Leonardo Haas Peçanha Lessa

Desenvolvimento da unidade de controle do módulo de aquisição de dados para o upgrade do experimento LHCb

Rio de Janeiro – RJ

Agosto / 2013

Leonardo Haas Peçanha Lessa

Desenvolvimento da unidade de controle do módulo de aquisição de dados para o upgrade do experimento LHCb

Dissertação apresentada à Coordenação de Formação Cientifica do Centro Brasileiro de Pesquisas Físicas para a obtenção do titulo de Mestre em Física com ênfase em Instrumentação Cientifica.

Orientador:
André Massafferri Rodrigues

Ministério da Ciência, Tecnologia e Inovação
Centro Brasileiro de Pesquisas Físicas
Mestrado Profissional Em Física

Rio de Janeiro – RJ

Agosto / 2013

Agradecimentos

Em primeiro lugar, agradeço aos meus pais e minha irmã, por todo o carinho, suporte e educação, dado em toda minha vida.

À minha esposa Cristina, que não me deixou desistir em momento algum do objetivo através de suas palavras de incentivo. Mesmo com minha ausência constante após o primeiro mês do nascimento do nosso filho Eric.

Ao meu orientador André Massafferri por todo suporte oferecido para a realização deste trabalho, pela grande compreensão e todo esforço feito para me manter no CERN. Além é claro do incentivo final para o término trabalho.

Ao pesquisador Ignácio Bediaga pelo apoio e colaboração no financiamento do projeto.

Ao pesquisador Alexandre Mello, pela compreensão relativa ao atraso para a apresentação desta dissertação.

Ao Niko Neufeld e Beat Jost do grupo *Online* do LHCb por me inserirem no grupo do CERN e pelo material fornecido para o desenvolvimento deste trabalho.

Ao colega de profissão e amigo, Diogo Di Calafiori, por disponibilizar o laboratório para os testes dos protótipos, sugestões e ideias e principalmente pela enorme paciência no trabalho de conserto das placas, incluindo a solda em componentes que dificilmente se enxergava.

Aos amigos brasileiros do CERN, pelos almoços, conversas e momentos de diversão vividos.

E finalmente aos familiares e amigos presentes em minha vida e que, de maneira direta ou indireta, me apoiaram e incentivaram para que seguisse em frente.

Resumo

O presente documento descreve o desenvolvimento do módulo de sistema de controle e suas camadas de software para o upgrade do sistema de aquisição de dados previsto no experimento LHCb. Este módulo é capaz de realizar o controle de uma placa de aquisição assim como outros dispositivos situados em áreas livres de radiação, de forma centralizada com a intermediação de um computador embarcado no módulo e o uso da comunicação Gigabit Ethernet através de barramentos de baixa velocidade como, I^2C e JTAG assim como a comunicação em alta velocidade utilizando o protocolo PCI Express. Também é descrito o desenvolvimento de uma placa de validação do módulo para uso na verificação do protótipo, particularmente importante para a produção em série. Serão apresentados os resultados obtidos através do uso de um kit de desenvolvimento, por consequência da impossibilidade da utilização do primeiro protótipo do módulo, com destaque a simplicidade de operação de dispositivos I^2C através do sistema operacional e desempenho na comunicação em alta velocidade.

Abstract

This document describes the development of the control system module and related software layers focused to the DAQ system upgrade of the LHCb experiment. This module is able to control a DAQ board as well as other generic devices located in radiation free areas. It can be managed centrally through an embedded computer module, using the Gigabit Ethernet communication to control, monitoring and configuring low-speed buses such as, I^2C and JTAG as well as high speed communication PCI Express protocol. It also describes the development of a validation board to be used to test the prototype and particularly important for mass production period. It is presented the results extracted using a development kit, since first prototype of the module was not properly working. We highlight the simplicity of operation I^2C devices through the operating system and performance in high-speed communication.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

ALICE	A Large Ion Collider Experiment
ATLAS	A Toroidal LHC ApparatuS
BE	Back-End
CAN	Controller Area Network
CCPC	Credit-card sized PC
CERN	Centre Européen pour la Recherche Nucléaire
CPPM	Centre de physique des particules de Marseille
DAQ	Data Acquisition System
DLL	Data Layer Link
DLLP	Data Layer Link Packet
DMA	Direct Memory Access
ECAL	Electromagnetic Calorimeter
ECS	Experiment Control System
EPFL	École Polytechnique Fédérale de Lausanne
FE	Front-End
FPGA	Field-programmable gate array
GBT	Gigabit Bidirectional Trigger and Data link
GPIO	General Purpose Input/Output
HCAL	Hadron Calorimeter
HPD	Hybrid Photon Detectors
HLT	High Level Trigger
I2C	Inter-Integrated Circuit
IT	Inner Tracker
JTAG	Joint Test Action Group
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty
LHCf	Large Hadron Collider forward
LLT	Large Hadron Collider beauty
MEP	Multiple Event Packets
OT	Outer tracker
PCIe	Peripheral Component Interconnect Express
PL	Physical Layer
PS	PreShower detector
PSB	Proton Syncrotron Booster
RICH	Ring Imaging Cherenkov Detector
SCADA	Supervisory Control And Data Acquisition
SDP	Scintilator Pad Detector
SPECS	Serial Protocol for ECS
SPI	Serial Peripheral Interface
SPS	Super Proton Synchrotron
TFC	Trigger and Fast Control
TL	Transaction Layer
TLP	Transaction Layer Packet

TOTEM	TOTal Elastic and diffractive cross section Measurement
TT	Trigger Tracker
TTC	Trigger, Timing and Control
VELO	Vertex Locator

1. Introdução

O experimento LHCb, um dos quatro grandes experimentos situados no acelerador LHC, foi projetado para realizar medidas de precisão envolvendo processos contendo os quarks b (*botton*) e c (*charm*), de modo a realizar estudos da violação da simetria CP e procura de Física além do modelo padrão das partículas elementares. Nos últimos anos o experimento vem publicando uma sequência de resultados de excelente qualidade, baseados nos dados acumulados durante os anos 2011 e 2012, o qual totalizou uma luminosidade integrada de 1 e 2 fb-1, respectivamente.

Recentemente a Colaboração LHCb definiu sua estratégia para o upgrade do experimento com o intuito de otimizar a utilização da alta luminosidade que será oferecida pelo acelerador LHC nos próximos anos. O novo regime permitirá o LHCb acumular uma luminosidade integrada de 50 fb-1 num período de 10 anos. A estatística de mésons contendo os *quarks* b e c fornecida proporcionará uma sensibilidade sem precedente à procura de nova Física.

A estratégia definida para o upgrade se divide em duas partes, ambas relacionadas à extinção do *trigger* de nível 0, destinado a reduzir a taxa de eventos a 1 MHz, e a subsequente operação de todos os seus sub-detectores a taxa máxima, correspondente a frequência de colisões do LHC, de 40 MHz. A primeira parte consiste na substituição de alguns detectores de modo a permitir sua utilização em altos regimes de radiação e grandes níveis de ocupação de traços, decorrentes do aumento na luminosidade. A segunda parte se refere unicamente à eletrônica e consiste na adaptação e redesenho da eletrônica de leitura dos detectores (*front-end electronics*) que não operam a 40 MHz, bem como da substituição completa do sistema de aquisição de dados, atualmente baseado em uma placa de aquisição denominada TELL1, por um capaz de processar a formatação de eventos de forma significativamente mais rápida e transmitir dados a uma taxa de 10 Gbs/link. No upgrade será necessária a produção de aproximadamente 300 placas de aquisição com as novas especificações. Estas placas, alocadas em região protegida da radiação (*counting room*), são monitorados e configurados a distância, a partir da sala de controle, parte integrada ao sistema central denominado ECS (*Experiment Control System*). Devido à complexidade de

alguns dispositivos que operam no *counting room* esse controle é desempenhado por unidades locais sofisticadas com a função de estabelecer comunicação em alta velocidade com a sala de controle através do protocolo Ethernet, realizar processamentos locais através de uma pequena CPU (*Credit-Card sized* PC ou CCPC), e ao mesmo tempo permitir a comunicação através de diferentes protocolos como I2C, JTag e SPI. Esses protocolos são utilizados para o controle de parâmetros importantes como temperatura, fluxo de dados, além de permitir o carregamento de firmware de dispositivos FPGAs (*Field-programmable gate array*), comumente encontrados nas placas de aquisição atuais.

No ano de 2009 duas implementações paralelas para o upgrade da placa de aquisição TELL1 estavam sendo estudadas. Uma delas em desenvolvimento pelo mesmo instituto que produziu a TELL1, a EPFL (*École polytechnique fédérale de Lausanne*), em Lausanne, com o projeto denominado TELL10. A segunda em desenvolvimento pelo CPPM (*Centre de physique des particules de Marseille*), em Marseille, cujo projeto foi denominado TELL40. Neste período o grupo de eletrônica do CBPF foi convidado a desenvolver a unidade de controle da nova placa de aquisição, para o upgrade do sistema empregado na TELL1, desenvolvido pelo grupo de eletrônica de Genova. O sistema de controle da TELL1 tinha como característica técnica a utilização de um CCPC de 130 MHz, comunicação ethernet de 10/100 Mbs, e o fornecimento de um barramento paralelo, quatro linhas I2C e três interfaces JTAG (*Joint Test Action Group*). Os pontos críticos no contexto do upgrade do sistema de aquisição consistia na ausência de um protocolo de alta velocidade, como PCIe (*Peripheral Component Interconnect Express*), no pequeno número de linhas de baixa velocidade, na inexistência do protocolo SPI (*Serial Peripheral Interface*) e na baixa velocidade da comunicação com a sala de controle, realizada via *Ethernet* à 100 Mb/s, além da dificuldade da implementação dos drivers de controle e aplicativos para os barramentos de baixa velocidade.

O desenho proposto pelo grupo do CBPF consistiu no desenvolvimento de um *mezzanino*, denominado ECS-CBPF, no qual seria conectado um modelo CCPC do fabricante Kontron com 1GHz de velocidade, e que disponibilizasse Ethernet de 1 Gbs, 6 linhas PCIe e um número configurável de linhas I2C, SPI e JTAG, com controladores implementados dentro de uma FPGA. Expansões seriam possíveis através de portas genéricas (GPIO - *General Purpose Input/Output*), desde que respeitado o limite suportado pela FPGA. Essa dissertação

trata do desenvolvimento de todo o pacote envolvido nesta unidade de controle; *hardware*, *software* e *firmware*. Neste contexto foi produzida também uma placa de validação do módulo ECS-CBPF, importante para o período de produção em série.

Nas várias reuniões que se seguiram, esse desenho se mostrou adequado para as necessidades do upgrade do sistema de controle da TELL1, e genérico suficiente para ser utilizado em outros dispositivos que operem no *counting room*. A geometria proposta inicialmente acompanhava o projeto da TELL10, o qual possuía uma grande modularidade na forma de *mezzaninos*, e, portanto já disponibilizava dimensões específicas.

No ano de 2011 a Colaboração optou pelo projeto da TELL40, uma vez que o grupo de CPPM se encontrava num estágio mais avançado na caraterização dos sinais de alta velocidade. O desenho da placa de aquisição TELL40 previa o desenvolvimento de uma placa mãe a qual serviria de base para quatro *mezzaninos* denominados AMC40, cada um contendo uma FPGA de altíssimo desempenho. Estas FPGAs desempenham todas as tarefas de formatação de fragmentos dos dados desempenhadas pela TELL40. Segundo o projeto apresentado pela CPPM a placa de aquisição TELL40 seria alocado num *crate* ATCA, amplamente utilizado na indústria de telecomunicações, considerado como uma concepção mais moderna dos tradicionais *crates* utilizados em altas energias. Este *crate* impõe restrições geométricas de modo que foi definido que a implementação do sistema de controle seria na própria placa TELL40, inviabilizando a inserção do *mezzanino* do CBPF. Porém basicamente todas as funcionalidades de software, principalmente os drivers, e eventualmente algumas firmwares podem ser utilizadas. O projeto do *mezzanino* ECS-CBPF foi discutido em várias reuniões da Eletrônica e *Online* do LHCb, onde o andamento do projeto foi apresentado tipicamente de quatro em quatro meses. Uma intensa interação com o grupo de CPPM ainda se verifica no presente momento, principalmente em relação ao suporte de software de baixo nível.

A produção do primeiro protótipo, que inclui o módulo ECS-CBPF e placa de validação, foi dividida em três etapas, produção e fabricação do layout do circuito impresso, e montagem. No momento da verificação foram constatados erros inviabilizaram a obtenção dos resultados. Por este motivo a mesma foi feita através de um kit de desenvolvimento contendo um FPGA similar ao módulo. A partir dos erros detectados na fase de verificação

do primeiro protótipo, um segundo protótipo foi produzido recentemente. Sua verificação esta prevista para o mês de agosto de 2013.

Esta dissertação está dividida em oito capítulos. No capítulo 2 e 3 serão apresentados, respectivamente, o Acelerador LHC com todo o aparato experimental e uma breve descrição do detector LHCb (*Large Hadron Collider beauty*). O capítulo 4 detalha o upgrade previsto para a parte de eletrônica, de *front-end* e aquisição do detector LHCb. O desenvolvimento do módulo ECS-CPBF e placa de validação são tratados respectivamente nos capítulos 5 e 6. Resultados obtidos com a utilização dos elementos criados neste trabalho são encontrados no Capítulo 7 e a conclusão e perspectivas futuras no Capítulo 8.

2. Acelerador LHC

O LHC [1][2](*Large Hadron Collider*) construído no complexo de pesquisas do CERN (*Centre Européen pour la Recherche Nucléaire*) está situado na fronteira entre a Suíça e a França. O acelerador encontra-se em um túnel localizado a uma média de 100 m abaixo da superfície, devido a considerações geográficas da região, a profundidade pode variar entre 50 m e 175 m.

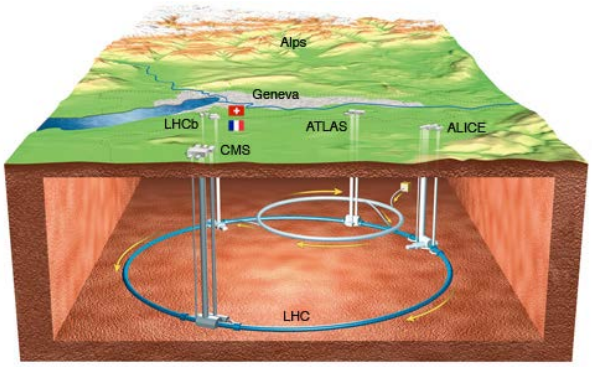


Figura 1 - Túnel do LHC

Dentro do acelerador, dois feixes de prótons circulam a uma velocidade próxima a da luz e com uma energia muito alta, antes de colidirem em um dos quatro pontos específicos. Estes feixes trafegam, em direções oposta em anéis separados mantidos em vácuo na ordem de 10^{-10} a 10^{-11} mbar[1] para evitar qualquer interferência de partículas encontradas no ar. Os feixes são guiados ao redor do acelerador por um forte campo magnético através de magnetos supercondutores resfriados a uma temperatura de -271^{o}C. São necessários 1232 dipolos magnéticos que ajudam a manter a direção dos feixes e 392 quadrupolos magnéticos para manter o foco.

Na configuração nominal do LHC, as colisões ocorrem a uma energia de 14 TeV no centro de massa e a uma luminosidade de 10^{34} cm^{-2} s^{-1} em uma frequência de 40MHz.

Para atingir a energia nominal os prótons são acelerados por uma série de pré-aceleradores, como mostrado na Figura 2. Inicialmente os feixes são produzidos a partir da ionização do gás de hidrogênio e passam por um acelerador linear (LINAC2) onde são acelerados até 50 MeV. Em seguida são injetados no PSB (*Proton Syncrotron Booster*) até

atingirem a energia de 1.4 GeV. Na etapa seguinte são acelerados pelo PS (*Proton Synchrontron*) até 25 GeV para no SPS (*Super Proton Synchrotron*) atingirem a energia de 450 GeV. Por ultimo, são injetados no LHC em anéis separados que circulam em direções opostas até a energia nominal.

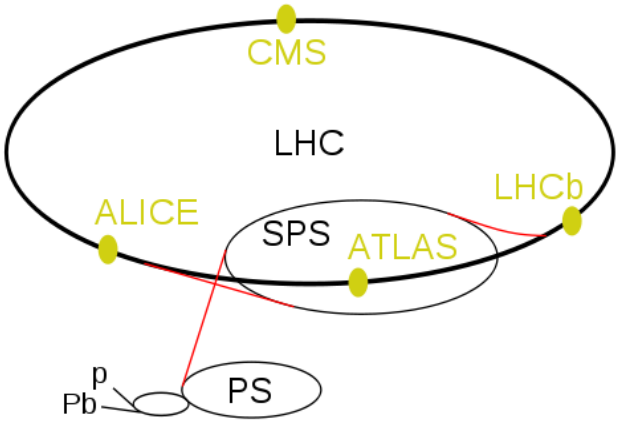


Figura 2 - Pré-Aceleradores

Nos quatro pontos de cruzamento dos feixes existem instalados um total de seis experimentos: ALICE[3] (*A Large Ion Collider Experiment*), ATLAS[4] (*Toroidal LHC ApparatuS*), CMS[5] (*Compact Muon Solenoid*), TOTEM[6] (*TOTal Elastic and diffractive cross section Measurement*), LHCf[7] (*Large Hadron Collider forward*) e o LHCb[8] (*Large Hadron Collider Beauty Experiment*).

O experimento ALICE tem como objetivo estudar através da colisão de ions pesados, a formação do estado da matéria conhecido como plasma de quarks e glúons. Acredita-se que tal plasma tenha existido no Universo, instante após o Big Bang.

O ATLAS e CMS são experimentos de propósito geral, que buscam bóson de Higgs e nova física na escala de TeV.

O experimento TOTEM é dedicado a medidas precisas da seção de choque da interação próton-próton, como também o estudo da estrutura do próton.

O LHCf é um experimento dedicado a medida de partículas neutras emitidas bem próximo a região de colisão. O principal objetivo físico é obter dados para calibração dos modelos de interação de hadrons que são usados no estudo de raios cósmicos de altíssimas energias.

O LHCb tem como principal objetivo estudar o fenômeno de violação de simetria Carga e Paridade (CP) nos decaimentos de sabores pesados, em busca de novos processos físicos, não descritos pelo Modelo Padrão.

3. LHCb (Large Hadron Collider Beauty)

O detector LHCb foi projetado para explorar um grande número de *hadrons*-b produzido no LHC a fim de fazer estudos precisos de assimetrias CP e decaimentos raros no sistema de mésons B. Sua geometria é de um experimento de alvo fixo com uma cobertura angular de 10 mrad a 300 mrad no eixo de projeção horizontal e até 250mrad no eixo de projeção vertical. O layout do detector mostrado na Figura 3 os principais sub-detectores são respectivamente: o detector de vértice (VELO), dois detectores Cherenkov para identificação de partículas (RICH1 e RICH2), um magneto, detectores de trajetórias (TT, T1-T3) e um sistema de calorímetros (SDP, PS, ECAL e HCAL), seguido de um sistema de múons (M1-M5).

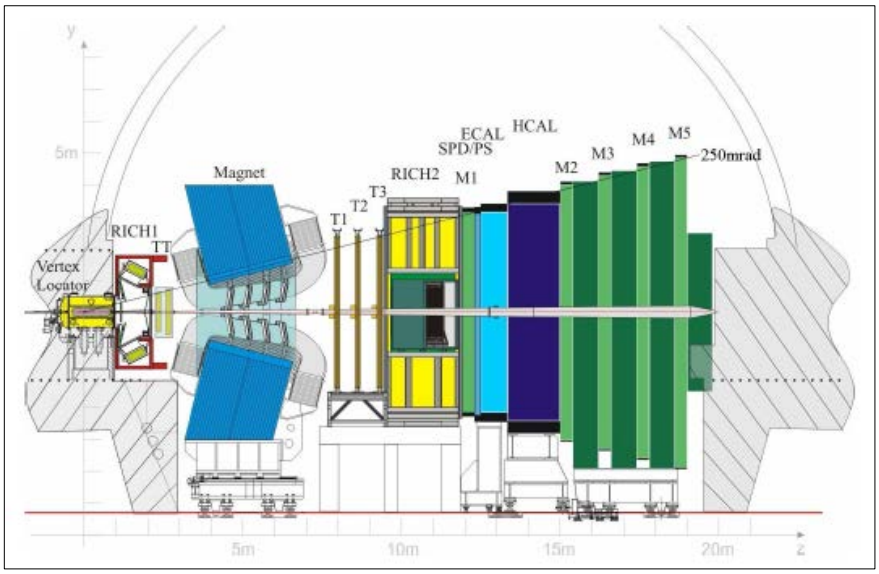


Figura 3 - Detector LHCb

3.1 Magneto

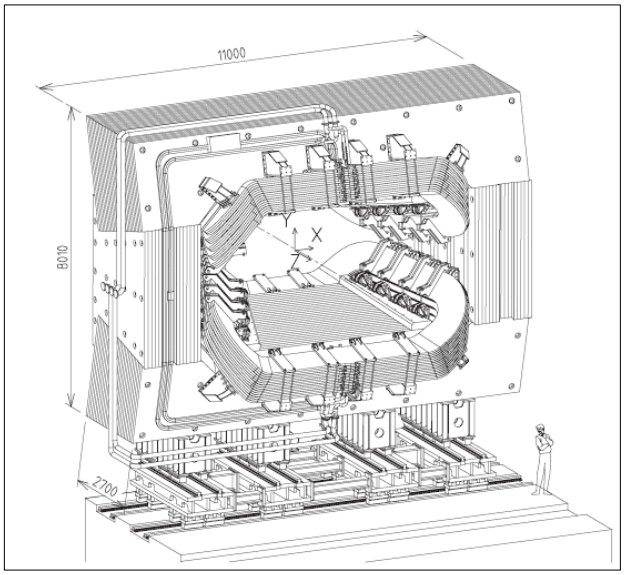


Figura 4 – Magneto

Um dipolo magnético, chamado de magneto[9], é usado no experimento LHCb para medida do momento de partículas carregadas, capaz de produzir um campo magnético integrado de 4 Tm. A polaridade do campo magnético pode ser invertida. Esta inversão é usada para o estudo e a redução de erros sistemáticos nas medidas de assimetria. A representação do magneto e o sistema de coordenadas utilizadas pode ser viso na Figura 4.

3.2 Sistema de traços

O sistema de traços tem como principal objetivo realizar reconstrução de trajetórias das partículas carregadas, que associada à medida de campo magnético possibilite extrair o momento. O sistema de traços do LHCb é composto por quatro detectores são eles: VELO, TT, IT e OT.

3.2.1 VELO

O VErtex LOcator [10] (VELO) é responsável pela medida precisa das coordenadas de traço próximas a região de interação, utilizada para medir a distância entre o vértice de produção dos mesons b e c dos vértices de decaimento. O VELO consiste em uma série de módulos de silício e onde cada módulo promove a medida das coordenadas r e ϕ, arranjados ao longo da direção do feixe, Figura 5.

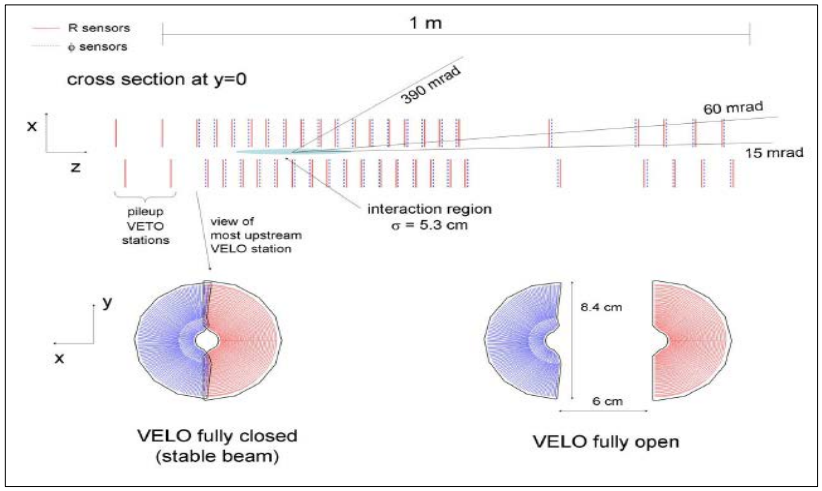


Figura 5 - Sub-detector VELO

Os sensores do VELO são localizados a uma distância radial do feixe cuja abertura é menor que a requerida pelo LHC durante a injeção e por esse motivo o sistema é retrátil. Os detectores são montados em um recipiente de vácuo separado dos sensores por uma folha fina e ondulada de alumínio. Isto é feito para minimizar a quantidade de material atravessado por uma partícula antes que esta atravesse o sensor.

3.2.2 Silicon Tracker

Dois detectores formam o Silicon Tracker: O *Trigger Tracker* (TT) e o *Inner Tracker* (IT), ambos utilizam sensores de microtiras, com aproximadamente 200 µm. O detector TT[11] está localizado entre o VELO e o magneto e é composto por duas estações com largura de 150 cm e altura de 130 cm. Cada estação é composta por quatro planos de sensores de silício com espessura de 500 µm. Um sensor é formado por 512 microtiras de silício espaçadas de 183 µm. Isso possibilita uma resolução espacial de aproximadamente 50 µm. As microtiras seguem orientações diferentes em cada plano. No primeiro plano, a orientação é vertical, no segundo plano as microtiras são rotacionadas de -5° na direção do

feixe, no terceiro de 5°, no último as microtiras são dispostas na vertical novamente, por esta geometria esses planos são chamados do tipo x-u-v-x.

O detector IT[12] encontra-se localizado depois do magneto, na região próxima ao feixe, onde se tem maior fluxo de partículas. Os sensores de silício seguem o mesmo *layout* de planos do TT, porém possuem 384 tiras espaçadas de 198 µm.

3.2.3 Outer Tracker

Na região mais externa, devido ao menor fluxo de partículas, no OT[13] emprega outra tecnologia de detecção.

O OT é composto por quatro planos de câmaras de arrasto do tipo *Straw tubes.* Cada estação é formada por quatro camadas duplas de tubos. Os *straw tubes* possuem 5 mm de diâmetro e são preenchidos com gás Ar/CO2/O2 = 70%/28.5%/1.5%. A resolução espacial é obtida a partir da medida de tempo entre o sinal do gás e a colisão do LHC.

3.3 RICH

Um dos requerimentos fundamentais do LHCb é a capacidade de separar píons de kaons em um decaimento B. Esta separação é feita através do detector RICH[14] (*Ring Imaging Cherenkov Detector*) numa faixa de momento de 1 a 100 GeV/c. Para esta finalidade existem três tipos de radiadores e dois detectores. O RICH1 localiza-se depois do detector VELO e o RICH2, antes da estação M1. O *layout* destes detectores pode ser observado na Figura 6.

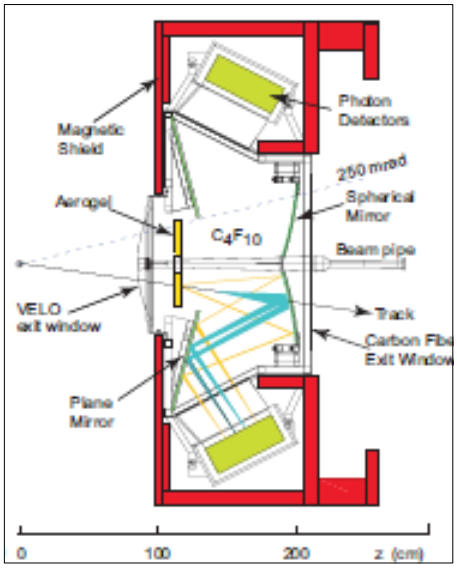


Figura 6 - Sub-detector RICH

No RICH1 são utilizados dois radiadores aerogel e C_4F_{10}, destinados a partículas de baixo momento. O elemento radiador do RICH2 é o CF_4 destinado a partículas de momento mais alto, variando de 15 a 100 GeV/c.

A luz de Cherenkov produzida nos radiadores é focalizada através de espelhos esféricos no RICH1, e por meio de espelhos planos no RICH2, para a detecção nos detectores de fótons híbridos (HPD – Hybrid Photon Detectors).

3.4 Calorímetros

Os calorímetros[15] realizam muitas funções importantes no LHCb. A primeira consiste na seleção de partículas candidatas para o primeiro nível de trigger (L0), que toma a decisão 4 μs após a interação. São responsáveis também pela discriminação entre fótons, elétrons e hadrons assim como a medida de sua energia e posição. O sistema é composto por quatro detectores: *Scintilator Pad Detector* (SPD), *PreShower detector* (PS), calorímetro eletromagnético (ECAL) e calorímetro hadrônico (HCAL).

A primeira camada de material ativo do calorímetro é o SPD. Sua principal função é detectar partículas carregadas antes da formação dos chuveiros de partículas, permitindo a separação entre fótons e elétrons. Segue-se uma parede 12 mm de chumbo usada para melhor separação de píons e elétrons através da formação e reconstrução de chuveiros eletromagnéticos.

O PS tem o mesmo princípio de funcionamento do SPD, e tem como objetivo definir a posição inicial dos chuveiros.

NO ECAL foi utilizada a tecnologia *Shashlik,* alternando placas cintiladoras de 4 mm e placas de chumbo de 2 mm de espessura.

O HCAL emprega a mesma tecnologia do ECAL porém é composto de placas cintiladoras de 4 mm e placas de ferro de 16 mm de espessura.

O SPD, PS e ECAL contêm 6.000 canais cada um, com três regiões distintas de granularidade ao redor do tubo do feixe, o HCAL contém 1.500 canais e é dividido em duas partes de granularidade distintas.

A variação de granularidade pode ser observada na Figura 7.

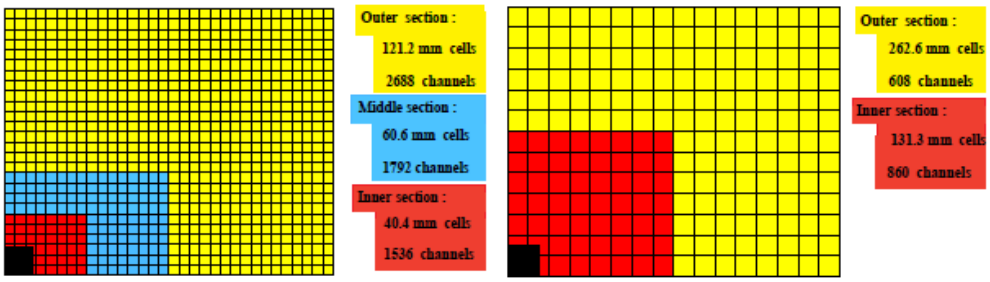


Figura 7 - Granularidade dos calorímetros

3.5 Sistema de Múons

O sistema de múons[16], mostrado na Figura 8, é composto por cinco estações (M1-M5) de forma retangular, situadas ao longo do eixo do tubo do feixe. O sistema é composto por 1380 câmaras, cobrindo uma área total de 435 m^2. Seu objetivo é fornecer informações para o *trigger* de múons e para a identificação de múons *off-line*.

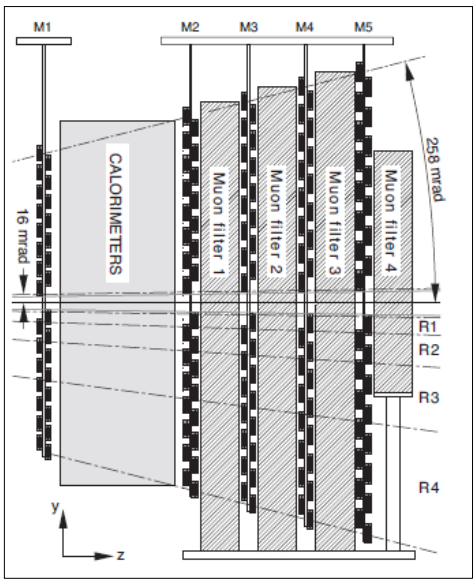


Figura 8 - Sistema de múons

A estação M1 localiza-se em frente aos calorímetros, próximo ao ponto de interação. É importante para a medida do momento transverso do traço identificado no *trigger* (L0). As outras quatro estacoes (M2-M5) são dispostas atrás dos calorímetros e são intercaladas absorvedores de ferro com 80 cm de espessura para selecionar os múons penetrantes. O momento dos múons para que atravessem as cinco estações é de aproximadamente 6 GeV/c.

3.6 Sistema de trigger

Durante a operação do LHC os dados obtidos pela colisão dos prótons são amostrados pelo detector a uma taxa de 40 MHz. Entretanto a velocidade de aquisição de dados é limitada pela eletrônica de cada detector e pela velocidade no armazenamento pelo cluster de computadores disponíveis. Para viabilizar a aquisição de dados um sistema de *trigger*[17] é utilizado para reduzir esta taxa para 2 KHz, selecionando eventos de interesse.

O *trigger* funciona em dois estágios o primeiro chamado de nível zero (L0) e o segundo *trigger* de alto nível (HTL – *High Level Trigger*).

3.7 Sistema Online

O propósito do sistema *online*[18][19] é garantir a transferência dos dados da eletrônica próxima aos detectores para os locais de armazenamento permanente. Isto inclui não somente o movimento de dados gerados pelos detectores, a configuração de todos os parâmetros de operação e o monitoramento dos mesmos. O sistema *online* também garante que todos os canais de aquisição do detector estejam propriamente sincronizados com o *clock* do LHC.

Basicamente o sistema *Online* consiste em três componentes, como mostrado na Figura 9, DAQ (*Data Acquisition System*), o TFC (*Trigger and Fast Control*) e o ECS (*Experiment Control System*).

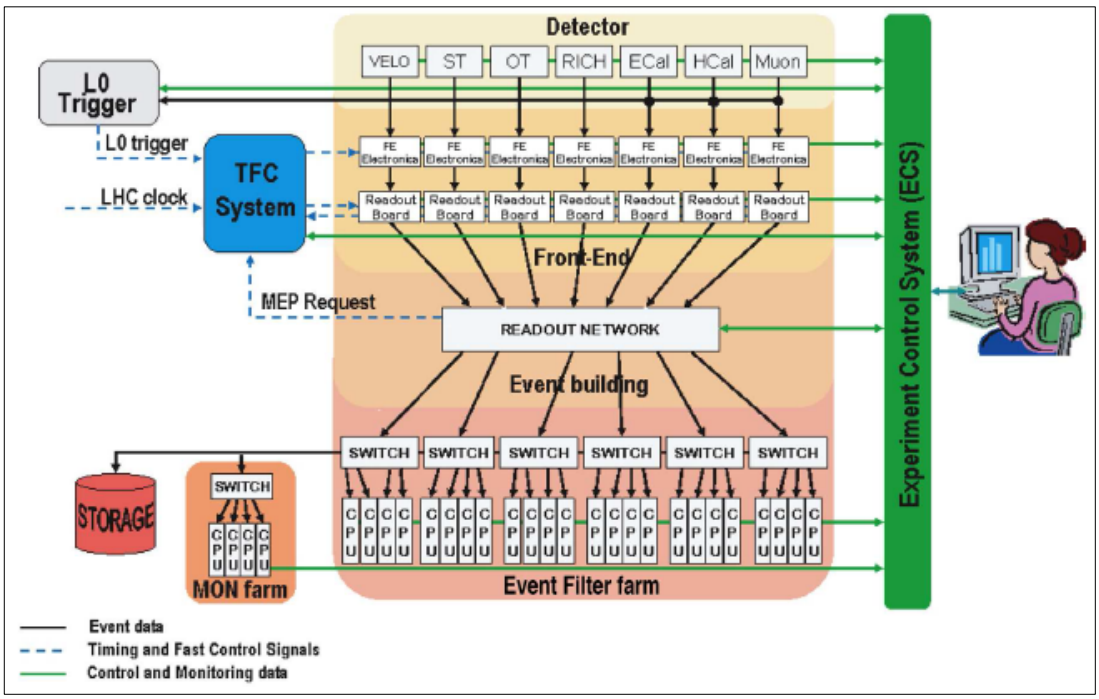


Figura 9 - Sistema *Online*

O sistema de aquisição de dados (DAQ) é responsável pelo transporte dos dados pertencentes a um dado *bunch-crossing* identificado pelo *trigger* correspondente ao detector para o local de armazenamento permanente. A arquitetura do DAQ segue como princípios:

- Simplicidade: Protocolos simples e um pequeno número de componentes com funcionalidades simples.

- Escalabilidade: Habilidade de reagir a modificações dos parâmetros do sistema, como o tamanho dos eventos, taxas de *trigger* ou o número de CPUs necessários para o algoritmo de *trigger.*
- *Links* ponto a ponto: Nenhum barramento é utilizado (fora das placas). Isto leva a um sistema mais robusto.
- Uso de produtos comerciais quando possível.

Estes princípios permitem a construção de um sistema robusto e confiável com a flexibilidade suficiente para adequar-se a possíveis novos requerimentos, motivados pela experiência com dados reais.

Os dados coletados pelos módulos da eletrônica próxima aos detectores do LHCb são coletados por placas de *readout* denominadas TELL1.

Os dados são recebidos pela placa TELL1 da eletrônica próxima aos detectores através de placas com links ópticos ou analógicos e processados em quatro FPGAs. Nas FPGAs um processamento de modo comum, *zero-suppression* ou compressão de dados, é realizado dependendo das necessidades individuais dos detectores. O resultados dos fragmentos dos dados são coletados por um quinto FPGA (*SyncLink*) e formatados em um pacote *raw* IP. Este enviado ao sistema de aquisição de dados através de uma placa de quatro canais *Gigabit-Ethernet*. A interface com o ECS é feita através de um CCPC montado como *mezzanino* na TELL1. Os sinais de sincronismo e *clock* são transmitidos através da interface de *Trigger, Timing and Control* (TTC). O fluxo de dados para o TFC é realizado através do sinal de *trottle* e é conduzido pelo FPGA *SyncLink*. O diagrama de blocos simplificado é apresentado na Figura 10.

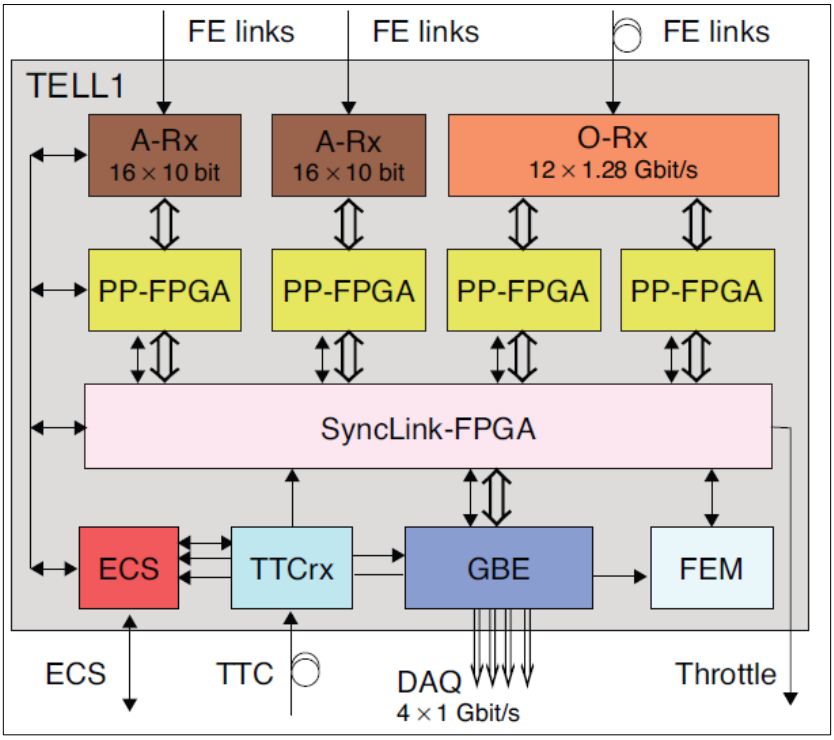


Figura 10 - Diagrama de blocos simplificado da TELL1

No *cluster* de *CPU*[20], o algoritmo HLT seleciona as interações de interesse em consequência de uma decisão positiva. Os dados são enviados então para o local de armazenamento permanente. Com o uso do HLT é esperada a redução da taxa global de trigger de 1 MHz para aproximadamente 2 KHz. O local de armazenamento permanente tem a capacidade de aproximadamente 40 TB, oferecendo um espaço de buffer suficiente para lidar com possíveis interrupções de transferências para o local de armazenamento permanente principal do CERN. Para a transferência de dados foi escolhida a tecnologia para *links Gigabit Ethernet*, principalmente devido a sua ampla utilização e aceitação no mercado de redes LAN, o que inclui seu baixo custo de aquisição. A grande variedade de velocidades de 10Mb/s a 10Gb/s e a disponibilidade de *switches* com grande capacidade de portas são outras características que foram levadas em consideração.

As quatro portas *Gigabit Ethernet* da placa TELL1 são usadas como estágio de saída. Algumas destas são conectadas a um grande switch de rede, fornecendo a conectividade entre a TELL1 e nós individuais do *Cluster*. Para superar um *overhead* significativo por pacote de *Ethernet*, o conceito de MEP (*Multiple Event Packets*) foi concebido. Em torno de dez pacotes de dados provenientes de diversas decisões de *trigger* são coletados em um único pacote IP e por sua vez transferidos de uma só vez pela rede. O tamanho do *cluster* de *CPU*

executando o algoritmo do HLT é determinado pela média de tempo de execução por evento, mas também possivelmente determinada pelo número máximo de banda em um nó de processamento individual. Se o tempo de execução for muito baixo, a banda de entrada pode constituir um fator limitante e o número de nós deve ser aumentado. O algoritmo HLT é executado em um *Cluster* de *CPU* bastante grande, onde se espera um número de 1000 a 2000 servidores do tipo 1U contendo *CPUs* com tecnologia *multi-core*. O tamanho inicial do cluster era de apenas 200 servidores e a máxima capacidade para instalação é de 2200 unidades. Este grande número de servidores esta organizado em 50 *sub-clusters* de 20 a 40 *CPUs* cada. A escalabilidade é garantida desde que um *sub-cluster* seja uma unidade funcional e não haja comunicação cruzada entre os *sub-clusters*.

A qualidade dos dados adquiridos é checada em um *cluster* de monitoramento separado que recebe os eventos aceitos pelo HLT e executa algoritmos definidos como por exemplo, o monitoramento da eficiência dos canais e a resolução de massa de um detector.

O sistema TFC trata todos os estágios do readout de dados entre a eletrônica de *front-end* e o *cluster* de processamento do sistema *online,* distribuindo o *clock* de sincronismo do feixe, as decisões do *trigger* L0, sinais síncronos de *reset* e comandos de *fast control*. O sistema é uma combinação de componentes eletrônicos comum a todos os experimentos do LHC e da eletrônica particular do LHCb. A arquitetura do TFC é mostrada na Figura 11 e pode ser descrita em temos de três componentes principais; a rede de distribuição do TFC, a rede de *trigger throttle* e o TFC *máster,* também conhecido como *Readout Supervisor*.

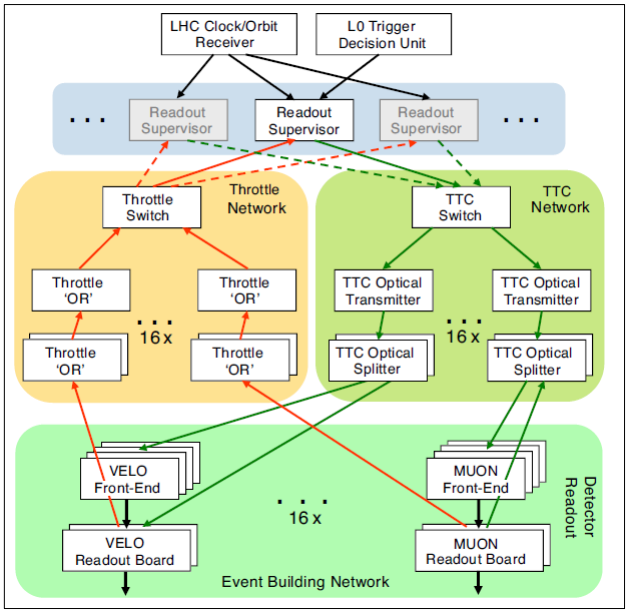


Figura 11 – Arquitetura do sistema TFC

A rede óptica de distribuição do TFC, assim como os transmissores e receptores, é baseada no sistema TTC do LHC desenvolvido no CERN. Além de transmitir o *clock* de sincronismo do feixe, o protocolo tem como característica a baixa latência do canal de *trigger* e um segundo canal com informações usadas para transmissão de comandos de controle codificados. Um switch foi desenvolvido e introduzido na rede de distribuição para permitir o particionamento dinâmico do LHCb, suportando as atividades independentes e concorrentes das atividades dos detectores, assim como verificação, calibração e teste.

A rede óptica de *throttle* é usada para transmitir o sinal de *back-pressure*, isto é, a inibição do sinal de *trigger*, para as partes assíncronas do *readout* para o *Readout Supervisor* em caso de congestionamento do caminho de dados. A rede incorpora um switch para suportar o requerimento de que o sistema de *readout* seja particionável e para permitir que os módulos executem a operação logica *OR* (OU) do sinal de *throttle* de cada subsistema localmente.

O elemento principal do TFC é o *Readout Supervisor.* Sua função é fazer a interface entre o sistema de trigger do LHCb e a cadeia de *readout*. Este elemento sincroniza as decisões de *trigger*, os comandos de sincronismo de feixe ao *clock* e o sinal da orbita do feixe, providos pelo LHC. Outra característica é a capacidade de produzir uma variedade de *auto-triggers* para a calibração e teste dos detectores, além de e realizar o controle de *trigger* em função da carga do *readout*. A fim de executar um balanceamento dinâmico da

carga sobre os nós do *cluster*, o *Readout Supervisor* seleciona e retransmite o destino do próximo grupo de eventos para as placas de *readout* baseado em um esquema de crédito em que cada nó do *cluster* envia os dados solicitados diretamente para o *Readout Supervisor*.

Para cada evento de *trigger* o *Readout Supervisor* transmite os dados pela rede dos *readout. E*ste será anexado com os dados do evento, contendo o identificador do evento, o tempo e a fonte de onde o *trigger* foi gerado.

O ECS garante o controle e monitoramento operacional de todo LHCb. Isto engloba não somente o controle tradicional de domínio do detector, como também o controle de alta e baixa tensão, temperatura, fluxo de gases, pressões, controle e monitoramento do *Trigger*, TFC e do Sistema de aquisição. Os componentes de *hardware* do ECS são um tanto diversos, principalmente em consequência da variedade de equipamentos que necessitam ser controlados, como *crates* padronizados e fontes de alimentação. No LHCb, um grande esforço foi feito para minimizar o número de diferentes tipos de interface e barramentos de conexão. São encontrados:

- SPECS (*Serial Protocol for ECS*) é um barramento serial de alta velocidade (10 Mb/s)
- CAN (*Controller Area Network*)
- *Ethernet*

Os dois primeiros, SPECS e CAN, são utilizados em equipamentos instalados na área de alta radiação próxima ao detector. Estas interfaces toleram um certo nível de radiação, porém não são construídos para este fim. Conexões *Ethernet* são somente utilizadas em áreas livres de radiação. A *Ethernet* é usada para controlar não somente computadores, mas também placas TELL1 através de um CCPC montado diretamente em cada placa. Esta escolha permite que computadores com interfaces comuns de rede controlem a eletrônica de *readout.*

O *software* de controle do ECS é baseado no PVSS II, um sistema comercial SCADA (Supervisory Control And Data Acquisition). Este *toolkit* fornece a infraestrutura necessária para a construção do sistema ECS, como a banco de dados da configuração do detector, a

comunicação entre componentes distribuídos, bibliotecas gráficas para a operação de painéis e um sistema de alarmes. Baseado no PVSS, um sistema hierárquico e distribuído foi desenvolvido como mostra a Figura 12.

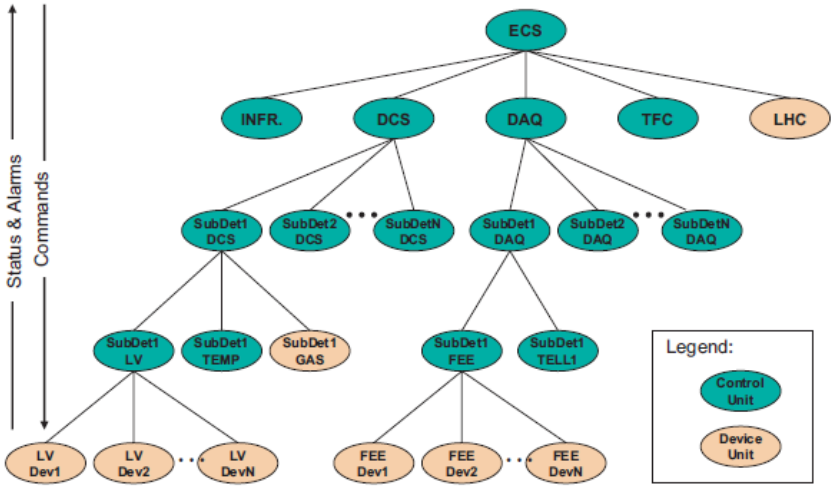


Figura 12 - Sistema hierárquico e distribuído

As *Device Units são* os componentes de acesso de baixo nível que modela os dispositivos físicos e tipicamente comunicam diretamente com o *hardware*. De uma maneira geral é implementado uma máquina de estado simples executada pelo controlador *Control Unit.* Exemplos de *Device Units* são: Fontes de alimentação e processos de *software*, como o HLT.

As transações e estados de alto nível são implementados nas *Control Units* que também contém a lógica local para a recuperação de erros de uma *Device Unit* Subordinada. Típicos exemplos de *Control Units* são: O sistema de Alta tensão, ou componentes que controlam *crates* de um detector ou todo o cluster de filtragem de eventos. As *Control Units* podem sem controladas por outras *Control Units*, para permitir a construção de uma hierarquia de profundidade arbitrária. O sequenciamento de estados é conseguido com o uso de um pacote de máquina de estados finito, baseado no SMI++ que permite a criação de uma lógica complexa. Os componentes distribuídos do sistema ECS são conectados através de uma larga rede *Ethernet*, constituída de centenas de links *Gigabit Ethernet*.

