }
    }
}
```

Figura 44 – Código C alterado - parte 1

```
56
57  void delay() {
58    volatile unsigned int j;
59
60    for (j = 0; j < (1000000); j++) ; // delay
61  }
62
63  void _mips_handle_exception(void* ctx, int reason) {
64    volatile int *IO_LEDR = (int*)0xbf800000;
65
66    *IO_LEDR = 0x8001;  // Display 0x8001 on LEDs to indicate error state
67    while (1) ;
68  }
69
70  void data_read()          //Função para receber os dados gravados em HW
71  {
72    int i = 0, j = 0;
73
74      while(j == 0)
75      {
76          control = *IO_CONTROLE;
77
78          if(((int)control == 1) && i<10000)
79          {
80              d[i] = *IO_MEMDATA;
81              d[i] = d[i]*pow(2, -29);
82              i++;
83          }
84
85          if(((int)control == 1) && i>9999)
86          {
87              x[i] = *IO_MEMDATA;
88              x[i] = x[i]*pow(2, -29);
89              i++;
90          }
91
92          if(i>19999)
93              j++;
94      }
95
96   return;
97  }
```

Figura 45 – Código C alterado - parte 2

```
98
99  void lms(const float d[10000], const float x[10000])     //Função para cálculo das saídas do filtro
100 {
101   dspcodegen_LMSFilter h;
102   float c[10000];
103   float unusedU0[10000];
104   dsp_LMSFilter_0 *obj;
105   float media = 0, limiar;
106   int ultimo = 0, borda = 0;
107   volatile float teste = 180.4;
108
109   /* % FILTRAGEM E RESULTADOS */
110   /* filtro adaptativo */
111   LMSFilter_LMSFilter(&h);
112   SystemCore_step(&h, x, d, unusedU0, c);
113
114   obj = &h.cSFunObject;
115
116   /* System object Destructor function: dsp.LMSFilter */
117   if (obj->S0_isInitialized) {
118     obj->S0_isInitialized = false;
119     if (!obj->S1_isReleased) {
120       obj->S1_isReleased = true;
121     }
122   }
```

Figura 46 – Código C alterado - parte 3

```
123
124
125   for (i = 0; i<10000; i++) {
126     media = media + fabs(c[i]);
127   }
128   media = media / (Fs*Time);
129   limiar = 2.5*media;
130
131   for (i=NumSamp/4; i<10000; i++) {
132     if (c[i] > limiar)
133     {
134       borda = borda + 1;
135       ultimo = i;
136       i = i + Fs/4;
137     }
138   }
139
140   fhr = borda *60*Fs/(ultimo-NumSamp/4);
141 }
142
143 void show()
144 {
145     *IO_LEDR = (int) fhr;   // write to red LEDs
146     *IO_DISPLAY = (int) fhr;
147 }
```

Figura 47 – Código C alterado - parte 4