4. Upgrade da eletrônica e sistema online do LHCb

Para que seja possível implementar a aquisição de dados como previsto no *upgrade* do LHCb[21] deve-se modificar a arquitetura da eletrônica[22]. A atual arquitetura, inclui um *buffer* em *pipeline*, impondo um limite na taxa de aquisição e de *trigger* de 1 MHz. Qualquer aumento acima desta taxa requer uma nova arquitetura. Para atingir taxas de 40 MHz necessita-se do uso de tecnológicas modernas que se adaptem ao uso na física de altas energias. Por exemplo, *links* ópticos de alta velocidade serão instalados para acomodar o aumento do volume de dados vindos do detector, assim como novos esquemas de compressão de dados que serão implementados para reduzir o número de *links*. Embora o objetivo seja eliminar o *trigger* baseado em *hardware*, um mecanismo de controle será desenvolvido para controlar o fluxo de dados na aquisição. Este controle poderá ser aprimorado com informações da física, da mesma maneira do mecanismo de *trigger* existente no *Level-0*, *LLT (Low Level Trigger)*.

Embora o *upgrade* exija grandes modificações na eletrônica do detector, algumas medidas estão sendo tomadas para minimizar o custo, tempo de desenvolvimento e esforço na instalação, são eles:

1. Reuso das partes existentes na eletrônica de *FE* que satisfaçam os requerimentos do *upgrade*.
2. Desenvolvimento de módulos e dispositivos comuns usados por todos os detectores.
3. Reutilizar o máximo possível da infraestrutura existente.

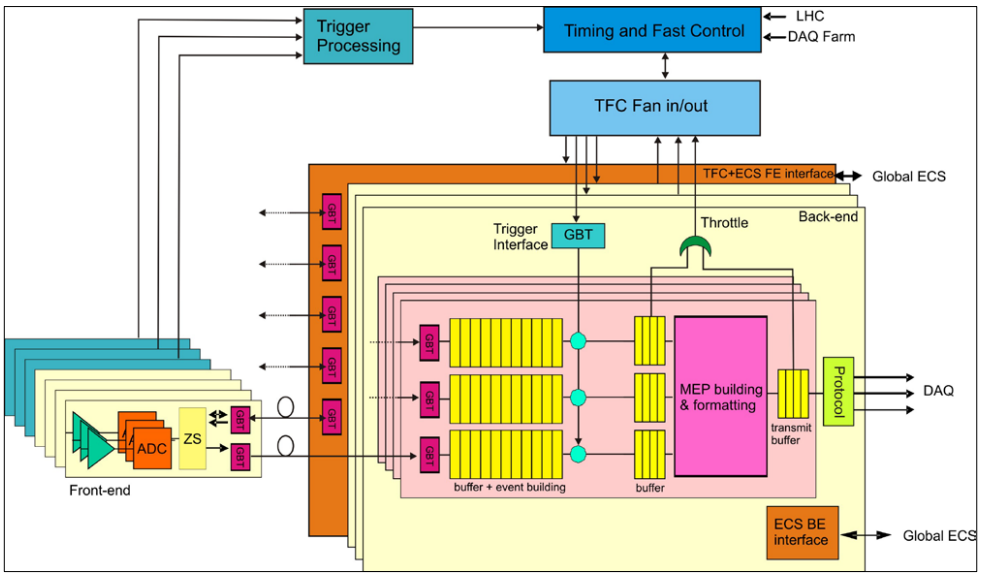


Figura 13 - Arquitetura geral do upgrade na eletrônica

A arquitetura geral do upgrade na eletrônica é mostrada na Figura 13. A eletrônica de FE sujeita a radiação, localizada junto aos detectores, é responsável pela amplificação e condicionamento do sinal gerado nos detectores. Estes sinais são digitalizados, comprimidos, formatados e então transmitidos via *links* de ópticos de alta velocidade. A eletrônica de *Back-End* (*BE)* baseada nas placas de aquisição TELL40 localiza-se em uma área protegida, afastada da radiação, e recebe os dados através dos *links* ópticos. Após a filtragem pelo LLT, os dados são digitalizados para transmissão através do sistema de aquisição de dados.

A transmissão da informação do *trigger* é feita pelo sistema TFC e contém o número de identificação de *bunch-crossing* onde o LLT teve uma decisão positiva. A configuração e monitoramento da eletrônica de BE e FE é feita através da interface ECS.

Links genéricos estão sendo desenvolvidos para o upgrade dos experimentos do LHC. Um mesmo link pode ser usado para a aquisição de dados, TFC e ECS. Por razões de custo e flexibilidade, os dados provenientes dos detectores serão transmitidos em links unidirecionais enquanto as funcionalidades de TFC e ECS irão se juntar em um número menor de *links* bidirecionais.

A estrutura de *bunch* do LHC será constituída de 3564 *bunches* espaçados por 24.95 ns. *Resets* regulares dos sistemas irão ser gerados para garantir o sincronismo e minimizar a perda de dados.

A taxa de ocupação de dados na banda disponível será determinada pela intensidade de colisões próton-próton no LHCb dominada pelo ruído do detector e da eletrônica. A energia nominal instantânea planejada para o upgrade e de 1 x 10^{33} $cm^{-2}s^{-1}$.

4.1 Eletrônica de *Front-end*

Um típico módulo de FE pode ser visto na Figura 14. Estes módulos consistem basicamente em circuitos integrados com a eletrônica de aquisição do FE conectados a *links* ópticos de alta velocidade bidirecionais, implementados usando o GBT (*Gigabit Bidirectional Trigger and Data link*) [23]. Os circuitos integrados da eletrônica de FE devem ser tolerantes a radiação presente no ambiente do detector. Áreas de alta radiação algumas vezes requerem circuitos integrados desenvolvidos sob medida. Entretanto, dispositivos encontrados comercialmente poderão ser utilizados desde que provem ser tolerantes aos níveis de radiação esperado.

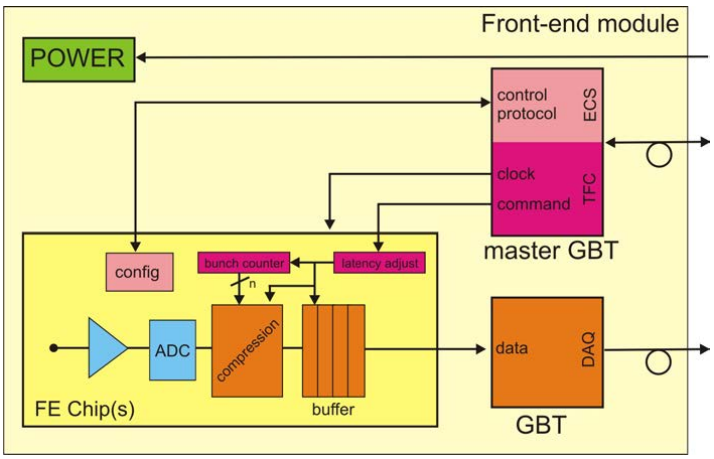


Figura 14 - Típico módulo de FE

Espera-se que o circuito integrado do FE deva ser capaz de amplificar, condicionar e digitalizar o sinal gerado pelo detector de acordo com seus respectivos requerimentos. O tempo total para a execução da aquisição dos sinais nestes circuitos deve ser rápido suficiente para permitir a identificação de uma colisão com o seu *bunch crossing,* enquanto que o tempo de recuperação para uma próxima colisão deve seguir as especificações do detector em questão. Cada detector deverá escolher o seu método de conversão analógico-digital dos sinais de forma a melhorar o desempenho, complexidade e custo.

Os circuitos digitais serão sincronizados com o *clock* de 40 MHz proveniente do GBT. O circuito deverá também permitir o ajuste de fase do *clock* para otimizar sua eficiência, respeitando colisões com espaços de 25 ns. O tamanho dos circuitos de armazenamento deverá ser tal que seja capaz de absorver flutuações estatísticas no tamanho de um evento comprimido para fazer um uso eficiente da largura de banda fornecida pelo GBT. Isto implica que dados de diferentes módulos de FE irão ser enviados de forma assíncrona para os módulos de BE. Informações adicionais serão adicionadas posteriormente nos pacotes de dados permitindo assim a reconstrução de eventos completos. A formatação mínima necessária é anexada ao cabeçalho do pacote contendo bits do *bunch-counter* (BXID). O BXID deverá ser gerado no circuito integrado do FE de forma síncrona em referência ao *clock*. Estes bits serão usados para rastrear os dados posteriormente no sistema.

Após o pacote estar completo, os dados serão enviados para a interface de DAQ do circuito integrado GBT e, em sequência, transmitidos via links ópticos.

4.2 Arquitetura geral da interface TFC/ECS no *Front-end*

Os sinais de controles e sincronismo do sistema serão enviados através dos links GBT da área protegida de radiação pelo *master-GBT*. Um pacote de dados será transmitido a cada período de 25 ns e transmitidos simultaneamente para todos os módulos de FE. Este pacote deverá ser decodificado pelo circuito integrado de FE e deverá se encarregar de algumas tarefas. As mais importantes serão:

1. Gerar o sinal de reset nos FE.
2. Gerar o pulso de calibração.
3. Gerar o sinal de reset para o contador BXID.

Para sincronizar o sistema devido a utilização de diferentes comprimentos de links, cada circuito integrado deverá ser capaz de atrasar os comandos do TFC por até 16 ciclos de *clock*. Isto permitirá a correção de qualquer diferença de latências entre os dispositivos do FE conectados ao mesmo barramento de comandos do TFC.

A configuração dos circuitos integrados do FE será feita pela interface ECS encontrada no circuito integrado do GBT-SCA[24]. Os circuitos integrados de FE deverão suportar ao menos um dos protocolos padrões disponíveis no GBT-SCA para este fim. Para permitir a leitura dos dados recém-configurados ou para qualquer outro tipo de verificação destes dados. Cada link de ECS deverá ser implementado como um link ponto-a-ponto entre o sistema do GBT e os dispositivos de FE.

Os protocolos padrões disponíveis no circuito integrado GBT-SCA são:

- Até 16 controladores I^2C
- Um controlador JTAG
- 32 canais de conversores A/D (Multiplexados)
- Um barramento de memória de 32 bits
- Até 4 controladores PIA (*Parallel Interface Adapter*)
- Um barramento SPI
- Até 4 canais de conversores D/A

A largura de banda disponível no GBT para a transmissão de informações ECS/TFC é grande, então, em muitos casos será mais eficiente o uso de um link bidirecional GBT para fornecer informações de ECS/TFC para muitos módulos de FE.

4.3 Placas de *readout*

Para acomodar os novos requerimentos do *upgrade*, o sistema de aquisição e controle atual também deverá ser modificado. Uma placa genérica que cobre todos os requisitos necessários está sendo desenvolvida pelo CPPM[25].

Dois componentes genéricos estão em fase de desenvolvimento para formar uma placa de *readout* compatível com o padrão ATCA[26], mostrada na Figura 15. São elas:

- AMC40: terá a capacidade de comunicar-se com a eletrônica de FE e *Clusters* de processamento.

- ATCA40 : placa-mãe que hospeda até quatro placas AMC40 como *mezzanino*.

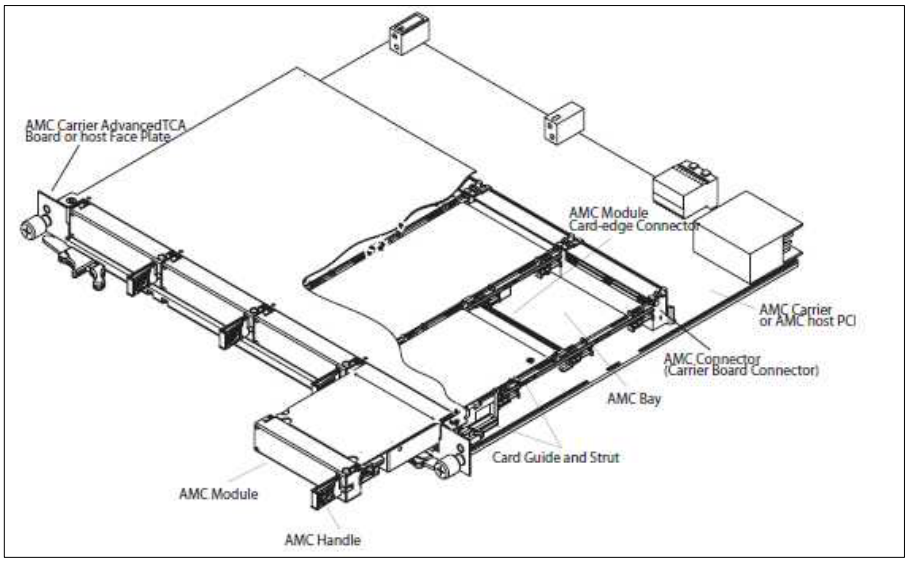


Figura 15 - Placa ATCA com quatro módulos AMC

A placa AMC40 está sendo desenvolvida para ser capaz de processar até 36 links ópticos de entrada e até 36 links ópticos de saída, em alta velocidade, através do *transceiver* do FPGA Stratrix V GX. Cada link deverá ser capaz de transmitir ou receber até 10 Gbits/s de dados. Um diagrama de blocos simplificado pode ser visto na Figura 16.

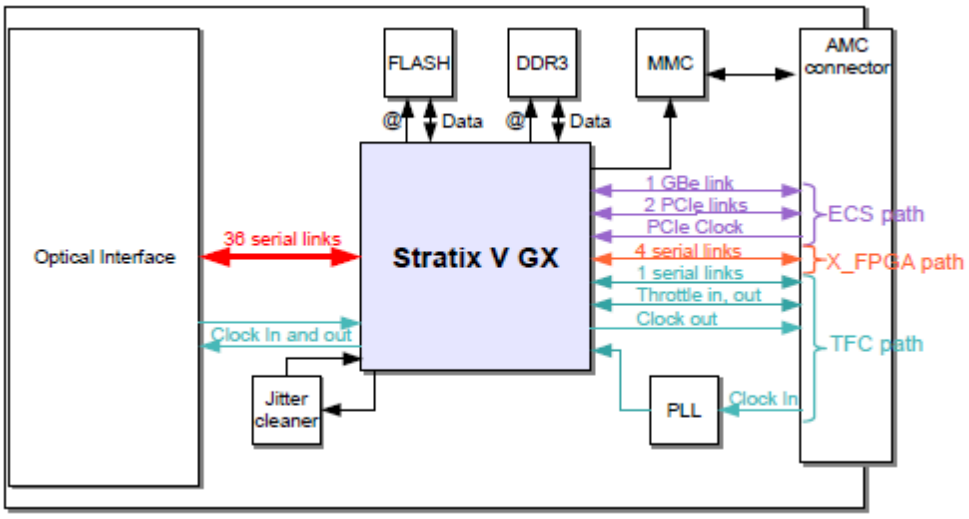


Figura 16 - Arquitetura da placa AMC40

O FPGA gerencia três tipos de caminhos através do conector AMC

- O caminho do ECS que pode ser através da conexão *Gigabit Ethernet* ou PCIe.
- O caminho entre as FPGAs feito através de quatro *links* seriais.

- O caminho do TFC feito por linhas de *clock* bidirecionais, linhas LVDS para transmissão de informações de *throttle* e links seriais de alta velocidade.

Os caminhos de ECS, TFC e *inter-FPGA*, descritos anteriormente, são gerenciados pela placa ATCA40. Seu diagrama de blocos pode ser visto na Figura 17.

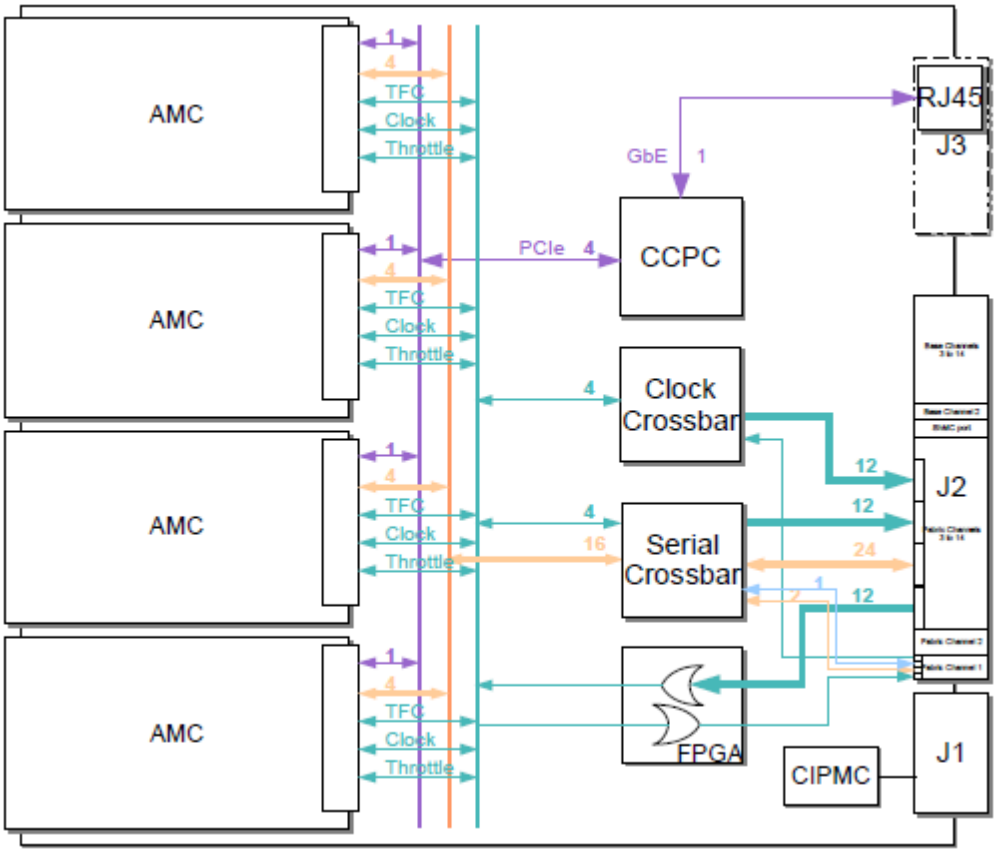


Figura 17 - Placa ATCA40

Um barramento de *clock* (*Clock Crossbar*) permitirá o roteamento do clock entre as placas AMC e o conector J2, ou entre placas AMC. O Barramento serial (*Serial Crossbar*) permitirá o roteamento dos links seriais de alta velocidade entre as placas AMC e o conector J2, ou entre placas AMC.

4.3.1 ECS nas placas de readout

Um CCPC poderá ler ou escrever registradores localizados dentro da FPGA através de quatro barramentos PCIe conectados as placas AMC. O CCPC deverá executar como sistema operacional o Linux e também será responsável pelo controle do *link Ethernet* com o sistema supervisório externo. Este é o principal componente que faz a ligação destas placas com o sistema ECS global. Um switch externo irá permitir o acesso a todos os CCPCs da mesma maneira que se faz na atual arquitetura encontrada nas placas TELL1.

Similarmente com a eletrônica de FE, os softwares de configuração e monitoramento das placas de *readout* deverão ser fornecidos centralmente:

- Bibliotecas de baixo nível e ferramentas de linhas de comando para os CCPC nas placas irão permitir o acesso a diferentes chips encontrados na placa de *readout* e seus *mezzaninos.*

- Um servidor executado no CCPC irá permitir a comunicação via TCP/IP e implementará comandos de alto nível para a configuração e monitoramento de diversos chips.

A integração com o componente ECS via WinCC-AO (ex PVSS) irá fornecer uma descrição de alto nível e acesso a todos os componentes eletrônicos.

O sistema ECS da placa de *readout* é o foco deste trabalho e no capítulo a seguir serão descritos todos os elementos necessários para o desenvolvimento deste sistema.

4.4 Arquitetura de *readout*

O fluxo de dados do ECS e TFC irão requerer links bidirecionais enquanto a aquisição será feita por links unidirecionais. Os FE em geral irão necessitar de menos links para ECS e TFC do que links para aquisição. O *crate* que gerenciará um detector poderá ajustar a relação entre o número de placas de *slow control* (FSC40) e aquisição(TELL40) para atingir o número de FE necessárias na eletrônica de FE, como mostrado na Figura 18.

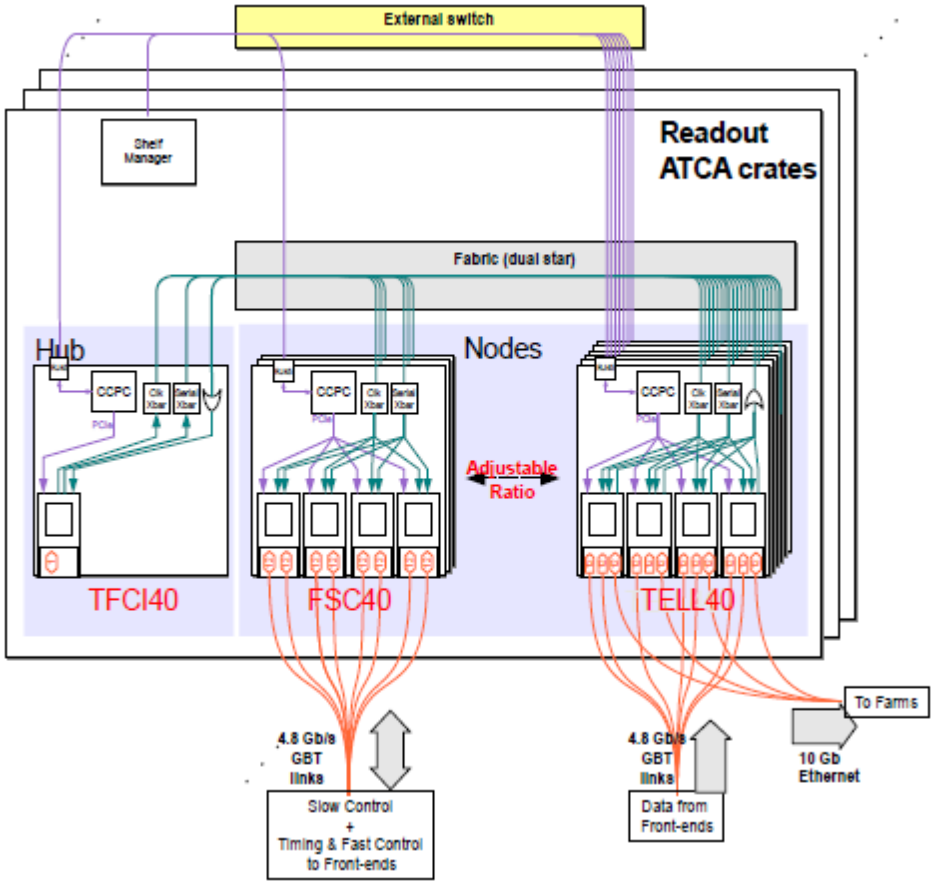


Figura 18 - Arquitetura de readout com switch externo

Cinco configurações podem ser tratadas com o mesmo hardware:

- TELL40 para aquisição de dados.
- FSC40 para *slow control* e interfaceamento TFC com os FE.
- ODIN40 para Supervisão do TFC, decisões de trigger de baixo nível e interface com o tempo do LHC.
- TFCI40 para o gerenciamento local do TFC em cada *crate*.
- TRIG40 para fornecer interface entre o antigo trigger L0, o sistema de TFC e DAQ

Na configuração TELL40, o sistema será capaz de coletar fragmentos de um evento da eletrônica de FE e juntá-los em um único pacote Ethernet que será enviado para o cluster de processamento. Três caminhos diferentes são ilustrados pela Figura 19.

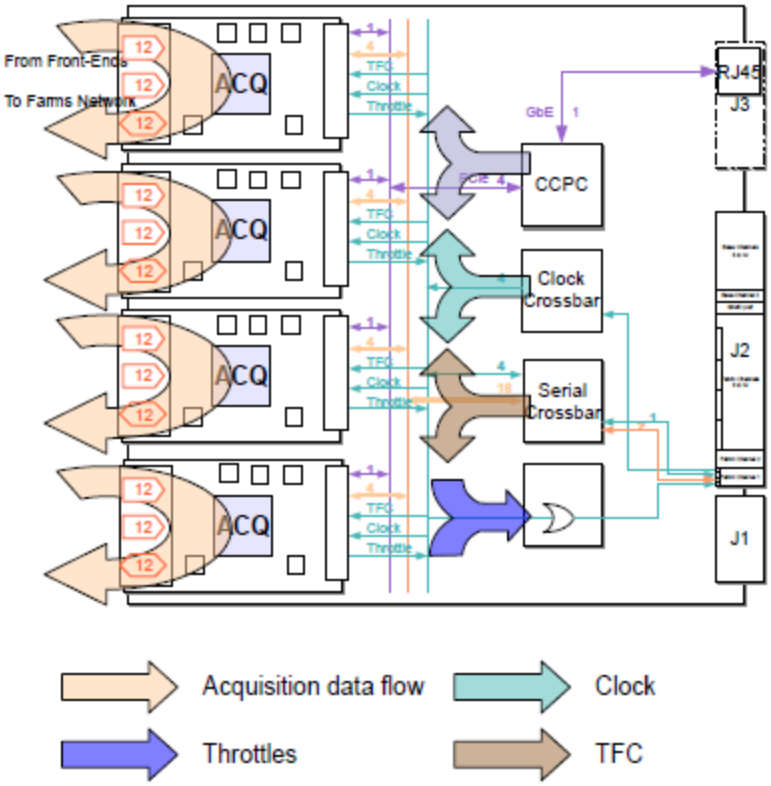


Figura 19 - Configuração TELL40

Os fragmentos dos eventos serão processados no FPGA da placa AMC40. Informações de *clock* e TFC serão recebidas pela placa TFCI40 localizadas no mesmo *crate*. E então distribuídas para as placas TELL40 através do *backplane* do *crate* e distribuídas para os FPGA da placa AMC40 pelo barramento local. Nesta configuração as informações de *clock* e TFC serão utilizadas pelo processamento local da TELL40. A informação de *throttle* gerada por cada placa AMC40 será então concatenada e enviada de volta para a única placa TFCI40.

Os dados do ECS das placas AMC40 serão recebidos pelo CCPC via rede dedicada e transmitidas para os FPGAs localizados nas placas AMC40 através dos links PCIe.

A placa na configuração FSC40 atuará como concentrador do ECS/TFC, os caminhos diferentes desta configuração podem ser visto na Figura 20.

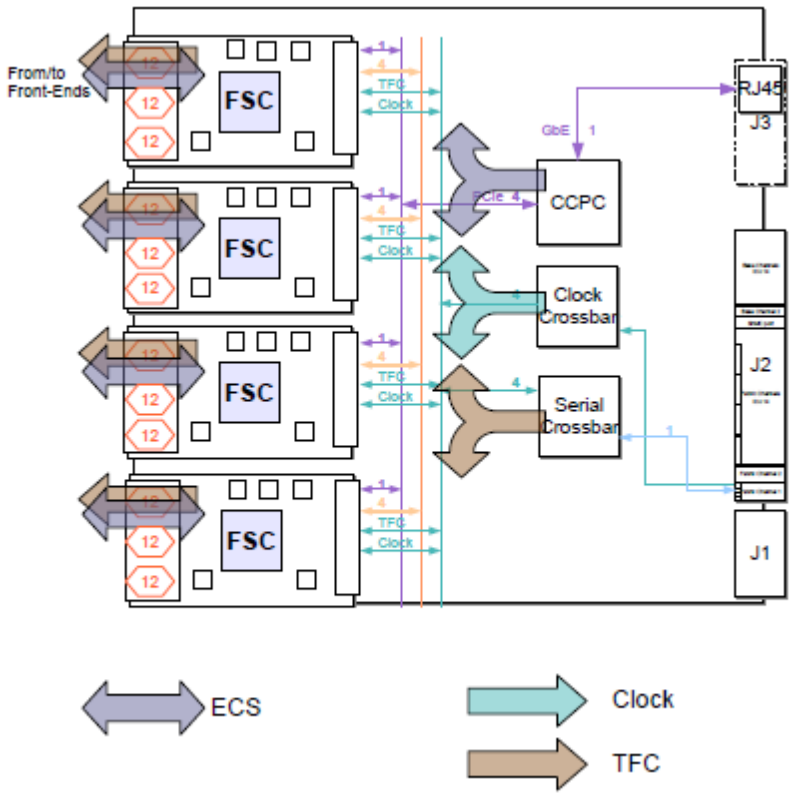


Figura 20 - Configuração FSC40

Os dados do ECS e TFC serão recebidos e transmitidos de forma similar a TELL40. Porém, todas estas informações serão concatenadas para serem enviadas pelo *link* GBT para os FE. Um mecanismo especial, embarcados nas FPGA, irá garantir que a fase do *clock* seja constante respeitando o principal com a precisão de 50 us RMS.

Até 144 links ECS poderão ser controlados por uma placa FSC40.

ODIN40 é o componente principal que tratará dos sinais de clock e distribuição de comandos. Sua configuração pode ser observada na Figura 21.

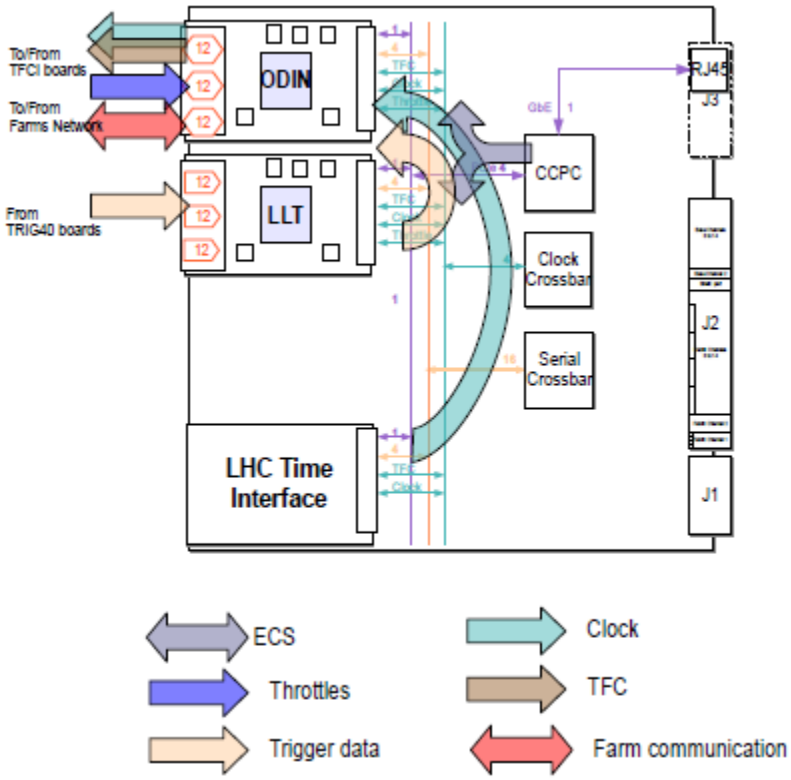


Figura 21 - Configuração ODIN40

O *clock* do LHC será recebido por uma placa dedicada (LHC Time Interface) e será enviado para a placa ODIN/AMC40 através da rede de *clock* local e seu *crossbar* associado. Os *triggers* candidatos do LLT serão computados pela placa TRIG40 e recebidos pela placa LLT/AMC40 que controlará a decisão final do LLT e o enviará para a placa ODIN/AMC40 através dos links seriais de alta velocidade que interconectam as diferentes placas AMC.

A função *Super*-ODIN, implementada na placa ODIN/AMC40, usa a decisão principal do LLT, a informação de *throttle* da placa TELL40 e a disponibilidade do cluster de processamento, para construir a informação do TFC que serão transmitidas pelas fibras ópticas para a placa TFCI40 em cada *crate*. Estra transmissão será feita sincronamente com o *clock* do LHC e em fase constante.

Até 24 *crates* poderão ser tratados por uma placa *Super*-ODIN. A comunicação com os *cluster* de processamento será feito por meio de 12 links ópticos bidirecionais usando o protocolo 10 Gb Ethernet.

A placa TFCI40 será a interface entre a placa ODIN40 local do *crate*. Sua configuração pode ser encontrada na Figura 22.

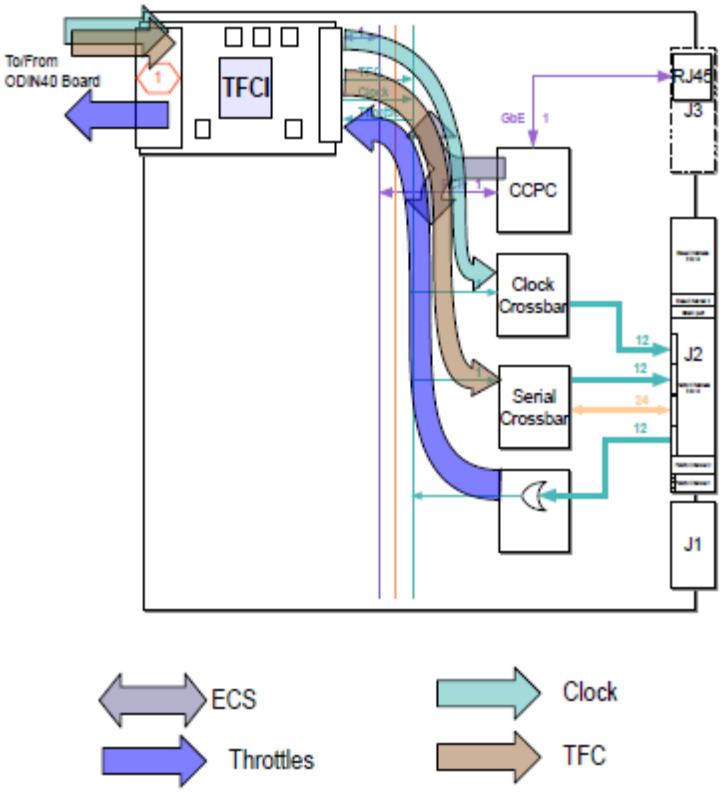


Figura 22 - Configuração TFCI40

As informações de *clock* e TFC serão recebidas pelos módulos ODIN40 através de links ópticos, e assim distribuídas para outras placas no mesmo *crate* (FSC40, TELL40 TRIG40) via ao *backplane* com ajuda do *crossbar* localizado na placa ATC40. Informações de *throttle* oriundas de placas TELL40 sofrerão a operação logica “OU” localmente e enviadas para a placa ODIN40 via *link* óptico.

A interface entre os processadores LLT de múons e dos calorímetros, e unidades de decisão LLT localizadas na placa ODIN40 será feita através da configuração TRIG40, ilustrada na Figura 23.

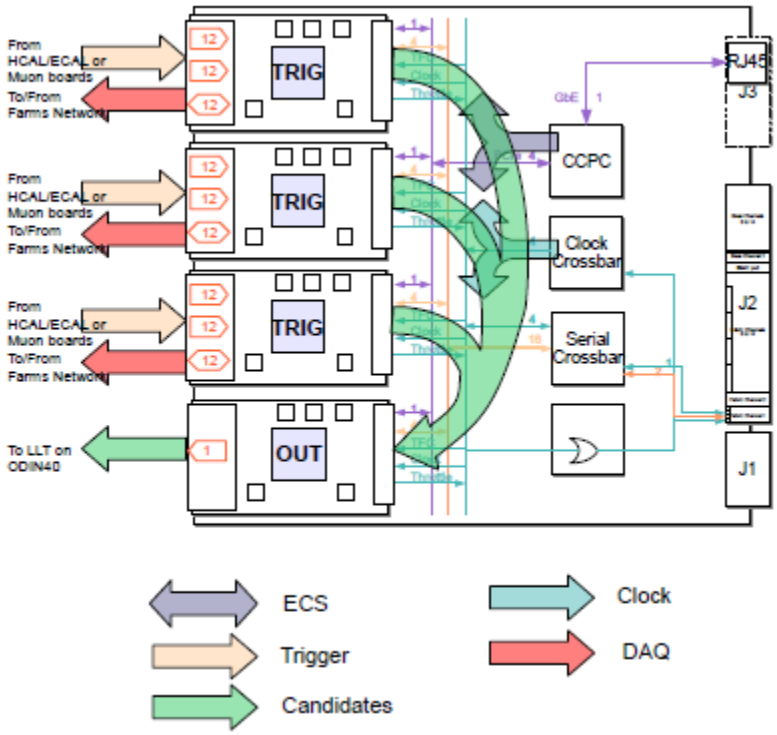


Figura 23 - Configuração TRIG40

O processadores de LLT enviarão os seus candidatos a placa TRIG/AMC40 a um taxa de 40 Mhz. Candidatos com o maior momento transverso serão selecionados. Os restantes serão enviados para a placa OUT/AMC40 pelos links ópticos, aonde a decisão final será tomada.

Os processadores de LLT enviarão os dados de entrada e os resultados do processamento a 1 MHz. Esta informação será escrita em disco e usada para verificar o comportamento do processador. Estes dados brutos serão recebidos pela placa TRG/AMC40 onde serão concatenados e comprimidos para posterior envio para o DAQ.

5. O Módulo ECS-CBPF

Motivados pela necessidade de upgrade do sistema de controle da placa TELL40, o CBPF desenvolveu um módulo de controle de experimento genérico, ECS-CBPF é uma placa em formato de *mezzanino* que foi projetada para fazer a conexão entre um sistema genérico a ser controlado, chamado neste documento de hospedeiro, e a sala de controle do experimento LHCb através de um link *Gigabit Ethernet*. Consiste em uma unidade de processamento e controle, contendo um CCPC modelo COMe-mTTi10 da Kontron [27] carregada com o sistema operacional Linux, uma unidade lógica programável Cyclone IV (FPGA) que faz o interfaceamento do CCPC com dispositivos de baixa velocidade como I^2C, JTAG e SPI a serem configurados e monitorados pela sala de controle e um *switch* PCIe PEX8609 da PLX [28] para a comunicação com outras unidades logicas programáveis em alta velocidade utilizando o barramento PCIe. Os componentes do módulo são alimentados por meio de uma tensão externa proveniente do hospedeiro.

O ECS-CBPF e seu diagrama de blocos são apresentando nas Figuras 24 e 25, respectivamente.

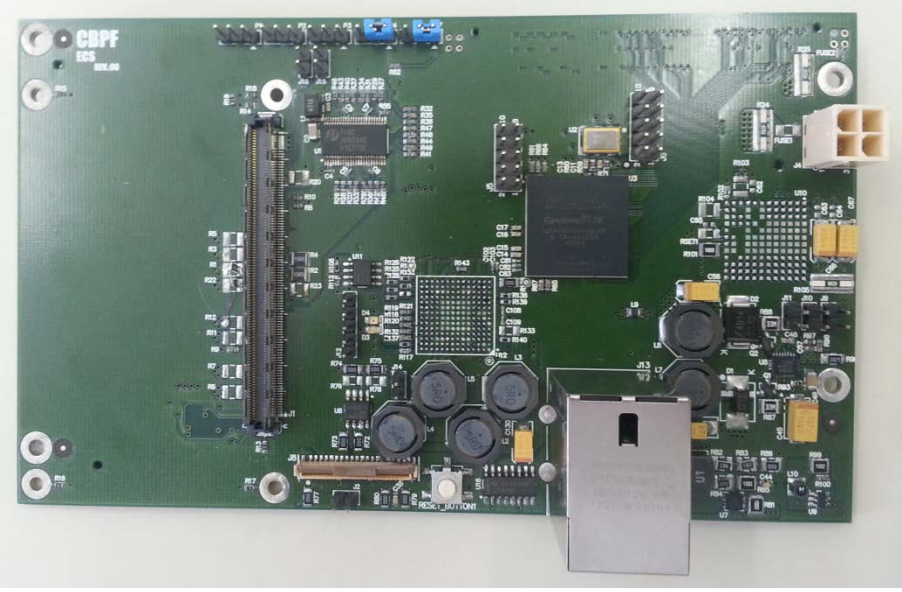


Figura 24 - Módulo Experiment Control System – CCPC CBPF

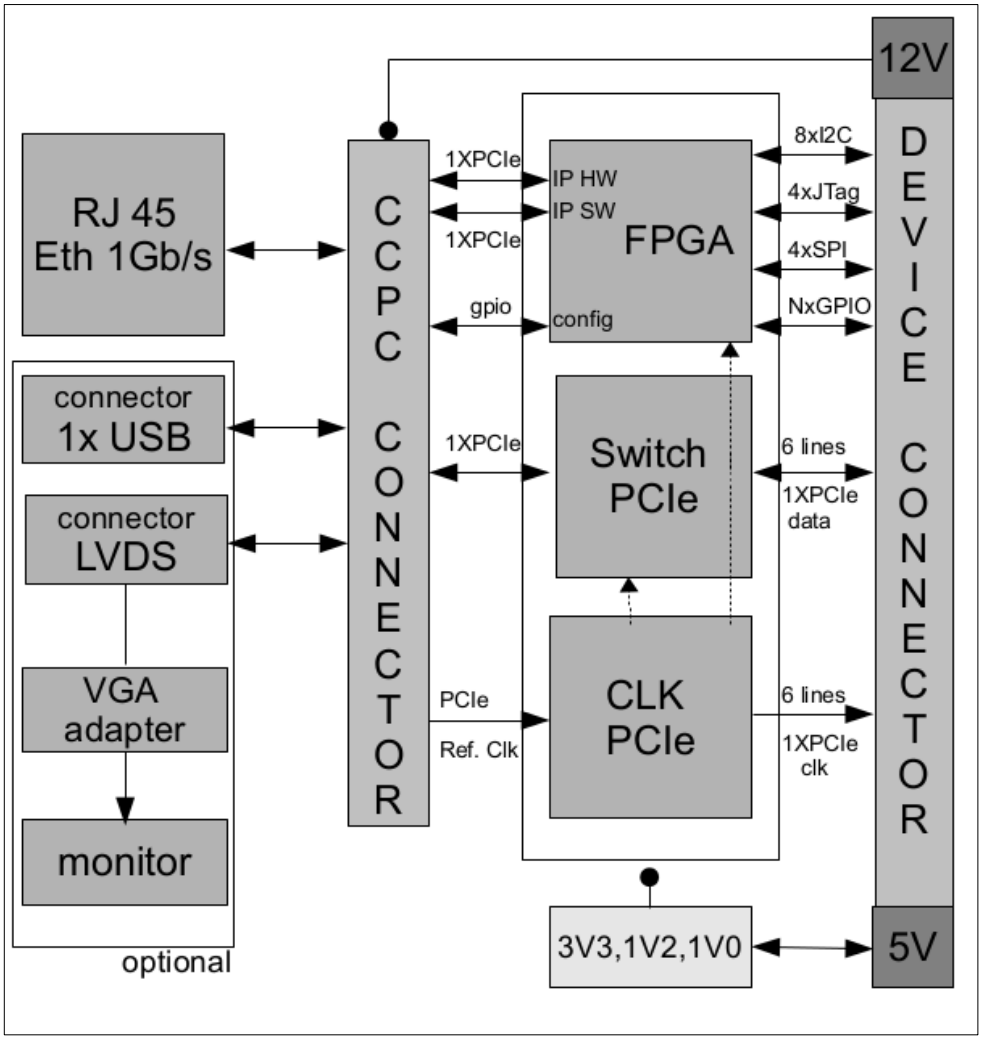


Figura 25 - Diagrama de blocos do módulo ECS-CBPF

As seguintes funcionalidades são encontradas:

- Computador de controle
- Interface padrão Gigabit Ethernet
- Seis canais de comunicação PCIe
- Oito controladores independentes I^2C
- Quatro controladores independentes JTAG
- 36 pinos de Entradas/Saídas genéricas

5.1 Protocolos e barramentos de comunicação

Para que a comunicação entre a sala de controle e a placa de aquisição seja realizada é necessária a criação entre alguns protocolos de comunicação que serão explicados a seguir.

5.1.1 Barramento PCIe

O padrão PCI Express [29] (PCIe) é uma evolução do PCI[30], barramento de comunicação entre um processador (*rootport*) e periféricos (*endpoint*) em um computador. Apesar de não se enquadrar fisicamente como um barramento, uma vez que o mesmo utiliza linhas seriais de transmissão e recepção para a comunicação, por questões históricas refere-se ao PCIe como um barramento. É importante ressaltar que no ponto de vista de *software* ou *drivers* nos sistemas operacionais o PCIe é totalmente compatível com o PCI.

O protocolo de comunicação utilizado pelo padrão PCIe é dividido em camadas[30]: camada de transação, camada de *link* de dados e camada física, como mostrado na Figura 26.

A primeira camada do PCIe é a camada de transação TL (*Transaction Layer.*) Esta interage com o dispositivo PCIe e a camada inferior a ela. A TL é responsável por iniciar o processo de transmissão de dados transformando uma requisição, provenientes de um dispositivo em um pacote de dados TLP (Transaction Layer Packet). O pacote é composto por informações em um cabeçalho que é codificado de tal forma a identificar o tipo de operação a ser realizada, como a quantidade de bytes necessários para o cabeçalho, e opcionalmente por bytes de dados, quando necessário.

A camada de link de dados DLL (*Data Layer Link*) interage com a primeira e terceira camadas. Seu principal objetivo é garantir a integridade dos dados que circulam pelo *link* PCIe. Para isto, a DLL adiciona uma sequencia numérica única no inicio do pacote e uma codificação para checagem de erros no final do mesmo. Utiliza-se a técnica de CRC (Checagem de Redundância Cíclica) para detecção de erros. Caso seja detectado um problema a DLL gera um pacote DLLP (*Data Layer Link Packet*) que é transmitido de volta para o dispositivo. Este pacote contém informações específicas, tais como o gerenciamento de hierarquia, notificação de erros, controle de fluxo de dados e outros.

A camada física (PL – *Physical Layer*) do PCIe é responsável pela transmissão e recepção das camadas anteriores através de sinais elétricos, composta por dois pares diferenciais, em geral do tipo LVDS ou PCML. Um par é responsável pela transmissão (TX) e outro pela recepção (RX). Juntos constituem em um *lane.* Pode-se combinar *lanes* em paralelo para se obter um *link.* Um *link* é composto por no mínimo um *lane* (1x), outras configurações disponíveis são: 2x, 4x, 8x, 16x e 32x. No padrão PCIe 1.1 [31] a capacidade de transporte de dados é de 2.5 Gb/s em cada direção, ou seja 250 MB/s de dados brutos (Sem levar em conta o *overhead* de cabeçalhos e outras informações), atingindo um máximo de 80 Gb/s na configuração 32x. A Figura 26 representa um pacote completo PCIe.

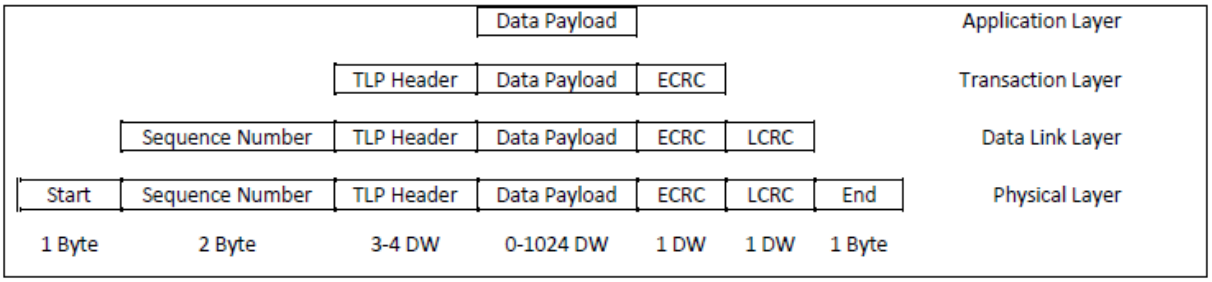


Figura 26 - Pacote de dados PCIe

5.1.2 Barramento I²C

O barramento I²C foi desenvolvido em meados dos anos 80 pela Philips, inicialmente concebido para fazer a interconexão entre uma CPU e *chips* controladores em televisores. Percebeu-se que conforme a medida que quantidade de periféricos aumenta em um barramento paralelo, as conexões se tornam mais complexas, aumentando os custos de projeto e desenvolvimento, além de tornar a placa do circuito mais suscetível à interferência eletromagnética (EMI). Por conta destes problemas, adotou-se um barramento serial de dois fios. Atualmente este barramento tornou-se padrão na indústria e fabricantes de semicondutores como Atmel, Intel, Microchip, Maxim, Texas Instrument entre outros, fabricam dispositivos compatíveis com este barramento com as mais diversas funções, como por exemplo memórias, displays, conversores A/D e D/A, sensores de temperatura, pressão e humidade, etc.

Fisicamente o barramento consiste em duas portas (fios ou linhas de comunicação) de coletor-aberto bidirecionais SDA (*Serial Data Line*) e SCL (*Serial Clock Line*). Os dispositivos são dispostos no barramento em uma arquitetura Mestre/Escravo, onde sempre o Mestre

inicializa uma comunicação. Alguns dispositivos podem atuar, não simultaneamente, como Mestre e Escravo. Como visto na Figura 27.

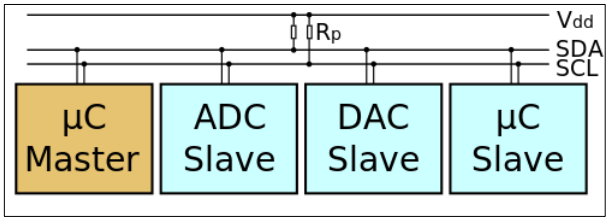


Figura 27 - Barramento I²C

O protocolo de comunicação I²C segue o seguinte fluxo:

- O dispositivo Mestre envia ao barramento um sinal START. Isto faz com que os dispositivos Escravos esperem por uma comunicação.
- O dispositivo Mestre envia um registro com o endereço do dispositivo Escravo a ser acessado e a direção da comunicação, leitura ou escrita.
    - Se o dispositivo não for o responsável pelo endereço, este irá ignorar os comandos e aguardará o sinal de STOP.
    - O dispositivo com o endereço solicitado enviará uma resposta com um sinal de ACK (*Acknowledge*).
- Logo após o dispositivo Mestre receber o sinal de ACK, inicia-se a transmissão ou requisição de dados. Após o termino das operações o Mestre envia o sinal de STOP e libera o barramento para outro Mestre atuar.

O processo descrito no fluxo à cima pode ser observado na Figura 28.

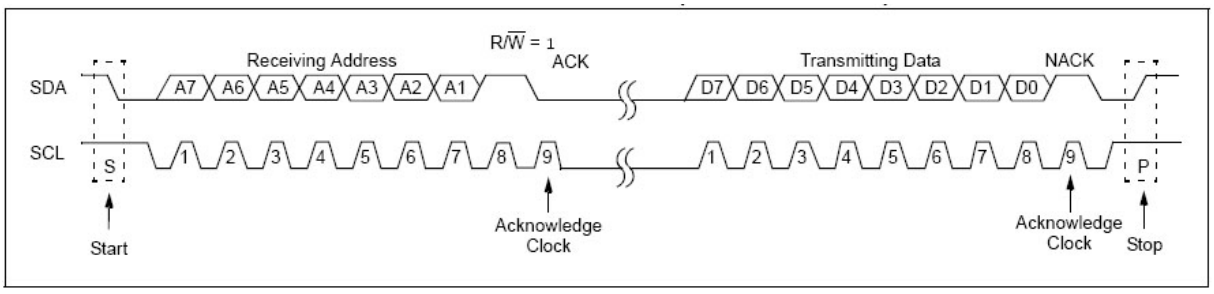


Figura 28 - Protocolo I²C

Os sinais de *clock* são gerados pelos dispositivos Mestres através da linha SCL e os dados são validos apenas quando o sinal SCL estiver em nível alto.

5.1.3 JTAG

Devido à miniaturização e discretização dos componentes eletrônicos na forma de circuitos integrados e placas de circuitos impressos com diversas camadas, acrescentou-se certa dificuldade para a realização dos testes através de pontas de prova na superfície destes circuitos. Em 1985 foi criado um grupo internacional (JTAG – *Joint Action Group*) para o desenvolvimento de uma solução deste problema. Após alguns anos de sua criação, esta solução foi adotada por grande parte da indústria e assim criou-se o padrão IEEE Std. 1149.1-1990: IEEE *Standatd Test Access Port and Boundary Scan Architecture*[32], também conhecido como JTAG. Este padrão utiliza componentes em *hardware* nos circuitos integrados, permitindo o controle através de *software*. Recentemente o grupo JTAG, desenvolveu e publicou um novo padrão o IEEE Std 1532.2-2010: *IEEE Standard for System Configuration of Programmable Devices* [33] e seu objetivo foi padronizar a configuração de dispositivos lógicos programáveis, as FPGA, através da interface JTAG.

A interface JTAG é composta por quatro pinos(um quinto opcional) e um conjunto de registradores de controle e de dados. Seu objetivo é permitir a realização de testes e monitoramento de níveis lógicos nos pinos de um circuito integrado que contém esta interface. Para selecionar o tipo de teste executado pela interface utiliza-se o *Instruction Register* (IR), o qual realiza monitoramento do nível de lógica do circuito o *Boundary Scan Register* (BSR) conectado diretamente a cada pino do dispositivo. Em conjunto com IR existe o *Data Register* (DR), para instruções de recebimento e envio de dados.

Os pinos existentes são:

- TCK: Sinal de *Clock.*
- TMS: Seleção do modo de teste (*Test Mode Select*).
- TDI: Entrada de dados.
- TDO: Saída de dados.
- TRST: Sinal de *Reset* (Opcional).

O Sinal TMS controla uma máquina de estado, chamada de *Test Access Port* (TAP), responsável pela execução dos testes e fluxo de dados do dispositivo. A Figura 29 ilustra o diagrama da máquina de estado do controlador TAP.

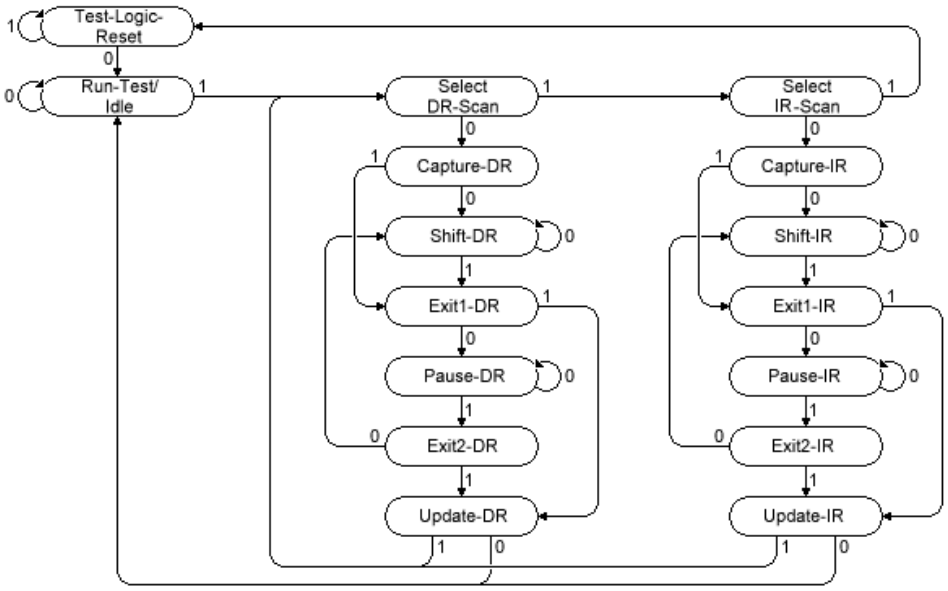


Figura 29 - Máquina de estado do controlador JTAG

O gerenciamento dos sinais, controle dos testes e configuração dos dispositivos que possuem uma interface JTAG, é realizado pelo controlador JTAG *master*, que pode controlar um ou mais dispositivos ligados em cadeia, como mostrado na Figura 30.

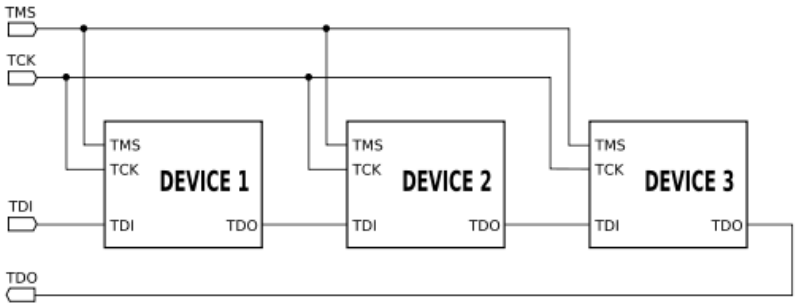


Figura 30 - Configuração de uma chain JTAG

5.2 Ferramentas de desenvolvimento

O projeto ECS-CCPC ECS foi desenvolvido utilizando ferramentas comerciais disponíveis, são elas:

- Kit de desenvolvimento PCIe Cyclone IV (DB4CGX15) [34] em conjunto com o programa da Altera, Quartus II [35], e seus módulos.
- O programa Altium Design Summer 09 [36], Utilizado para o desenho do esquemático (Apêndice A) da placa e confecção de layout para posterior produção da placa.

5.3 Kit de desenvolvimento Cyclone IV (DB4CGX15)

O kit de desenvolvimento DB4CGX15, mostrado na Figura 31, utiliza um FPGA da família Cyclone IV (EP4CGX15C6N), de baixo custo e consumo se comparado com outras famílias do mesmo fabricante. Possui um bloco em hardware dedicado para o protocolo PCIe possibilitando assim um maior desempenho e economia em relação a recursos utilizados dos elementos lógicos disponíveis no FPGA. No modelo utilizado pelo kit encontram-se disponíveis 15.000 elementos lógicos para o desenvolvimento de aplicações além de um chip de configuração (EPCS16), memória do tipo DDR2 (256MB), programador USB integrado e um conector com pinos de Entrada/Saída.

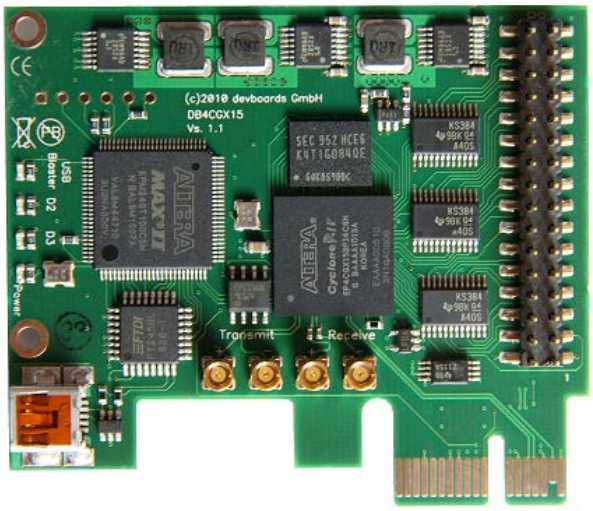


Figura 31 - Kit de desenvolvimento DB4CGX15

Com a utilização deste kit foi possível projetar e verificar o funcionamento do protocolo PCIe, assim como a conexão com os outros controladores encontrados no projeto a serem descritos no tópico do desenvolvimento do *firmware,* antes mesmo do desenvolvimento da placa.

Baseado no desenvolvimento utilizando o kit foi possível estimar a utilização de recursos no FPGA possibilitando assim à escolha do chip adequado as características do ECS-CBPF.

5.4 Desenvolvimento do Hardware

5.4.1 Distribuição de alimentação

O módulo recebe como entrada do sistema hospedeiro a tensão de alimentação de 12V e gera todas as tensões necessárias para seu funcionamento. A Figura 32 mostra a representação da distribuição das tensões envolvidas no projeto.

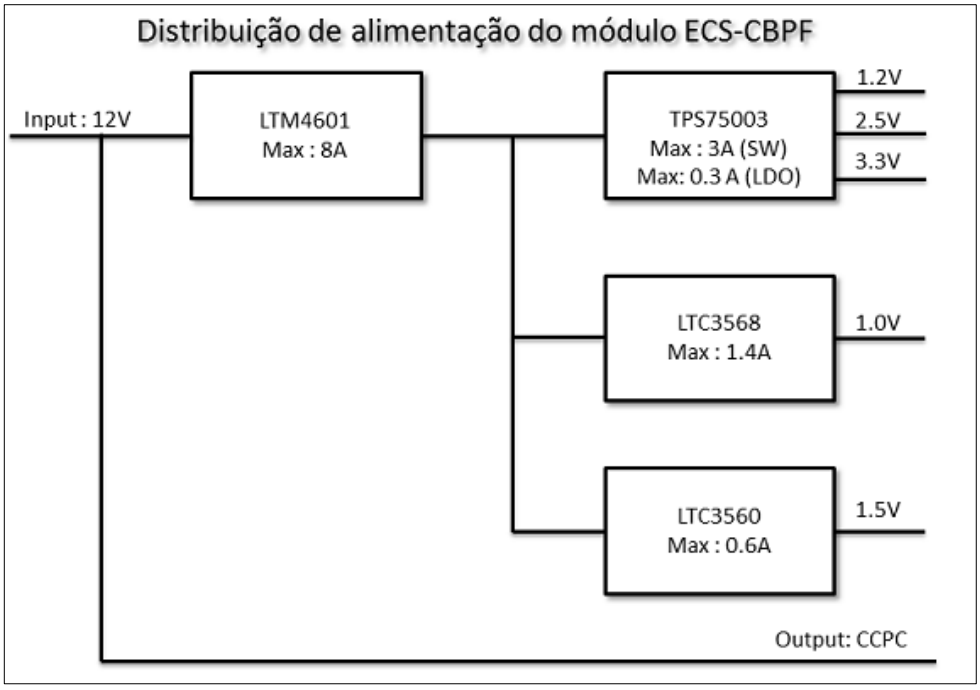


Figura 32 - Distribuição da alimentação do módulo ECS

No primeiro estágio a tensão de entrada é diretamente conectada ao CCPC e ao regulador LTM4601[37] . Este gera 5 V o qual é convertido em tensões mais baixas através de dois reguladores; o primeiro LTC3568[38] fornece a tensão de 1.0 V, e TPS75003[39] fornece as tensões de 1.2 V, 2.5 V e 3.3 V.

5.4.2 Distribuição de Clock

Para que todos as seis saídas PCIe fornecidas pelo *switch PCIe* estejam disponíveis é necessário que o *clock* de referência do PCIe seja distribuído. Para este fim utiliza-se o circuito PI6C20800[40] como mostra a Figura 33.

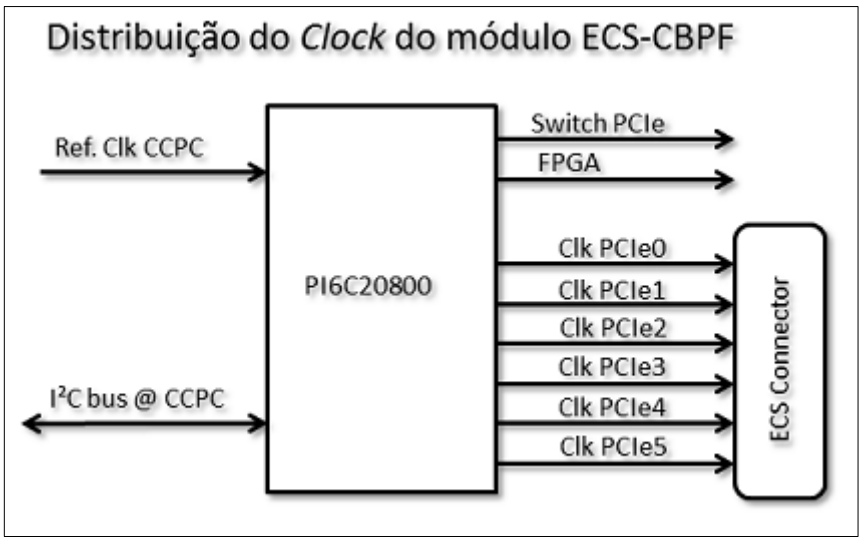


Figura 33 - Distribuição de clock de referência PCIe

Um circuito oscilador a cristal de 125 Mhz do tipo LVPECL é conectado diretamente ao FPGA e utilizado como referência dos componentes internos do FPGA.

5.4.3 CCPC

O CCPC é um módulo do tipo COM Express R2.0 com conector Type 10[41] encontrado comercialmente e fabricado por diversas empresas. O módulo possui um circuito altamente integrado com baixo consumo de energia e são compostos por uma unidade de processamento (CPU) e memória, além de dispositivos de I/O encontrados em um computador comum, como USB, áudio, gráfico e rede. Todos os sinais de I/O são mapeados em um conector de alta densidade e podem ser utilizados em circuitos específicos. A escolha de um sistema padronizado proporciona a fácil substituição caso seja necessário o aumento do desempenho em sistemas futuros.

O módulo COM Express utilizado neste projeto é o nanoETXexpressTTi 1.0Ghz 512/mSD da empresa Kontron e pode ser visto na Figura 34.

Figura 34 - Módulo CCPC Kontron

Este módulo possui:

- Um processador Intel Atom E640T (1 Ghz).
- Até 3Gb de memória RAM DDR *onboard.*
- Quatro linhas PCIe, uma reservada para conexão com o controlador de sistema (*Southbridge*).
- Uma Interface Gigabit Ethernet.
- Um controlador SATA (não utilizado).
- Seis *Links* USB (um link utilizado no protótipo).
- Audio.
- Vídeo (DVI disponível no protótipo).

No protótipo da placa ECS-CBPF as três linhas PCIe disponíveis no conector são conectadas, duas distribuídas diretamente ao FPGA e outra para um Switch PCIe. Uma porta USB e saída DVI de vídeo estão disponíveis na placa de forma opcional para facilitar a interação com o CCPC e corrigir possíveis erros.

5.4.4 Switch PCIe

A arquitetura PCIe utiliza comunicação através de ligações ponto-a-ponto. Desta maneira a transmissão de pacotes entre o *rootport* e diferentes *endpoint*, ou entre dois *endpoints*, pode ser feita através do chaveamento das portas disponíveis. Um switch PCIe foi

utilizado para aumentar o número de links PCIe disponíveis na placa com o intuito de serem roteadas para a placa hospedeira. A distribuição dos links PCIe pode ser vista na Tabela 1.

Fonte	Disponibilidade de links	Destino
CCPC	3	2 para o FPGA
		1 para o PCIe Switch
PCIe Switch	8	6 para o Conector ECS

Tabela 1 - Roteamento dos links PCIe

Foi utilizado o circuito integrado PEX8609 da PLX *technology* capaz de fornecer oito portas em 1x de *downstream* (*endpoints*) e uma de *upstream* (*rootport*). Devido a limitação da distribuição do *clock* de referência destas oito portas seis são utilizadas. Outra característica importante deste circuito é a possibilidade de realizar operações de *hotplug*. Essa característica garante que o *firmware* dos dispositivos PCIe da placa hospedeira possam ser programados após o CCPC ter feito seu *boot* e estar com o sistema operacional em execução.

5.4.5 FPGA

A conexão entre o CCPC e os controladores de dispositivos de baixa velocidade I²C e JTAG é feita através de um FPGA via *link* PCIe. O circuito utilizado na placa ECS-CBPF é o Cyclone IV EP4CGX30CF19C8 [42] da Altera.

Este FPGA possui 150 pinos de I/O, além de 30K elementos lógicos para o desenvolvimento do circuito, quatro *transceivers* de alta velocidade dedicados e um bloco em hardware para o protocolo PCIe.

Um circuito auxiliar contendo uma memória não volátil está conectado ao FPGA para fornecer um espaço de armazenamento para o *firmware* e a realização da programação do FPGA assim que a placa for ligada.

5.5 Desenvolvimento do Firmware

Para desenvolvimento do *firmware* do projeto foi utilizado o programa Quartus II da Altera, que cobre todas as fases do fluxo de desenvolvimento de um projeto em FPGA, como a realização de testes, etapas de síntese, disposição e roteamento dos elementos lógicos. Como linguagem de descrição de hardware foi utilizada a linguagem Verilog, componentes

disponibilizados pelo fabricante conhecidos como *megafunctions* e ferramenta para geração automática de sistemas SOPC Builder.

Através do editor de blocos do programa, foi possível desenvolver o sistema utilizando blocos que representam a lógica descrita em Verilog, símbolos primitivos, assim como a designação dos pinos de I/O utilizados pelo sistema, como sinais de *Clock*, *Reset*, linhas PCIe e outros.

A Figura 35 mostra uma captura de tela com o sistema de blocos do projeto ECS-CBPF, desenvolvido em ambiente gráfico.

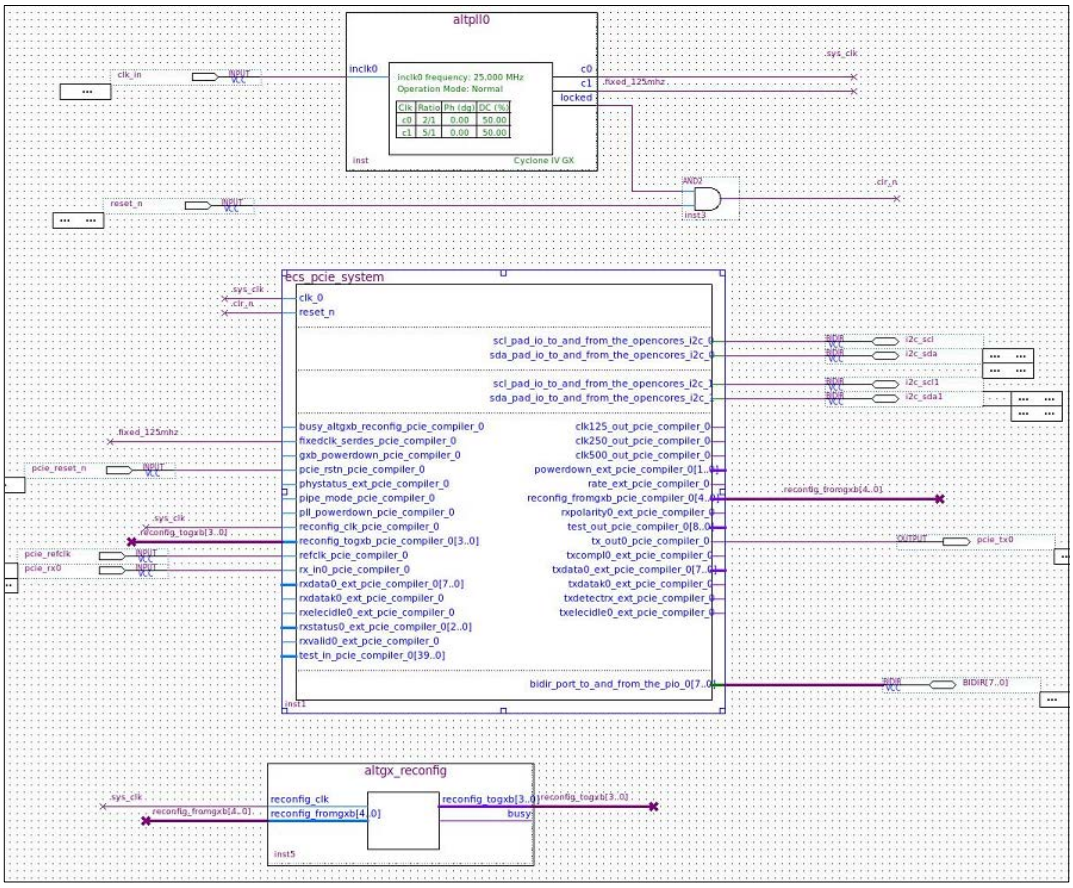


Figura 35 - Tela do projeto ECS-CBPF em ambiente gráfico

5.5.1 O SOPC Builder

O SOPC Builder [43] é uma ferramenta para o desenvolvimento de sistemas em alto nível incorporada ao Quartus II. Optou-se pela utilização desta ferramenta, pois os componentes de hardware são interconectados de maneira automatizada e otimizada para a utilização do bloco de hardware nativo PCIe.

A Figura 36 mostra as interconexões e o ambiente de trabalho SOPC. É possível observar os canais de compartilhamento entre os dispositivos mestre e escravo na arquitetura do SOPC; a interconexão entre o PCI Express (pcie_compiler_0), o controlador I²C (opencores_i2c_0 e 1) e os pinos de I/O genéricos para geração dos sinais JTAG (pio_0).

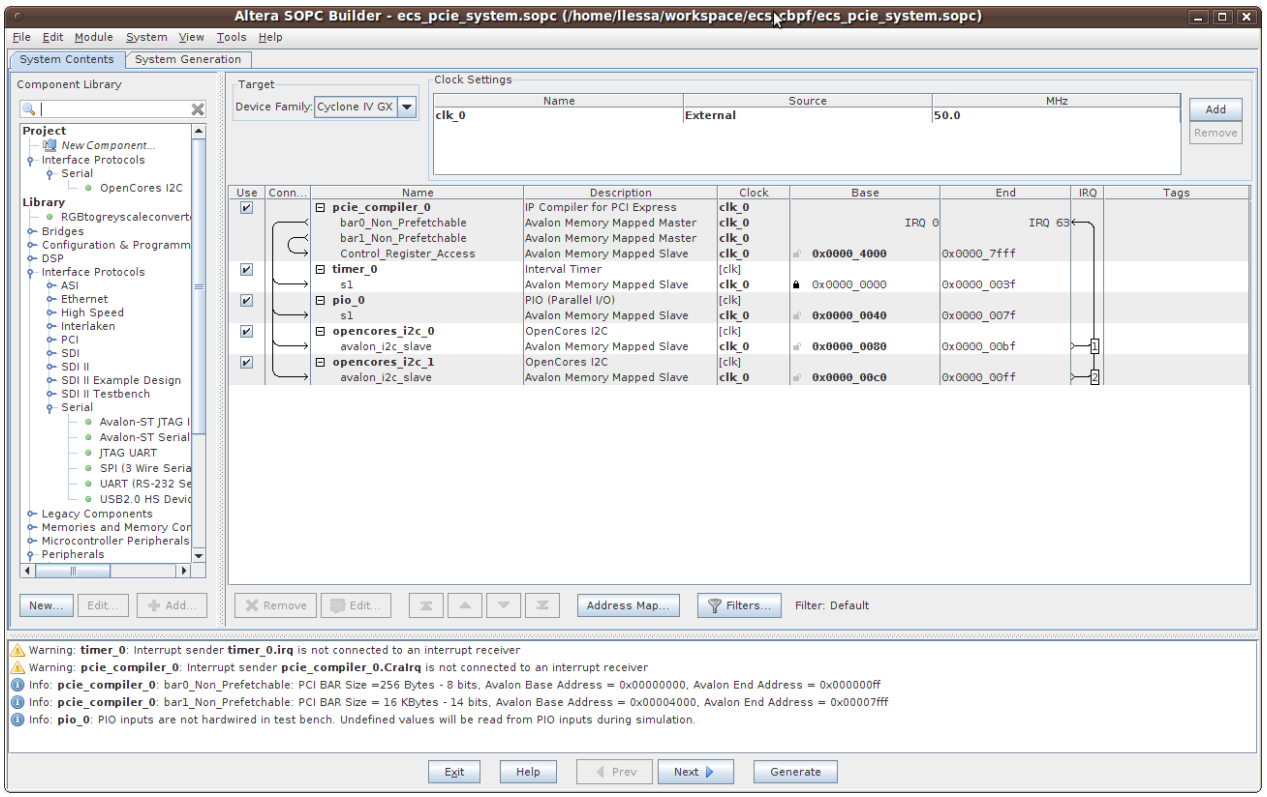


Figura 36 - Tela de integração entre dispositivos no SOPC

Após a geração do código fonte em Verilog pela ferramenta SOPC, foi encontrado um bug na ferramenta relativo à conexão das interrupções dos controladores para o despachador de interrupções do PCIe. Este erro foi corrigido de forma manual no arquivo Verilog seguindo as recomendações da Altera [44].

5.5.1.1 O compilador PCI Express

Com o uso do compilador fornecido pela Altera[45] foi possível configurar parâmetros relacionados como o comportamento do componente na estrutura do sistema, endereçamento dos componentes internos ao FPGA, bem como o comportamento elétrico, seguindo a especificação PCIe.

No caso deste projeto, o PCIe está configurado como *endpoint,* e para fazer acessos ao barramento de interconexões da Altera chamado de Avalon-MM [46] através de

transações do tipo *Requester/Completer*, com esta configuração é possível fazer o acesso através do CCPC via PCIe dos dispositivos escravos no barramento interno ao FPGA, neste caso I²C e JTAG, assim como qualquer controlador mestre acessar a memória do computador.

Outra característica importante é a disponibilidade do acesso direto ao registrador de controle do componente PCIe. Este acesso permite configurar o mapeamento de memórias do tradutor de endereços dinâmicos e identificar, se necessário, onde ocorreu uma interrupção dentro do barramento interno ao FPGA.

5.5.1.2 OpenCores I²C controller core

O controlador I²C utilizado neste projeto foi o controlador desenvolvido pela *Opencores*[47] consiste em um controlador de código aberto implementado utilizando o barramento *Wishbone*[48]. Por esta razão foi necessária à criação de um código de interconexão e compatibilização entre o barramento Avalon da Altera e o *Wishbone* e também a criação do componente no SOPC *Builder*. Este controlador segue as especificações para o protocolo I²C compatíveis com o padrão Philips[49]. Com ele é possível realizar a operação de múltiplos mestres no mesmo barramento, um sistema de temporização programável por *software*, modo de operação por interrupção ou *polling* e endereçamento de dispositivos de 7 ou 10 bits.

5.5.1.3 PIO Core (JTAG)

Para a geração dos sinais JTAG necessários utilizou-se o controlador *PIO Core* do pacote *Embedded Peripherals IP* da Altera [50]. Este controlador está conectado ao barramento Avalon-MM, permitindo a comunicação de sinais nas portas de I/O através do CCPC via PCIe.

Todo o controle dos sinais JTAG é gerado através de máquinas de estado embutidas no *software* que acessa os registradores do PIO Core.

5.6 Desenvolvimento de *Software*

5.6.1 Sistema Operacional

Um sistema operacional é composto por partes responsáveis do uso básico e administrativo de um sistema de computador. Isto inclui um Kernel, drivers de dispositivos, boot loader, linha de comando e outras interfaces com o usuário, e sistema básico de arquivos e utilitários.

O sistema operacional executado pelo CCPC é o Linux, baseado no sistema operacional proprietário Unix[51]. Este sistema operacional foi desenvolvido sobre o modelo de distribuição de software de código livre e aberto onde qualquer desenvolvedor ao redor do mundo pode contribuir com códigos, corrigir falhas ou sugerir modificações. Seu principal componente é o Kernel[52], cujo objetivo é o interfaceamento entre CPU e seus periféricos, suporte em baixo nível a sistemas primários de um sistema operacional, além do suporte as principais ferramentas no ambiente de interação com o usuário, como ferramentas de sistema, bibliotecas de desenvolvimento de software e compiladores.

O Kernel Linux foi originalmente escrito e publicado em 1991, pelo cientista da computação Linus Torvalds, para o uso em sistemas Intel x86. Atualmente o Kernel e as ferramentas necessárias continuam a ser portadas para diversas arquiteturas de CPU como, por exemplo: ARM, PowerPC, SPARC, NIOS II. Este Kernel é utilizado em diversas aplicações, desde computadores pessoais a servidores, como também em sistemas embarcados como televisores, roteadores de rede, telefones celulares.

O conjunto Kernel, contendo ferramentas e aplicativos, é chamado de distribuição. Foi adotado para este projeto a distribuição CERN *Scientific* Linux (CERN SCL) por ser uma distribuição que oferece o suporte e *softwares* necessários dentro dos experimentos do LHC.

Componentes típicos encontrados no Kernel são os tratadores para as requisições do serviço de interrupção, um agendador de tarefas para a divisão do tempo utilizado no processador por múltiplos processos, além de um sistema de gerenciamento de memória para gerenciar o endereçamento dos processos e serviços do sistema, como rede e comunicação entre processos. Nos sistemas operacionais modernos com unidade de gerenciamento de memória protegida, o Kernel reside tipicamente em um estado elevado

do sistema comparado com um processo normal do usuário, isto inclui um espaço de memória protegido e acesso total ao *hardware*. Este estado do sistema é chamado de *Kernel-space*. Os processos de usuário são executados no *user-space*, espaço diferente e separado do *Kernel-space*.

Os aplicativos executados em um sistema se comunicam com o Kernel através de chamadas ao sistema (*system calls*), como mostrado na Figura 37.

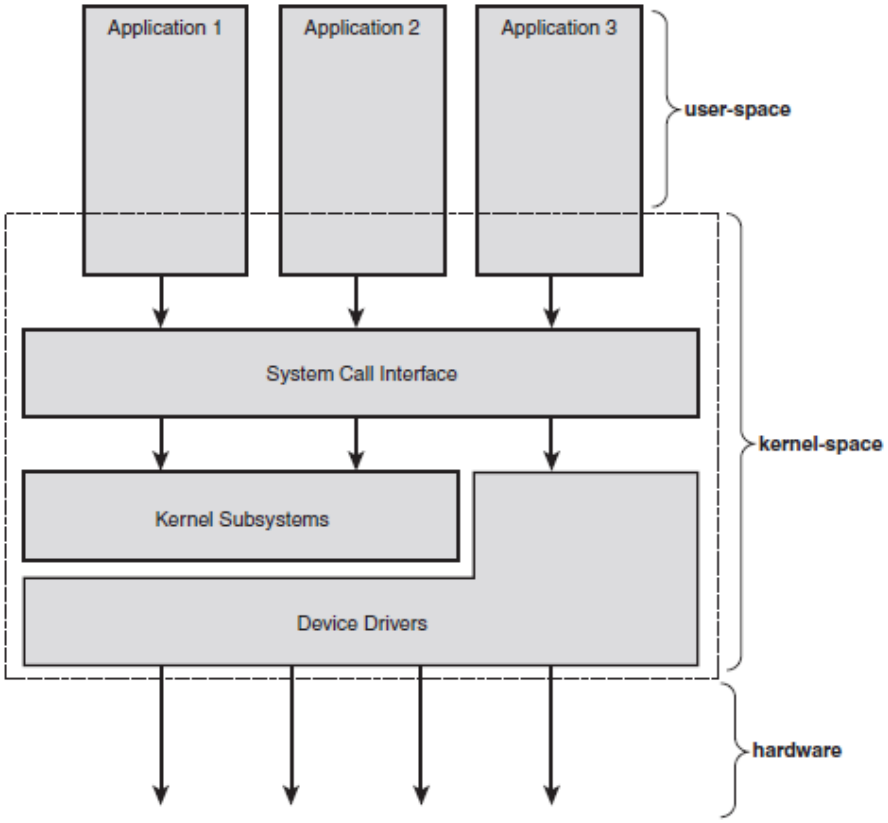


Figura 37 - Relação entre aplicações, Kernel e hardware

O Kernel também gerencia o hardware do sistema, nas arquiteturas em que o Linux é suportado. Quando um hardware necessita se comunicar com o sistema, este emite um sinal que literalmente interrompe o processador e aciona um mecanismo no Kernel para o tratamento e identificação deste sinal. Outra responsabilidade é a alocação de recursos, como o endereçamento dos dispositivos.

Pode-se dividir o Kernel de acordo com o papel desempenhado em cinco módulos, como mostrado na Figura 38.

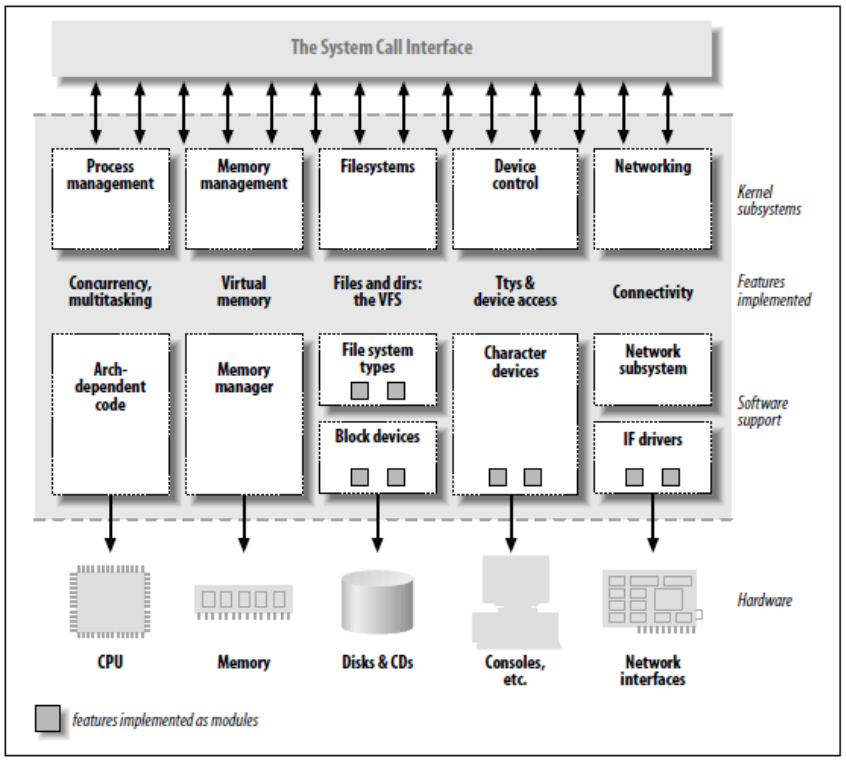


Figura 38 - Diagrama das divisões dos módulos no Kernel

5.6.1.1 Gerenciamento de processos (Process management)

O Kernel é responsável pela criação e destruição dos processos assim como o tratamento da conexão com o mundo exterior (Entrada/Saída). A comunicação entre diferentes processos (através de sinais, *pipes* ou comunicação entre processos primários) é uma funcionalidade básica fundamental do sistema assim como o seu tratamento pelo Kernel. Outra funcionalidade consiste no agendador (*scheduler*), que controla a forma de como os processos são compartilhados pela CPU.

5.6.1.2 Gerenciamento de memória (Memory management)

A memória de um computador e a política de como utiliza-los é crítica para o desempenho do sistema. O Kernel constrói um espaço de endereços virtuais para todos os processos levando em conta a limitação dos recursos disponíveis. Diferentes partes do Kernel interagem com o subsistema de gerenciamento de memória através de um conjunto de chamadas, variando de um comando simples de alocação (*malloc/free*) a um conjunto de funções mais complexas.

5.6.1.3 Sistema de arquivos (Filesystems)

Por herdar características dos sistemas Unix, que utiliza fortemente o conceito de sistema de arquivos, pode-se dizer que quase tudo no Linux pode ser tratado como um arquivo. O Kernel constrói uma estrutura de sistema de arquivos em cima de hardwares que não possuem uma estrutura definida, resultando assim em uma abstração o conceito de arquivo. O Kernel Linux suporta múltiplos tipos de sistemas de arquivos, facilitando a organização dos dados no meio físico.

5.6.1.4 Controle de dispositivos (Device Control)

Quase todas as operações do sistema em algum momento mapeiam ou acessam dispositivos físicos. Com exceção do processador, memória e outros poucos dispositivos, todas as operações de controle do dispositivo são executada por um código que é específico ao dispositivo a ser acessado. Este código é chamado de *device driver*[53], ou simplesmente *driver*. O Kernel deve ter embutido em sua estrutura os drivers específicos para os periféricos presentes em um sistema, do disco rígido ao teclado. Este aspecto do Kernel será explorado neste trabalho.

5.6.1.5 Rede (Networking)

Os dispositivos de rede devem ser gerenciados pelo sistema operacional, pois muitas das operações de rede não são específicas a um processo além de constituírem eventos assíncronos. Os pacotes devem ser coletados, identificados e despachados antes de um processo cuidar deles. O sistema é encarregado pela entrega dos pacotes de dados através dos programas e interfaces de rede, e devem controlar a execução dos programas de acordo com a atividade da rede. Além disto, todos os problemas de roteamento e resolução de endereços são resolvidos pelo Kernel.

5.6.1.6 Classes de dispositivos

A forma com que o Linux enxerga os dispositivos pode ser dividida entre três classes fundamentais de tipos de dispositivo. Cada módulo, em geral, implementa um destes tipos e são classificados em: módulo de caractere (*char module*), módulo de blocos (*block module*), e módulo de rede (*network module*). Esta classificação dos módulos em diferentes tipos não é rígida, podendo-se optar pela programação de um módulo com diferentes *drivers* em um único código. Porém, de modo geral, cria-se um módulo para cada nova funcionalidade

implementada, pois esta decomposição é um elemento chave para a escalabilidade e extensibilidade.

Um dispositivo de caractere (*char device*) é aquele que pode ser acessado como um fluxo de dados (como um arquivo). Este tipo de *driver* deve ter em seu código pelo menos quatro chamadas básicas de sistema: *open, close, read e write*. São exemplos de drivers o console de texto (/dev/console) e as portas seriais (/dev/ttyS0). Estes dispositivos podem ser acessados como um arquivo no sistema de arquivos. A única diferença relevante entre um *char device* e um arquivo regular é que em um arquivo sempre pode se mover o ponteiro de leitura/escrita para frente e para trás enquanto um *char device* é apenas um canal de dados e deve ser acessado sequencialmente. Para dispositivos que dispõem de área de dados, pode se mover os ponteiros como arquivos normais, e esta forma de acesso geralmente se aplicam a memória de um DAQ, onde o programa acessa todos os dados adquiridos usando funções do tipo *mmap* ou *lseek*.

Igualmente aos *char devices*, os dispositivos de bloco (*block devices*) são acessados através do sistema de arquivos no diretório /dev. Um *block device* é um dispositivo que pode hospedar um sistema de arquivos. Em sistemas Unix um *block device* pode tratar somente operações de leitura/escrita de um ou mais blocos, tipicamente constituídos de 512 bytes (ou um número na potência de dois). O Linux permite que as aplicações façam operação de leitura/escrita como se fossem um *char device* com isso permitindo a transferência de qualquer número de bytes. Como resultado, *char* e *block devices* diferem apenas na maneira como os dados são gerenciados internamente no Kernel e na maneira de como o código é escrito, dado que as interfaces são completamente diferentes.

Qualquer transação de rede é efetuada através de uma interface, isto é, um dispositivo capaz de trocar dados com outros sistemas (*hosts*). Em geral, uma interface é um dispositivo de *hardware*. Uma interface de rede é encarregada de enviar e receber pacotes de dados, guiadas através do subsistema de rede do Kernel. Muitas conexões de rede (especialmente as que usam TCP) são orientadas ao fluxo de dados em oposição aos dispositivos de rede, que de modo geral, são modeladas em torno da transmissão e recepção de pacotes. Um driver de rede gerencia conexões individuais, ele apenas trata de pacotes. Não sendo um dispositivo orientado ao fluxo de dados, uma interface de rede não

pode ser facilmente mapeada como um arquivo. A maneira como Kernel resolve isto é associando um nome único (eth0). A comunicação feita entre o Kernel e o driver de rede é completamente diferente da utilizada por um *char* ou *block device*.

Existem outras maneiras de se classificar módulos de driver que são ortogonais aos tipos de dispositivos explicados acima. Geralmente, alguns tipos de drivers funcionam com uma camada de suporte de funções do Kernel para um dado tipo de dispositivo. É importante citar os módulos USB. Cada módulo USB funciona dentro do subsistema USB e eles podem ser vistos no sistema como um *char device* (Porta serial USB), um *block device* (Pendrive ou HD) ou um dispositivo de rede (USB Ethernet ou Wifi).

Outras classes de driver foram adicionadas recentemente, isto inclui drivers para I^2C, SPI, *Fireware*, entre outros.

5.6.2 ECS-CBPF Drivers

Para a comunicação do FPGA contendo os controladores de baixa velocidade com o CCPC através do PCIe foi necessário o desenvolvimento de um driver específico para este fim. O driver foi escrito utilizando a estrutura de um módulo de Kernel, possibilitando assim o carregamento e descarregamento do mesmo com o sistema em execução.

Como pode ser observado na Figura 39, o driver possui duas camadas. A primeira, ECS PCIe Driver é responsável pela alocação de recursos do PCIe, gerenciamento das interrupções e instanciação dos drivers responsáveis pelos controladores de baixa velocidade que estão localizados na próxima camada. São eles o I^2C Ocores, JTAG Port, o *driver* SPI não foi implementado neste trabalho.

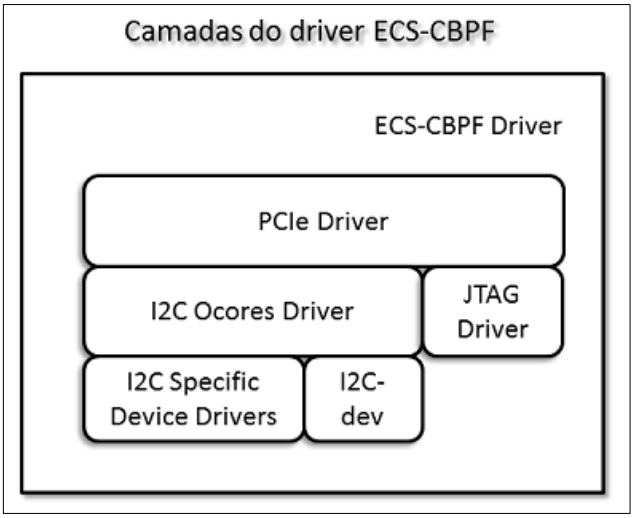


Figura 39 - Camadas do driver ECS-CBPF

5.6.2.1 PCIe ECS driver

O principal driver para o sistema ECS-CBPF é encontrado no arquivo driver/pcie_ecs.c (Apêndice B) , foi desenvolvido como um módulo do Kernel e deve ser carregado após o sistema estar pronto para execução através da ferramenta insmod.

Em sua estrutura duas funções primárias, chamadas de *module entry point,* responsáveis pelo controle da instanciação e remoção do módulo. Estas funções recebem como parâmetro uma função

```
module_init(pcie_ecs_init);
module_exit(pcie_ecs_exit);
```

A função pcie_ecs_init(void), uma vez executada, é responsável pelo registro do driver do dispositivo PCIe com o uso da função de *entry point* para drivers PCI/PCIe pci_register_driver(&pci_driver). É importante ressaltar que dispositivos PCI e PCIe, apesar de não serem compatíveis fisicamente são transparentes no ponto de vista do sistema operacional. O parâmetro recebido pela função é uma estrutura de dados padrão do Kernel e é descrita a seguir.

```
static struct pci_driver pci_driver = {
        .name = DRV_NAME,       // Nome único de identificação
        .id_table = ecs_ids,     // Array contendo VENDOR_ID e DEVICE_ID
        .probe = ecs_probe,      // função de inicialização e alocação dos recursos do PCIe
        .remove = ecs_remove,  // função de remoção do driver e desalocamento dos
                                 //recursos PCIe
};
```

Outra função importante é a ecs_probe(struct pci_dev *dev, const struct pci_device_id *id), responsável pela identificação do dispositivo PCIe contendo através dos identificadores VENDOR_ID e DEVICE_ID. Uma vez identificado, o dispositivo é habilitado através da função pci_enable_device(dev), onde o parâmetro é a estrutura padrão do Kernel, struct pci_dev.

No dispositivo PCIe do sistema ECS-CBPF, os registros dos controladores de baixa velocidade e do dispositivo PCIe podem ser obtidos a partir do registrador de endereço base (BAR – *Base Address Register*) do PCIe. Existem dois BAR, o BAR0 onde encontra-se os registradores dos controladores de baixa velocidade e o BAR1, onde se encontram os registradores de controle do dispositivo PCIe, o CRA (*Control Register Address*)

Os endereços base do PCIe são definidos pelo sistema no momento do boot do CCPC pela BIOS. Este recurso deve ser solicitado pelo driver para uso pelo sistema operacional utilizando o endereçamento virtual do Kernel através da função pci_resource_start(dev, x), onde o parâmetro dev contém a estrutura de dados do dispositivo e x representa o endereço a ser alocado.

Outro recurso importante que deve ser utilizado refere a de interrupção do dispositivo. Esta alocação é feita pela função request_irq(dev->irq, pcie_irq_handler, IRQF_SHARED, DRV_NAME, (void *)ecs_device). O parâmetro pcie_irq_handler é a chamada para a função responsável pelo tratamento das interrupções recebidas pelo sistema operacional. Esta função, quando solicitada, desabilita outras interrupções do dispositivo, identifica qual o controlador que gerou a interrupção, executa a função de tratamento de interrupção externa contida no driver do controlador e reabilita as interrupções no dispositivo.

Com todos os recursos alocados o driver começa a inicialização dos drivers dos controladores de baixa velocidade através das funções, ecs_i2c_init(ecs_device) e ecs_jtag_init(ecs_device), sendo a primeira responsável pela inicialização dos controladores I²C e a segunda pelo controlador JTAG.

Caso ocorra algum erro na alocação dos recursos, inicialização dos controladores ou remoção do driver pela ferramenta rmmod a função ecs_remove(struct pci_dev *dev) é responsável pela finalização do driver.

5.6.2.2 i2c-ocores driver

O Kernel Linux dispõe em sua arquitetura um subsistema para o protocolo I²C [54]. A relação dos componentes necessários para o funcionamento do protocolo no sistema operacional pode ser visto na Figura 40.

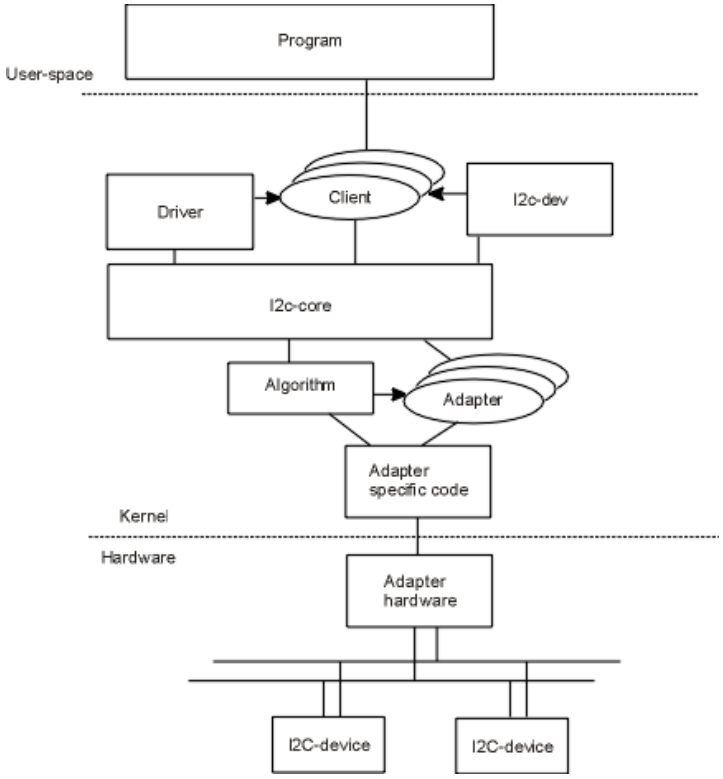


Figura 40 - Camadas do subsistema I²C

A camada superior, *Program*, representa todos os programas no *user-space* que irão acessar os chips, seja pela interface /dev utilizando o driver i2c–dev (a ser descrito posteriormente) ou através da sysfs, utilizando os drivers específicos para cada chip.

Pode-se dividir o os componentes do subsistema I²C em duas categorias, *barramentos* (*buses*) e *dispositivos* (*devices*). A categoria *barramento* é dividida em duas subcategorias, algoritmos (*algorithms*) e adaptadores (*adapters*) enquanto que a categoria *dispositivos* é dividida em driver e cliente (*client*).

Os adaptadores representam efetivamente o barramento e utilizam o algoritmo em sua estrutura de dados. Em geral existe uma combinação de adaptador e algoritmo, específico a um dado controlador, o que não ocorre no caso de operações *bit-banging (*o *software* controla pinos de I/O*)*, para este tipo de operação o algoritmo é comum a todos os adaptadores.

O *bloco cliente* representa um chip I²C no sistema, como por exemplo, uma memória EEPROM, um conversor A/D, um termômetro. Sua estrutura possui alguns parâmetros como o endereço físico no barramento I²C do chip, o adaptador utilizado para comunicação e o driver do chip.

O driver é o responsável pela lógica específica e operação dos dispositivos I²C. Por padrão o Linux inclui em sua árvore diversos *drivers* de chips encontrados comercialmente. Desta maneira, usando um controlador compatível com o subsistema I²C do Linux estes drivers podem ser utilizados no sistema ECS-CBPF.

Um código do controlador I²C OpenCores está disponível na árvore de código fontes do Kernel do Linux [55]. No entanto, foi originalmente desenvolvido para a utilização com o microprocessador embarcado OpenRISC de modo que uma adaptação do *arquivo driver/i2c-ocores.c* (Apêndice B) para a arquitetura do módulo ECS-CBPF foi necessária.

O driver i2c-ocores é inicializado pela função `ecs_i2c_init(struct ecs_device *ecs_device)` do driver pcie_ecs. Nesta função os recursos alocados no barramento PCIe são passados para o driver do adaptador i2c-ocores para registro e carregamento do sistema.

Por ser originalmente escrito para sistemas embarcados, o *driver* i2c-ocores espera que a interrupção seja tratada de forma direta pelo driver do adaptador. Entretanto quem gerência a interrupção no caso deste projeto é o tratador de interrupção do PCIe. A função *ocores_process* foi exportada do código original para o código do driver pcie_ecs para resolver esta questão. Com ela quando uma interrupção vinda do controlador I²C é gerada o

tratador de interrupções é capaz de reconhecer quem a gerou e chamar a função necessária para a continuação do processo.

5.6.2.3 JTAG Port driver

O *driver* JTAG *Port* é responsável pela interface JTAG no ECS-CBPF pode ser encontrado no arquivo driver/ecs_jtag.c (Apêndice B).

Sua inicialização é feita no driver principal do sistema, o PCIe ECS Driver, pela função ecs_jtag_init, com o objetivo de criar uma estrutura de dados do tipo ecs_jtag, carregar o endereço de base para os registradores do controlador JTAG através das instrução abaixo.

```
jtag->start = ecs_device->base_addr_bar0 + PIO0_OFFT;
```

A variável start contém o endereço base dos registradores dos controladores de baixa velocidade acrescentada do offset do endereço do controlador PIO no barramento Avalon (0x0040). Posteriormente o driver principal executa a função exportada do driver JTAG Port, init_jtag(jtag), onde o parâmetro jtag é a estrutura do tipo ecs_jtag.

A função init_jtag cria um arquivo no diretório /dev do sistema operacional, o qual pode ser acessado pelos softwares através das operações de leitura e escrita.

5.6.3 Programas e bibliotecas

5.6.3.1 i2c-dev

Em geral, os dispositivos I²C são controlados por um driver do Kernel. Porém é possível o acesso a todos os dispositivos em um controlador através do *user-space* . Para que a operação no *user-space* (programas e bibliotecas) seja possível, deve-se carregar o módulo padrão do Kernel i2c-dev.

Cada controlador I²C, quando registrado recebe um número único, contado a partir de 0 e contendo um máximo de 256 controladores. Para conhecer o número do controlador correspondente, pode se examinar o diretório /sys/class/i2c-dev/ ou utilizar a ferramenta "i2c-detect -l" para obter uma lista formatada dos controladores disponíveis no sistema. Esta ferramenta esta presente no pacote i2c-tools a ser descrita posteriormente.

Para acessar o controlador por um programa o primeiro passo é incluir no cabeçalho do programa a diretiva #include <Linux/i2c-dev.h>. As descrições completas da interface e API podem ser encontradas na documentação do *Kernel Linux - dev-interface* [56].

5.6.3.2 i2c-tools

O pacote de ferramentas i2c-tools [57], originalmente desenvolvido como parte do projeto lm-sensors responsável por fornecer ferramentas para o monitoramento de parâmetros ambiental no Linux e presente como ferramenta padrão na maior parte das distribuições, contém um conjunto heterogêneo de ferramentas para o uso dos controladores e dispositivos I²C no *user-space*. Para o uso destas ferramentas o módulo do Kernel i2c-dev, deve ser carregado.

As ferramentas contidas no pacote são:

- i2cdetect : programa responsável pela identificação dos controladores e dispositivos anexados em um dado controlador.
- i2cdump : programa responsável pela leitura de todos os registradores de um dispositivo I²C.
- i2cget / i2cset : ferramenta utilizada para comandos de leitura e escrita, respectivamente.

5.6.3.3 Altera Jam STAPL Player Software

O software Altera Jam STAPL [58] permite a programação de chips (CPLD ou FPGA) da Altera utilizando linhas de comando. O software Quartus permite a geração do arquivo de programação STAPL (*Standard Test and Programming Language*). Por utilizar interfaces de programação proprietárias da Altera, o software foi modificado para ser compatível com o controlador JTAG disponível no ECS-CBPF através da interface /dev/jtag .

5.6.3.4 UrJTAG

A ferramenta UrJTAG [59] é um pacote de software de código aberto, que habilita o trabalho com dispositivos de hardware compatíveis com o JTAG. Este pacote possui uma arquitetura modular e aberta com a possibilidade de desenvolvimento de extensões diversas, como, por exemplo, sistemas de testes de placas, programadores de memórias

flash. Esta ferramenta é genérica, pois pode ser utilizada com diversos tipos de dispositivos e fabricantes, além de possuir maior flexibilidade de operações realizadas utilizando o JTAG, este, um fator limitador do software Jam STAPL Player da altera. Por possuir o código aberto e documentação para a inclusão de novas interfaces, foi possível o desenvolvimento do código necessário para o funcionamento desta ferramenta com a interface JTAG do ECS-CBPF.

5.7 Geometria e Interconexões

5.7.1 Geometria

A placa ECS-CBPF possui as dimensões físicas de 150 mm de largura, 225 mm de comprimento e uma altura de 40 mm, incluindo o conector de conexão entre o módulo e a placa hospedeira, o *mezzanino* do CCPC e seu dissipador de calor, além de uma margem para a ventilação da placa. A Figura 41 apresenta o modelo mecânico do módulo, assim como a posição dos furos de fixação mecânica.

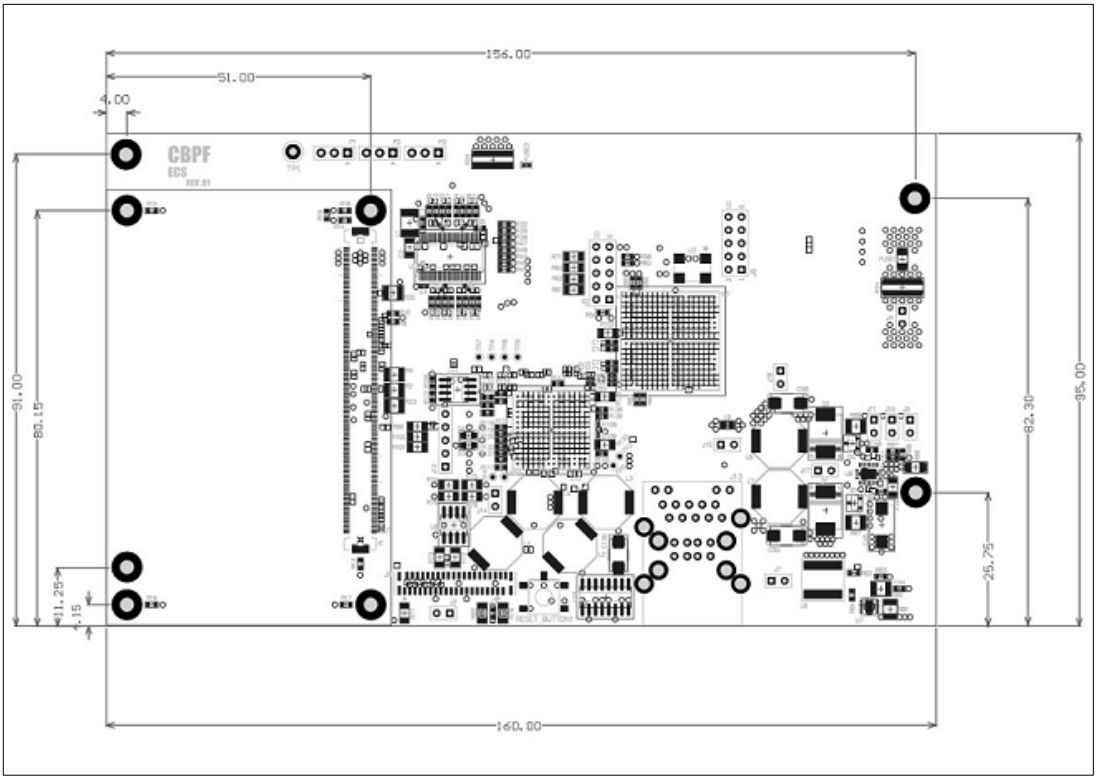


Figura 41 - Esquema mecânico do módulo ECS-CBPF

5.7.2 Conector LVDS (Vídeo)

O conector responsável pela saída de vídeo LVDS na placa ECS-CBPF é o J6 com o uso do kit LVDS-DVI Converter 24bit (P.N: 96007-0000-00-1) da fabricante Kontron o sinal LVDS é transformado em uma saída para monitores DVI.

5.7.3 RJ45 e USB

O conector integrado J13 fornece a entrada para conexão de rede e conector para dispositivos USB, através de um conector RJ45 e dois conectores USB tipo A.

5.7.4 Conector JTAG

A programação do chip de memória que faz o carregamento do FPGA e outro tipo de acesso JTAG através da interface de programação USB Blaster é feita através do conector J5.

5.7.5 Conector Samtec

O conector utilizado para a conexão entre a placa ECS-CBPF e a placa hospedeira é o par do fabricante Samtec, QFS-078-04.25L-D-PC4 (encontrado na parte inferior da placa ECS-CCPF CBPF) e o QMS-078-05.75-L-D-PC4 (para a placa hospedeira). Este conector foi utilizado por fornecer as características elétricas necessárias para a operação com sinais de alta velocidade como é o caso do PCIe, além de possuir uma alta densidade de pinos em uma área reduzida.

A altura relativa entre as placas é de 10 mm. Quando encaixados os conectores e componentes montados na placa hospedeira não devem ultrapassar esta medida. Entretanto não se recomenda a utilização da superfície abaixo da placa ECS-CBPF para a montagem de componentes, pois a dissipação de calor entre as placas pode ocasionar o mau funcionamento de ambas.

6. Placa de validação do módulo ECS-CBPF

Uma placa de validação foi desenvolvida e produzida para realizar os testes funcionais e lógicos necessários do módulo ECS-CBPF. A Figura 42 representa o diagrama de blocos da placa de validação enquanto a Figura 43 mostra a placa após sua fabricação.

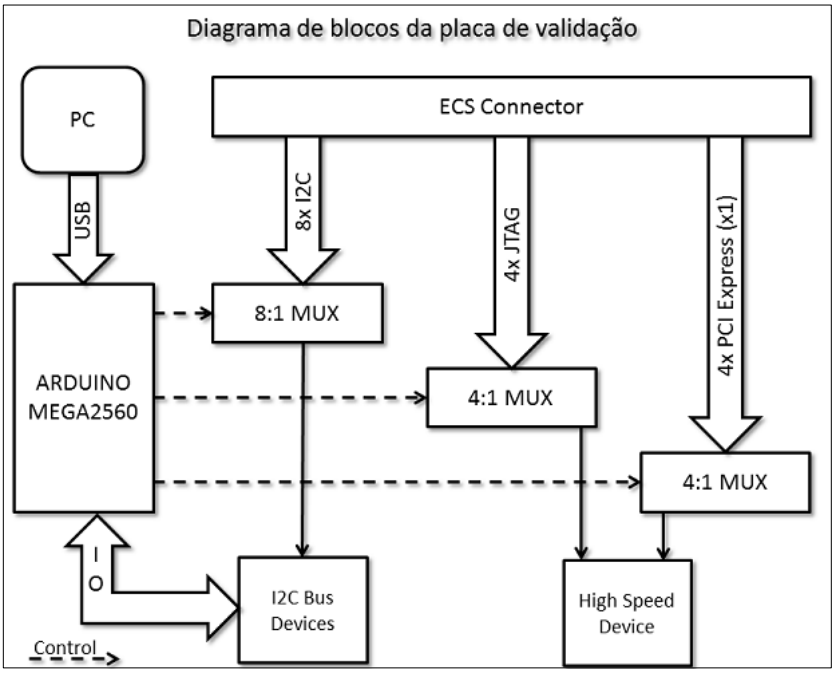


Figura 42 - Diagrama de Blocos placa de validação

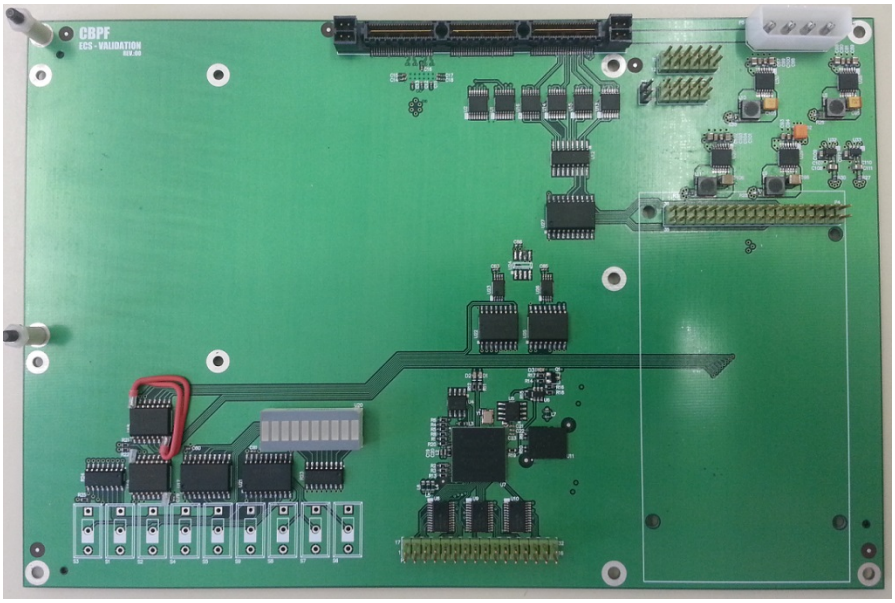


Figura 43 - Foto da placa de validação

A placa de validação possui as seguintes funcionalidades:

- Microcontrolador para controle e validação dos testes e comunicação USB
- Dispositivos I²C
    - Conversor Analógico-Digital para leitura de tensão.
    - Sensor de temperatura.
    - Memória EEPROM (Leitura/Escrita).
    - Interação manual e visual com usuário (Botões e LEDs).
    - Interação automática através do módulo de controle.
- Bloco para teste de programação e comunicação em alta velocidade
    - FPGA conectado ao barramento PCIe e 256 MB de memória DDR, configurado pelo controlador JTAG disponível no ECS-CBPF.

O acesso aos dispositivos do barramento é realizado através de um multiplexador que seleciona um dos oito controladores fornecidos pelo módulo ECS-CBPF disponibilizando 8 linhas I²C, 4 PCIe e 4 JTAG.

6.1 Desenvolvimento do Hardware

6.1.1 Distribuição de alimentação

A alimentação desta placa é feita através de um conector padrão para discos rígidos encontrado em fontes ATX de computador, tem como entrada a tensão de 12V diretamente distribuída para o módulo mezzanino ECS-CBPF e 5V conectado aos reguladores que produzem as tensões necessárias ao circuito, como mostrado na Figura 44.

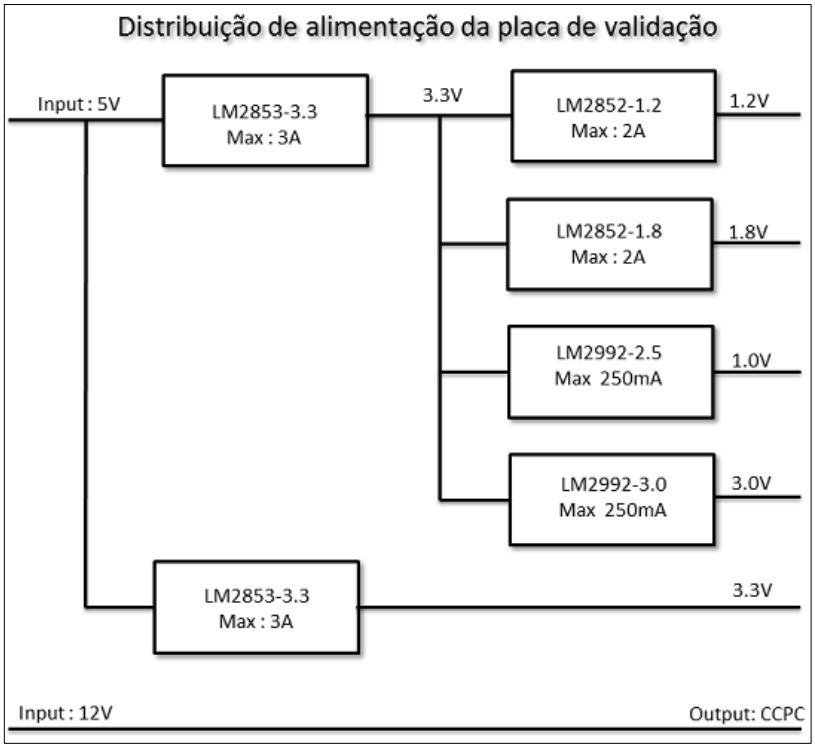


Figura 44 - Distribuição de alimentação da placa de validação

Dois reguladores LM2853-3.3 [60] reduzem a tensão de entrada para 3.3V. Uma deles é utilizado para alimentar outros reguladores que geram as tensões de 1.2V, 1.8V, 2.5V e 3.0 através dos reguladores LM2852-1.2, LM2852-1.8 [61], LP2992-2.5 e LP2992-3.0[62], O outro regulador de 3.3V é usado para alimentar os circuitos e dispositivos auxiliares da placa.

6.1.2 Módulo de controle de teste

O dispositivo responsável pelo controle de teste é formado pela plataforma de prototipagem de código aberto Arduino Mega 2560[63]. Este dispositivo contém um microcontrolador ATMEGA 2560 [64] e seu *software* faz a seleção dos barramentos multiplexados e a comunicação com um computador através do USB, como mostrado no diagrama da Figura 42. O endereçamento dos barramentos multiplexados é feito através do registrador PORT.C em conjunto com um sinal de controle que habilita a escrita no *Latch* do circuito 74LVC373[65]. A Tabela 2 descreve o mapeamento dos bits no registrador.

MSB							LSB
7	6	5	4	3	2	1	0
X	I2C_2	I2C_1	I2C_2	PCIe_1	PCIe_0	JTAG_1	JTAG_0

Tabela 2 - Bits de controle dos multiplexadores de barramento

Para a leitura e escrita de dados utilizados no teste automático de interação com o dispositivo I^2C, utiliza-se o bit 0 do registrador PORT.A.

6.1.3 Dispositivos I^2C

O barramento I^2C presente na placa de validação conta com os seguintes dispositivos: dois conversores A/D de 8 bits PCF8591[66], duas memórias do tipo EEPROM de 8 KB AT24C08B [67], um sensor digital de temperatura LM75[68], dois extensores de E/S de 8 bits PCF8574[69].

Como todos os dispositivos estão no mesmo barramento, os endereços do protocolo I^2C foram configurados de modo que não exista conflito entre. A Tabela 3 mostra os endereços dos dispositivos e suas seguintes funções.

Dispositivo	Função	Endereço
PCF8574	8Bits E/S – Botões e LED	0x20
PCF8574	8Bits E/S – Conectado ao Arduino	0x21
PCF8591	Conversor A/D – 3V3, 3V3, 2V5 e 3V0	0x48
PCF8591	Conversor A/D – 1V8 e 1V2	0x49
LM75	Sensor de temperatura	0x4A
AT24C08B	Memória EEPROM	0x50 e 0x51
AT24C08B	Memória EEPROM	0x52 e 0x53

Tabela 3 - Endereços do barramento I2C

Os conversores A/D são usados para o monitoramento das tensões disponíveis na placa de validação e usam como referência a tensão de 3.3 V, oferecendo uma resolução de 0.0129V por bit.

As memórias EEPROM são utilizadas testes nos de validação da integridade de dados no barramento. Um processo de *software* controlado pelo sistema de controle da placa de validação escreve um bloco de dados conhecido e esses dados são lidos e comparados.

O termômetro I²C é responsável pela medida da temperatura ambiente ao redor da placa de validação.

Os extensores de E/S são utilizados em conjunto com botões para interação manual com usuário e LED para visualização de dados de 8 bits através do barramento I²C, assim como para comunicação de forma automática através do módulo de controle de testes.

6.1.4 Dispositivo para teste de configuração e comunicação em alta velocidade

Para a realização de teste de comunicação de dados em alta velocidade através do PCIe e configuração remota, utiliza-se um circuito composto por um FPGA Altera Cyclone IV EP4CGX30, que possui dois *transceivers* de alta velocidade integrado para comunicação PCIe e memória DDR de 256 Mb.

O firmware de verificação é carregado no FPGA através de um dos quatro controladores JTAG disponíveis na placa ECS CBPF, selecionado na placa de validação através do circuito integrado 74CBTLV3253 [70].

Após a configuração, o sistema está apto a comunicar-se com o CCPC através de uma das quatro linhas PCIe disponíveis na placa ECS-CBPF. Uma das linhas de comunicação PCIe é selecionada na placa de validação através do circuito integrado de chaveamento PCIe PI2PCIE2214[71]. Em seguida o sistema operacional do CCPC reconhece o dispositivo no barramento para o posterior carregamento do *driver* responsável pela execução dos *softwares* de teste de transferência de dados.

6.2 Desenvolvimento do *Firmware*

Para a realização dos testes de medida de taxa de transferência de dados em alta velocidade foi desenvolvido um firmware com um sistema capaz de realizar transferência de dados da memória externa anexada ao FPGA para o CCPC através do barramento PCIe. Esta transferência é feita por meio de um controlador de acesso direto a memória (DMA) possibilitando assim um melhor desempenho na transferência de dados. Uma representação deste sistema é mostrada na figura x.

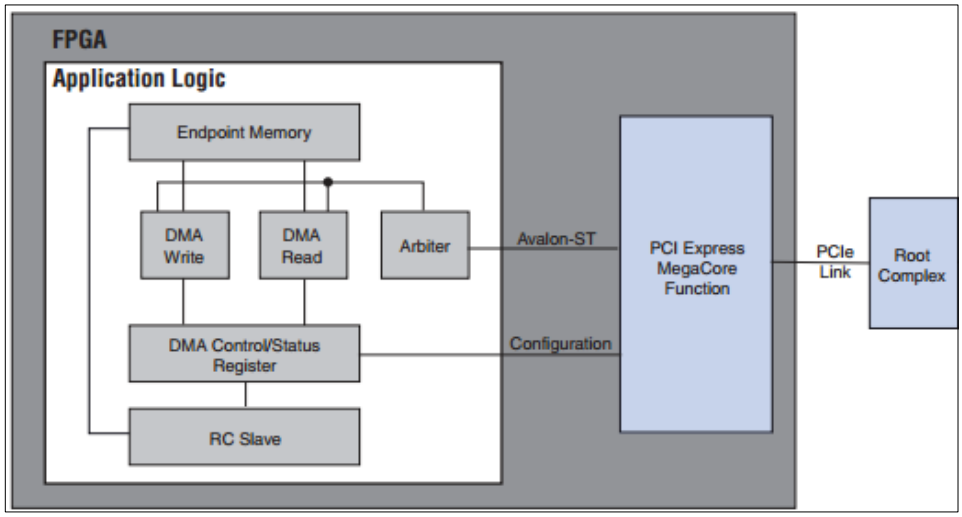


Figura 45 - Diagrama de blocos do firmware da placa de validação

Novamente utilizou-se o SOPC para a geração do sistema. As configurações do *PCI Express compiler* são exatamente as mesmas utilizadas no *firmware* do ECS-CBPF. Os controladores de baixa velocidade foram substituídos por um controlador DMA e uma memória de 4 KB de blocos internos ao FPGA.

O controlador DMA utilizado foi o *DMA Controller Core* [49], do pacote *Embedded Peripherals IP* da Altera. Este controlador é usado para a transferência entre dados de memórias do sistema sem a intervenção da CPU, aliviando a carga no processamento do CPU, como consequência aumentando o desempenho das transferências. Este método aproveita de forma eficiente a utilização do espaço destinado ao transporte de dados efetivos (*payload*) em uma transmissão do pacote do protocolo PCIe.

Os parâmetros utilizados para a configuração do controlador para obter o máximo de desempenho foram:

- Tamanho da transferência de 13 bits.

- Habilitar transmissões em massa com o tamanho máximo de 1024 palavras.
- Profundidade do buffer de FIFO de 256.

O bloco de memória *on-chip* de onde são lidos ou escritos os dados internos ao FPGA através do controlador DMA para a memória RAM do computador foi configurado com tamanho do barramento de 64 bits e 4096 bytes de espaço.

6.3 Desenvolvimento do *driver*

O driver responsável pela transferência de dados em alta velocidade foi desenvolvido de forma similar aos drivers do módulo ECS-CBPF uma vez que a parte responsável pela comunicação PCIe é a mesma. O driver foi escrito para se comportar como um *char device* e por se tratar de um driver com um único propósito, todo o código do driver se encontra no arquivo `pcie_dma.c`. (Apêndice B) Após o carregamento do driver o dispositivo deve ser acessado através do arquivo `/dev/pcie_dma`.

Após o carregamento do módulo, a função de inicialização e reconhecimento do dispositivo PCIe é executada pela função `dma_probe`. Esta função de forma similar a descrita no item do driver `pcie_ecs` do módulo ECS-CBPF; procura o dispositivo PCIe no barramento, aloca os recursos e inicializa o controlador DMA através da função `dma_init`.

A função `dma_init(void)` é responsável pela configuração do controlador DMA através do registrador de controle. Inicialmente o controlador deve ser resetado ao menos duas vezes para que os parâmetros estejam corretamente configurados. O procedimento que se segue consiste na configuração do registrador de controle com os parâmetros de transferências utilizando duas palavras de dados, habilitação do uso de interrupções e a geração de interrupções depois do esvaziamento do contador de número de bytes a serem transferidos.

A transferência dos dados da memória do computador para a memória interna do FPGA é realizada pelas funções `dma_read` e `dma_write`. Estas funções executam a função `dma_transfer (struct ecs_device *dev, int write, void *buffer, size_t count)`, onde o parâmetro `void *buffer` é o espaço de memória alocado pelo driver pela função de inicialização de onde os dados da memória do computador serão lidos ou escritos. Após a

execução da função dma_transfer, inicializa-se o processo de espera de um evento por uma interrupção. Este evento seguirá o fluxo de execução da função de leitura/escrita e transferência dos dados para a área reservada pelo programa que executou as operações no *user-space*.

A função dma_transfer é responsável pela configuração do controlador do DMA com os parâmetros dos endereços de memória de leitura e escrita, direção da transferência (memória do PC para memória do FPGA ou vice e versa), tamanho de dados a serem transferidos e inicialização da transferência.

7. Resultados

7.1 Verificação dos protótipos

Devido à complexidade na confecção do *layout*, fabricação e montagem das placas, três empresas foram contratadas para a execução das fases anteriormente mencionadas. A empresa CEM Engenharia foi responsável pelo desenvolvimento do layout das placas de circuito impresso presentes neste trabalho. A empresa Micropress foi responsável pelo processo de produção das placas de circuito impresso e a empresa Eurotronics, pela montagem dos componentes nas placas de circuito impresso.

No Apêndice C é detalhado o procedimento de verificação do protótipo, cujos erros inviabilizaram a obtenção dos resultados. Esse detalhamento é importante no contexto desta dissertação para evitar futuras falhas.

É importante ressaltar que a processo de verificação dos protótipos, foi realizado em dois meses e que os resultados obtidos foram cruciais para a correção dos erros encontrados, a realização das modificações necessárias no esquemático, no layout e por consequência a produção de um segundo protótipo.

Com a impossibilidade da utilização do FPGA, devido aos problemas encontrados, anteriormente descritos na placa do primeiro protótipo do módulo ECS-CBPF, a verificação do sistema foi realizada com a utilização do kit de desenvolvimento DB4CGX15, um PC com o sistema operacional Linux Ubuntu 9.04, os drivers necessários para a realização dos testes, uma versão de firmware compatível com a quantidade de espaço de elementos lógicos e pinos de I/O disponíveis no FPGA do kit de desenvolvimento e para a obtenção dos resultados, quando possível foram feitos utilizando a placa de validação.

7.2 Verificação do protocolo I2C

O procedimento para a verificação do protocolo I2C foi realizado com a utilização do O kit de desenvolvimento diretamente anexado ao barramento PCIe do PC e a conexão entre

os pinos do controlador I2C (SCL e SDA) no conector P3 para acesso externo aos dispositivos I2C disponíveis na placa de validação.

O primeiro passo realizado foi o carregamento do código de firmware na memória estática do kit de desenvolvimento utilizando a função de programação do aplicativo Quartus II e a interface JTAG/USB. Após a programação completa do código foi necessário realizar a reinicialização do computador para que no momento do boot o dispositivo PCIe fosse reconhecido pelo BIOS do computador e seus recursos reconhecidos pelo sistema.

Com o sistema operacional em execução, o comando lspci –v –d 0x1172:0x0004, foi executado para verificar se o dispositivo PCIe foi realmente reconhecido pelo sistema operacional e, assim, iniciar o carregamento dos drivers pcie_ecs e i2c_ocores através da ferramenta insmod. O resultado deste procedimento pode ser observado através do comando dmesg, como ilustrado na Figura 46.

```
[  454.483496] ECS JTAG Driver Loaded...
[  477.000290] ECS JTAG Driver Unloaded...
[  478.796661] ECS JTAG Driver Loaded...
[  478.802832] I2C ocores driver loaded
[  478.810158] PCIe ECS Driver Loaded...
[  478.810188]  Probing ECS Device
[  478.810207] ecs 0000:02:00.0: PCI INT A -> GSI 18 (level, low) -> IRQ 18
[  478.810211]   Base Address Register(BAR0) @ 0xd8a04000
[  478.810212]   Size of 256 bytes
[  478.810214]   Base Address Register(BAR1) @ 0xd8a00000
[  478.810215]   Size of 16384 bytes
[  478.810224] Succesfully requested IRQ #18 with dev_id 0xf0eb5f40
[  478.810273] i2c_ocores: i2c drv addr: 0xf2c52c00
[  478.810302] i2c_ocores: i2c drv addr: 0xf2c52d54
[  478.810318] pcie_ecs: i2c drv [0] addr: 0xf2c52c00
[  478.810320] pcie_ecs: i2c drv [1] addr: 0xf2c52d54
[  478.810322] pcie_ecs: jtag start base addr: 0xd8a04040
[  478.810323] pcie_ecs: jtag size: 0x40
[  478.810366] pcie_ecs: jtag start base addr: 0xf80f6040
[  527.281779] i2c /dev entries driver
llessa@pcie-ecs:~/sandbox/ecs-jtag/driver$
```

Figura 46 - Procedimento de carregamento dos drivers

Com os drivers carregados e seu funcionamento verificado foi realizado o carregamento do driver i2c-dev, responsável pelo acesso dos controladores I2C por aplicações do usuário. Desta maneira, para verificar qual controlador I²C conectado aos dispositivos na placa de validação, foi utilizado o comando i2cdetect –l e, posteriormente, o comando i2cdetect 1, utilizado para a visualização do endereço dos dispositivos conectados no barramento, como pode ser visto na Figura 47.

```
llessa@pcie-ecs:~/sandbox/ecs-jtag/driver$ sudo i2cdetect -l
i2c-0   smbus           SMBus PIIX4 adapter at fc00             SMBus adapter
i2c-1   i2c             i2c-ocores                              I2C adapter
i2c-2   i2c             i2c-ocores                              I2C adapter
llessa@pcie-ecs:~/sandbox/ecs-jtag/driver$ sudo i2cdetect 1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 21 -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 49 -- -- -- -- -- --
50: 50 51 52 53 -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
llessa@pcie-ecs:~/sandbox/ecs-jtag/driver$
```

Figura 47 - Procedimento de carregamento dos drivers

Os primeiros dispositivos a serem verificados foram os conversores A/D com endereços no barramento 0x48 e 0x49. O driver foi carregado seguindo o quarto método de instanciamento de dispositivos i2c[73], através do comando abaixo. O programa adc_i2c (Apêndice B) foi utilizado para obter a medida, com resolução de 0.0129 V/bit, dos níveis de tensão presente nos reguladores da placa de validação. Como mostrado na Figura 48.

```
echo pcf8591 0x48 > /sys/bus/platform/devices/ocores-i2c.0/i2c-adapter/i2c-1
echo pcf8591 0x49 > /sys/bus/platform/devices/ocores-i2c.0/i2c-adapter/i2c-1
```

```
llessa@pcie-ecs:~/sandbox/ecs-jtag/app$ ./adc_i2c
Regulator [3V3A] = 3.3000 V
Regulator [3V3B] = 3.3000 V
Regulator [3V0 ] = 3.0024 V
Regulator [2V5 ] = 2.5235 V
Regulator [1V8 ] = 1.7859 V
```

Figura 48 - Resultado do nível de tensão nos reguladores

Outro dispositivo utilizado para a verificação do funcionamento do I²C foi o sensor de temperatura DS1821[74]. Este componente foi inserido no barramento através de uma placa de prototipagem, uma vez que o sensor de temperatura LM75 foi removido por apresentar problemas. Seu carregamento foi realizado de forma similar ao componente PCF8591 conforme mostrado abaixo.

```
echo ds1891 0x4a > /sys/bus/platform/devices/ocores-i2c.0/i2c-adapter/i2c-1
```

Uma aquisição de dados com os valores da temperatura do sensor foi obtida através do script app/temp_acq.sh (Apêndice B) durante o período do dia 04/08/2013 as 01:00 ao dia 05/08/2013 as 11:10 com intervalo de 10 s em cada medida. A Figura 49 mostra o comportamento da temperatura em (^{o}C) em função do tempo.

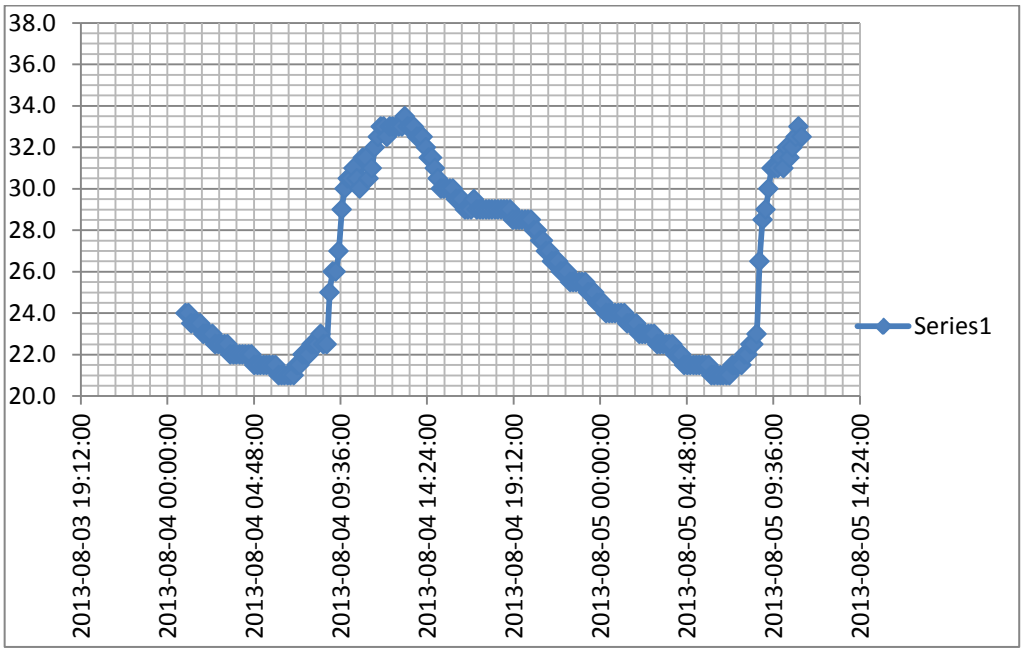


Figura 49 - Gráfico da medida de temperatura ambiente

Com a utilização dos dispositivos I^{2}C pôde se verificar o perfeito funcionamento da integração entre o *hardware* e *drivers* através de aplicações no nível do usuário, além da simplicidade de operação utilizando as ferramentas e *drivers* disponíveis pela comunidade de desenvolvimento do Kernel.

7.3 Verificação da interface JTAG

Os resultados da verificação do funcionamento da interface JTAG, foram realizados com a saída da interface JTAG foi conectada no conector P1 da placa de validação e, posteriormente, conectada aos pinos JTAG da placa de desenvolvimento Arduino.

Depois de efetuado o carregamento dos *drivers* pcie_ecs e ecs_jtag foi utilizada a ferramenta JAM Player como mostrado na Figura 50. O código utilizado para o carregamento do FPGA consiste em um código simples, apenas para verificar o perfeito carregamento da interface JTAG do módulo ECS-CBPF. Este, após sintetizado pelo programa Quartus II, foi convertido em um arquivo do tipo jam, este código consiste em um sinal de *clock*, um circuito *pll* (Phase-Locked Loop) e um divisor de frequência. A saída do divisor de frequência e o sinal *locked* do pll são exportados para os pinos do FPGA. Desta maneira, foi possível observar através de um osciloscópio(Figura 51) sua execução. O diagrama de blocos do código carregado no FPGA através da interface JTAG do módulo ECS pode ser visto na Figura 52.

```
llessa@pcie-ecs:~/sandbox/ecs-jtag/jamplayer$ sudo ./jamplayer ../app/JTAG_Validation.jam -s/dev/jtag -aconfigure -v
Jam STAPL Player Version 2.5 (20040526)
Copyright (C) 1997-2004 Altera Corporation

CRC matched: CRC value = 1F84
Export: key = "JAM_STATEMENT_BUFFER_SIZE", value = 3547
NOTE "CREATOR" = "QUARTUS II JAM COMPOSER 11.1"
NOTE "DATE" = "2013/07/31"
NOTE "DEVICE" = "EP2C8"
NOTE "FILE" = "JTAG_Validation.sof"
NOTE "TARGET" = "1"
NOTE "IDCODE" = "020B20DD"
NOTE "USERCODE" = "FFFFFFFF"
NOTE "CHECKSUM" = "000C68FB"
NOTE "SAVE_DATA" = "DEVICE_DATA"
NOTE "SAVE_DATA_VARIABLES" = "V0, A12, A13, A25, A42, A93, A43, A92, A94, A95, A105, A109, A111"
NOTE "STAPL_VERSION" = "JESD71"
NOTE "JAM_VERSION" = "2.0"
NOTE "ALG_VERSION" = "55"
Device #1 IDCODE is 020B20DD
configuring SRAM device(s)...
DONE
Exit code = 0... Success
Elapsed time = 00:00:02
```

Figura 50 - Processo de carrega utilizando a ferramenta JAM Player

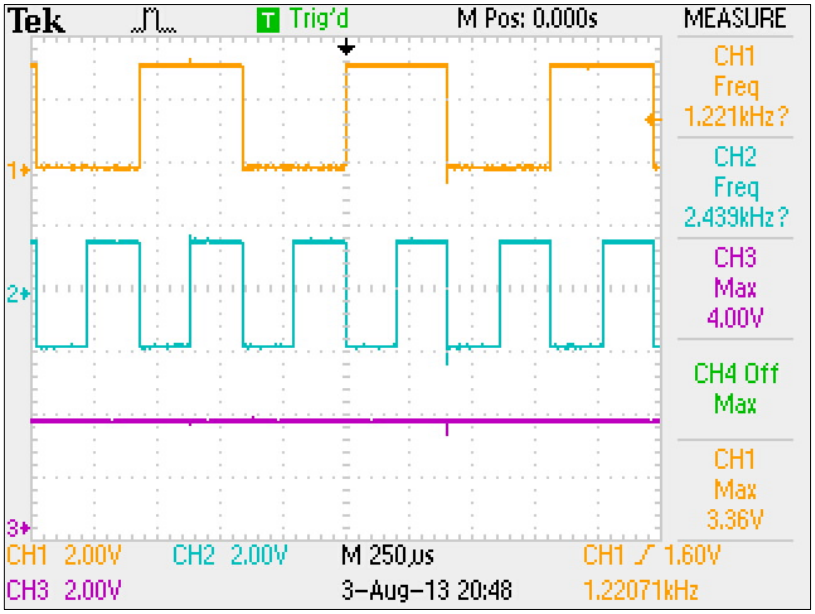


Figura 51 - Captura do código em execução após o carregamento

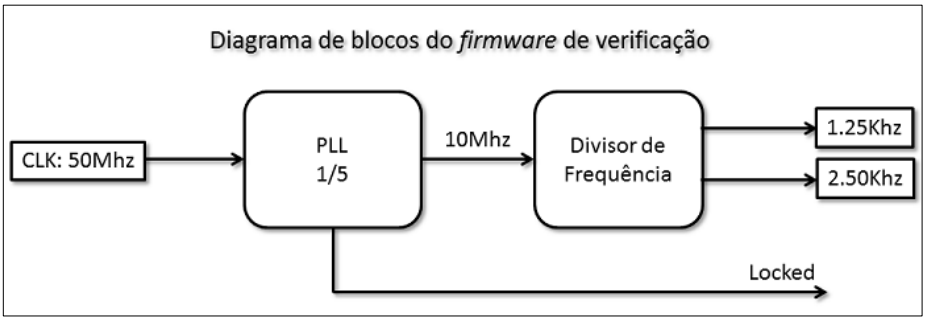


Figura 52 - Diagrama de blocos do firmware de verificação

Outra ferramenta utilizada na verificação do funcionamento da interface JTAG consiste na ferramenta urJTAG. Com ela foi possível realizar a operação de identificação do dispositivo (IDCODE) de dois dispositivos que continham portas JTAG, o FPGA da placa de validação e a placa de desenvolvimento Arduino, como pode ser observado na Figura 53.

```
llessa@pcie-ecs:~/sandbox/ecs-jtag/urjtag-0.10/src$ sudo ./jtag

UrJTAG 0.10 #1502
Copyright (C) 2002, 2003 ETC s.r.o.
Copyright (C) 2007, 2008, 2009 Kolja Waschk and the respective authors

UrJTAG is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
There is absolutely no warranty for UrJTAG.

WARNING: UrJTAG may damage your hardware!
Type "quit" to exit, "help" for help.

jtag> cable ECS ecsdev /dev/jtag
Initializing ppdev port /dev/jtag
jtag> idcode
Reading 0 bytes if idcode
Read 11011101 00100000 10000000 00000010 00000000 00000000 00000000 00000000
jtag> idcode
Reading 0 bytes if idcode
Read 00111111 00010000 10000000 01001001 00000000 00000000 00000000 00000000
jtag>
```

Figura 53 - Utilização da ferramenta urJTAG

Os resultados obtidos com estas verificações mostrou que a interface e os softwares estão operacionais, não somente para FPGAs da Altera, mas também dispositivos de outros fabricantes.

7.4 Verificação de transmissão em alta velocidade PCIe

Os resultados da verificação de transmissão em alta velocidade foram realizados utilizando o kit de desenvolvimento anexado diretamente em um PC. O *firmware* desenvolvido para a placa de validação foi programado através da interface USB/JTAG de forma similar a realizada na verificação do protocolo I^2C.

Após a finalização da programação do kit de desenvolvimento, o PC foi reiniciado e após o boot completo, foi iniciado o procedimento de carregamento do driver pcie_dma. A verificação do carregamento do driver no sistema operacional foi obtida pelo comando `dmesg`.

Um arquivo com dados randômicos de 4096 bytes, tamanho exato da memória interna ao FPGA, foi gerado para servir de referência aos dados transmitidos. O comando padrão do Linux `dd` foi utilizado para a realização das transferências de dados entre a memória do computador e a memória interna do FPGA utilizando o controlador DMA. A Figura 54 mostra as transferências realizadas entre PC x FPGA. Os arquivos transmitidos e

recebidos foram comparados com a utilização da ferramenta diff, foi constatado que os dados foram transmitidos de forma correta.

```
llessa@pcie-ecs:~/sandbox/pcie_DMA_QSYS$ sudo dd if=dummy_4k.dat of=/dev/ecs bs=
4096 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.000186267 s, 22.0 MB/s
llessa@pcie-ecs:~/sandbox/pcie_DMA_QSYS$ sudo dd if=/dev/ecs of=output.dat bs=40
96 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.0002464 s, 16.6 MB/s
llessa@pcie-ecs:~/sandbox/pcie_DMA_QSYS$ diff dummy_4k.dat output.dat
llessa@pcie-ecs:~/sandbox/pcie_DMA_QSYS$
```

Figura 54 - Transferência de dados entre a memória do PC e FPGA

Para evitar que o tempo de inicialização do controlador influencie no calculo do tempo pelo de transferência pelo comando dd. A medida do tempo de transferência foi realizada diretamente pelo *driver*. Esta medida começa a ser efetuada no momento em que controlador recebe o comando de iniciação, isto é, após todas as configurações já terem sido realizadas, e finalizada quando o controlador envia um sinal de interrupção ao sistema.

Através do resultado da medida do tempo de forma indireta pelo *driver* foi possível o cálculo aproximado da velocidade de transmissão pelo barramento PCIe denominado *throughput* do PCIe. A quantidade de dados transmitidas no barramento PCIe teórica, após a camada física é de 250 MB/s. Para uma medida mais precisa seria necessário o uso um osciloscópio com analisador de protocolos.

O *throughput* dos dados é extremamente dependente do tamanho do *payload* e neste caso, limitado a 256 bytes pelo bloco físico PCIe da FPGA Cyclone IV. No cálculo da eficiência de transmissão de um pacote PCIe deve-se levar em conta o *overhead* do pacote, que no caso deste sistema é de 20 bytes pois utiliza-se uma TLP com 3 DWORD (32 bits). O calculo da eficiência pode ser feito a partir da fórmula abaixo.

$$Eficiência\ [\%] = 100\% \times \frac{Tamanho\ do\ Payload}{Tamanho\ do\ Payload \times overhead}$$

A Tabela 5 mostra a eficiência em função do tamanho do *payload*.

Payload	Eficiência [%]	*Throughput* [MB/s]
16	44.44	111.11
32	61.54	153.85
64	76.19	190.48
128	86.49	216.22
256	92.75	231.88
512	96.24	240.60
1024	98.08	245.21
2048	99.03	247.58
4096	99.51	248.79

Tabela 4 - Eficiência e Throughput

Além dos pacotes TLP, são transmitidos pacotes do tipo DLLP e PLP que possuem 8 e 4 bytes, respectivamente, e não possuem nenhuma informação do usuário. Os DLLP são utilizados para a acusação de recebimento de um pacote, controle de fluxo e algumas funções de gerenciamento de energia. Os pacotes do tipo DLLP são originados e terminados na camada de *link* de dados, o PLP na camada física, estes pacotes não são visíveis para a aplicação. Os pacotes do tipo PLP são responsáveis pela sincronização do *clock* e devem ser transmitidos regularmente após a transmissão de 1180 a 1538 bytes.

Os pacotes TLP só são transmitidos se o receptor possuir espaço suficiente no seu *buffer* de recebimento. Por isto pacotes DLLP de controle de fluxo são transmitidos em ambas às direções para informar o transmissor e receptor do espaço disponível em cada um.

A partir destas informações foi possível calcular o valor de *throughput* para a operação de escrita, levando-se em conta dois pacotes do tipo DLLP por pacote TLP transmitido e um pacote PLP a cada 1180 bytes transmitido, o valor real aproximado para um *payload* de 256 bytes, em uma transmissão de 4096 bytes é de 230,90 MB/s.

Para a operação de leitura um pacote TLP vazio deve ser enviado na direção de escrita. Após o recebimento, um pacote TLP de resposta é retornado com os dados solicitados. Em seguida o processo de transmissão se inicia com dois pacotes DLLP enviados a cada TLP transmitido, um pacote TLP a cada pacote recebido (na direção de escrita), um

pacote PLP a cada 1180 bytes. Essas operações de leitura são implementadas com um tempo fixo de 500 ns. O valor real aproximado para um *payload* de 256 bytes e uma transmissão de 4096 bytes é de 149.18 MB/s.

O cálculo da velocidade de transmissão dos dados na direção de escrita e leitura, foram obtidos a partir da média de 10.000 medidas do tempo, para diferentes quantidades de dados transmitidos. Foram utilizado os valores de 4096, 2048, 1024 bytes. As distribuições obtidas foram ajustadas por distribuições gaussianas onde o valor central corresponde ao tempo de transmissão. E o erro associado foi obtido a partir da largura da gaussiana correspondente. Os valores das médias do tempo de transmissão e erros associados obtidos para as operações de escrita e leitura encontram-se nas Tabelas 5 e 6, respectivamente.

bytes	t(0) (us)	Erro t(0) (us)
1024	24.633 +- 0.016	1.351 +- 0.017
2046	31.108 +- 0.039	3.285 +- 0.025
4098	40.120 +- 0.013	0.850 +- 0.016

Tabela 5 - Operação de escrita

Bytes	t(0) (us)	Erro t(0) (us)
1024	21.662 +- 0.015	0.5422 +- 0.0093
2046	27.439 +- 0.017	0.628 +- 0.011
4098	42.540 +- 0.011	0.858 +- 0.011

Tabela 6 - Operação de leitura

A partir dos pontos obtidos foi possível construir um gráfico do numero de bytes transmitidos, para operação de escrita e leitura em função do tempo. Mostrados respectivamente nas Figuras 55 e 56

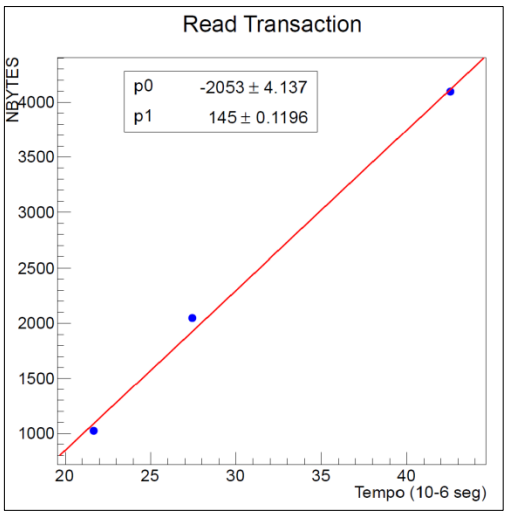


Figura 55 - Transação de Leitura

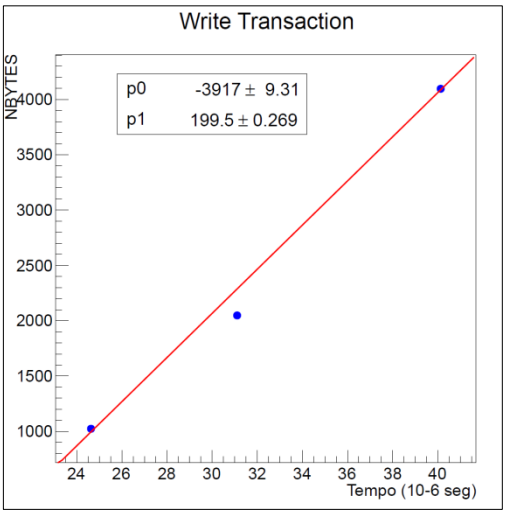


Figura 56 - Transação de Escrita

Para a operação de escrita foi observado o valor de 199.5 ± 0.3 MB/s e para a operação de leitura o valor de 145.0 ± 0.11 MB/s, estes valores indicam que a transmissão esta sendo realizada próximo aos valores teóricos levando-se em conta o cálculo dos pacotes presentes em transações de leitura. O valor obtido para a escrita está abaixo do esperado para um *payload* de 256 bytes, possivelmente por alguma limitação interna ao FPGA, como latência para acesso a memória, quantidade de buffer disponível para o sistema, o tamanho da FIFO do controlador DMA ou mesmo a influencia do Sistema Operacional na medida do tempo. Os resultados encontrados fornecem um aumento considerável do desempenho se comparados com a primeira versão do driver desenvolvida [75], em torno de 5 MB/s.

8. Conclusão e perspectivas futuras

Este trabalho se refere ao desenvolvimento de um módulo de controle, ECS-CBPF, tendo em vista as modificações previstas para o upgrade do experimento LHCb, em particular para a placa de aquisição TELL40. Apesar de ser projetado para uma determinada aplicação, o desenho do módulo se mostrou flexível por conter uma unidade de lógica programável e uma quantidade suficiente de portas de entrada e saída para a utilização em outras aplicações. O projeto pode ser definido como um pacote que envolve *hardware*, tanto do módulo como da respectiva unidade de verificação, como *firmware* e *software*.

De acordo com os requerimentos necessários ao upgrade do LHCb o módulo deve conter um computador de bordo CCPC moderno, protocolo Gigabit Ethernet para comunicação com a sala de controle, e fornecer protocolos de alta velocidade PCIe e de baixa velocidade I2C, JTAG e SPI, para controle, monitoramento e configuração de dispositivos na placa de aquisição.

O desenvolvimento do *firmware* e dos *drivers* foi inicialmente realizado utilizando um kit de desenvolvimento contendo um FPGA e o barramento de comunicação PCIe. Desta maneira foi possível estimar o uso de recursos necessários para o desenvolvimento do *hardware* do módulo e verificar o funcionamento dos elementos de *software*. Para produção do módulo foram necessárias quatro etapas, são elas, o desenho do esquemático elétrico, a produção do layout do circuito impresso, a fabricação do circuito impresso e aquisição dos componentes e por fim a montagem dos componentes no circuito impresso.

Para a verificação e validação do funcionamento do módulo tanto na etapa de verificação inicial do funcionamento do protótipo quanto na etapa de produção em série. Foi desenvolvida uma placa de validação que oferece os recursos necessários para a verificação do funcionamento e extração de resultados dos elementos disponíveis no módulo ECS-CBPF, esta verificação pode ser feita de forma automática através de um *software* de controle via USB.

No decorrer do deste projeto houveram algumas mudanças em relação ao desenvolvimento do upgrade da placa de aquisição de dados para o experimento LHCb. Os

requisitos inicialmente utilizados para o desenvolvimento das características físicas e funcionais do módulo ECS-CBPF tinham como objetivo se enquadrar com as especificações do upgrade do sistema atual (TELL1), projeto este, denominado TELL10 e desenvolvido pela EPFL, em Lausanne. Esta placa tinha como filosofia a utilização de *mezzaninos* e a utilização de *crates* com espaço suficiente para acomodar o módulo ECS-CBPF. No período que o módulo ECS-CBPF e placa de validação encontravam-se em fase de produção, a Colaboração optou pela utilização de outro projeto que vinha sendo desenvolvido em paralelo ao TELL10 denominado TELL40 e desenvolvido pelo CPPM, em Marseille. Por utilizar *crates* de nova geração e com limitação de espaço, após algumas reuniões com o grupo de Marseille, optou-se utilizar como referência o desenho do módulo ECS-CBPF e implementar um sistema similar diretamente na placa de aquisição.

A TELL40 utiliza um modelo da mesma família do FPGA especificado para o módulo ECS-CBPF e seus requisitos funcionais são parecidos, como o controle através de um CCPC e Gigabit Ethernet, comunicação através do protocolo PCIe e a utilização de barramento de baixa velocidade como I^2C e SPI para o controle e monitoramento de dispositivos na placa. Neste ponto a parte de software (*firmware* e *drivers*) do módulo ECS-CBPF encontrava-se em uma fase mais avançada e totalmente operacional, no que diz respeito ao protocolo I^2C. Foi dado ao grupo de Marseille o suporte necessário para a operação do CCPC e a integração do *firmware* e *drivers* desenvolvidos pelo grupo do CBPF através de uma visita técnica ao laboratório do grupo do CPPM.

Os problemas encontrados no primeiro protótipo, principalmente o relacionado à fonte de tensão que ocasionou o comprometimento de componentes essenciais, como o FPGA e o Switch PCIe. Inviabilizou a utilização do protótipo para a obtenção dos resultados deste trabalho. Os resultados foram obtidos utilizando o kit de desenvolvimento com um *firmware* que possui recursos limitados em relação à quantidade de controladores, mas completo em relação às funcionalidades fornecidas se comparado à quantidade de controladores necessários para o sistema final ECS-CBPF. Os problemas encontrados no primeiro protótipo e suas correções foram de extrema importância para a produção de um segundo protótipo. Atualmente este protótipo encontra-se produzido e seu teste deve ser iniciado durante o mês de agosto de 2013.

Em relação aos resultados obtidos a partir do kit de desenvolvimento, foi possível observar o perfeito funcionamento de forma robusta do controlador e drivers I^2C implementados nesse trabalho. Assim como a integração total com o sistema operacional, facilidade de operação, utilização de *drivers* já existentes e transparência de questões técnicas relativas ao funcionamento do controlador e protocolo I^2C no desenvolvimento dos aplicativos pelo usuário final. Foi possível carregar um código em um FPGA bem como a utilização de outros tipos de dispositivos que contenham a interface JTAG, através de linhas de comandos. Esta operação no contexto do sistema final será realizada de forma remota e o PC substituído pelo CCPC.

Os resultados de desempenho em transferências de dados em alta velocidade utilizando o protocolo PCIe, podem ser considerados ótimos devido a limitação da quantidade de dados que pode ser transmitida utilizando a família de FPGA utilizada para a verificação. As velocidades obtidas foram de 199.5 MB/s na operação de escrita e 145 MB/s na operação de leitura. Estas taxas indicam uma significativa melhora no aumento do desempenho, se comparado ao primeiro *driver* escrito [75] para uma comunicação PCIe cujo o valor máximo obtido foi na ordem de 5 MB/s.

Alguns pontos não foram explorados neste trabalho, como o desenvolvimento do *driver* e inclusão do controlador SPI no firmware, o desenvolvimento da aplicação de controle e software necessários para o controle da placa de validação, resolução dos eventuais problemas do segundo protótipo.

O desenvolvimento deste trabalho levou a publicação de uma nota técnica interna no CERN com a identificação LHCb-INT-2013-046 referente ao *hardware*. Em um futuro próximo serão publicadas outras duas notas técnicas abordando a parte de *software*/*drivers* e desempenho do sistema final.

Referências Bibliográficas

[1] L. Evans, P.Bryant, LHC machine. JINST, 3:S08001, 2008

[2] LHC Design Report. Disponível em: <http://lhc.web.cern.ch/lhc/LHC-DesignReport.html>

[3] K. Aamodt et al. (The ALICE collaboration), The ALICE Experiment at the LHC, JINST 3, S08002 (2008) <http://alice.cern.ch>

[4] G. Aad et al. (The ATLAS collaboration), The ATLAS Experiment at the LHC,JINST 3, S08003 (2008)<http://atlas.ch>

[5] R. Adolphi et al. (The CMS collaboration), The CMS Experiment at the LHC,JINST 3, S08004 (2008) <http://cms.cern.ch>

[6] The TOTEM experiment <http://totem-experiment.web.cern.ch/totem-experiment>. Acesso em:03 mar. 2013

[7] O.Adriani et al. LHCb Collaboration, The LHCf detector at the CERN Large Hadron Collider, JINST 3, S08006, 2008.

[8] A.A. Alves Jr. et al (The LHCb Collaboration), The LHCb Detector at the LHC, JINST 3, S08005 (2008) <http://lhcb.cern.ch/>. Acesso em: 03 mar. 2013

[9] The LHCb Collaboration, LHCb Magnet: Technical Design Report, CERNLHCC-2000-007 (2000)

[10] The LHCb Collaboration, LHCb VELO: Technical Design Report, CERNLHCC-2001-011 (2001)

[11] LHCb Collaboration. LHCb Technical Design Report: Reoptimized detector design and performance. CERN-LHCC 2003-030.

[12] The LHCb Collaboration, LHCb Inner Tracker: Technical Design Report, CERN-LHCC-2002-029 (2002)

[13] The LHCb Collaboration, LHCb Outer Tracker: Technical Design Report, CERN-LHCC-2002-029 (2001)

[14] The LHCb Collaboration, LHCb RICH: Technical Design Report, CERNLHCC-2000-037 (2000)

[15] The LHCb Collaboration, LHCb Calorimeters: Technical Design Report, CERN-LHCC-2000-036 (2000)

[16] The LHCb Collaboration, LHCb Muon System: Technical Design Report,CERN-LHCC-2001-010 (2001)

[17] The LHCb Collaboration, LHCb Trigger System: Technical Design Report,CERN-LHCC-2003-031 (2003)

[18] The LHCb Collaboration, LHCb Online System : Technical Design Report,CERN-LHCC-2001-040 (2001)

[19] The LHCb Collaboration, Addendum to the LHCb Online System : Technical Design Report,CERN-LHCC-2005-039 (2005)

[20] ANTUNES. R.N et al 2005, LHCb Computing Technical Design Report CERN-LHCC-2005-019

[21] The LHCb Collaboration, Letter of Intent for the LHCb Upgrade, CERN-LHCC-2011-001 (2011)

[22] Wyllie K et al, LHCb Technical Note: Electronics Architecture of the LHCb Upgrade, LHCb-PUB-2011-011 (2011)

[23] MOREIRA, P. et al, GBTx Specification V1.5 Draft, CMS Document 3857-v3 (2011)

[24] Alessandro Gabrielli et al, GBT-SCA The Slow Control Adapter for the GBT System REV4.3 (2011)

[25] J-P. Cachemiche et al, Readout board specifications for the LHCb upgrade, EDMS 1251709 (2012)

[26] PICMG 3.0 Revision 2.0 AdvancedTCA Base Specification <http://www.picmg.org>

[27] KONTRON COMe-mTTi10 <http://emea.kontron.com/products/computeronmodules/com+express/com+express+mini/comemtt10.html>

[28] PLX PEX8609 <http://www.plxtech.com/products/expresslane/pex8609>

[29] BUDRUK, R.; ANDERSON, D.; SHANLEY, T. PCI Express System Architecture. 2nd. ed. [S.l.]: MindShare, Inc, 2003.

[30] SHANLEY, T.; ANDERSON, D. PCI System Architecture. 4rd. ed. [S.l.]: MindShare, Inc, 1999.

[31] PCI-SIG. PCIe Base Spec 1.1 <http://www.pcisig.com/specifications/pciexpress/> Acesso em 18 mar. 2012

[32] IEEE, Inc. IEEE Std 1149.1 - Standard Test Access Port and Boundary Scan Architecture. [S.l.]: IEEE, Inc.

[33] IEEE, Inc. IEEE Std 1532 - Standard for In-System Configuration of Programmable Devices. [S.l.]: IEEE, Inc.

[34] DEVBOARDS. <http://www.devboards.de/en/home/boards/product-details/article/db4cgx15/>

[35] ALTERA Corporation. Quartus II Handbook Version 10.0. [S.l.], 2010.

[36] ALTIUM Ltd. Altium Designer. <http://www.altium.com>

[37] LINEAR Tecnology. LTM4601 Datasheet. <http://www.linear.com>

[38] LINEAR Tecnology. LTC3568 Datasheet. <http://www.linear.com>

[39] TEXAS Instrument. TPS95003 Datasheet. <http://www.ti.com>

[40] PERICOM. PI6C20800 Datasheet. <http://www.pericom.com>

[41] PICMG COM.0 Revision 2.0 COM Express. <http://www.picmg.org>

[42] ALTERA Corporation. Cyclone IV Device Handbook. [S.l.], 2011.

[43] ALTERA Corporation. SOPC Builder - User Guide. [S.l.], 2010.

[44] ALTERA Corporation. Solution ID: rd08032010_560 <http://www.altera.com/support/kdb/solutions/rd08032010_560.html>

[45] ALTERA Corporation. PCI Express Compiler - User Guide. [S.l.], 2011.

[46] ALTERA Corporation. Avalon Interface Specifications. [S.l.] 2011.

[47] OPENCORES. I2C controller core <http://opencores.org/project,i2c>

[48] OPENCORES. Wishbone Specification [S.l.], 2010.

[49] PHILIPS Semiconductors. The I 2C-bus specification - version 2.1. [S.l.]: Philips, 2000.

[50] ALTERA Corporation. Embedded Peripherals IP User Guide. [S.l], 2011.

[51] BACH, Maurice J. The design of the Unix operating system. New Jersey: Prentice Hall, 1990.

[52] BOVET, Daniel P.; CESATI, Marco. Understanding the Linux kernel. Sebastopol: O'Reilly, 2005

[53] LDD3

[54] Linux I2C Subsystem <https://i2c.wiki.kernel.org/index.php/Main_Page>

[55] Linux Kernel <http://www.kernel.org>

[56] Dev-interface - The Linux Kernel Archives <https://www.kernel.org/doc/Documentation/i2c/dev-interface>

[57] I2C Tools for Linux <http://www.lm-sensors.org/wiki/I2CTools>

[58] Altera Jam STAPL <https://www.altera.com/download/programming/jam/jam-index.jsp>

[59] UrJTAG - Universal JTAG library, server and tools <http://www.urjtag.org>

[60] TEXAS Instruments. LM2853 Datasheet <http://www.ti.com>

[61] TEXAS Instruments. LM2852 Datasheet <http://www.ti.com>

[62] TEXAS Instruments. LP2992 Datasheet <http://www.ti.com>

[63] ARDUINO. <http://www.arduino.cc>

[64] ATMEL Corporation. ATMEGA 2560 Datasheet <http://www.atmel.com>

[65] NXP Semiconductors. 74LVC373 Datasheet <http://www.nxp.com>

[66] NXP Semiconductors. PCF8591 Datasheet <http://www.nxp.com>

[67] ATMEL Corporation. AT24C08B Datasheet <http://www.atmel.com>

[68] TEXAS Instruments. LM75 Datasheet <http://www.ti.com>

[69] NXP Semiconductors. PCF8574 Datasheet <http://www.nxp.com>

[70] NXP Semiconductors. 74CBTLV3253 Datasheet <http://www.nxp.com>

[71] PERICOM. PI2PCIE2214 Datasheet <http://www.pericom.com>

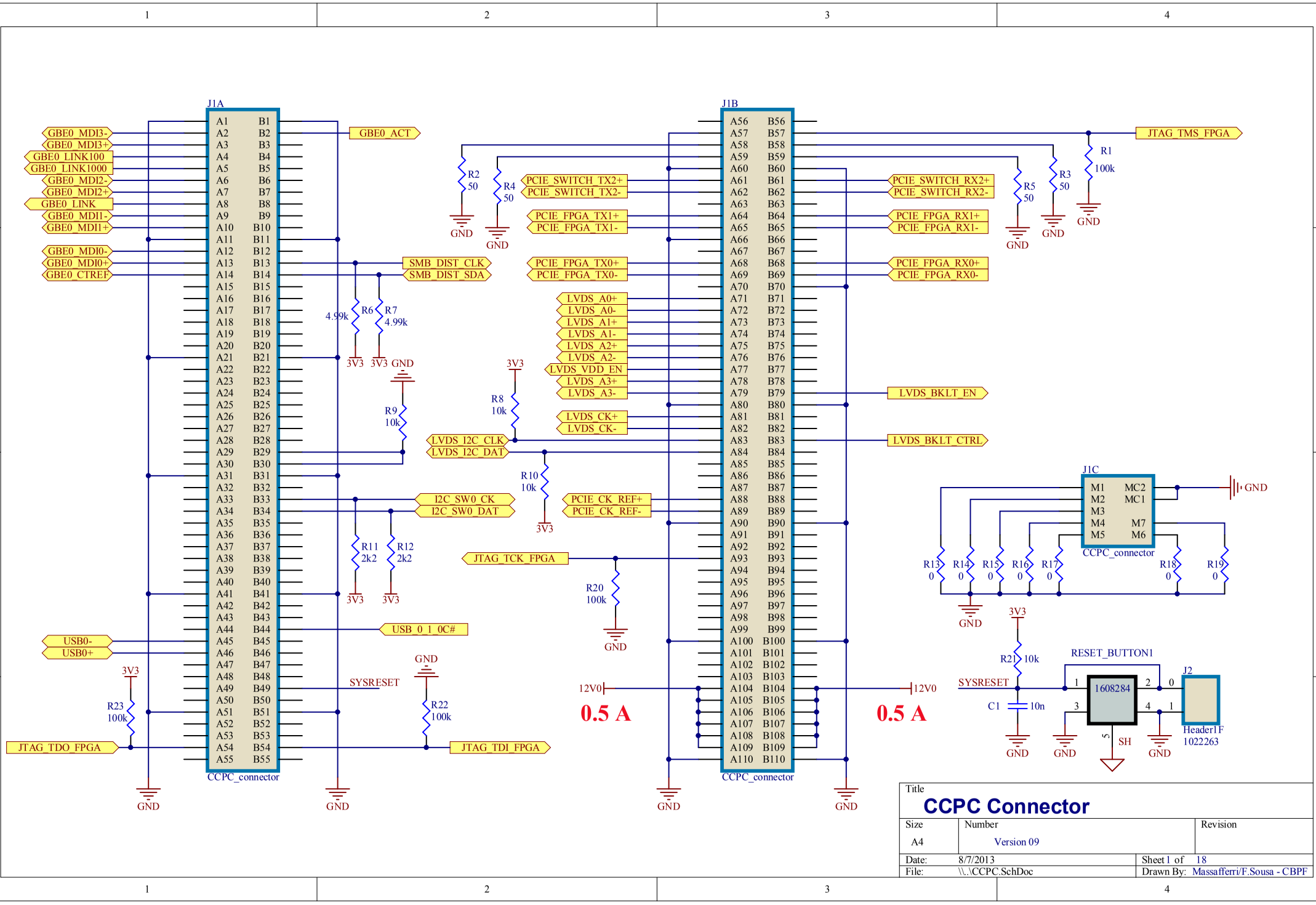
[72] HOROWITZ, P.; HILL, W. The Art of electronics. 2. ed. [S.l.]: Cambridge University Press, 1989.

[73] Instantiating-Devices - The Linux Kernel Archives <https://www.kernel.org/doc/Documentation/i2c/instantiating-devices>

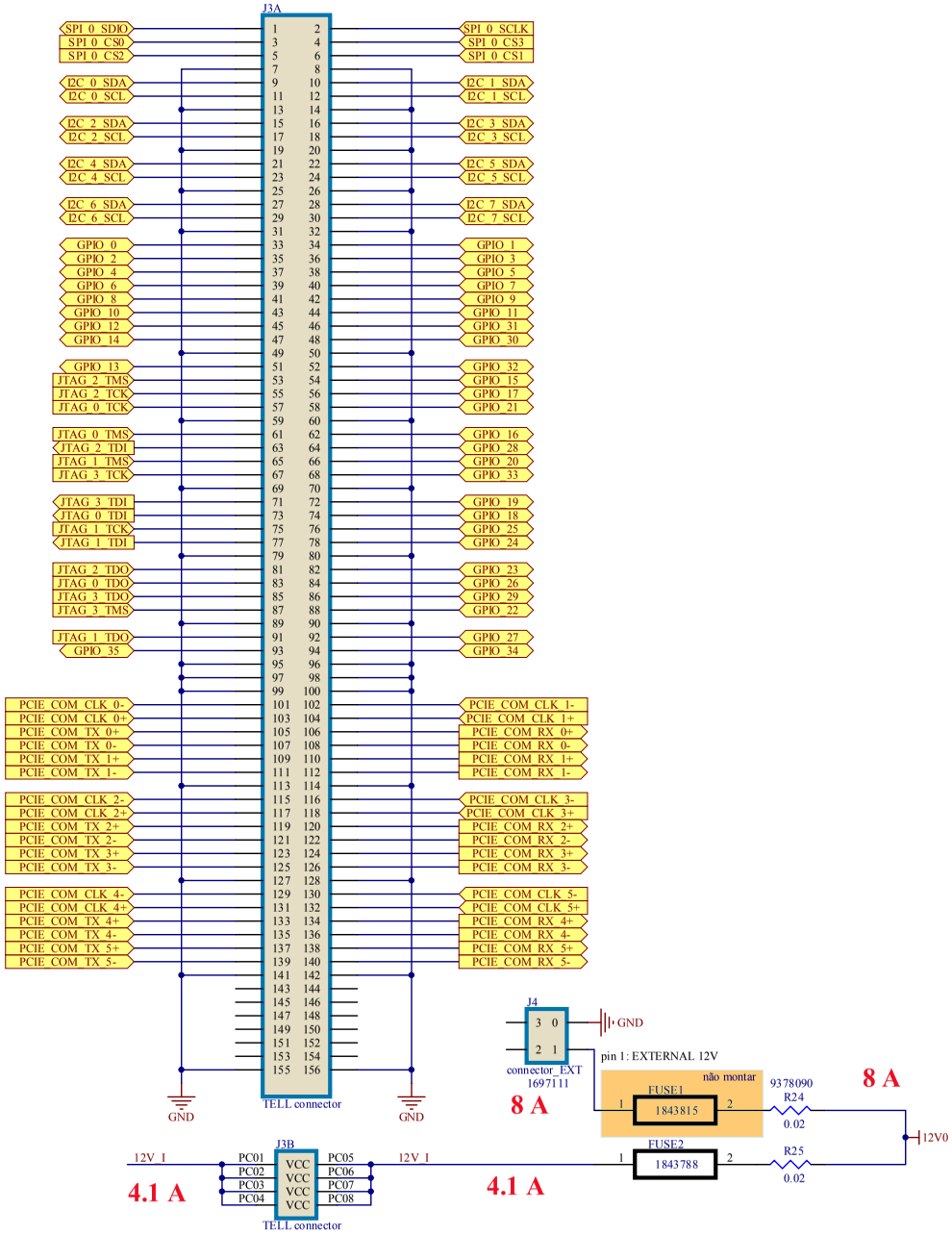
[74] MAXIM Semiconductors. DS1821 Datasheet < www.maximintegrated.com>

[75] LESSA, L.H.P. Desenvolvimento de um barramento PCI Express para a comunicação entre um processador e uma FPGA. Programa de Capacitação Institucional do MCT/CBPF, 2010.

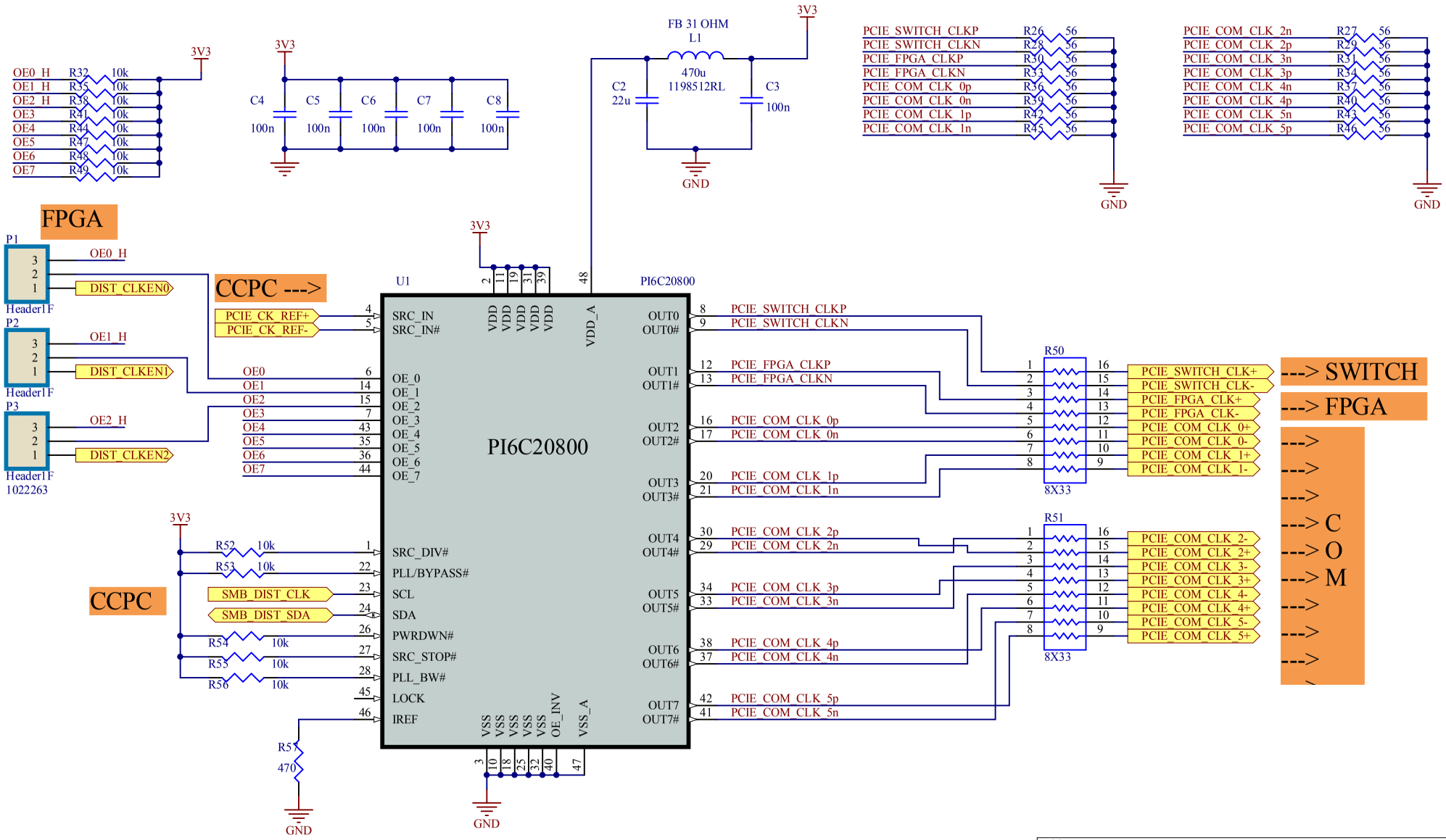
Apêndice A – Esquemáticos



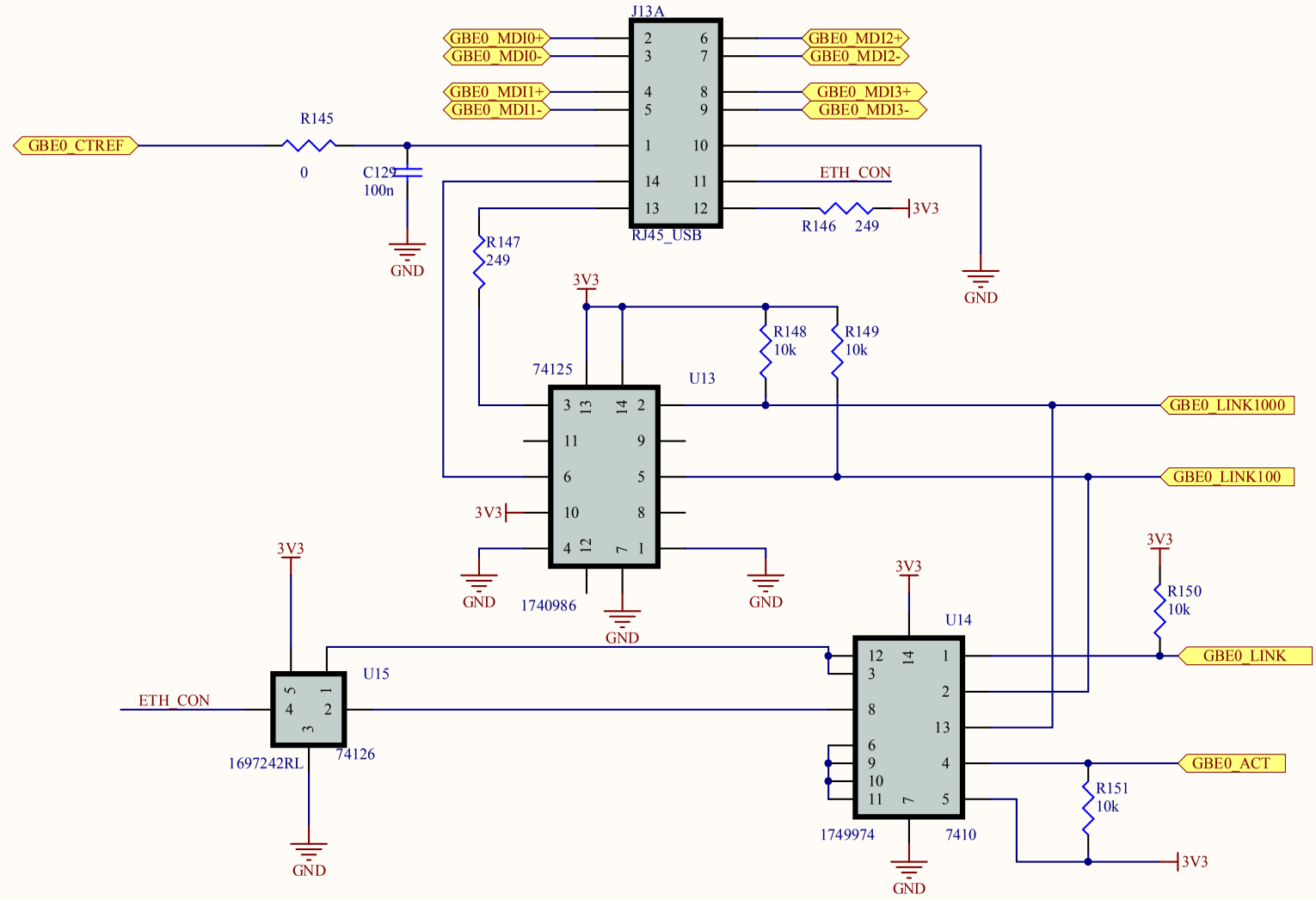
Title	**CCPC Connector**	
Size: A4	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 1 of 18
File: \\..\CCPC.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF



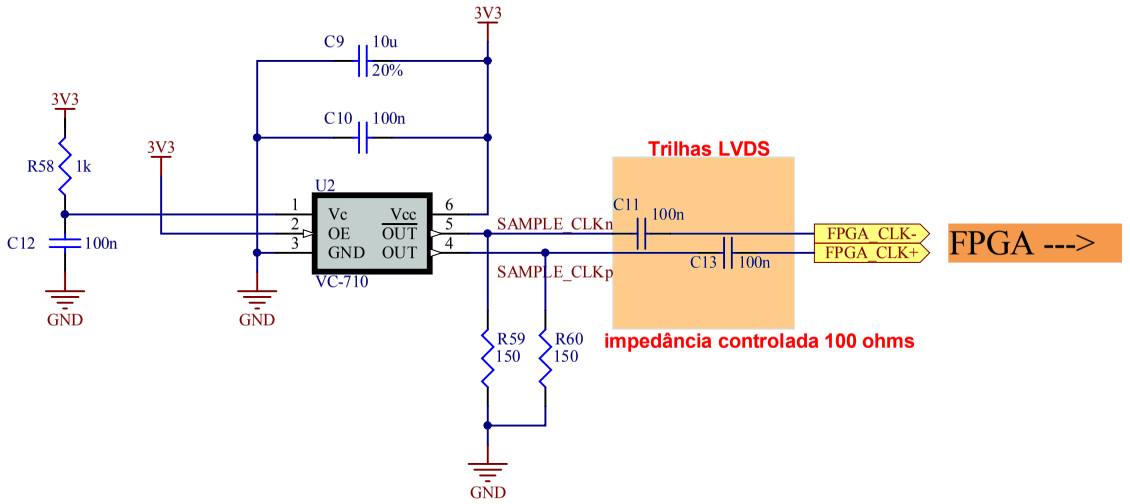
Title	**TELL connector**	
Size: A3	Number: Version 09	Revision:
Date: 8/7/2013	Sheet 14 of 18	
File: \\..\connectorTELL.SchDoc	Drawn By: Massafferri/F.Sousa - CBPF	



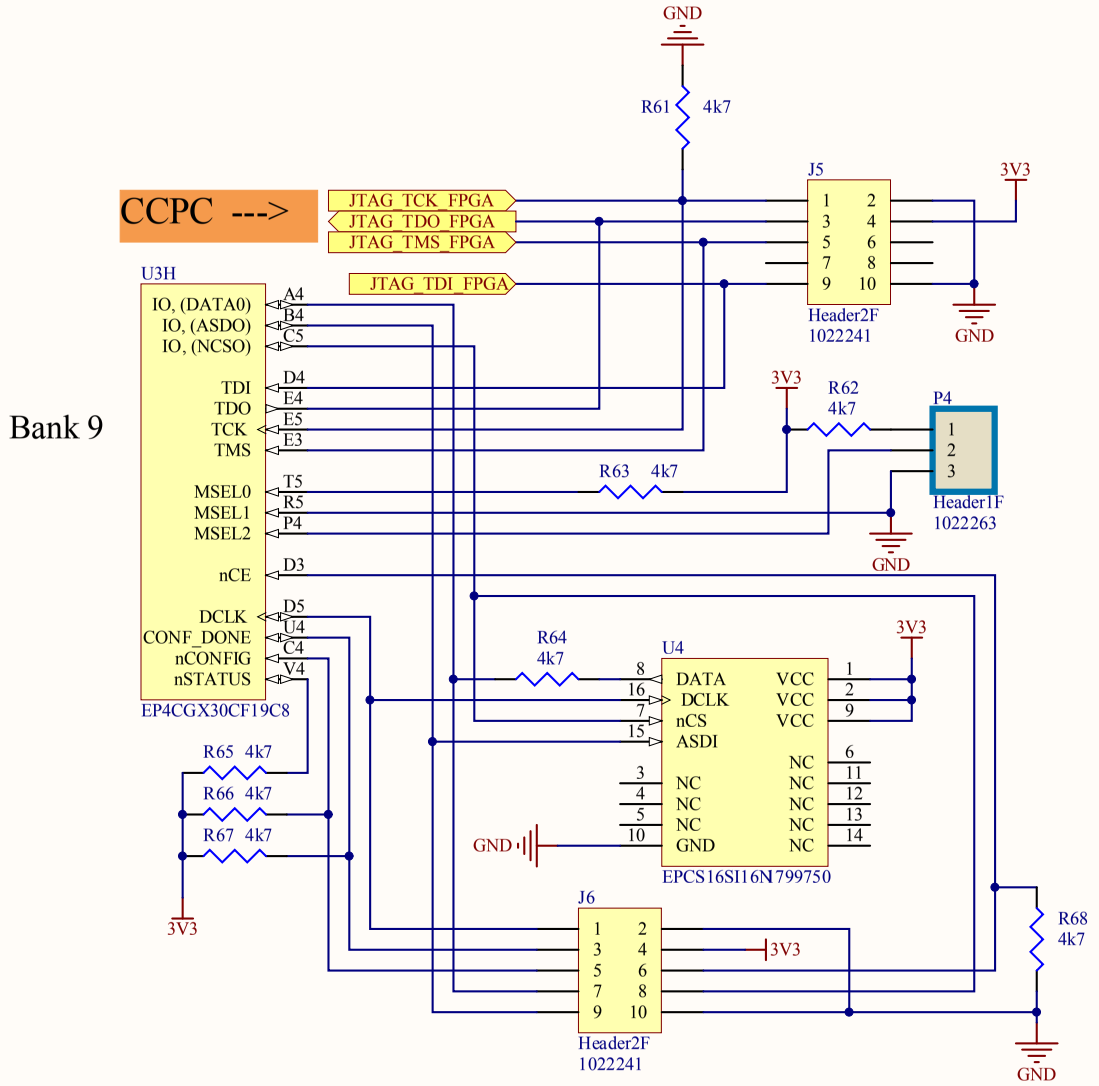
Title	**Clock distribution**	
Size: A4	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 3 of 18
File: \\..\distCLK_PCIe.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF



Title	**Ethernet**	
Size: A	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 2 of 18
File: \\..\Ethernet.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF



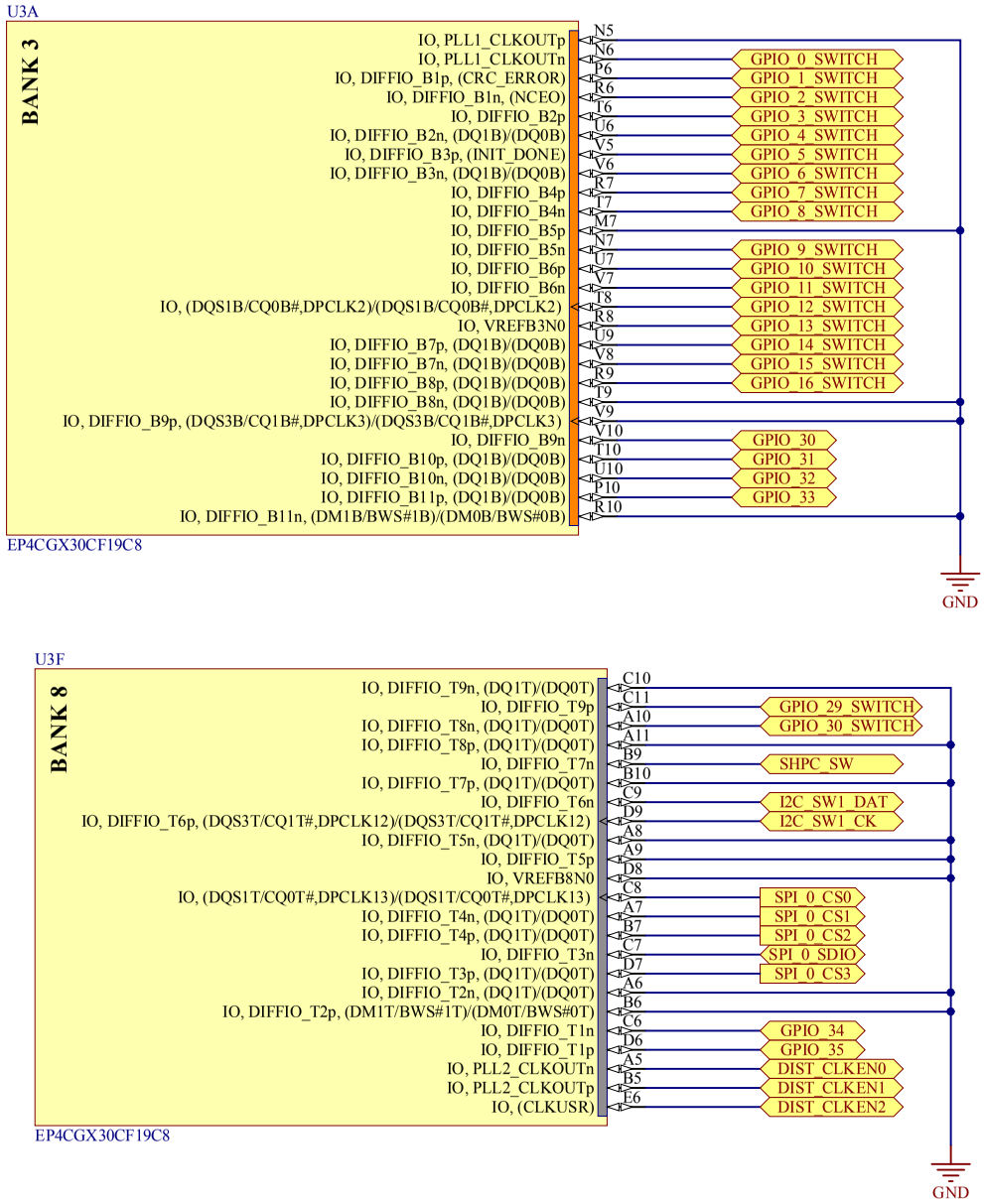
Title	**FPGA Clock**		
Size: A4	Number: Version 09		Revision:
Date: 8/7/2013		Sheet 10 of 18	
File: \\..\fpga_CLK.SchDoc		Drawn By: Fernando Sousa - CBPF	



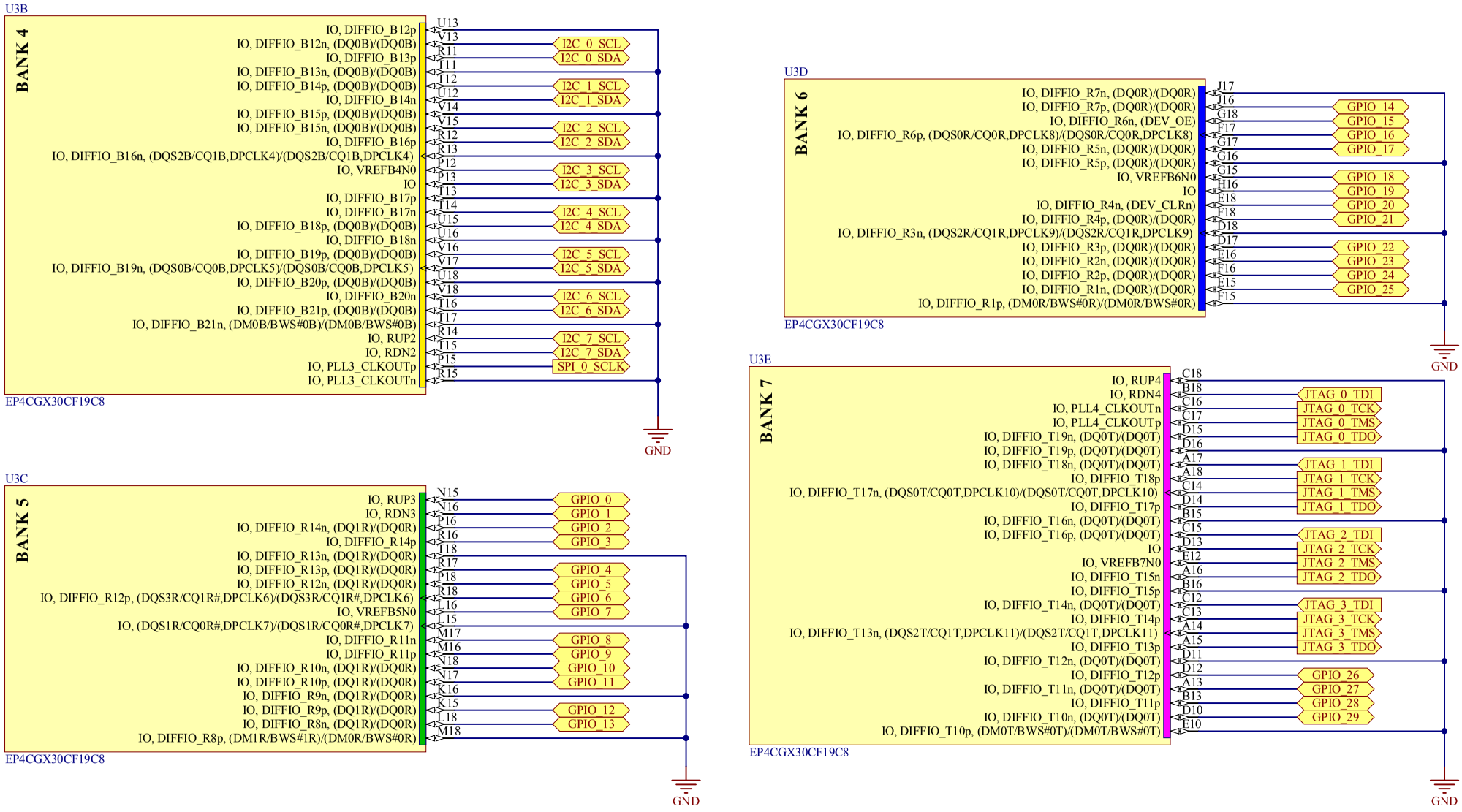
Title

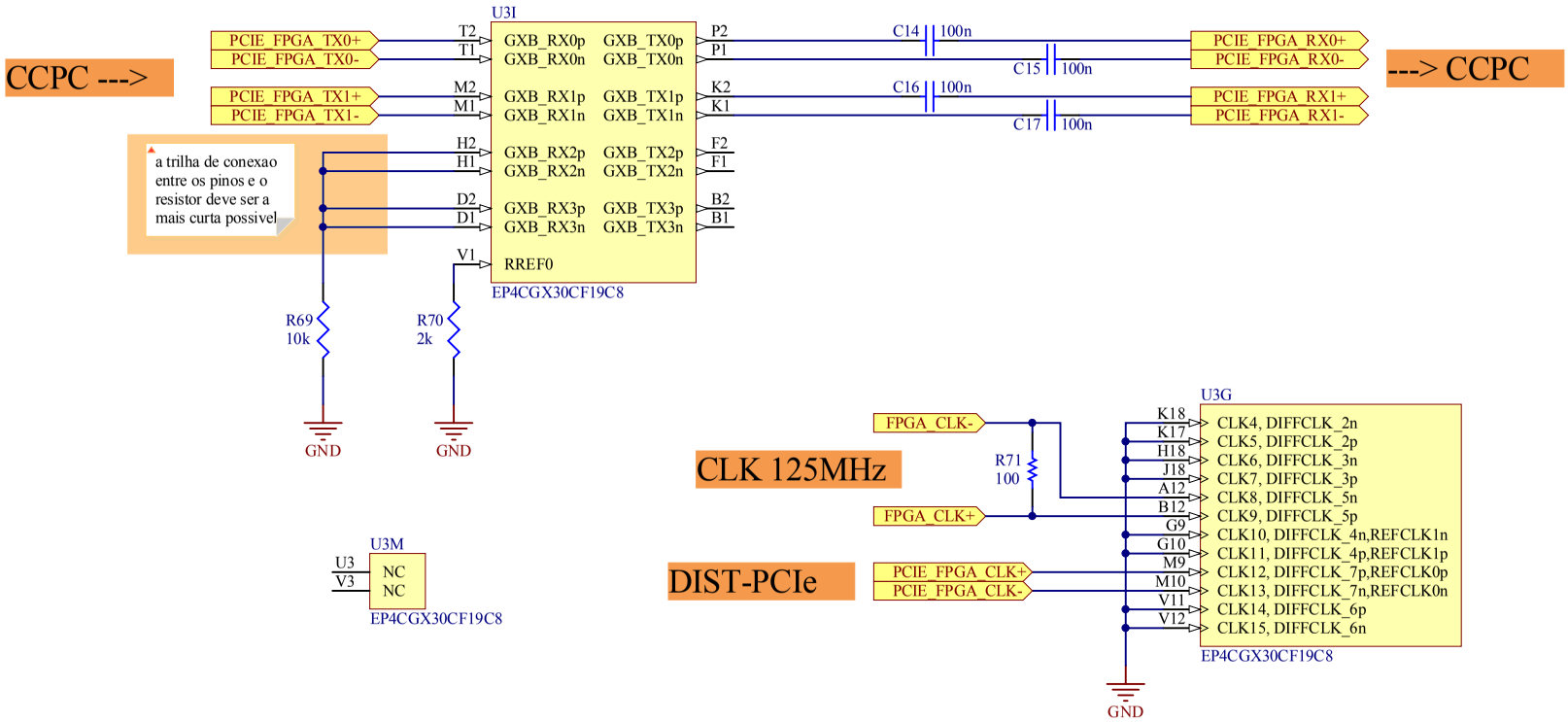
FPGA - Configuration

Size	Number	Revision
A	Version 09	
Date: 8/7/2013		Sheet 6 of 18
File: \\..\fpga_config.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

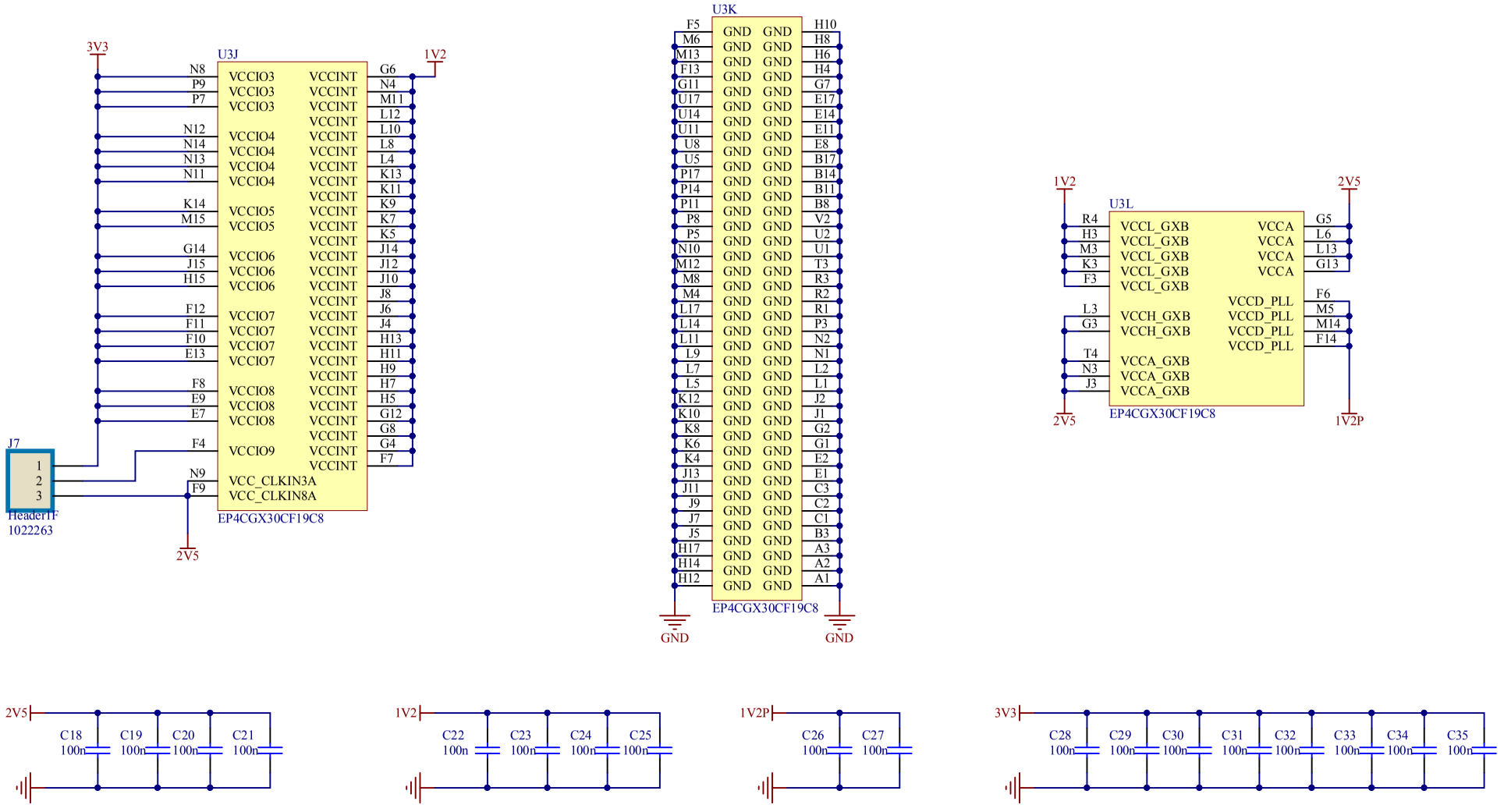


Title	**FPGA Hot Plug & IO**	
Size: A4	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 8 of 18
File: \\..\fpga_HP-IO.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

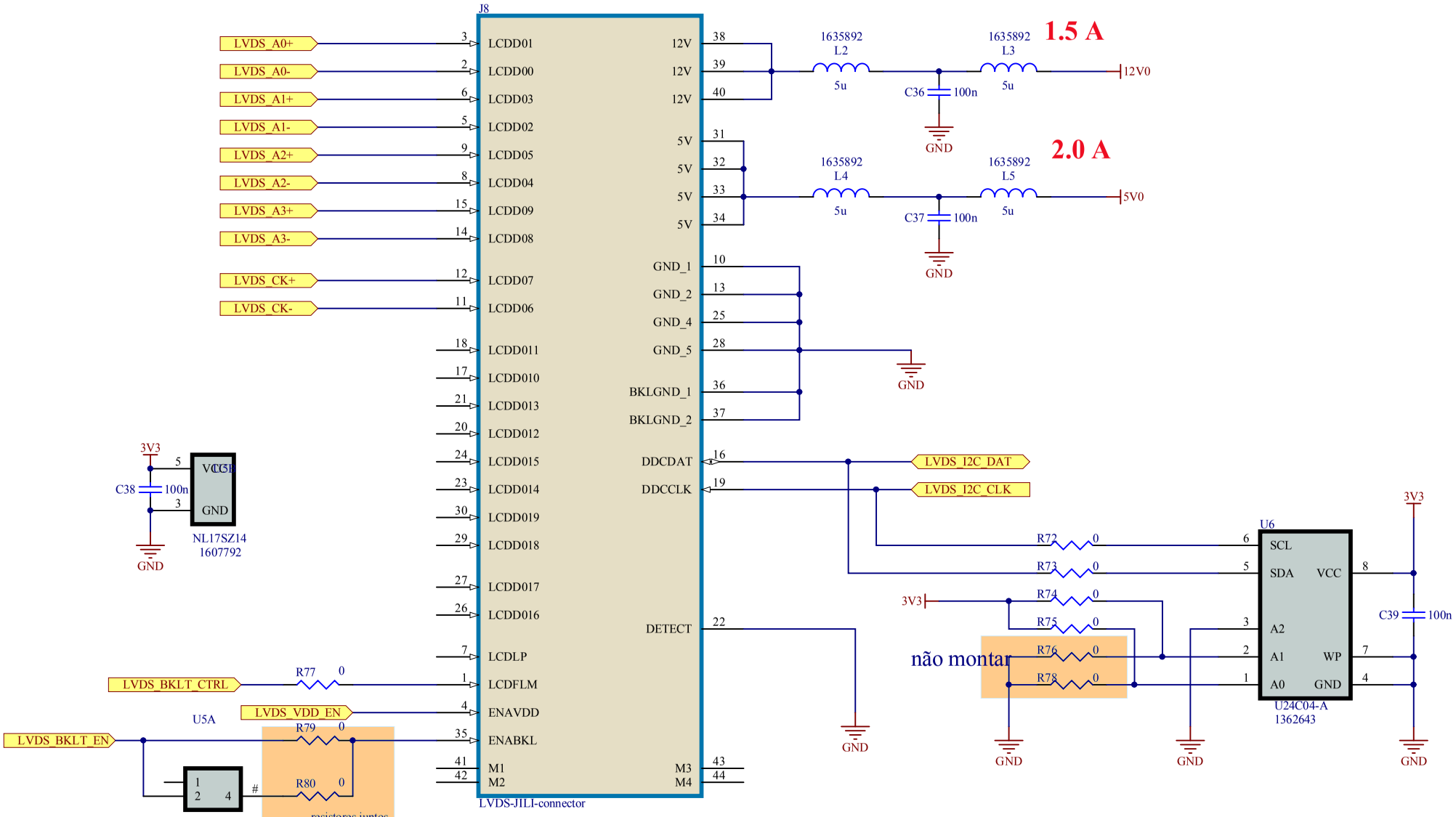

U3B
BANK 4
IO, DIFFIO_B12p
IO, DIFFIO_B12n, (DQ0B)/(DQ0B)
IO, DIFFIO_B13p
IO, DIFFIO_B13n, (DQ0B)/(DQ0B)
IO, DIFFIO_B14p, (DQ0B)/(DQ0B)
IO, DIFFIO_B14n
IO, DIFFIO_B15p, (DQ0B)/(DQ0B)
IO, DIFFIO_B15n, (DQ0B)/(DQ0B)
IO, DIFFIO_B16p
IO, DIFFIO_B16n, (DQS2B/CQ1B,DPCLK4)/(DQS2B/CQ1B,DPCLK4)
IO, VREFB4N0
IO
IO, DIFFIO_B17p
IO, DIFFIO_B17n
IO, DIFFIO_B18p, (DQ0B)/(DQ0B)
IO, DIFFIO_B18n
IO, DIFFIO_B19p, (DQ0B)/(DQ0B)
IO, DIFFIO_B19n, (DQS0B/CQ0B,DPCLK5)/(DQS0B/CQ0B,DPCLK5)
IO, DIFFIO_B20p, (DQ0B)/(DQ0B)
IO, DIFFIO_B20n
IO, DIFFIO_B21p, (DQ0B)/(DQ0B)
IO, DIFFIO_B21n, (DM0B/BWS#0B)/(DM0B/BWS#0B)
IO, RUP2
IO, RDN2
IO, PLL3_CLKOUTp
IO, PLL3_CLKOUTn
U13
V13
R11
T11
T12
U12
V14
V15
R12
R13
P12
P13
T13
T14
U15
U16
V16
V17
U18
V18
T16
T17
R14
T15
P15
R15
I2C_0_SCL
I2C_0_SDA
I2C_1_SCL
I2C_1_SDA
I2C_2_SCL
I2C_2_SDA
I2C_3_SCL
I2C_3_SDA
I2C_4_SCL
I2C_4_SDA
I2C_5_SCL
I2C_5_SDA
I2C_6_SCL
I2C_6_SDA
I2C_7_SCL
I2C_7_SDA
SPI_0_SCLK
GND
EP4CGX30CF19C8
U3C
BANK 5
IO, RUP3
IO, RDN3
IO, DIFFIO_R14n, (DQ1R)/(DQ0R)
IO, DIFFIO_R14p
IO, DIFFIO_R13n, (DQ1R)/(DQ0R)
IO, DIFFIO_R13p, (DQ1R)/(DQ0R)
IO, DIFFIO_R12n, (DQ1R)/(DQ0R)
IO, DIFFIO_R12p, (DQS3R/CQ1R#,DPCLK6)/(DQS3R/CQ1R#,DPCLK6)
IO, VREFB5N0
IO, (DQS1R/CQ0R#,DPCLK7)/(DQS1R/CQ0R#,DPCLK7)
IO, DIFFIO_R11n
IO, DIFFIO_R11p
IO, DIFFIO_R10n, (DQ1R)/(DQ0R)
IO, DIFFIO_R10p, (DQ1R)/(DQ0R)
IO, DIFFIO_R9n, (DQ1R)/(DQ0R)
IO, DIFFIO_R9p, (DQ1R)/(DQ0R)
IO, DIFFIO_R8n, (DQ1R)/(DQ0R)
IO, DIFFIO_R8p, (DM1R/BWS#1R)/(DM0R/BWS#0R)
N15
N16
P16
R16
T18
R17
P18
R18
L16
L15
M17
M16
N18
N17
K16
K15
L18
M18
GPIO_0
GPIO_1
GPIO_2
GPIO_3
GPIO_4
GPIO_5
GPIO_6
GPIO_7
GPIO_8
GPIO_9
GPIO_10
GPIO_11
GPIO_12
GPIO_13
GND
EP4CGX30CF19C8
U3D
BANK 6
IO, DIFFIO_R7n, (DQ0R)/(DQ0R)
IO, DIFFIO_R7p, (DQ0R)/(DQ0R)
IO, DIFFIO_R6n, (DEV_OE)
IO, DIFFIO_R6p, (DQS0R/CQ0R,DPCLK8)/(DQS0R/CQ0R,DPCLK8)
IO, DIFFIO_R5n, (DQ0R)/(DQ0R)
IO, DIFFIO_R5p, (DQ0R)/(DQ0R)
IO, VREFB6N0
IO
IO, DIFFIO_R4n, (DEV_CLRn)
IO, DIFFIO_R4p, (DQ0R)/(DQ0R)
IO, DIFFIO_R3n, (DQS2R/CQ1R,DPCLK9)/(DQS2R/CQ1R,DPCLK9)
IO, DIFFIO_R3p, (DQ0R)/(DQ0R)
IO, DIFFIO_R2n, (DQ0R)/(DQ0R)
IO, DIFFIO_R2p, (DQ0R)/(DQ0R)
IO, DIFFIO_R1n, (DQ0R)/(DQ0R)
IO, DIFFIO_R1p, (DM0R/BWS#0R)/(DM0R/BWS#0R)
J17
J16
G18
F17
G17
G16
G15
H16
E18
F18
D18
D17
E16
F16
E15
F15
GPIO_14
GPIO_15
GPIO_16
GPIO_17
GPIO_18
GPIO_19
GPIO_20
GPIO_21
GPIO_22
GPIO_23
GPIO_24
GPIO_25
GND
EP4CGX30CF19C8
U3E
BANK 7
IO, RUP4
IO, RDN4
IO, PLL4_CLKOUTn
IO, PLL4_CLKOUTp
IO, DIFFIO_T19n, (DQ0T)/(DQ0T)
IO, DIFFIO_T19p, (DQ0T)/(DQ0T)
IO, DIFFIO_T18n, (DQ0T)/(DQ0T)
IO, DIFFIO_T18p
IO, DIFFIO_T17n, (DQS0T/CQ0T,DPCLK10)/(DQS0T/CQ0T,DPCLK10)
IO, DIFFIO_T17p
IO, DIFFIO_T16n, (DQ0T)/(DQ0T)
IO, DIFFIO_T16p, (DQ0T)/(DQ0T)
IO
IO, VREFB7N0
IO, DIFFIO_T15n
IO, DIFFIO_T15p
IO, DIFFIO_T14n, (DQ0T)/(DQ0T)
IO, DIFFIO_T14p
IO, DIFFIO_T13n, (DQS2T/CQ1T,DPCLK11)/(DQS2T/CQ1T,DPCLK11)
IO, DIFFIO_T13p
IO, DIFFIO_T12n, (DQ0T)/(DQ0T)
IO, DIFFIO_T12p
IO, DIFFIO_T11n, (DQ0T)/(DQ0T)
IO, DIFFIO_T11p
IO, DIFFIO_T10n, (DQ0T)/(DQ0T)
IO, DIFFIO_T10p, (DM0T/BWS#0T)/(DM0T/BWS#0T)
C18
B18
C16
C17
D15
D16
A17
A18
C14
D14
B15
C15
D13
E12
A16
B16
C12
C13
A14
A15
D11
D12
A13
B13
D10
E10
JTAG_0_TDI
JTAG_0_TCK
JTAG_0_TMS
JTAG_0_TDO
JTAG_1_TDI
JTAG_1_TCK
JTAG_1_TMS
JTAG_1_TDO
JTAG_2_TDI
JTAG_2_TCK
JTAG_2_TMS
JTAG_2_TDO
JTAG_3_TDI
JTAG_3_TCK
JTAG_3_TMS
JTAG_3_TDO
GPIO_26
GPIO_27
GPIO_28
GPIO_29
GND
EP4CGX30CF19C8
Title
FPGA - IO
Size
A4
Number
Version 09
Revision
Date: 8/7/2013
Sheet 7 of 18
File: \\..\fpga_IO.SchDoc
Drawn By: Massafferri/F.Sousa - CBPF



Title	**FPGA Transceivers PCIe**	
Size: A4	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 9 of 18
File: \\..\fpga_PCIe.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF



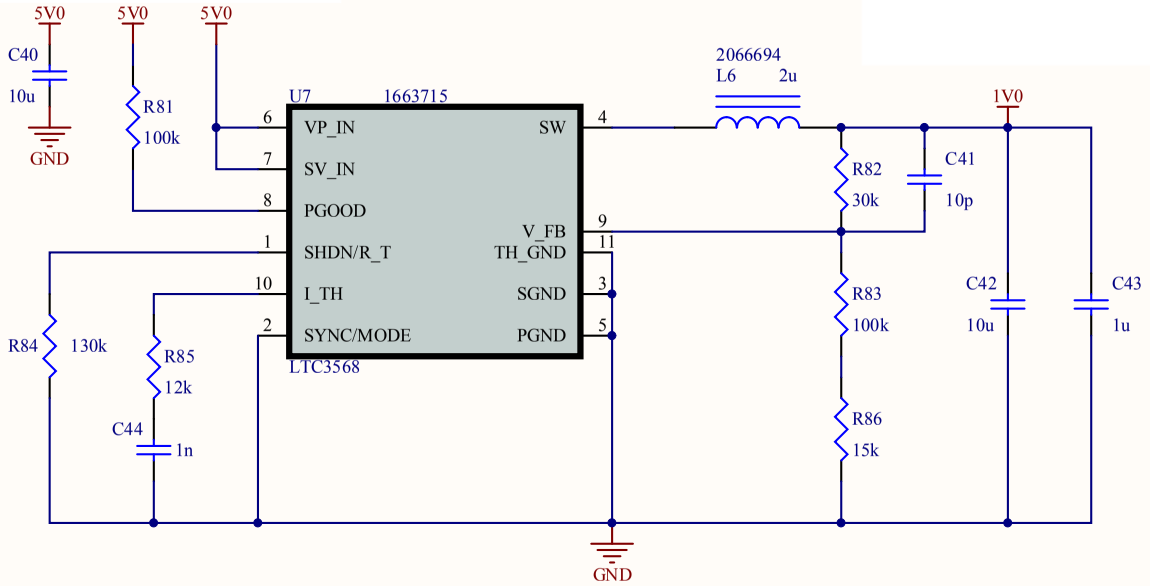
Title	**FPGA - Power**	
Size: A4	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 5 of 18
File: \\..\fpga_power.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF



Title	**LVDS Connector**	
Size: A4	Number: Version 09	Revision: 18
Date: 8/7/2013		Sheet 4 of
File: \\..\LVDSmonitor.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

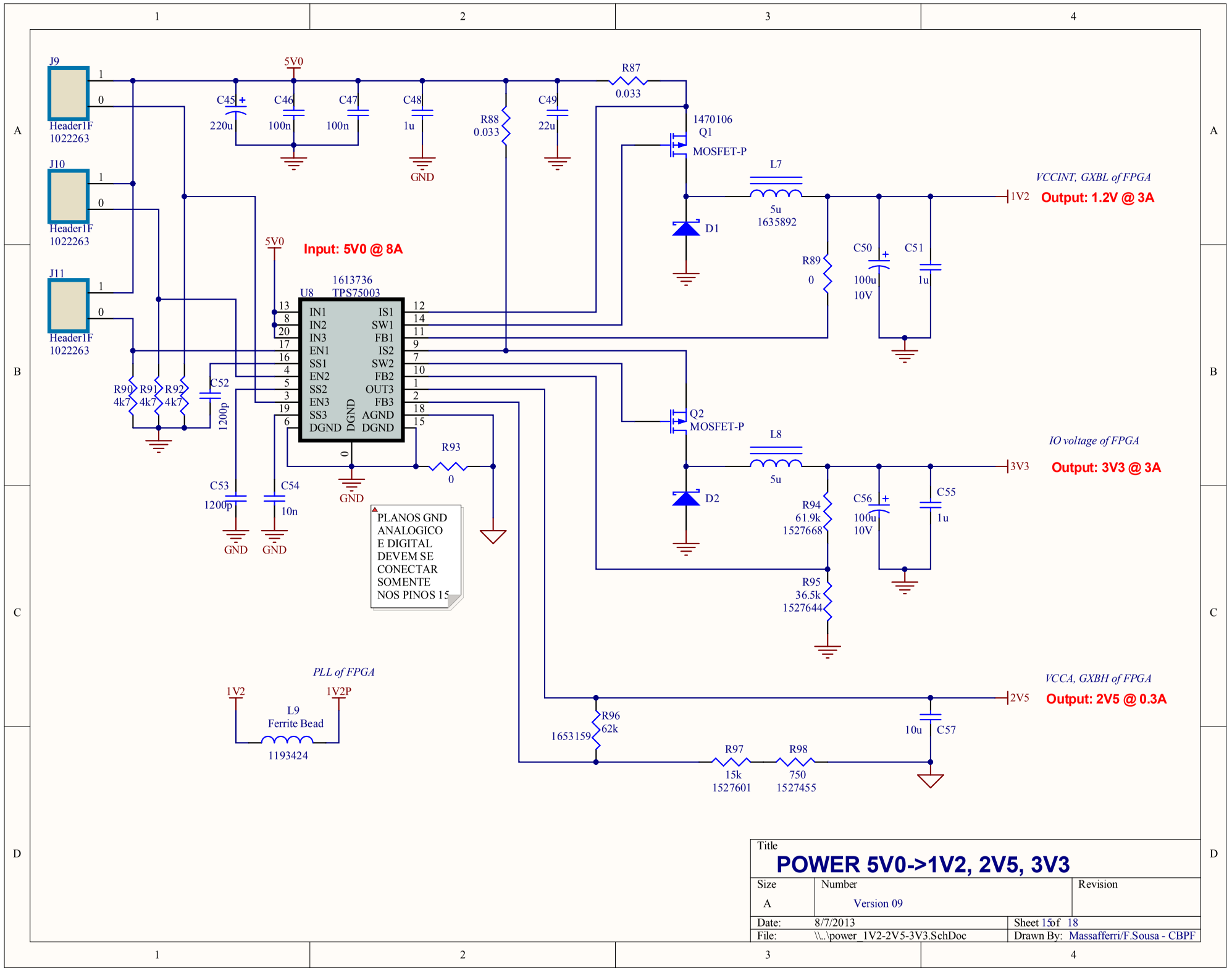
input 5V0 @ 0.3 A

output 1V0 @ 1.4 A



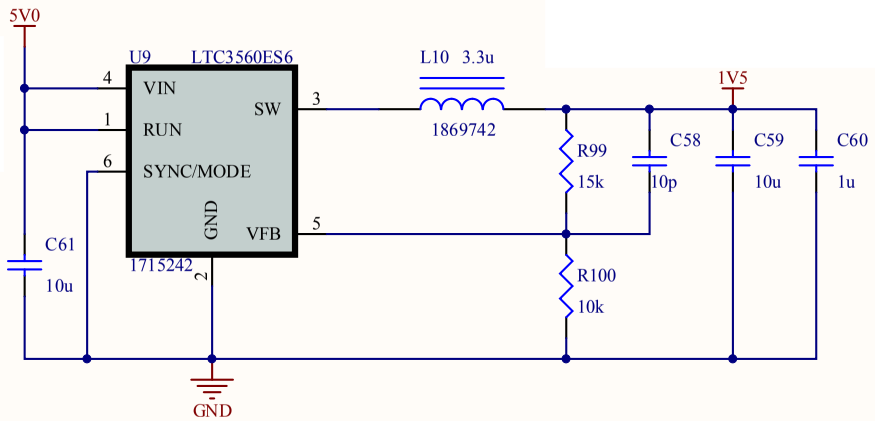
CORE & GXB FOR SWITCH

Title: **POWER 5V0->1V0**		
Size: A	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 15 of 18
File: \\..\power_1V0.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF


J9
Header1F
1022263
J10
Header1F
1022263
J11
Header1F
1022263
5V0
C45
220u
C46
100n
C47
100n
C48
1u
GND
R88
0.033
C49
22u
R87
0.033
1470106
Q1
MOSFET-P
L7
5u
1635892
D1
VCCINT, GXBL of FPGA
1V2
Output: 1.2V @ 3A
R89
0
C50
100u
10V
C51
1u
5V0
Input: 5V0 @ 8A
1613736
U8
TPS75003
IN1
IN2
IN3
EN1
SS1
EN2
SS2
EN3
SS3
DGND
IS1
SW1
FB1
IS2
SW2
FB2
OUT3
FB3
AGND
DGND
R90
4k7
R91
4k7
R92
4k7
C52
1200p
C53
1200p
C54
10n
R93
0
PLANOS GND
ANALOGICO
E DIGITAL
DEVEM SE
CONECTAR
SOMENTE
NOS PINOS 15
Q2
MOSFET-P
L8
5u
D2
IO voltage of FPGA
3V3
Output: 3V3 @ 3A
R94
61.9k
1527668
C56
100u
10V
C55
1u
R95
36.5k
1527644
PLL of FPGA
1V2
1V2P
L9
Ferrite Bead
1193424
VCCA, GXBH of FPGA
2V5
Output: 2V5 @ 0.3A
R96
62k
1653159
R97
15k
1527601
R98
750
1527455
10u
C57
Title
POWER 5V0->1V2, 2V5, 3V3
Size
A
Number
Version 09
Revision
Date:
8/7/2013
Sheet 15 of 18
File:
\\..\power_1V2-2V5-3V3.SchDoc
Drawn By:
Massafferri/F.Sousa - CBPF

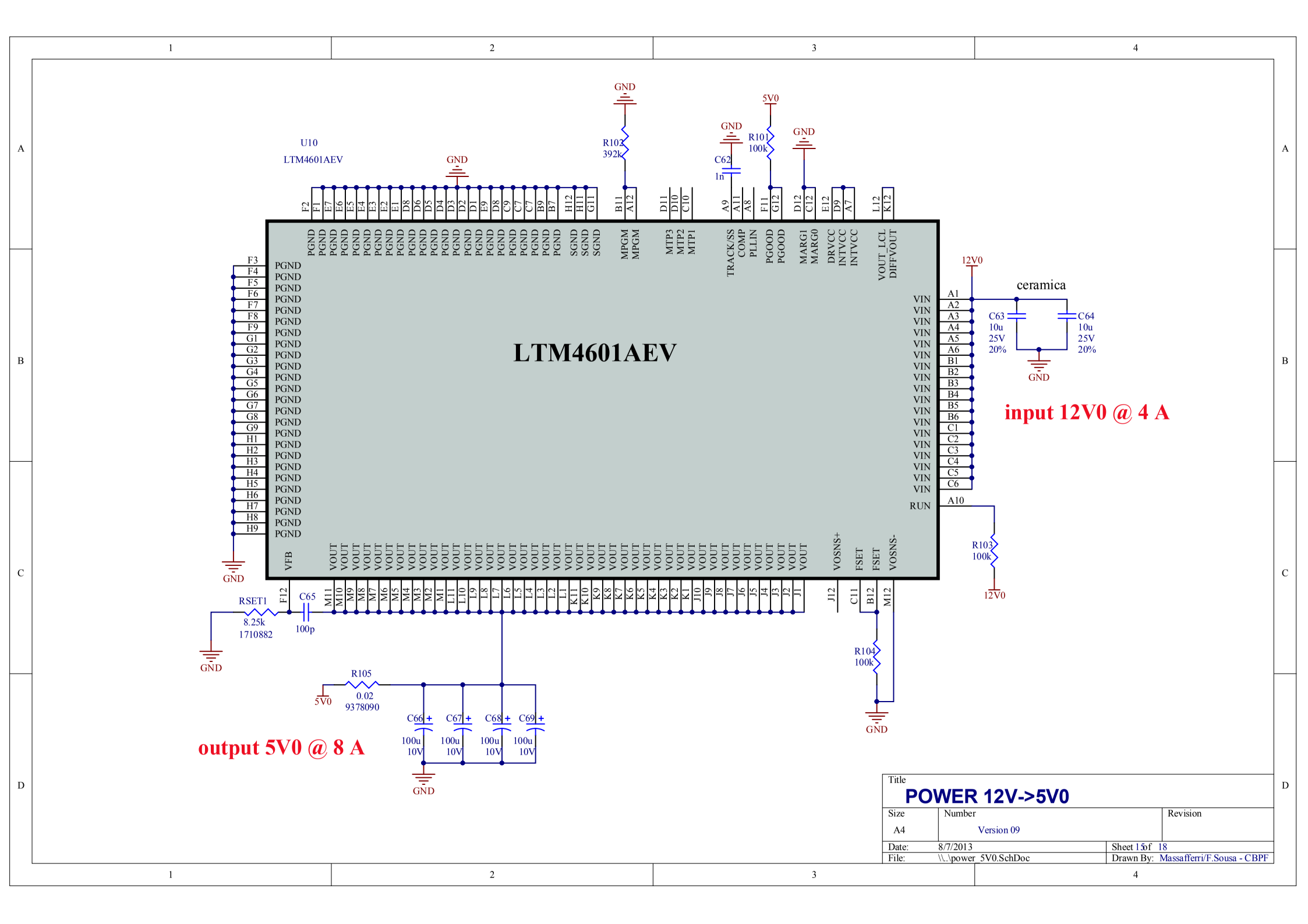
input 5V0 @ 0.2 A

output 1V5 @ 0.6 A

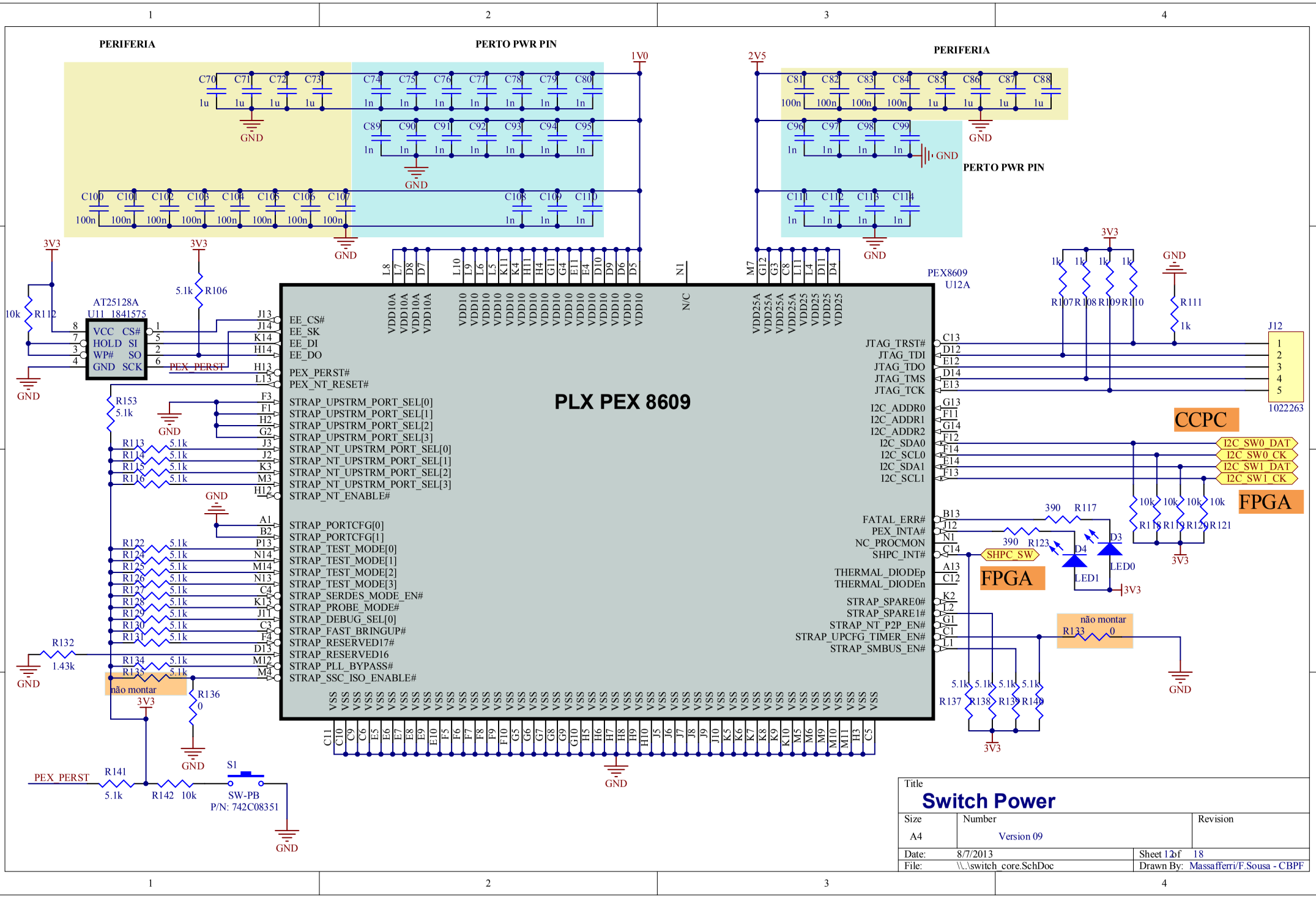


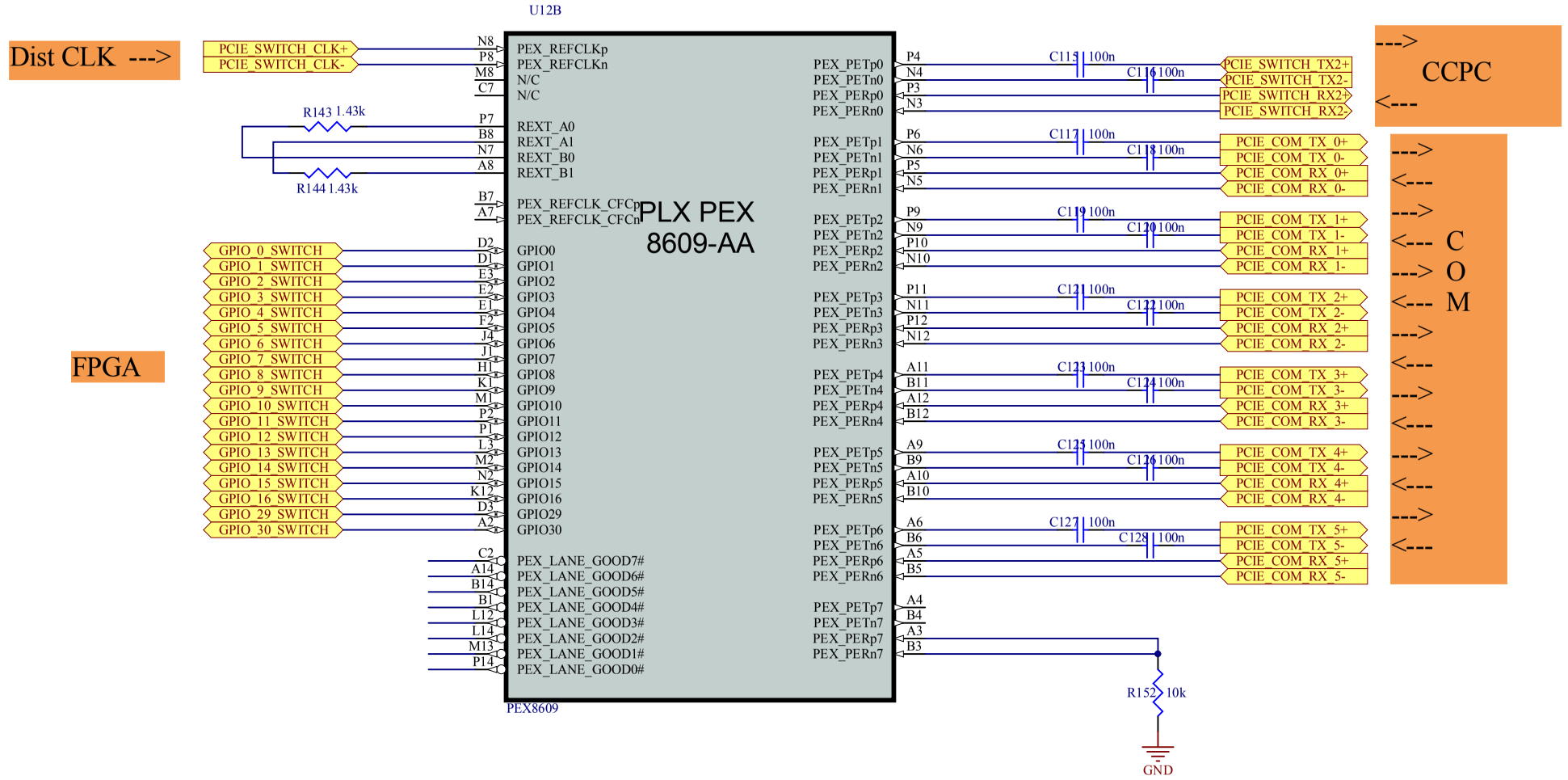
SERDES FOR SWITCH

Title: **POWER 5V0->1V5**		
Size: A	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 15 of 18
File: \\..\power_1V5.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

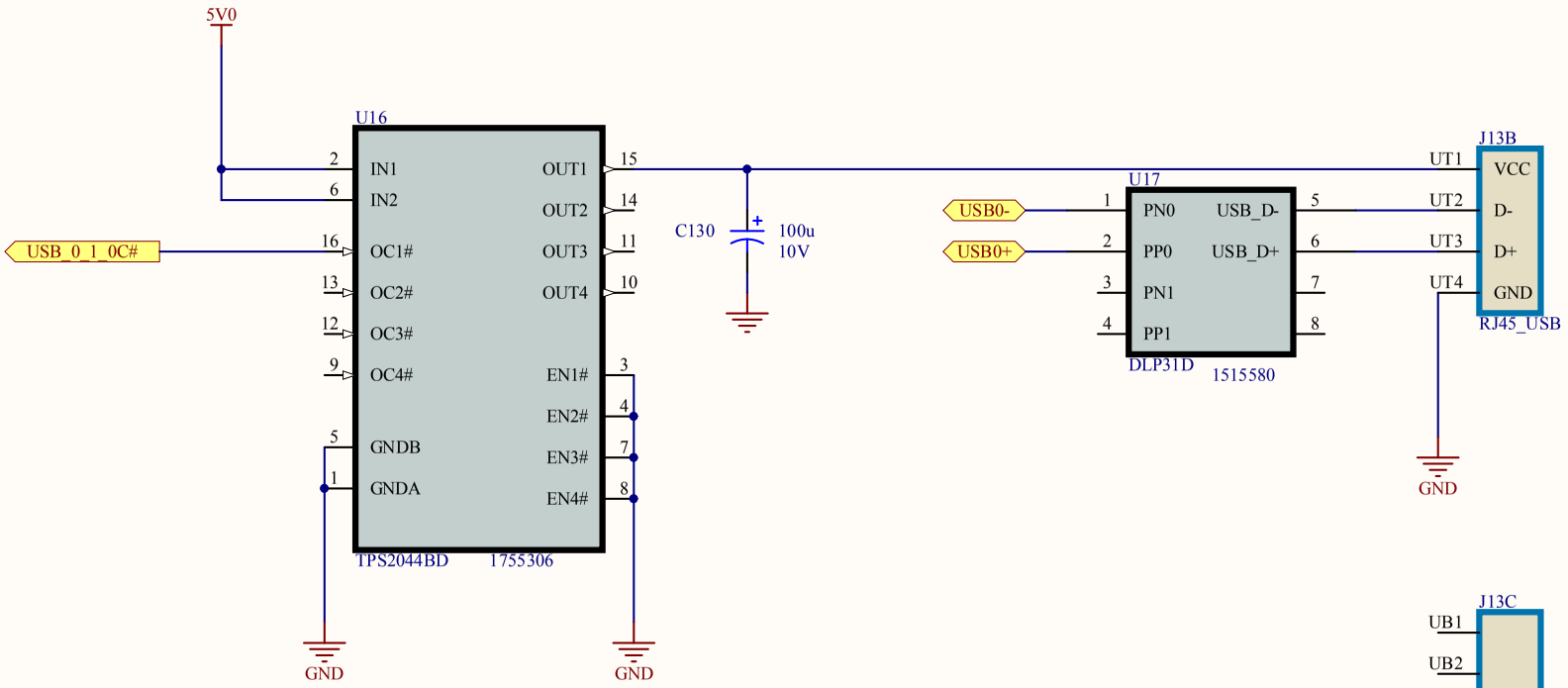


Title	**POWER 12V->5V0**	
Size: A4	Number: Version 09	Revision
Date: 8/7/2013		Sheet 15 of 18
File: \\..\power_5V0.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

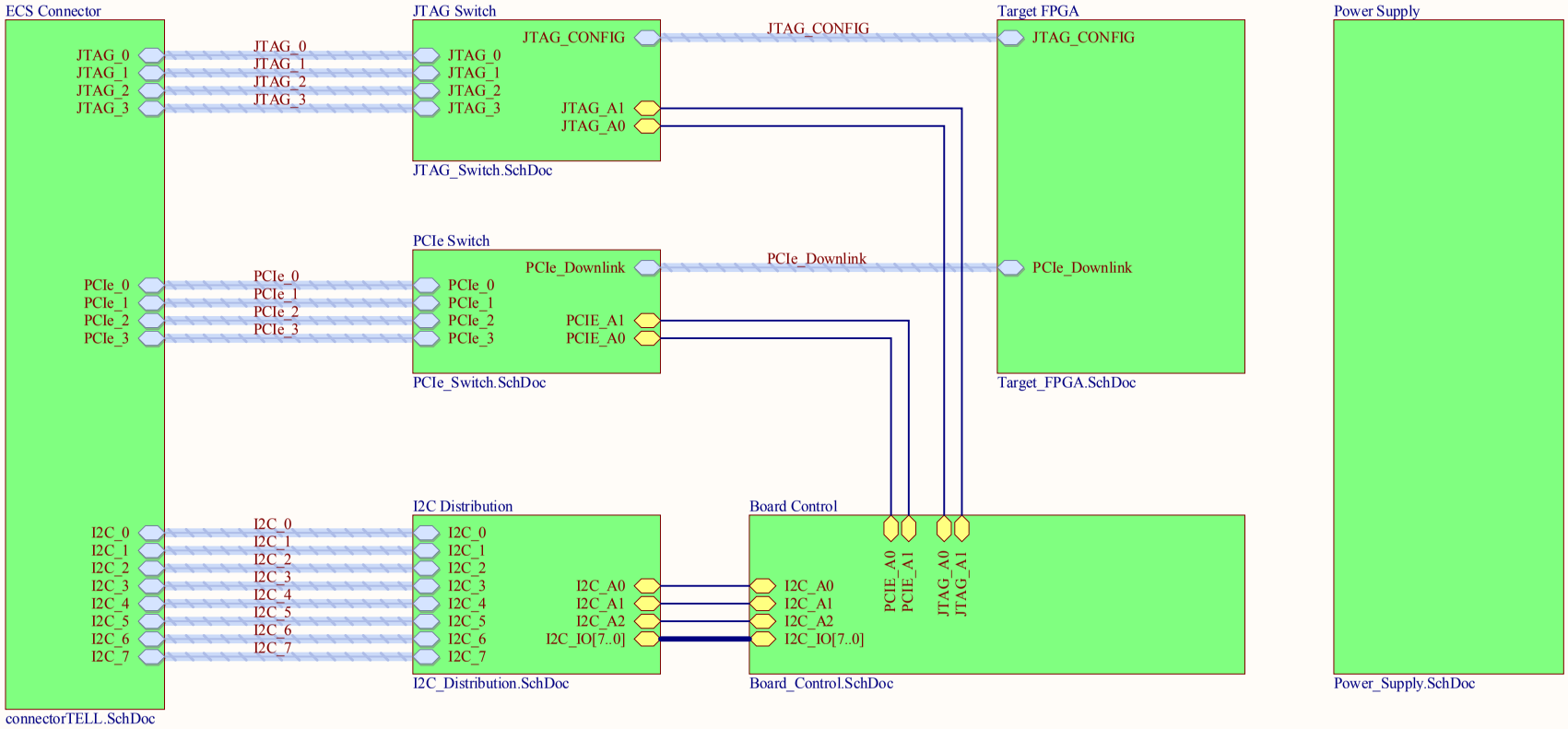

PERIFERIA
PERTO PWR PIN
1V0
2V5
PERIFERIA
PERTO PWR PIN
PEX8609
U12A
PLX PEX 8609
AT25128A
U11_1841575
EE_CS#
EE_SK
EE_DI
EE_DO
PEX_PERST#
PEX_NT_RESET#
STRAP_UPSTRM_PORT_SEL[0]
STRAP_UPSTRM_PORT_SEL[1]
STRAP_UPSTRM_PORT_SEL[2]
STRAP_UPSTRM_PORT_SEL[3]
STRAP_NT_UPSTRM_PORT_SEL[0]
STRAP_NT_UPSTRM_PORT_SEL[1]
STRAP_NT_UPSTRM_PORT_SEL[2]
STRAP_NT_UPSTRM_PORT_SEL[3]
STRAP_NT_ENABLE#
STRAP_PORTCFG[0]
STRAP_PORTCFG[1]
STRAP_TEST_MODE[0]
STRAP_TEST_MODE[1]
STRAP_TEST_MODE[2]
STRAP_TEST_MODE[3]
STRAP_SERDES_MODE_EN#
STRAP_PROBE_MODE#
STRAP_DEBUG_SEL[0]
STRAP_FAST_BRINGUP#
STRAP_RESERVED17#
STRAP_RESERVED16
STRAP_PLL_BYPASS#
STRAP_SSC_ISO_ENABLE#
JTAG_TRST#
JTAG_TDI
JTAG_TDO
JTAG_TMS
JTAG_TCK
I2C_ADDR0
I2C_ADDR1
I2C_ADDR2
I2C_SDA0
I2C_SCL0
I2C_SDA1
I2C_SCL1
FATAL_ERR#
PEX_INTA#
NC_PROCMON
SHPC_INT#
THERMAL_DIODEp
THERMAL_DIODEn
STRAP_SPARE0#
STRAP_SPARE1#
STRAP_NT_P2P_EN#
STRAP_UPCFG_TIMER_EN#
STRAP_SMBUS_EN#
CCPC
FPGA
FPGA
I2C_SW0_DAT
I2C_SW0_CK
I2C_SW1_DAT
I2C_SW1_CK
SHPC_SW
LED0
LED1
não montar
não montar
PEX_PERST
SW-PB
P/N: 742C08351
1022263
Title
Switch Power
Size
A4
Number
Version 09
Revision
Date: 8/7/2013
Sheet 12 of 18
File: \\..\switch_core.SchDoc
Drawn By: Massafferri/F.Sousa - CBPF



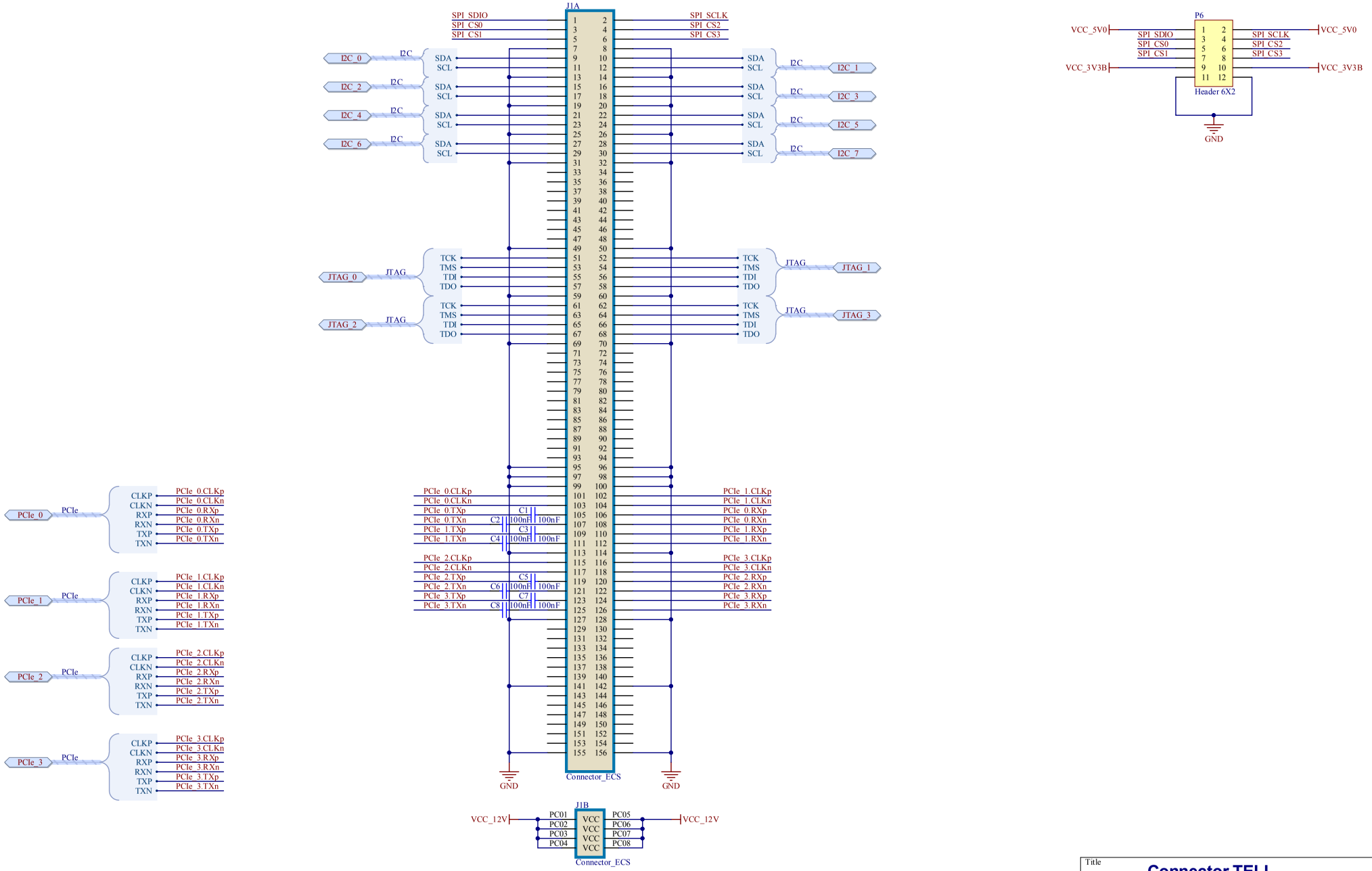
Title: **Switch PCIe**		
Size: A4	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 1 of 18
File: \\..\switch_PCIe.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF



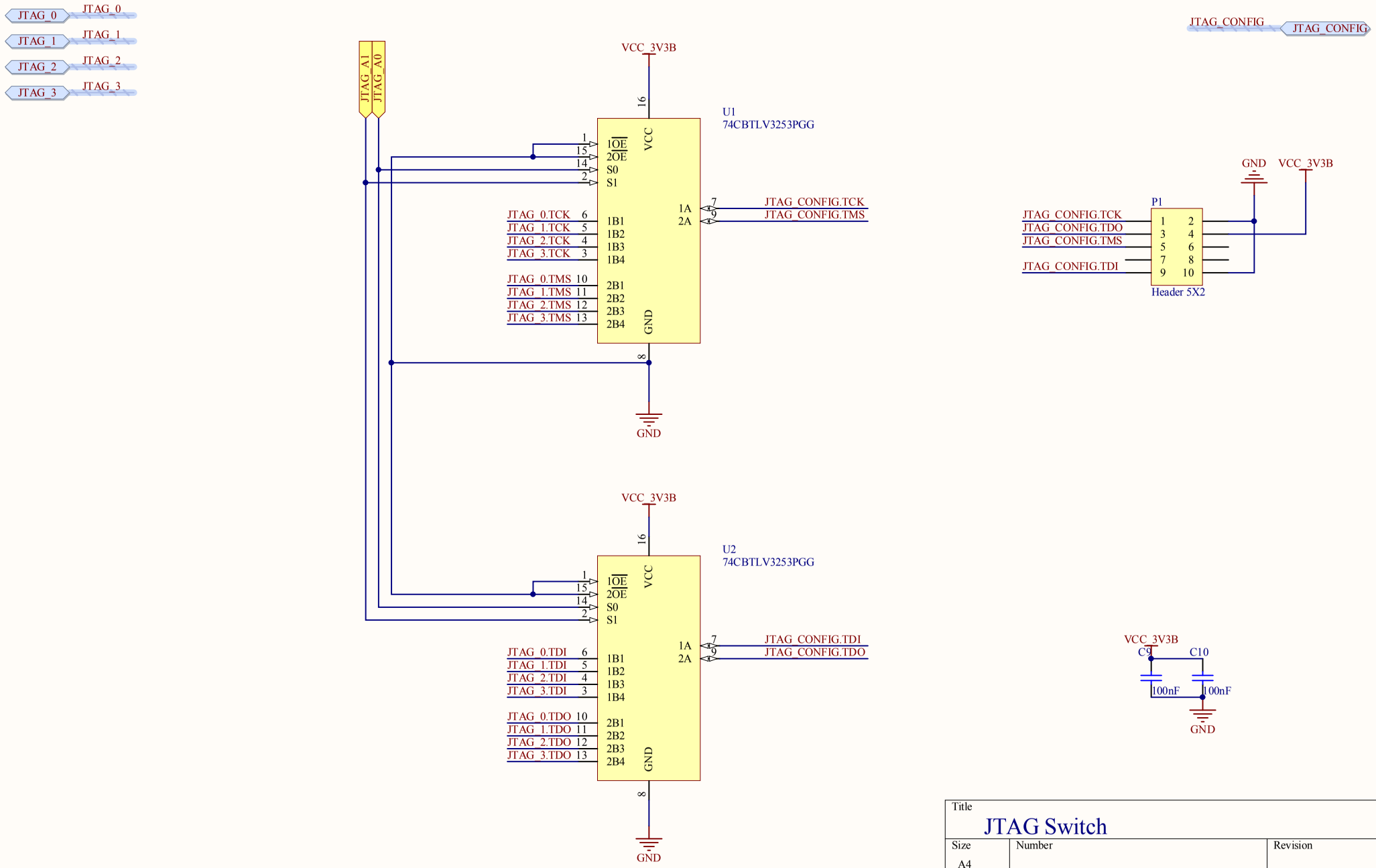
Title: **USB**		
Size: A	Number: Version 09	Revision:
Date: 8/7/2013		Sheet 2 of 18
File: \\..\USB.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

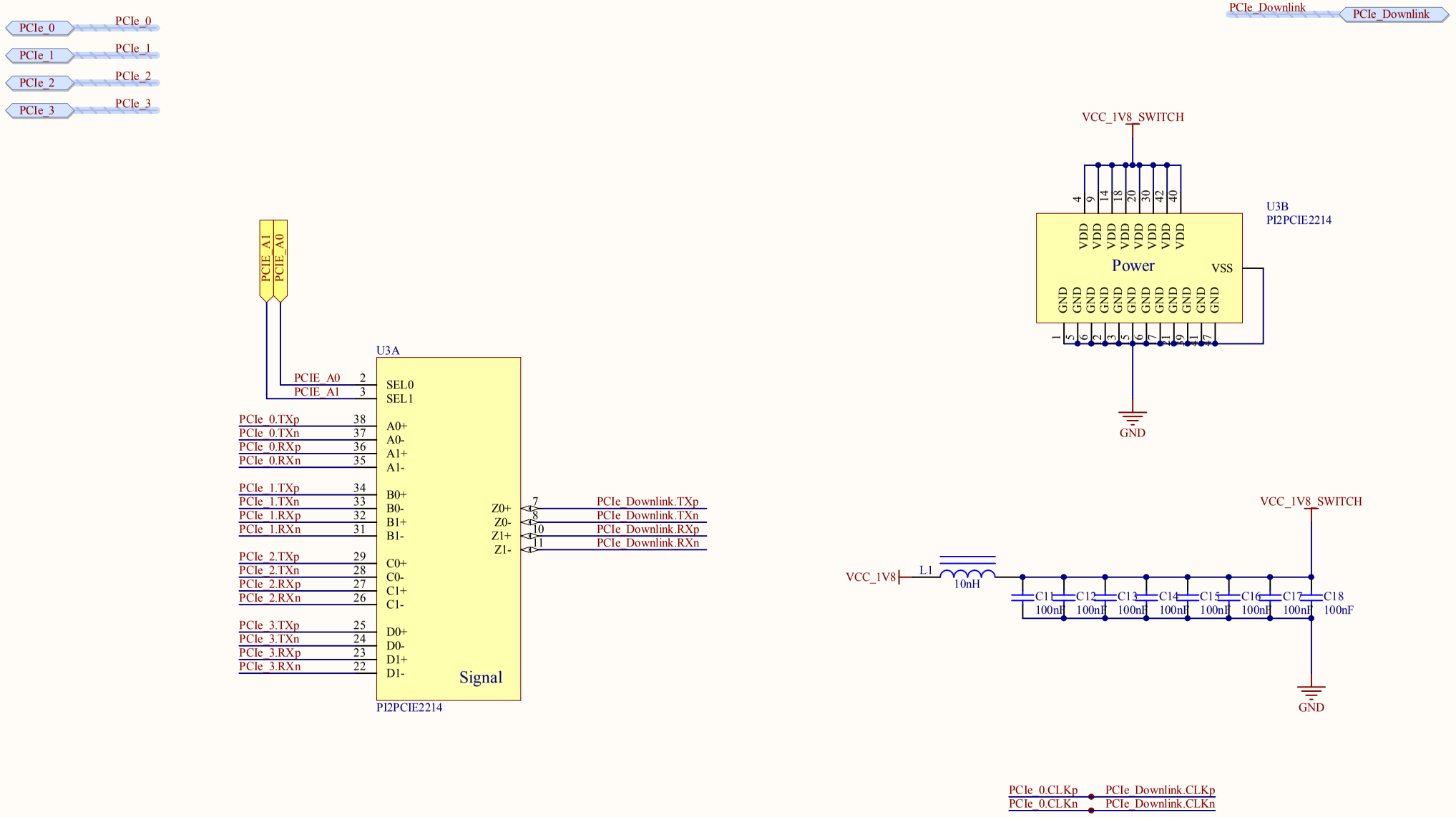


Title	ECS Validation Board Block Diagram	
Size: A4	Number:	Revision:
Date: 8/7/2013		Sheet of
File: \\..\Block_Diagram.SchDoc		Drawn By: Eng. Leonardo Lessa

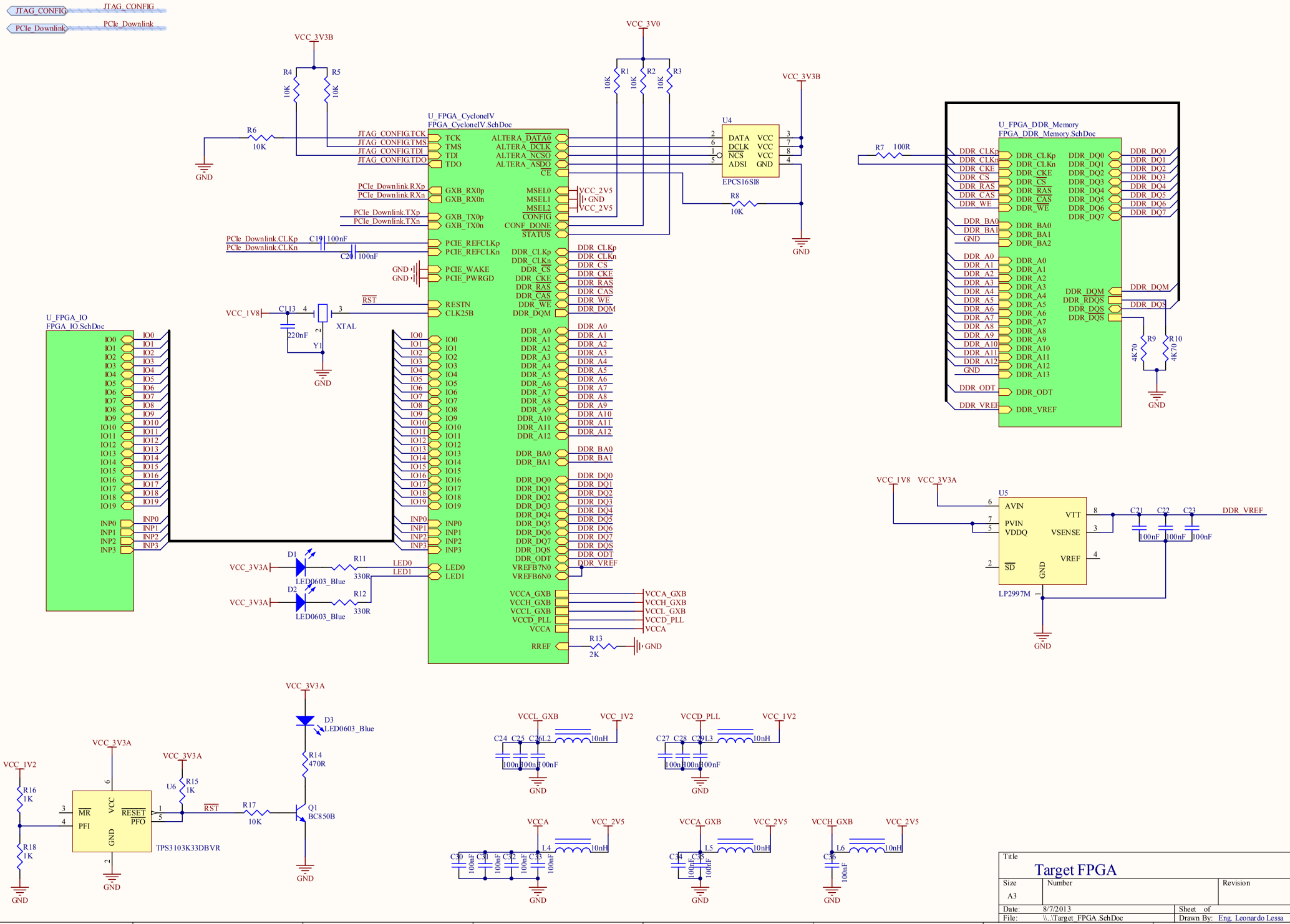


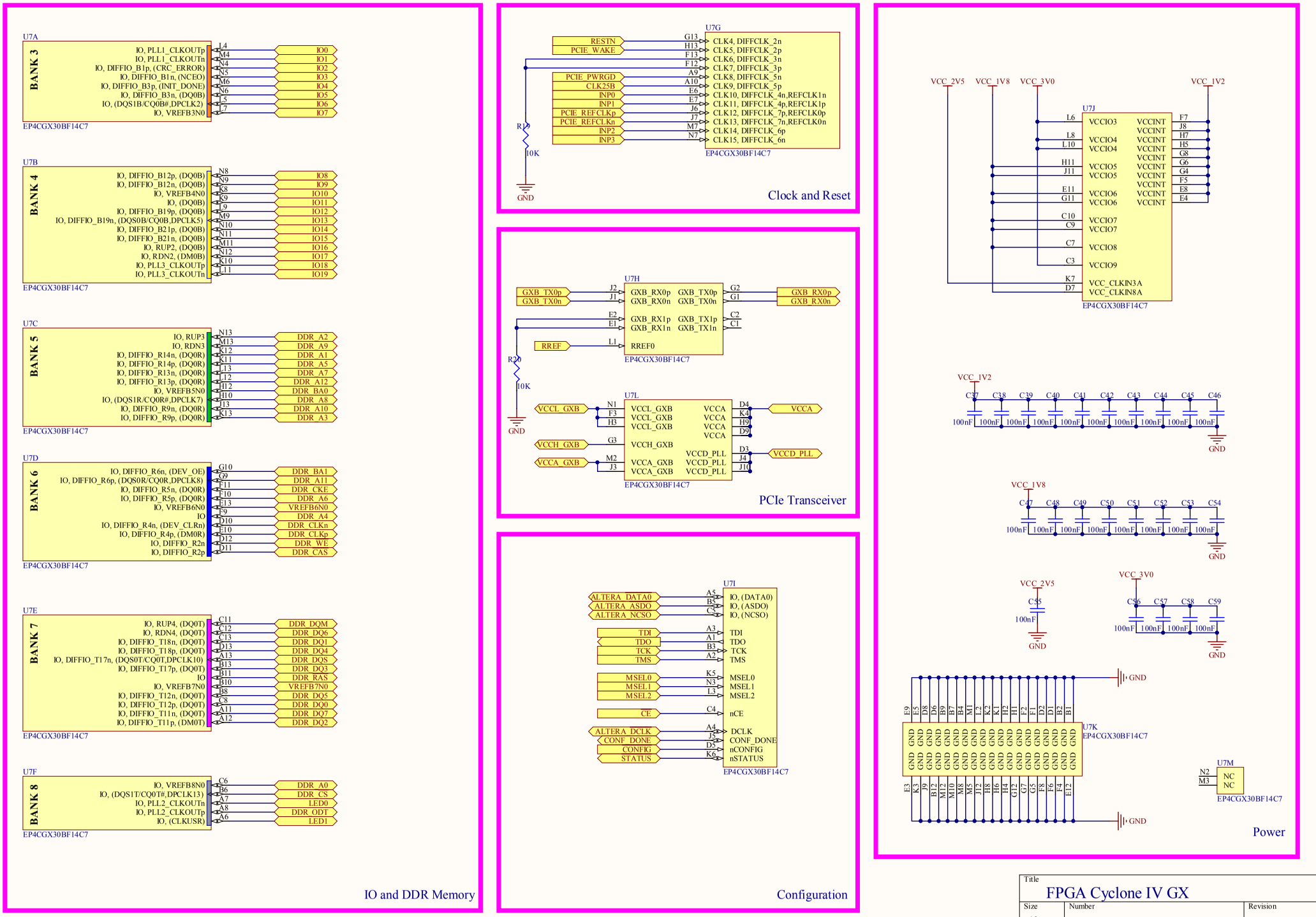
Title	**Connector TELL**	
Size: A3	Number	Revision
Date: 8/7/2013		Sheet of
File: \\..\connectorTELL.SchDoc		Drawn By: Massafferri/F.Sousa - CBPF

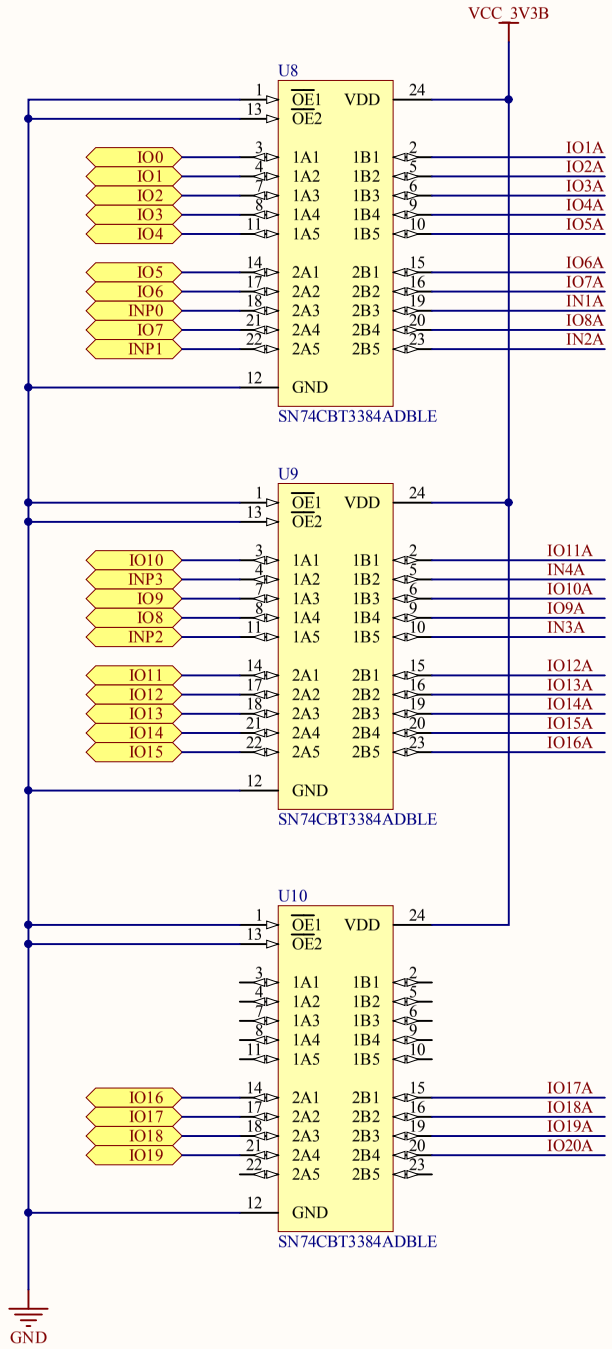

JTAG_0
JTAG_1
JTAG_2
JTAG_3
JTAG_A1
JTAG_A0
VCC_3V3B
U1
74CBTLV3253PGG
1OE
2OE
S0
S1
1A
2A
1B1
1B2
1B3
1B4
2B1
2B2
2B3
2B4
VCC
GND
JTAG_0.TCK
JTAG_1.TCK
JTAG_2.TCK
JTAG_3.TCK
JTAG_0.TMS
JTAG_1.TMS
JTAG_2.TMS
JTAG_3.TMS
JTAG_CONFIG.TCK
JTAG_CONFIG.TMS
U2
74CBTLV3253PGG
JTAG_0.TDI
JTAG_1.TDI
JTAG_2.TDI
JTAG_3.TDI
JTAG_0.TDO
JTAG_1.TDO
JTAG_2.TDO
JTAG_3.TDO
JTAG_CONFIG.TDI
JTAG_CONFIG.TDO
JTAG_CONFIG
P1
Header 5X2
JTAG_CONFIG.TDO
JTAG_CONFIG.TDI
C9
C10
100nF
Title
JTAG Switch
Size
A4
Number
Revision
Date: 8/7/2013
Sheet of
File: \\..\JTAG_Switch.SchDoc
Drawn By: Eng. Leonardo Lessa

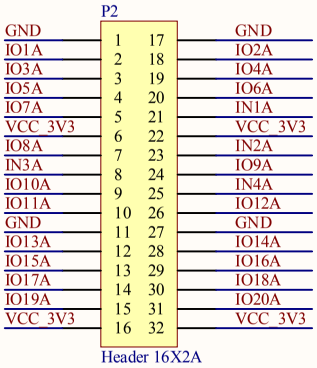


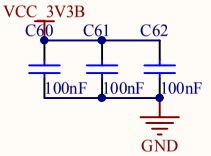
Title	PCIe Switch	
Size: A4	Number:	Revision:
Date: 8/7/2013		Sheet of
File: \\..\PCIe_Switch.SchDoc		Drawn By: Eng. Leonardo Lessa


JTAG_CONFIG
JTAG_CONFIG
PCIe_Downlink
PCIe_Downlink
VCC_3V3B
VCC_3V0
R1 10K
R2 10K
R3 10K
R4 10K
R5 10K
R6 10K
R7 100R
R8 10K
R9 4K70
R10 4K70
R11 330R
R12 330R
R13 2K
U_FPGA_CycloneIV
FPGA_CycloneIV.SchDoc
U4
EPCS16SI8
DATA
DCLK
NCS
ASDI
VCC
GND
JTAG_CONFIG.TCK
JTAG_CONFIG.TMS
JTAG_CONFIG.TDI
JTAG_CONFIG.TDO
TCK
TMS
TDI
TDO
ALTERA_DATA0
ALTERA_DCLK
ALTERA_NCSO
ALTERA_ASDO
CE
PCIe_Downlink.RXp
PCIe_Downlink.RXn
PCIe_Downlink.TXp
PCIe_Downlink.TXn
GXB_RX0p
GXB_RX0n
GXB_TX0p
GXB_TX0n
MSEL0
MSEL1
MSEL2
CONFIG
CONF_DONE
STATUS
VCC_2V5
GND
PCIe_Downlink.CLKp
PCIe_Downlink.CLKn
C19 100nF
C20 100nF
PCIE_REFCLKp
PCIE_REFCLKn
PCIE_WAKE
PCIE_PWRGD
RESTN
CLK25B
RST
VCC_1V8
C13 220nF
XTAL
Y1
U_FPGA_IO
FPGA_IO.SchDoc
IO0
IO1
IO2
IO3
IO4
IO5
IO6
IO7
IO8
IO9
IO10
IO11
IO12
IO13
IO14
IO15
IO16
IO17
IO18
IO19
INP0
INP1
INP2
INP3
D1
D2
LED0603_Blue
VCC_3V3A
LED0
LED1
VREFB7N0
VREFB6N0
DDR_CLKp
DDR_CLKn
DDR_CS
DDR_CKE
DDR_RAS
DDR_CAS
DDR_WE
DDR_DQM
DDR_A0
DDR_A1
DDR_A2
DDR_A3
DDR_A4
DDR_A5
DDR_A6
DDR_A7
DDR_A8
DDR_A9
DDR_A10
DDR_A11
DDR_A12
DDR_BA0
DDR_BA1
DDR_DQ0
DDR_DQ1
DDR_DQ2
DDR_DQ3
DDR_DQ4
DDR_DQ5
DDR_DQ6
DDR_DQ7
DDR_DQS
DDR_ODT
DDR_VREF
VCCA_GXB
VCCH_GXB
VCCL_GXB
VCCD_PLL
VCCA
RREF
U_FPGA_DDR_Memory
FPGA_DDR_Memory.SchDoc
DDR_RDQS
U5
LP2997M
AVIN
PVIN
VDDQ
VTT
VSENSE
VREF
SD
C21
C22
C23
100nF
U6
TPS3103K33DBVR
MR
PFI
RESET
PFO
R14 470R
R15 1K
R16 1K
R17 10K
R18 1K
Q1
BC850B
D3
C24 C25 C26 L2 10nH
C27 C28 C29 L3 10nH
C30 C31 C32 C33 L4 10nH
C34 C35 L5 10nH
C36 L6 10nH
Title
Target FPGA
Size
A3
Number
Revision
Date: 8/7/2013
File: \\..\Target_FPGA.SchDoc
Sheet of
Drawn By: Eng. Leonardo Lessa

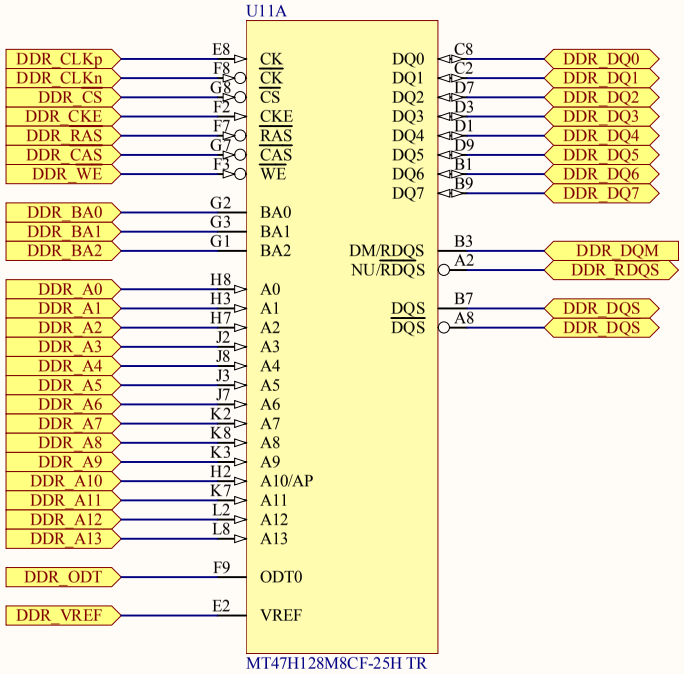

U7A
BANK 3
IO, PLL1_CLKOUTp
IO, PLL1_CLKOUTn
IO, DIFFIO_B1p, (CRC_ERROR)
IO, DIFFIO_B1n, (NCEO)
IO, DIFFIO_B3p, (INIT_DONE)
IO, DIFFIO_B3n, (DQ0B)
IO, (DQS1B/CQ0B#,DPCLK2)
IO, VREFB3N0
L4 M4 N4 N5 M6 N6 L5 L7
IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7
EP4CGX30BF14C7
U7B
BANK 4
IO, DIFFIO_B12p, (DQ0B)
IO, DIFFIO_B12n, (DQ0B)
IO, VREFB4N0
IO, (DQ0B)
IO, DIFFIO_B19p, (DQ0B)
IO, DIFFIO_B19n, (DQS0B/CQ0B,DPCLK5)
IO, DIFFIO_B21p, (DQ0B)
IO, DIFFIO_B21n, (DQ0B)
IO, RUP2, (DQ0B)
IO, RDN2, (DM0B)
IO, PLL3_CLKOUTp
IO, PLL3_CLKOUTn
N8 N9 K8 K9 L9 M9 N10 N11 M11 N12 K10 L11
IO8 IO9 IO10 IO11 IO12 IO13 IO14 IO15 IO16 IO17 IO18 IO19
EP4CGX30BF14C7
U7C
BANK 5
IO, RUP3
IO, RDN3
IO, DIFFIO_R14n, (DQ0R)
IO, DIFFIO_R14p, (DQ0R)
IO, DIFFIO_R13n, (DQ0R)
IO, DIFFIO_R13p, (DQ0R)
IO, VREFB5N0
IO, (DQS1R/CQ0R#,DPCLK7)
IO, DIFFIO_R9n, (DQ0R)
IO, DIFFIO_R9p, (DQ0R)
N13 M13 K12 K11 L13 L12 H12 H10 J13 K13
DDR_A2 DDR_A9 DDR_A1 DDR_A5 DDR_A7 DDR_A12 DDR_BA0 DDR_A8 DDR_A10 DDR_A3
EP4CGX30BF14C7
U7D
BANK 6
IO, DIFFIO_R6n, (DEV_OE)
IO, DIFFIO_R6p, (DQS0R/CQ0R,DPCLK8)
IO, DIFFIO_R5n, (DQ0R)
IO, DIFFIO_R5p, (DQ0R)
IO, VREFB6N0
IO
IO, DIFFIO_R4n, (DEV_CLRn)
IO, DIFFIO_R4p, (DM0R)
IO, DIFFIO_R2n
IO, DIFFIO_R2p
G10 G9 F11 E10 E13 F9 D10 E10 D12 D11
DDR_BA1 DDR_A11 DDR_CKE DDR_A6 VREFB6N0 DDR_A4 DDR_CLKn DDR_CLKp DDR_WE DDR_CAS
EP4CGX30BF14C7
U7E
BANK 7
IO, RUP4, (DQ0T)
IO, RDN4, (DQ0T)
IO, DIFFIO_T18n, (DQ0T)
IO, DIFFIO_T18p, (DQ0T)
IO, DIFFIO_T17n, (DQS0T/CQ0T,DPCLK10)
IO, DIFFIO_T17p, (DQ0T)
IO
IO, VREFB7N0
IO, DIFFIO_T12n, (DQ0T)
IO, DIFFIO_T12p, (DQ0T)
IO, DIFFIO_T11n, (DQ0T)
IO, DIFFIO_T11p, (DM0T)
C11 C12 C13 D13 A13 B13 B11 B10 B8 C8 A11 A12
DDR_DQM DDR_DQ6 DDR_DQ1 DDR_DQ4 DDR_DQS DDR_DQ3 DDR_RAS VREFB7N0 DDR_DQ5 DDR_DQ0 DDR_DQ7 DDR_DQ2
EP4CGX30BF14C7
U7F
BANK 8
IO, VREFB8N0
IO, (DQS1T/CQ0T#,DPCLK13)
IO, PLL2_CLKOUTn
IO, PLL2_CLKOUTp
IO, (CLKUSR)
C6 B6 A7 A8 A6
DDR_A0 DDR_CS LED0 DDR_ODT LED1
EP4CGX30BF14C7
IO and DDR Memory
U7G
RESTN
PCIE_WAKE
PCIE_PWRGD
CLK25B
INP0
INP1
PCIE_REFCLKp
PCIE_REFCLKn
INP2
INP3
G13 H13 F13 F12 A9 A10 E6 E7 J6 J7 M7 N7
CLK4, DIFFCLK_2n
CLK5, DIFFCLK_2p
CLK6, DIFFCLK_3n
CLK7, DIFFCLK_3p
CLK8, DIFFCLK_5n
CLK9, DIFFCLK_5p
CLK10, DIFFCLK_4n,REFCLK1n
CLK11, DIFFCLK_4p,REFCLK1p
CLK12, DIFFCLK_7p,REFCLK0p
CLK13, DIFFCLK_7n,REFCLK0n
CLK14, DIFFCLK_6p
CLK15, DIFFCLK_6n
EP4CGX30BF14C7
R19
10K
GND
Clock and Reset
U7H
GXB_TX0p
GXB_TX0n
J2 J1
GXB_RX0p GXB_TX0p
GXB_RX0n GXB_TX0n
G2 G1
GXB_RX0p
GXB_RX0n
E2 E1
GXB_RX1p GXB_TX1p
GXB_RX1n GXB_TX1n
C2 C1
RREF
L1
RREF0
EP4CGX30BF14C7
R20
10K
GND
U7L
VCCL_GXB
VCCH_GXB
VCCA_GXB
N1 F3 H3
G3
M2 J3
VCCL_GXB
VCCA_GXB
VCCD_PLL
D4 K4 H9 D9
D3 J4 J10
VCCA
VCCD_PLL
EP4CGX30BF14C7
PCIe Transceiver
U7I
ALTERA_DATA0
ALTERA_ASDO
ALTERA_NCSO
A5 B5 C5
IO, (DATA0)
IO, (ASDO)
IO, (NCSO)
TDI
TDO
TCK
TMS
A3 A1 B3 A2
TDI
TDO
TCK
TMS
MSEL0
MSEL1
MSEL2
K5 N3 L3
MSEL0
MSEL1
MSEL2
CE
C4
nCE
ALTERA_DCLK
CONF_DONE
CONFIG
STATUS
A4 J5 D5 K6
DCLK
CONF_DONE
nCONFIG
nSTATUS
EP4CGX30BF14C7
Configuration
VCC_2V5
VCC_1V8
VCC_3V0
VCC_1V2
U7J
VCCIO3
VCCIO4
VCCIO4
VCCIO5
VCCIO5
VCCIO6
VCCIO6
VCCIO7
VCCIO7
VCCIO8
VCCIO9
VCC_CLKIN3A
VCC_CLKIN8A
L6
L8
L10
H11
J11
E11
G11
C10
C9
C7
C3
K7
D7
VCCINT
F7 J8 H7 H5 G8 G6 G4 F5 E8 E4
EP4CGX30BF14C7
C37 C38 C39 C40 C41 C42 C43 C44 C45 C46
C47 C48 C49 C50 C51 C52 C53 C54
C55
C56 C57 C58 C59
100nF
GND
U7K
GND
E9 E5 D8 D6 B9 B7 B4 M1 L2 K2 K1 H2 H1 F2 F1 D2 D1 B2 B1
E3 K3 J9 B12 M12 M10 M8 M5 J12 H8 H6 H4 G12 G7 G5 F8 F6 F4 E12
EP4CGX30BF14C7
U7M
N2 M3
NC
NC
EP4CGX30BF14C7
Power
Title
FPGA Cyclone IV GX
Size
A3
Number
Revision
Date: 8/7/2013
Sheet of
File: \\..\FPGA_CycloneIV.SchDoc
Drawn By: Eng. Leonardo Lessa

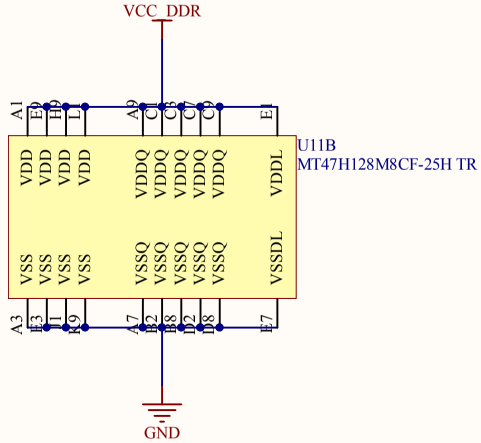


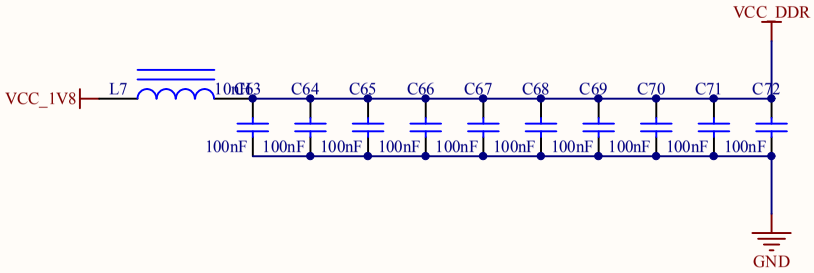




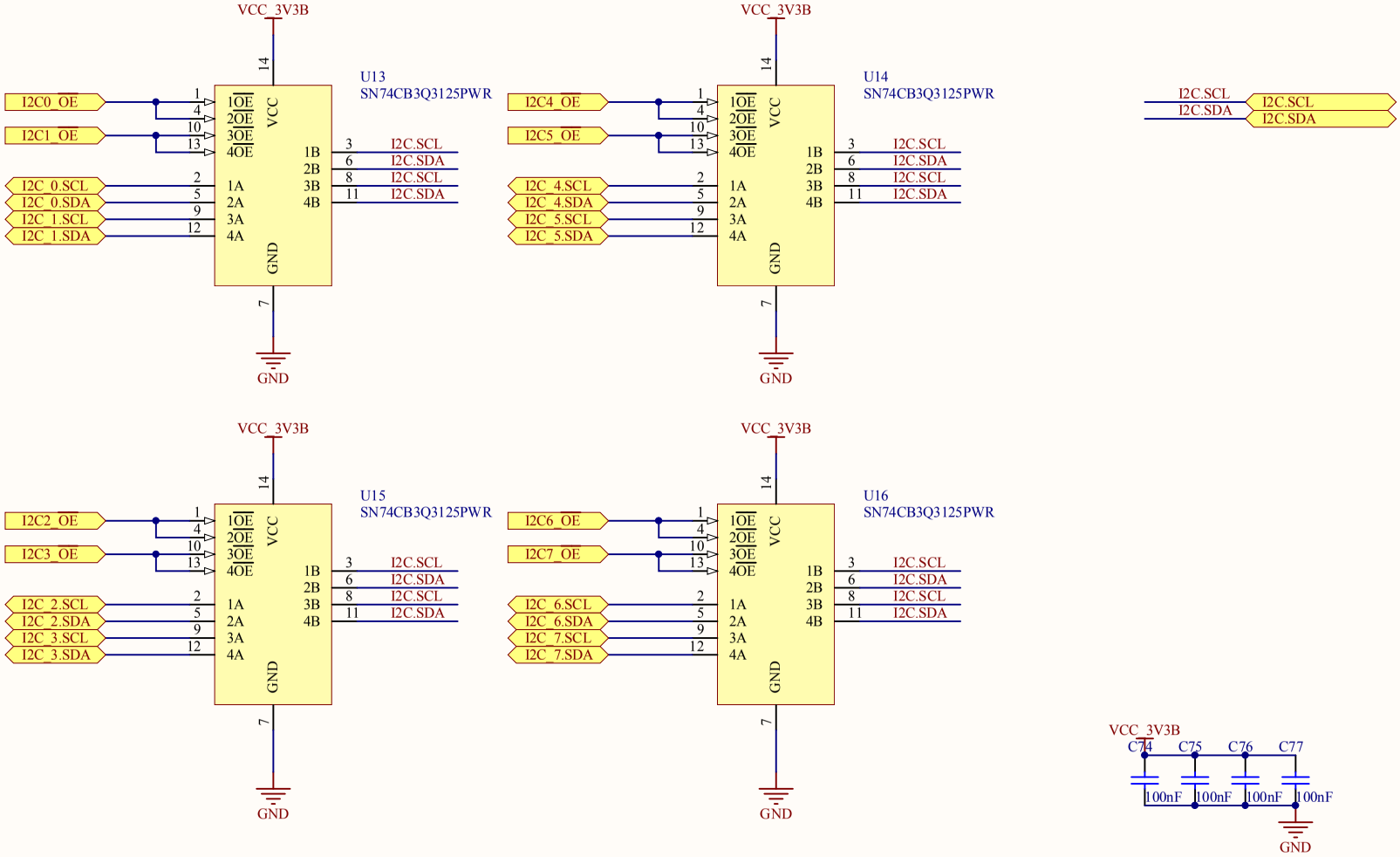
Title: FPGA I/O Header		
Size: A4	Number:	Revision:
Date: 8/7/2013		Sheet of
File: \\..\FPGA_IO.SchDoc		Drawn By: Eng. Leonardo Lessa



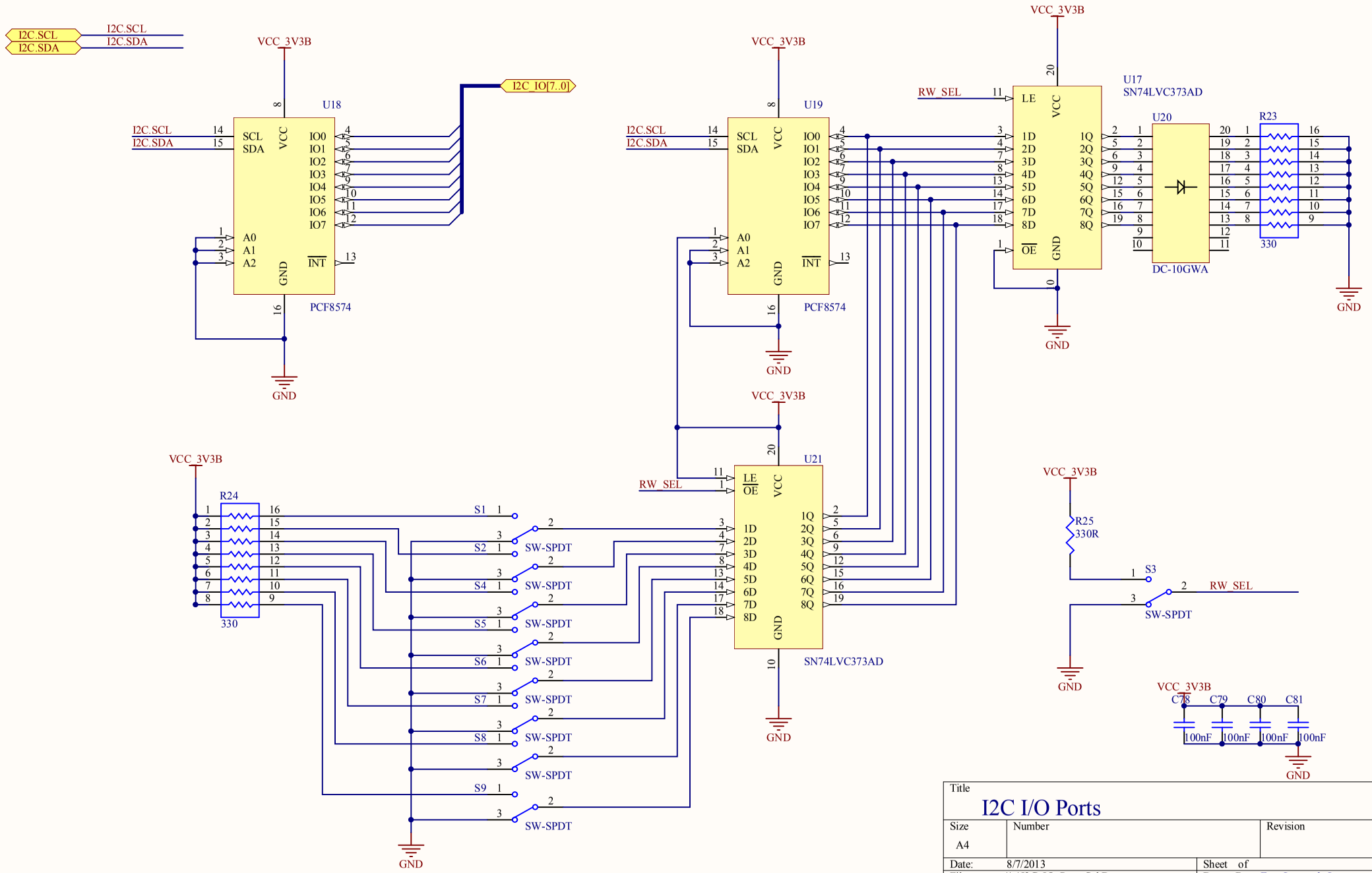




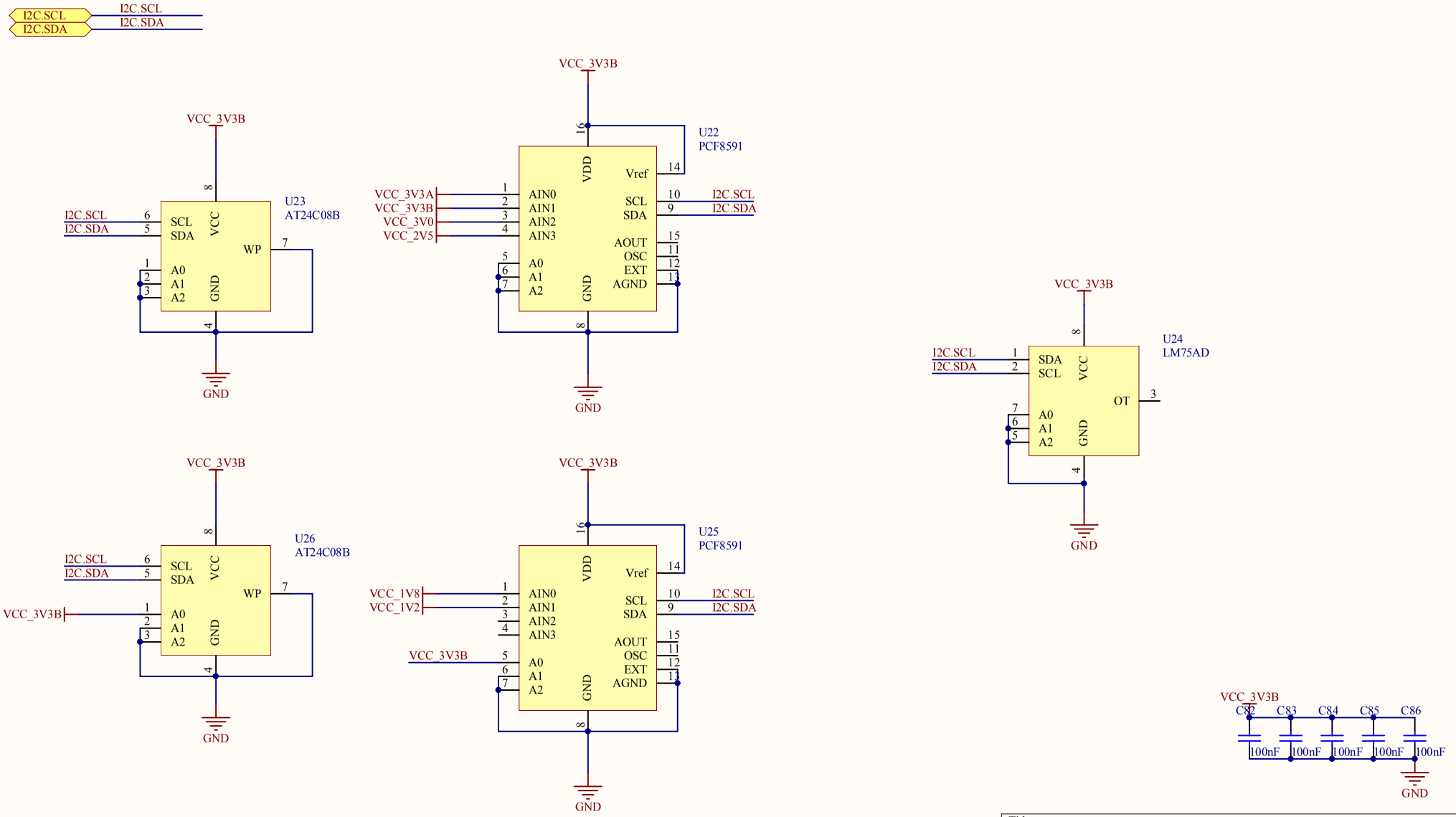
Title: FPGA DDR Memory		
Size: A4	Number:	Revision:
Date: 8/7/2013		Sheet of
File: \\..\FPGA_DDR_Memory.SchDoc		Drawn By: Eng. Leonardo Lessa

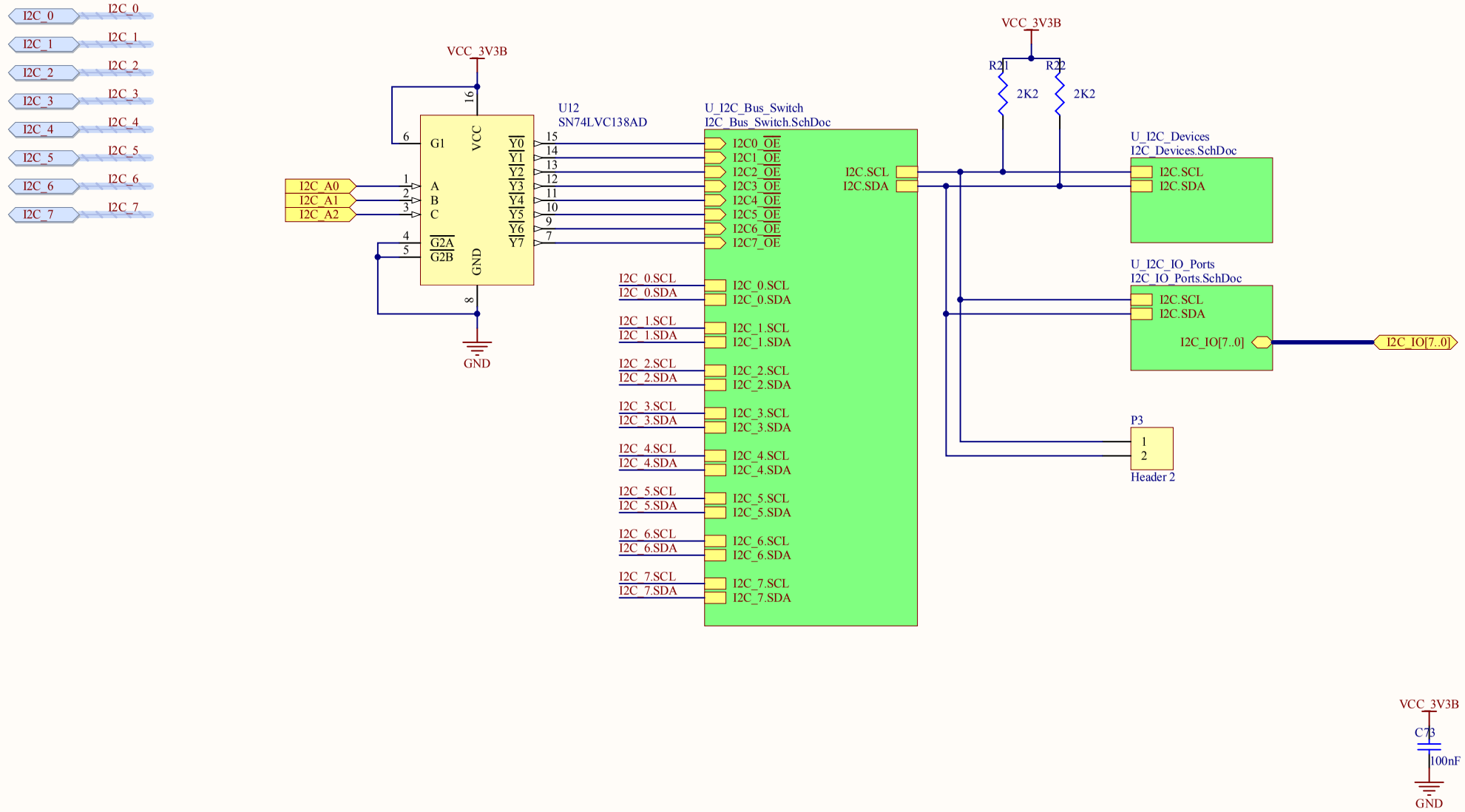


Title	I2C Bus Switch	
Size: A4	Number:	Revision:
Date: 8/7/2013	Sheet of	
File: \\..\I2C_Bus_Switch.SchDoc	Drawn By: Eng. Leonardo Lessa	

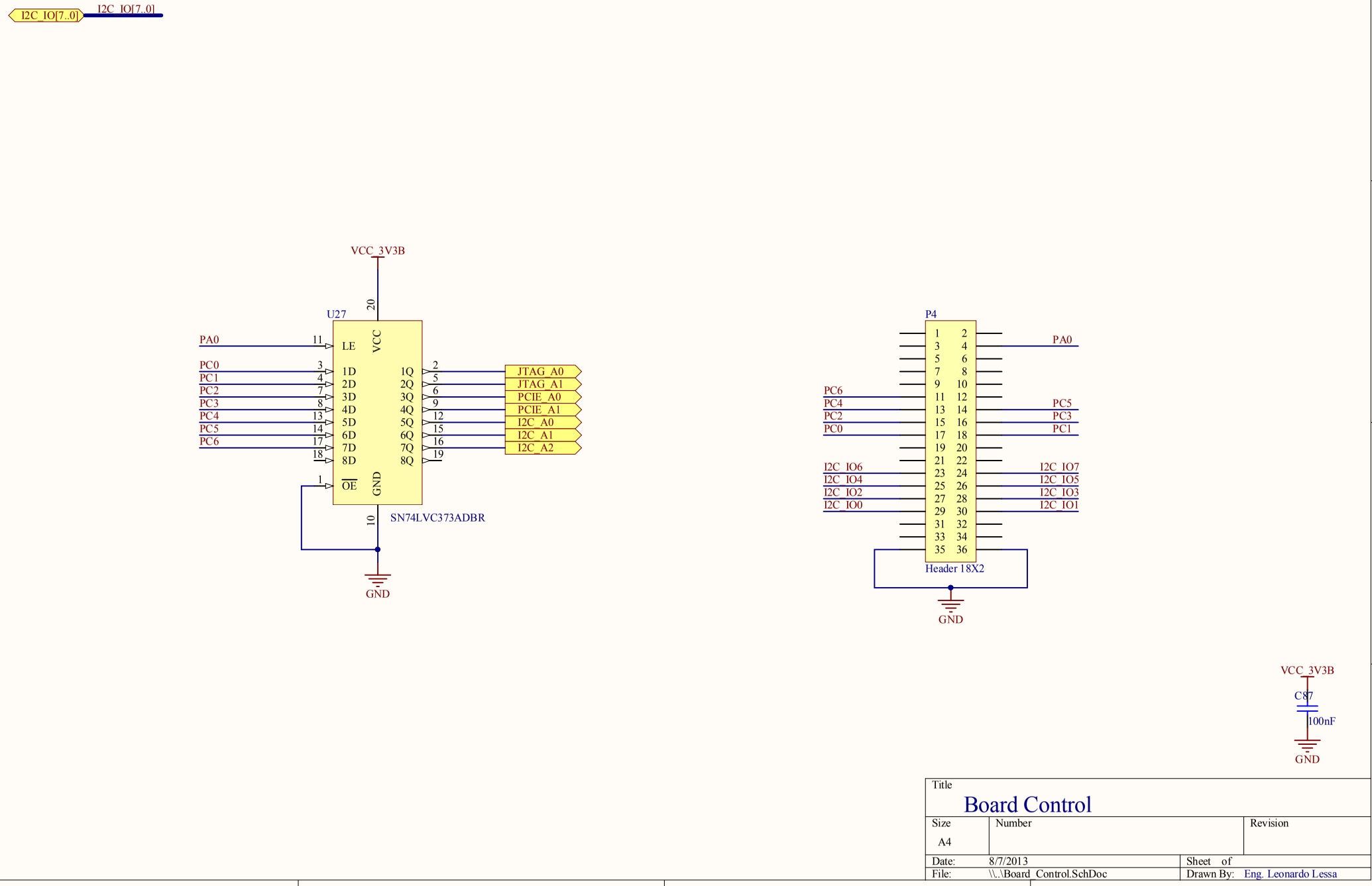


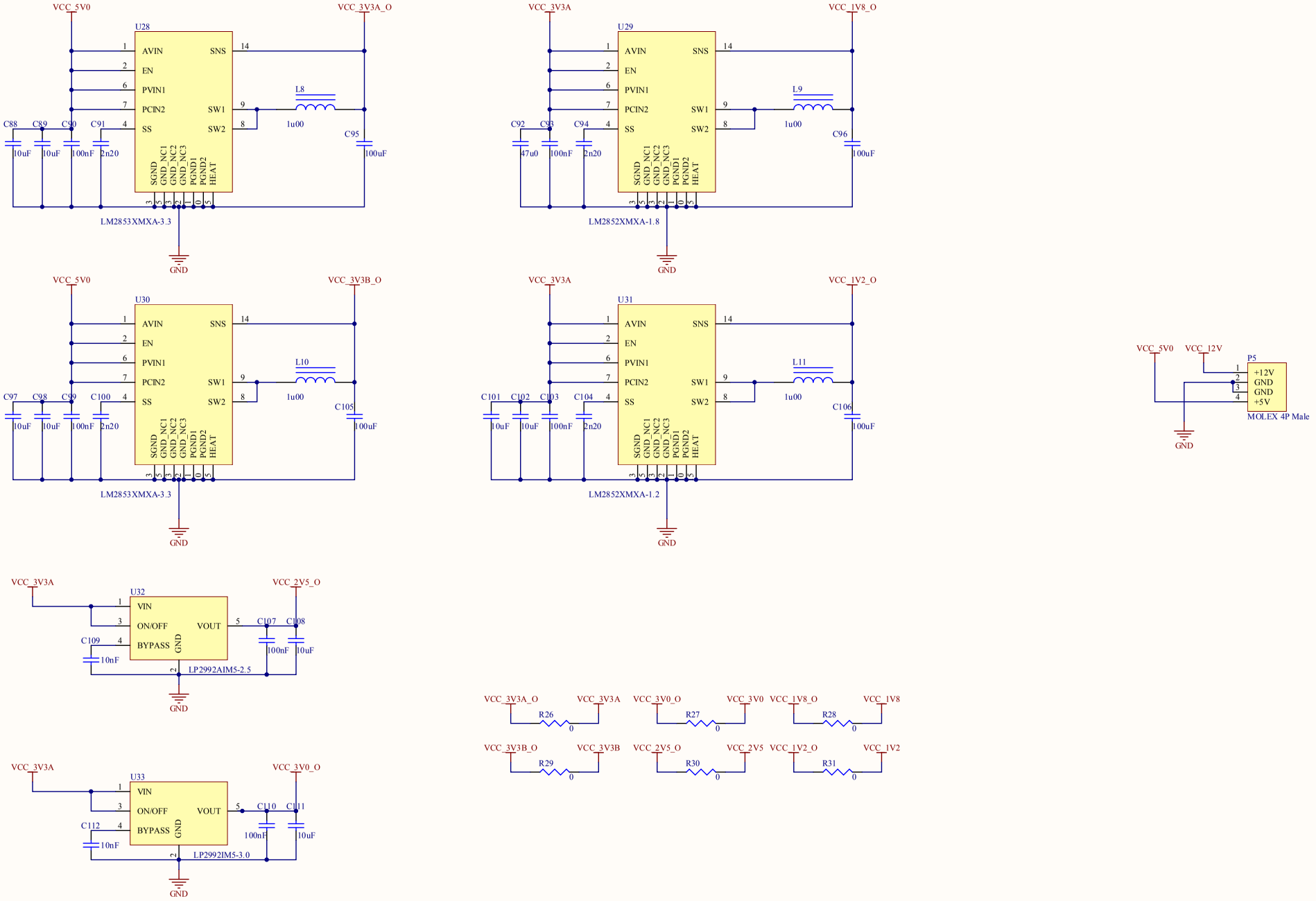
Title	I2C I/O Ports	
Size: A4	Number:	Revision:
Date: 8/7/2013	Sheet of	
File: \\..\I2C_IO_Ports.SchDoc	Drawn By: Eng. Leonardo Lessa	


I2C.SCL
I2C.SDA
I2C.SCL
I2C.SDA
VCC_3V3B
U23
AT24C08B
SCL
SDA
VCC
WP
A0
A1
A2
GND
U22
PCF8591
VDD
Vref
AIN0
AIN1
AIN2
AIN3
AOUT
OSC
EXT
AGND
VCC_3V3A
VCC_3V3B
VCC_3V0
VCC_2V5
U24
LM75AD
SDA
SCL
OT
U26
AT24C08B
U25
PCF8591
VCC_1V8
VCC_1V2
C82
C83
C84
C85
C86
100nF
GND
Title
I2C Devices
Size
A4
Number
Revision
Date:
8/7/2013
Sheet of
File:
\\..\I2C_Devices.SchDoc
Drawn By:
Eng. Leonardo Lessa



Title	I2C Distribution Circuit	
Size: A4	Number:	Revision:
Date: 8/7/2013	Sheet of	
File: \\..\I2C_Distribution.SchDoc	Drawn By: Eng. Leonardo Lessa	


I2C_IO[7..0]
I2C_IO[7..0]
VCC_3V3B
U27
20
PA0
11
LE
VCC
PC0
3
1D
1Q
2
JTAG_A0
PC1
4
2D
2Q
5
JTAG_A1
PC2
7
3D
3Q
6
PCIE_A0
PC3
8
4D
4Q
9
PCIE_A1
PC4
13
5D
5Q
12
I2C_A0
PC5
14
6D
6Q
15
I2C_A1
PC6
17
7D
7Q
16
I2C_A2
18
8D
8Q
19
1
OE
GND
10
SN74LVC373ADBR
GND
P4
PA0
PC6
PC4
PC5
PC2
PC3
PC0
PC1
I2C_IO6
I2C_IO7
I2C_IO4
I2C_IO5
I2C_IO2
I2C_IO3
I2C_IO0
I2C_IO1
Header 18X2
GND
VCC_3V3B
C87
100nF
GND
Title
Board Control
Size
A4
Number
Revision
Date:
8/7/2013
Sheet of
File:
\\..\Board_Control.SchDoc
Drawn By:
Eng. Leonardo Lessa



Title	Power Supply	
Size: A3	Number	Revision
Date: 8/7/2013		Sheet of
File: \\..\Power_Supply.SchDoc		Drawn By: Power Supply

Apêndice B – Código Fonte

driver/pcie_ecs.
driver/i2c-ocores.c
driver/ecs_jtag.c
app/adc_i2c.c
app/temp_acq.sh

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/interrupt.h>

#include <linux/slab.h>
#include <linux/init.h>

#include <linux/i2c.h>
#include <linux/i2c-ocores.h>
#include <linux/platform_device.h>

#include "pcie_ecs.h"
#include "ecs_jtag.h"

static struct ocores_i2c *i2c;
static struct ecs_jtag *jtag;

static struct pci_device_id ecs_ids[] = {
    { PCI_DEVICE(MY_PCI_VENDOR_ID, MY_PCI_DEVICE_ID), },
    { 0, }
};

static struct ocores_i2c_platform_data ecs_i2c_data = {
    .regstep    = 8,        /* eight bytes between registers */
    .clock_khz  = 50000,    /* input clock of 50MHz */
};

static struct ecs_device *ecs_device;
static struct resource i2c_resources[2];
static struct platform_device *ecs_i2c;

extern void ocores_process(struct ocores_i2c *);
extern void init_jtag(struct ecs_jtag *);
extern void rem_jtag(struct ecs_jtag *);

MODULE_DEVICE_TABLE(pci, ecs_ids);

static int ecs_i2c_init(struct ecs_device *ecs_device){

    int status, i;

    if (!(ecs_i2c = (struct platform_device *) kzalloc(sizeof(struct platform_device), GFP_KERNEL))){
        status = -ENOMEM;
        return status;
    }

    i2c_resources[0].start = ecs_device->base_addr_bar0 + I2C_0_OFFT;
    i2c_resources[0].end = (i2c_resources[0].start + I2C_SIZE) - 1;
    i2c_resources[0].flags = IORESOURCE_MEM;

    i2c_resources[1].start = ecs_device->base_addr_bar0 + I2C_1_OFFT;
    i2c_resources[1].end = (i2c_resources[1].start + I2C_SIZE) - 1;
    i2c_resources[1].flags = IORESOURCE_MEM;

    ecs_i2c->name = "ocores-i2c";
```

```
57      ecs_i2c->id = 0;
58      ecs_i2c->dev.platform_data = &ecs_i2c_data;
59      ecs_i2c->resource = i2c_resources;
60      ecs_i2c->num_resources = ARRAY_SIZE(i2c_resources);
61
62      request_mem_region(ecs_device->base_addr_bar1, ecs_device->size_bar1, DRV_NAME);
63      ecs_device->cra = ioremap_nocache(ecs_device->base_addr_bar1 + CRA_OFFT, CRA_SIZE
    );
64
65      status = platform_device_register(ecs_i2c);
66
67      if (!(i2c = (struct ocores_i2c *) kzalloc(sizeof(struct ocores_i2c) * ARRAY_SIZE(
    i2c_resources), GFP_KERNEL))){
68          status = -ENOMEM;
69          return status;
70      }
71
72      i2c = platform_get_drvdata(ecs_i2c);
73      for (i=0; i < ARRAY_SIZE(i2c_resources); i++){
74          printk(KERN_DEBUG "pcie_ecs: i2c drv [%d] addr: 0x%x\n", i, (unsigned int)&
    i2c[i]);
75      }
76
77      return status;
78  }
79
80  static int ecs_jtag_init(struct ecs_device *ecs_device){
81
82      int status;
83
84      if (!(jtag = (struct ecs_jtag *) kzalloc(sizeof(struct ecs_jtag), GFP_KERNEL))){
85          status = -ENOMEM;
86          return status;
87      }
88
89      jtag->start = ecs_device->base_addr_bar0 + PIO0_OFFT;
90      jtag->size = PIO0_SIZE;
91
92
93      // include an error handler
94      init_jtag(jtag);
95
96      return 0;
97
98  }
99
100 static irqreturn_t pcie_irq_handler(int irq, void *data){
101
102
103     int ret;
104
105     struct ecs_device *ecs_device = (struct ecs_device *)data;
106     if (!ecs_device)
107         return IRQ_NONE;
108
109     // Disable IRQ From CRA
110     iowrite32(0<<7, (u32*)(ecs_device->cra + CRA_IER));
```

```
111
112         ret = ioread32(ecs_device->cra + CRA_ISR);
113         ret = (ret & 0x3F00) >> 8;
114
115     //  printk (KERN_DEBUG "CRA Status : 0x%x\n", ret);
116
117         if (ret == 1){
118             ocores_process(&i2c[0]);
119         }
120
121         if (ret == 2){
122             ocores_process(&i2c[1]);
123         }
124
125     //  printk (KERN_DEBUG "IRQ Handled\n");
126
127         // Enable IRQ From CRA
128         iowrite32(1<<7, (u32*)(ecs_device->cra + CRA_IER));
129         return IRQ_HANDLED;
130
131     }
132
133     static int ecs_probe(struct pci_dev *dev, const struct pci_device_id *id) {
134
135         int status;
136
137         printk(KERN_DEBUG "\tProbing ECS Device\n");
138         if (pci_enable_device(dev)) {
139             printk(KERN_ERR "Failed to enable PCI device\n");
140             status = -ENODEV;
141             goto error_no_mem;
142         }
143
144         //Allocating a pointer on Kernel Space for ECS Device Driver
145         if (!(ecs_device = (struct ecs_device *) kzalloc(sizeof(struct ecs_device),
    GFP_KERNEL))){
146             status = -ENOMEM;
147             goto error_no_mem;
148         }
149
150         strcpy(ecs_device->name, DRV_NAME);
151         ecs_device->pci_dev = dev;
152
153         if (!(pci_resource_start(dev, 0))) {
154             printk(KERN_ERR "\t\tBAR0 disabled! Giving up.\n");
155             status = -ENOMEM;
156             goto error_base0;
157         }
158
159         if (!(pci_resource_start(dev, 1))) {
160             printk(KERN_ERR "\t\tBAR1 disabled! Giving up.\n");
161             status = -ENOMEM;
162             goto error_base0;
163         }
164
165         ecs_device->base_addr_bar0 = pci_resource_start(dev, 0);
166         ecs_device->size_bar0 = pci_resource_len(dev, 0);
```

```
167       ecs_device->base_addr_bar1 = pci_resource_start(dev, 1);
168       ecs_device->size_bar1 = pci_resource_len(dev, 1);
169
170       printk(KERN_DEBUG "\t Base Address Register(BAR0) @ 0x%x\n\t Size of %u bytes\n",
171                                   ecs_device->base_addr_bar0,
172                                   ecs_device->size_bar0);
173
174       printk(KERN_DEBUG "\t Base Address Register(BAR1) @ 0x%x\n\t Size of %u bytes\n",
175                                   ecs_device->base_addr_bar1,
176                                   ecs_device->size_bar1);
177
178       request_irq(dev->irq, pcie_irq_handler, IRQF_SHARED, DRV_NAME, (void *)ecs_device
    );
179       printk(KERN_DEBUG "Succesfully requested IRQ #%d with dev_id 0x%p\n", dev->irq,
    ecs_device);
180
181       ecs_i2c_init(ecs_device);
182       ecs_jtag_init(ecs_device);
183
184       // just a test with the timer
185       request_mem_region(ecs_device->base_addr_bar0 + TIMER0_OFFT, TIMER0_SIZE ,
    DRV_NAME);
186       ecs_device->timer0 = ioremap_nocache(ecs_device->base_addr_bar0 + TIMER0_OFFT,
    TIMER0_SIZE);
187
188       iowrite32(1<<7, (u32*)(ecs_device->cra + CRA_IER));
189       iowrite16((u16)0x08, (u16*)(ecs_device->timer0 + TIMER0_CRR));
190
191       return 0;
192
193   error_no_mem:
194       pci_disable_device(dev);
195   error_base0:
196       kfree(ecs_device);
197       return 0;
198
199
200   }
201
202   static void ecs_remove(struct pci_dev *dev) {
203
204       /* clean up any allocated resources and stuff here.
205        * like call release_region();
206        */
207
208       platform_device_del(ecs_i2c);
209       rem_jtag(jtag);
210       free_irq(dev->irq, ecs_device);
211       iounmap(ecs_device->cra);
212       pci_disable_device(dev);
213       kfree(ecs_device);
214
215   }
216
217   static struct pci_driver pci_driver = {
218       .name = DRV_NAME,
219       .id_table = ecs_ids,
```

```
220         .probe = ecs_probe,
221         .remove = ecs_remove,
222     };
223 
224     static int __init pcie_ecs_init(void) {
225 
226         printk(KERN_INFO "PCIe ECS Driver Loaded...\n");
227         return pci_register_driver(&pci_driver);
228     }
229 
230     static void __exit pcie_ecs_exit(void) {
231 
232         printk(KERN_INFO "PCIe ECS Driver Unloaded...\n");
233         pci_unregister_driver(&pci_driver);
234     }
235 
236     module_init(pcie_ecs_init);
237     module_exit(pcie_ecs_exit);
238 
239     MODULE_AUTHOR("Leonardo Lessa <llessa@cbpf.br>");
240     MODULE_DESCRIPTION("PCIe Driver for ECS Device");
241     MODULE_LICENSE("GPL");
242 
```

```c
/*
 * i2c-ocores.c: I2C bus driver for OpenCores I2C controller
 * (http://www.opencores.org/projects.cgi/web/i2c/overview).
 *
 * Peter Korsgaard <jacmet@sunsite.dk>
 *
 * This file is licensed under the terms of the GNU General Public License
 * version 2.  This program is licensed "as is" without any warranty of any
 * kind, whether express or implied.
 *
 * Modified by Leonardo Lessa <llessa@cern.ch> for CBPF ECS
 *
 *
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/platform_device.h>
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/wait.h>
#include <linux/i2c-ocores.h>
#include <asm/io.h>

struct ocores_i2c {
    void __iomem *base;
    int regstep;
    wait_queue_head_t wait;
    struct i2c_adapter adap;
    struct i2c_msg *msg;
    int pos;
    int nmsgs;
    int state; /* see STATE_ */
    int clock_khz;
};

/* registers */
#define OCI2C_PRELOW        0
#define OCI2C_PREHIGH       1
#define OCI2C_CONTROL       2
#define OCI2C_DATA      3
#define OCI2C_CMD       4 /* write only */
#define OCI2C_STATUS        4 /* read only, same address as OCI2C_CMD */

#define OCI2C_CTRL_IEN      0x40
#define OCI2C_CTRL_EN       0x80

#define OCI2C_CMD_START     0x91
#define OCI2C_CMD_STOP      0x41
#define OCI2C_CMD_READ      0x21
#define OCI2C_CMD_WRITE     0x11
#define OCI2C_CMD_READ_ACK  0x21
#define OCI2C_CMD_READ_NACK 0x29
#define OCI2C_CMD_IACK      0x01

```

```
58  #define OCI2C_STAT_IF       0x01
59  #define OCI2C_STAT_TIP      0x02
60  #define OCI2C_STAT_ARBLOST  0x20
61  #define OCI2C_STAT_BUSY     0x40
62  #define OCI2C_STAT_NACK     0x80
63
64  #define STATE_DONE      0
65  #define STATE_START     1
66  #define STATE_WRITE     2
67  #define STATE_READ      3
68  #define STATE_ERROR     4
69
70  static inline void oc_setreg(struct ocores_i2c *i2c, int reg, u8 value)
71  {
72      iowrite8(value, i2c->base + reg * i2c->regstep);
73  }
74
75  static inline u8 oc_getreg(struct ocores_i2c *i2c, int reg)
76  {
77      return ioread8(i2c->base + reg * i2c->regstep);
78  }
79
80  static void ocores_process(struct ocores_i2c *i2c)
81  {
82      struct i2c_msg *msg = i2c->msg;
83      u8 stat = oc_getreg(i2c, OCI2C_STATUS);
84
85      if ((i2c->state == STATE_DONE) || (i2c->state == STATE_ERROR)) {
86          /* stop has been sent */
87          oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_IACK);
88          wake_up(&i2c->wait);
89          return;
90      }
91
92      /* error? */
93      if (stat & OCI2C_STAT_ARBLOST) {
94          i2c->state = STATE_ERROR;
95          oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_STOP);
96          return;
97      }
98
99      if ((i2c->state == STATE_START) || (i2c->state == STATE_WRITE)) {
100         i2c->state =
101             (msg->flags & I2C_M_RD) ? STATE_READ : STATE_WRITE;
102
103         if (stat & OCI2C_STAT_NACK) {
104             i2c->state = STATE_ERROR;
105             oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_STOP);
106             return;
107         }
108     } else
109         msg->buf[i2c->pos++] = oc_getreg(i2c, OCI2C_DATA);
110
111     /* end of msg? */
112     if (i2c->pos == msg->len) {
113         i2c->nmsgs--;
114         i2c->msg++;
```

```
115          i2c->pos = 0;
116          msg = i2c->msg;
117
118          if (i2c->nmsgs) {   /* end? */
119              /* send start? */
120              if (!(msg->flags & I2C_M_NOSTART)) {
121                  u8 addr = (msg->addr << 1);
122
123                  if (msg->flags & I2C_M_RD)
124                      addr |= 1;
125
126                  i2c->state = STATE_START;
127
128                  oc_setreg(i2c, OCI2C_DATA, addr);
129                  oc_setreg(i2c, OCI2C_CMD,  OCI2C_CMD_START);
130                  return;
131              } else
132                  i2c->state = (msg->flags & I2C_M_RD)
133                      ? STATE_READ : STATE_WRITE;
134          } else {
135              i2c->state = STATE_DONE;
136              oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_STOP);
137              return;
138          }
139      }
140
141      if (i2c->state == STATE_READ) {
142          oc_setreg(i2c, OCI2C_CMD, i2c->pos == (msg->len-1) ?
143                OCI2C_CMD_READ_NACK : OCI2C_CMD_READ_ACK);
144      } else {
145          oc_setreg(i2c, OCI2C_DATA, msg->buf[i2c->pos++]);
146          oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_WRITE);
147      }
148
149  }
150
151  EXPORT_SYMBOL(ocores_process);
152
153  /*
154  static irqreturn_t ocores_isr(int irq, void *dev_id)
155  {
156      struct ocores_i2c *i2c = dev_id;
157
158      ocores_process(i2c);
159
160      return IRQ_HANDLED;
161  }
162  */
163
164  static int ocores_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
165  {
166      struct ocores_i2c *i2c = i2c_get_adapdata(adap);
167
168      i2c->msg = msgs;
169      i2c->pos = 0;
170      i2c->nmsgs = num;
171      i2c->state = STATE_START;
```

```
172
173         oc_setreg(i2c, OCI2C_DATA,
174                 (i2c->msg->addr << 1) |
175                 ((i2c->msg->flags & I2C_M_RD) ? 1:0));
176
177         oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_START);
178
179         if (wait_event_timeout(i2c->wait, (i2c->state == STATE_ERROR) ||
180                         (i2c->state == STATE_DONE), HZ))
181             return (i2c->state == STATE_DONE) ? num : -EIO;
182         else
183             return -ETIMEDOUT;
184     }
185
186     static void ocores_init(struct ocores_i2c *i2c)
187     {
188         int prescale;
189         u8 ctrl = oc_getreg(i2c, OCI2C_CONTROL);
190
191         /* make sure the device is disabled */
192         oc_setreg(i2c, OCI2C_CONTROL, ctrl & ~(OCI2C_CTRL_EN|OCI2C_CTRL_IEN));
193
194         prescale = (i2c->clock_khz / (5*100))-1;
195         oc_setreg(i2c, OCI2C_PRELOW, prescale & 0xff);
196         oc_setreg(i2c, OCI2C_PREHIGH, prescale >> 8);
197
198         /* Init the device */
199         oc_setreg(i2c, OCI2C_CMD, OCI2C_CMD_IACK);
200         oc_setreg(i2c, OCI2C_CONTROL, ctrl | OCI2C_CTRL_IEN | OCI2C_CTRL_EN);
201
202     }
203
204
205     static u32 ocores_func(struct i2c_adapter *adap)
206     {
207         return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL;
208     }
209
210     static const struct i2c_algorithm ocores_algorithm = {
211         .master_xfer    = ocores_xfer,
212         .functionality  = ocores_func,
213     };
214
215     static struct i2c_adapter ocores_adapter = {
216         .owner      = THIS_MODULE,
217         .name       = "i2c-ocores",
218         .class      = I2C_CLASS_HWMON | I2C_CLASS_SPD,
219         .algo       = &ocores_algorithm,
220     };
221
222
223     static int __devinit ocores_i2c_probe(struct platform_device *pdev)
224     {
225
226         struct ocores_i2c *i2c;
227         struct ocores_i2c_platform_data *pdata;
228         struct resource **res;
```

```
229     int ret, i;
230
231     if(!(res = (struct resource **) kzalloc(sizeof(struct resource *) * pdev->
    num_resources, GFP_KERNEL))){
232         ret = -ENOMEM;
233         return ret;
234     }
235
236     if(!(i2c = (struct ocores_i2c *) kzalloc(sizeof(struct ocores_i2c) * pdev->
    num_resources, GFP_KERNEL))){
237         ret = -ENOMEM;
238         return ret;
239     }
240
241     platform_set_drvdata(pdev, i2c);
242
243     pdata = (struct ocores_i2c_platform_data*) pdev->dev.platform_data;
244     if (!pdata)
245         return -ENODEV;
246
247
248     for(i=0; i<pdev->num_resources; i++) {
249
250         res[i] = platform_get_resource(pdev, IORESOURCE_MEM, i);
251         if (!res[i])
252             return -ENODEV;
253
254         if (!request_mem_region(res[i]->start, resource_size(res[i]),
255                 pdev->name)) {
256             dev_err(&pdev->dev, "Memory region busy\n");
257             ret = -EBUSY;
258             goto request_mem_failed;
259         }
260
261         i2c[i].base = ioremap_nocache(res[i]->start, resource_size(res[i]));
262         if (!i2c[i].base) {
263             dev_err(&pdev->dev, "Unable to map registers\n");
264             ret = -EIO;
265             goto map_failed;
266         }
267
268         i2c[i].regstep = pdata->regstep;
269         i2c[i].clock_khz = pdata->clock_khz;
270         ocores_init(&i2c[i]);
271
272         init_waitqueue_head(&i2c[i].wait);
273
274         printk(KERN_DEBUG "i2c_ocores: i2c drv addr: 0x%x\n",(unsigned int)&i2c[i]);
275
276         i2c[i].adap = ocores_adapter;
277         i2c_set_adapdata(&i2c[i].adap, &i2c[i]);
278         i2c[i].adap.dev.parent = &pdev->dev;
279
280         ret = i2c_add_adapter(&i2c[i].adap);
281         if (ret) {
282             dev_err(&pdev->dev, "Failed to add adapter\n");
283         }
```

```
284
285         }
286
287         for (i = 0; i < pdata->num_devices; i++)
288             i2c_new_device(&i2c->adap, pdata->devices + i);
289
290         return 0;
291
292     map_failed:
293         release_mem_region(res[i]->start, resource_size(res[i]));
294     request_mem_failed:
295         kfree(i2c);
296
297         return ret;
298
299
300     }
301
302     static int __devexit ocores_i2c_remove(struct platform_device* pdev)
303     {
304         struct ocores_i2c *i2c = platform_get_drvdata(pdev);
305         struct resource *res;
306         int i;
307
308         /* disable i2c logic */
309         oc_setreg(i2c, OCI2C_CONTROL, oc_getreg(i2c, OCI2C_CONTROL)
310               & ~(OCI2C_CTRL_EN|OCI2C_CTRL_IEN));
311
312         /* remove adapter & data */
313
314         for(i=0; i<pdev->num_resources; i++) {
315             i2c_del_adapter(&i2c[i].adap);
316         }
317
318         platform_set_drvdata(pdev, NULL);
319
320         iounmap(i2c->base);
321
322         for(i=0; i<pdev->num_resources; i++) {
323             res = platform_get_resource(pdev, IORESOURCE_MEM, i);
324             if (res)
325                 release_mem_region(res->start, resource_size(res));
326         }
327
328         kfree(i2c);
329
330         return 0;
331     }
332
333     #ifdef CONFIG_PM
334     static int ocores_i2c_suspend(struct platform_device *pdev, pm_message_t state)
335     {
336         struct ocores_i2c *i2c = platform_get_drvdata(pdev);
337         u8 ctrl = oc_getreg(i2c, OCI2C_CONTROL);
338
339         /* make sure the device is disabled */
340         oc_setreg(i2c, OCI2C_CONTROL, ctrl & ~(OCI2C_CTRL_EN|OCI2C_CTRL_IEN));
```

```
341
342         return 0;
343     }
344
345     static int ocores_i2c_resume(struct platform_device *pdev)
346     {
347         struct ocores_i2c *i2c = platform_get_drvdata(pdev);
348
349         ocores_init(i2c);
350
351         return 0;
352     }
353     #else
354     #define ocores_i2c_suspend  NULL
355     #define ocores_i2c_resume   NULL
356     #endif
357
358     /* work with hotplug and coldplug */
359     MODULE_ALIAS("platform:ocores-i2c");
360
361     static struct platform_driver ocores_i2c_driver = {
362         .probe   = ocores_i2c_probe,
363         .remove  = __devexit_p(ocores_i2c_remove),
364         .suspend = ocores_i2c_suspend,
365         .resume  = ocores_i2c_resume,
366         .driver  = {
367             .owner = THIS_MODULE,
368             .name = "ocores-i2c",
369         },
370     };
371
372     static int __init ocores_i2c_init(void)
373     {
374         printk(KERN_DEBUG "I2C ocores driver loaded\n");
375         return platform_driver_register(&ocores_i2c_driver);
376     }
377
378     static void __exit ocores_i2c_exit(void)
379     {
380         platform_driver_unregister(&ocores_i2c_driver);
381     }
382
383     module_init(ocores_i2c_init);
384     module_exit(ocores_i2c_exit);
385
386     MODULE_AUTHOR("Peter Korsgaard <jacmet@sunsite.dk>");
387     MODULE_DESCRIPTION("OpenCores I2C bus driver");
388     MODULE_LICENSE("GPL");
389
```

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/random.h>
#include <linux/errno.h>

#include <asm/io.h>
#include <linux/ioport.h>
#include <linux/miscdevice.h>

#include "ecs_jtag.h"

static struct miscdevice ecs_jtag_dev;
struct ecs_jtag *jtag_dev;

static int jtag_open(struct inode *inode, struct file *file)
{
    // OOOOIIII = 0x0f
    iowrite8((u8)0x0f, jtag_dev->base + 0x08);

    return 0;
}

static int jtag_release(struct inode *inode, struct file *file)
{
    //iowrite8((u8)0x00, jtag_dev->base);
    return 0;
}
// read data on JTAG device
ssize_t jtag_read(struct file *filp, char __user *buf, size_t count,
                  loff_t *f_pos)
{
    unsigned int data;


    data = ioread8(jtag_dev->base + PIO_DATA);
    copy_to_user(buf, &data, count);

    return count;
}
// write data on JTAG device
ssize_t jtag_write(struct file *filp, const char __user *buf, size_t count,
                  loff_t *f_pos)
{
    int retval = 0;
//  int tdi, tms, tck;

    char *k_buffer;
    k_buffer = (char *) kzalloc(count * sizeof(char), GFP_KERNEL);

    if(copy_from_user(k_buffer, buf, count)){
        retval = -EFAULT;
        goto out;
    }
```

```
58
59  //  tdi = *k_buffer & 0x04 ? 1 : 0;
60  //  tms = *k_buffer & 0x02 ? 1 : 0;
61  //  tck = *k_buffer & 0x01 ? 1 : 0;
62
63  //  if (tck == 1)
64  //      printk(KERN_DEBUG "TDI = %d | TMS = %d | TCK = %d \n", tdi, tms, tck);
65
66      iowrite8((u8) *k_buffer, jtag_dev->base);
67
68      retval = count;
69
70  out:
71      kfree(k_buffer);
72      return retval;
73  }
74
75  static struct file_operations ecs_jtag_fops = {
76      .owner   = THIS_MODULE,
77      .open    = jtag_open,
78      .read    = jtag_read,
79      .write   = jtag_write,
80      .release = jtag_release
81  };
82
83  static int init_jtag(struct ecs_jtag *jtag)
84  {
85
86          int retval;
87          printk(KERN_DEBUG "pcie_ecs: jtag start base addr: 0x%x\n",
88                                (unsigned int)jtag->start);
89          printk(KERN_DEBUG "pcie_ecs: jtag size: 0x%x\n",
90                                (unsigned int)jtag->size);
91
92          ecs_jtag_dev.minor = MISC_DYNAMIC_MINOR;
93          ecs_jtag_dev.name = "jtag";
94          ecs_jtag_dev.fops = &ecs_jtag_fops;
95          retval = misc_register(&ecs_jtag_dev);
96          if (retval)
97              return retval;
98
99          if (!request_mem_region(jtag->start, jtag->size,
100             ecs_jtag_dev.name)) {
101         printk(KERN_DEBUG "Memory region busy\n");
102         retval = -EBUSY;
103         goto request_mem_failed;
104
105     }
106
107     jtag->base = ioremap_nocache(jtag->start, jtag->size);
108         printk(KERN_DEBUG "pcie_ecs: jtag start base addr: 0x%x\n",
109                               (unsigned int)jtag->base);
110     jtag_dev = jtag;
111
112     return 0;
113
114 request_mem_failed:
```

```
115
116         return retval;
117
118     }
119
120     EXPORT_SYMBOL(init_jtag);
121
122     static int rem_jtag(struct ecs_jtag *jtag)
123     {
124         iounmap(jtag->base);
125         release_mem_region(jtag->start, jtag->size);
126         misc_deregister(&ecs_jtag_dev);
127         return 0;
128     }
129
130     EXPORT_SYMBOL(rem_jtag);
131
132     static int ecs_jtag_init(void)
133     {
134         printk(KERN_ALERT "ECS JTAG Driver Loaded...\n");
135         return 0;
136     }
137
138     static void ecs_jtag_exit(void)
139     {
140         printk(KERN_ALERT "ECS JTAG Driver Unloaded...\n");
141     }
142
143     module_init(ecs_jtag_init);
144     module_exit(ecs_jtag_exit);
145
146     MODULE_AUTHOR("Leonardo Lessa <llessa@cbpf.br>");
147     MODULE_DESCRIPTION("PIO JTAG Driver");
148     MODULE_LICENSE("GPL");
149
```

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/interrupt.h>

#include <linux/miscdevice.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#include <linux/slab.h>
#include <linux/init.h>

#include <linux/dma-mapping.h>

#include "pcie_dma.h"

static struct dma_device *dma_device;

static DECLARE_WAIT_QUEUE_HEAD(wq);

static struct pci_device_id dma_ids[] = {
    { PCI_DEVICE(MY_PCI_VENDOR_ID, MY_PCI_DEVICE_ID), },
    { 0, }
};

MODULE_DEVICE_TABLE(pci, dma_ids);

static int dma_init(void) {

    // SOFTWARE RESET DMA Controller
    unsigned int control, status = 0;
    IOWR_DMA(SOFTRESET, dma_device->dma_base_addr, DMA_CONTROL);
    IOWR_DMA(SOFTRESET, dma_device->dma_base_addr, DMA_CONTROL);

    // set data width and Interruption on Lenght.
    IOWR_DMA(DOUBLEWORD | I_EN| LEEN, dma_device->dma_base_addr, DMA_CONTROL);

    control = IORD_DMA(dma_device->dma_base_addr, DMA_CONTROL);
    status = IORD_DMA(dma_device->dma_base_addr, DMA_STATUS);

    printk (KERN_DEBUG "\t DMA: Control Register 0x%x\n", control);
    printk (KERN_DEBUG "\t DMA: Status Register 0x%x\n", status);

    return 0;

}

static int dma_transfer (struct dma_device *dev, int write, void *buffer,
             size_t count) {
    unsigned int ret;

    ret = IORD_DMA(dev->dma_base_addr, DMA_CONTROL);
    IOWR_DMA(ret & 0xFFFFFFF7, dma_device->dma_base_addr, DMA_CONTROL);

    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
```

```
58
59      iowrite32((dev->bus_addr & 0xFFFFFFFC), dev->cra_base_addr + MM_TABLE);
60
61      // 64Bits Address translation
62  //  iowrite32((dev->bus_addr & 0xFFFFFFFC) | 0x1, dev->cra_base_addr + MM_TABLE);
63  //  iowrite32(0x00000000, dev->cra_base_addr + MM_TABLE + 4);
64
65      ret = ioread32(dev->cra_base_addr + MM_TABLE);
66      printk(KERN_DEBUG "\t\t CRA: LOW MM_TABLE = 0x%x\n", ret);
67
68      // 64Bits Address translation
69  //  ret = ioread32(dev->cra_base_addr + MM_TABLE + 4);
70  //  printk(KERN_DEBUG "\t\t CRA: HIGH MM_TABLE = 0x%x\n", ret);
71
72      printk(KERN_DEBUG "\t\t DMA: phy_addr = 0x%x\n", dev->bus_addr);
73      printk(KERN_DEBUG "\t\t DMA: buffer = 0x%x\n", (unsigned int)dev->k_buffer);
74      printk(KERN_DEBUG "\t\t DMA: size = %d Bytes\n", (int)count);
75      printk(KERN_DEBUG "\t\t DMA: dma_dir = %s\n", write ? "Write" : "Read");
76
77      if(!write){
78          IOWR_DMA(TX + (dev->bus_addr & 0xFFFFF), dma_device->dma_base_addr,
    DMA_WR_ADDR);
79          IOWR_DMA(FPGA_MEM, dev->dma_base_addr, DMA_RD_ADDR);
80          IOWR_DMA(count, dev->dma_base_addr, DMA_LEN);
81      }
82      else{
83          IOWR_DMA(TX + (dev->bus_addr & 0xFFFFF), dma_device->dma_base_addr,
    DMA_RD_ADDR);
84          IOWR_DMA(FPGA_MEM, dev->dma_base_addr, DMA_WR_ADDR);
85          IOWR_DMA(count , dev->dma_base_addr, DMA_LEN);
86      }
87
88  //  printk (KERN_DEBUG "\t\t DMA: Before Start\n");
89
90      ret = IORD_DMA(dev->dma_base_addr, DMA_CONTROL);
91      printk (KERN_DEBUG "\t\t DMA: Control Register 0x%x\n", ret);
92      ret = IORD_DMA(dev->dma_base_addr, DMA_STATUS);
93      printk (KERN_DEBUG "\t\t DMA: Status Register 0x%x\n", ret);
94      ret = IORD_DMA(dev->dma_base_addr, DMA_RD_ADDR);
95      printk (KERN_DEBUG "\t\t DMA: RD ADDR 0x%x\n", ret);
96      ret = IORD_DMA(dev->dma_base_addr, DMA_WR_ADDR);
97      printk (KERN_DEBUG "\t\t DMA: WR ADDR 0x%x\n", ret);
98      ret = IORD_DMA(dev->dma_base_addr, DMA_LEN);
99      printk (KERN_DEBUG "\t\t DMA: LEN 0x%x\n", ret);
100
101     // Start DMA Transfer
102     ret = IORD_DMA(dev->dma_base_addr, DMA_CONTROL);
103     IOWR_DMA(ret | GO, dma_device->dma_base_addr, DMA_CONTROL);
104     dma_device->before = ktime_get_real();
105
106 /*
107     printk (KERN_DEBUG "\t\t DMA: After Start\n");
108     ret = IORD_DMA(dev->dma_base_addr, DMA_CONTROL);
109     printk (KERN_DEBUG "\t\t DMA: Control Register 0x%x\n", ret);
110     ret = IORD_DMA(dev->dma_base_addr, DMA_STATUS);
111     printk (KERN_DEBUG "\t\t DMA: Status Register 0x%x\n", ret);
112     ret = IORD_DMA(dev->dma_base_addr, DMA_RD_ADDR);
```

```
113         printk (KERN_DEBUG "\t\t DMA: RD ADDR 0x%x\n", ret);
114         ret = IORD_DMA(dev->dma_base_addr, DMA_WR_ADDR);
115         printk (KERN_DEBUG "\t\t DMA: WR ADDR 0x%x\n", ret);
116         ret = IORD_DMA(dev->dma_base_addr, DMA_LEN);
117         printk (KERN_DEBUG "\t\t DMA: LEN 0x%x\n", ret);
118     */
119
120         return 0;
121     }
122
123     static ssize_t dma_read(struct file *filp, char *buffer, size_t length, loff_t *
        offset){
124
125         int retval = 0;
126
127         dma_transfer(dma_device, 0, dma_device->k_buffer, length);
128
129         wait_event_interruptible(wq, dma_device->flag != 0);
130         dma_device->flag = 0;
131
132         if(copy_to_user(buffer, dma_device->k_buffer, length)){
133             retval = -EFAULT;
134             goto out;
135         }
136
137         retval = length;
138
139     out:
140         return retval;
141
142     }
143
144     static ssize_t dma_write(struct file *filp, const char *buffer, size_t length, loff_t
         * offset) {
145
146         int retval = 0;
147
148         if(copy_from_user(dma_device->k_buffer, buffer, length)){
149             retval = -EFAULT;
150             goto out;
151         }
152
153         dma_transfer(dma_device, 1, dma_device->k_buffer, length);
154
155         wait_event_interruptible(wq, dma_device->flag != 0);
156         dma_device->flag = 0;
157
158         retval = length;
159
160     out:
161         return retval;
162     }
163
164     int dma_open(struct inode *inode, struct file *filp) {
165
166         int num;
167
```

```
168         if ((num = MINOR(inode->i_rdev)) != dma_MINOR)
169             return -ENODEV;
170 
171         filp->private_data = (void *) dma_device;
172         return 0;
173     }
174 
175     int dma_release(struct inode *inode, struct file * filp) {
176 
177         int num;
178 
179         if ( (num = MINOR(inode->i_rdev)) != dma_MINOR)
180             return -ENODEV;
181 
182         return 0;
183     }
184 
185     static struct file_operations dma_fops = {
186 
187     //  .ioctl   =  dma_ioctl,
188     //  .mmap    =  dma_mmap,
189     //  .llseek   =  dma_lseek,
190         .read    =  dma_read,
191         .write   =  dma_write,
192         .open    =  dma_open,
193         .release =  dma_release,
194 
195     };
196 
197     static struct miscdevice dmamisc = {
198 
199         .name = DRV_NAME,
200         .minor = dma_MINOR,
201         .fops = &dma_fops,
202     };
203 
204     static irqreturn_t pcie_irq_handler(int irq, void *data) {
205 
206         int ret;
207 
208         struct dma_device *dma_device = (struct dma_device *)data;
209         if (!dma_device)
210             return IRQ_NONE;
211 
212         dma_device->after = ktime_get_real();
213 
214         // Stop interruptions from the DMA Controller
215         iowrite32((u32)0<<7, (u32*)(dma_device->cra_base_addr + CRA_IER));
216 
217         // Read Interrupt status register and clear
218         ret = ioread32((u32*)dma_device->cra_base_addr + CRA_ISR);
219         iowrite32((u32)0x00, (u32*)dma_device->cra_base_addr + CRA_ISR);
220 
221         // Calculate the transaction time
222         dma_device->diff = ktime_sub(dma_device->after, dma_device->before);
223             printk(KERN_DEBUG "\t\t DEBUG: dT = %lld ns\n", ktime_to_ns(dma_device->diff
    ));
```

```
224
225         dma_device->flag = 1;
226         wake_up_interruptible(&wq);
227
228     //  printk (KERN_DEBUG "IRQ Handled\n");
229     //  printk (KERN_DEBUG "CRA Status : 0x%x\n", ret);
230
231     //  ret = IORD_DMA(dma_device->dma_base_addr, DMA_STATUS);
232     //  printk (KERN_DEBUG "DMA Status : 0x%x\n", ret);
233
234         // Clear the DMA Status register
235         IOWR_DMA(0x00, dma_device->dma_base_addr, DMA_STATUS);
236
237         // Restart interruptions from the DMA Controller (QSYS)
238         iowrite32(0xFFFFFFFF, (u32*)(dma_device->cra_base_addr + CRA_IER));
239         // Restart interruptions from the DMA Controller (SOPC)
240     //  iowrite32((u32)1<<7, (u32*)(dma_device->cra_base_addr + CRA_IER));
241
242         return IRQ_HANDLED;
243     }
244
245
246     static int dma_probe(struct pci_dev *dev, const struct pci_device_id *id) {
247
248         int status;
249         unsigned int ret;
250
251         printk(KERN_DEBUG "\tProbing dma Device\n");
252
253         pci_set_master(dev);
254
255         if (pci_enable_device(dev)) {
256             printk(KERN_ERR "Failed to enable PCI device\n");
257             status = -ENODEV;
258             goto error_no_mem;
259         }
260
261         //Allocating a pointer on Kernel Space for dma Device Driver
262         if (!(dma_device = (struct dma_device *) kzalloc(sizeof(struct dma_device),
    GFP_KERNEL))){
263             status = -ENOMEM;
264             goto error_no_mem;
265         }
266
267         strcpy(dma_device->name, DRV_NAME);
268         dma_device->pci_dev = dev;
269
270         if (!(pci_resource_start(dev, 0))) {
271             printk(KERN_ERR "\t\tBAR0 disabled! Giving up.\n");
272             status = -ENOMEM;
273             goto error_base;
274         }
275
276         if (!(pci_resource_start(dev, 2))) {
277             printk(KERN_ERR "\t\tBAR2 disabled! Giving up.\n");
278             status = -ENOMEM;
279             goto error_base;
```

```
280     }
281
282     dma_device->BAR0 = pci_resource_start(dev, 0);
283     dma_device->BAR0_size = pci_resource_len(dev, 0);
284
285     dma_device->BAR2 = pci_resource_start(dev, 2);
286     dma_device->BAR2_size = pci_resource_len(dev, 2);
287
288
289     printk(KERN_DEBUG "\t Base Address Register(BAR0) @ 0x%lx\n\t Size of %lu
    bytes\n",
290                                     dma_device->BAR0,
291                                     dma_device->BAR0_size);
292
293     printk(KERN_DEBUG "\t Base Address Register(BAR2) @ 0x%lx\n\t Size of %lu
    bytes\n",
294                                     dma_device->BAR2,
295                                     dma_device->BAR2_size);
296
297     if (!(dma_device->mem_base_addr = ioremap_nocache(dma_device->BAR0, dma_device->
    BAR0_size))) {
298         printk(KERN_DEBUG "Could not Remap BAR0\n");
299         status = -ENOMEM;
300         goto error_base;
301     }
302
303     request_mem_region(dma_device->BAR2 + CRA_BASE_ADDR, CRA_SIZE, DRV_NAME);
304     dma_device->cra_base_addr = ioremap_nocache(dma_device->BAR2 + CRA_BASE_ADDR,
    CRA_SIZE);
305
306     request_mem_region(dma_device->BAR2 + DMA_BASE_ADDR, DMA_SIZE, DRV_NAME);
307     dma_device->dma_base_addr = ioremap_nocache(dma_device->BAR2 + DMA_BASE_ADDR,
    DMA_SIZE);
308
309     request_irq(dev->irq, pcie_irq_handler, IRQF_SHARED, DRV_NAME, (void *)dma_device
    );
310     printk(KERN_DEBUG "Succesfully requested IRQ #%d with dev_id 0x%p\n", dev->irq,
    dma_device);
311
312     printk(KERN_DEBUG "\t Remapped Local Address 0x%lx\n"\
313              "\t To Virtual Address 0x%lx : %d bytes of length\n",
314              dma_device->BAR0, (unsigned long)dma_device->mem_base_addr,
315              (int)dma_device->BAR0_size);
316
317     printk(KERN_DEBUG "\t Remapped DMA Avalon address 0x%lx\n"\
318             "\t To Virtual Address 0x%lx : %d bytes of length\n",
319             (unsigned long)DMA_BASE_ADDR, (unsigned long)dma_device->dma_base_addr,
320             (int)DMA_SIZE);
321
322     printk(KERN_DEBUG "\t Remapped PCIe CRA address 0x%lx\n"\
323             "\t To Virtual Address 0x%lx : %d bytes of length\n",
324             (unsigned long)CRA_BASE_ADDR, (unsigned long)dma_device->cra_base_addr,
325             (int)CRA_SIZE);
326
327     if (misc_register(&dmamisc)) {
328         printk(KERN_DEBUG "Could not register misc device\n");
329         goto error_init_dma;
```

```
330         }
331
332         ret = ioread32(dma_device->cra_base_addr + CRA_ISR);
333         printk(KERN_DEBUG "\t CRA ISR : 0x%x\n", le32_to_cpu(ret));
334         ret = 0;
335
336         iowrite32(0xFFFFFFFF, (u32*)(dma_device->cra_base_addr + CRA_IER));
337
338         ret = ioread32(dma_device->cra_base_addr + CRA_IER);
339         printk(KERN_DEBUG "\t CRA IER : 0x%x\n", ret);
340
341         dma_device->flag = 0;
342         dma_init();
343
344         dma_device->k_buffer = (char *) kzalloc(0x2000, GFP_DMA);
345         dma_device->bus_addr = virt_to_phys(dma_device->k_buffer);
346
347         return 0;
348
349     error_init_dma:
350         iounmap((void *) dma_device->mem_base_addr);
351         iounmap((void *) dma_device->dma_base_addr);
352         iounmap((void *) dma_device->cra_base_addr);
353
354     error_no_mem:
355         pci_disable_device(dev);
356     error_base:
357         kfree(dma_device);
358         return 0;
359     }
360
361     static void dma_remove(struct pci_dev *dev) {
362
363         /* clean up any allocated resources and stuff here.
364          * like call release_region();*/
365
366         free_irq(dev->irq, dma_device);
367         iounmap((void *) dma_device->mem_base_addr);
368         iounmap((void *) dma_device->dma_base_addr);
369         iounmap((void *) dma_device->cra_base_addr);
370
371         kfree(dma_device->k_buffer);
372
373         misc_deregister(&dmamisc);
374         kfree(dma_device);
375
376     }
377
378     static struct pci_driver pci_driver = {
379         .name = DRV_NAME,
380         .id_table = dma_ids,
381         .probe = dma_probe,
382         .remove = dma_remove,
383     };
384
385     static int __init pcie_dma_init(void) {
386
```

```
387         printk(KERN_INFO "PCIe dma Driver Loaded...\n");
388         return pci_register_driver(&pci_driver);
389     }
390 
391     static void __exit pcie_dma_exit(void) {
392 
393         printk(KERN_INFO "PCIe dma Driver Unloaded...\n");
394         pci_unregister_driver(&pci_driver);
395     }
396 
397     module_init(pcie_dma_init);
398     module_exit(pcie_dma_exit);
399 
400 
401     MODULE_AUTHOR("Leonardo Lessa <llessa@cbpf.br>");
402     MODULE_DESCRIPTION("PCIe Driver for DMA transfers");
403     MODULE_LICENSE("GPL");
404 
```

```c
// Leonardo Lessa
// Aquisicao da tensao dos reguladores via adc i2c utilizando chamadas no sysfs

#include <stdio.h>


#define N 0
char *path="/sys/bus/i2c/devices";
char buffer[50];
char *devices[2] = {"1-0048", "1-0049"};
char *adc_1[4] = {"3V3A", "3V3B", "3V0 ", "2V5 "};
char *adc_2[2] = {"1V8 ", "1V2 "};
float vref = 3.29;

float count_to_volt(int cnt, int adc_2){

float volt;

if (adc_2)
    volt = (cnt * vref) / 2550;
else
    volt = (cnt * vref) / 255;

return volt;

}


int main(){

FILE *file, *fout;
unsigned char tmp[4];
unsigned int adc_value;

int i,j,z;


fout = fopen("adc.txt", "w");
for (z=0; z<N; z++){
    for (i=0; i<2; i++){
        for (j=0; j<4; j++){
            sprintf (buffer, "%s/%s", path, devices[i]);
            sprintf (buffer, "%s/in%d_input", buffer, j);
            file = fopen(buffer, "r");
            fread(&tmp, 3, 1, file);
            adc_value = atoi (tmp);
            if (i == 0) {
                printf("Regulator [%s] = %.4f V\n",adc_1[j], count_to_volt(adc_value,
0));
                fprintf(fout, "%.4f ", count_to_volt(adc_value,0));
            }
            else if (j == 0){
                printf("Regulator [%s] = %.4f V\n",adc_2[0], count_to_volt(adc_value,
0));
                fprintf(fout, "%.4f ", count_to_volt(adc_value,0));
            }
            else if (j == 1){
```

```
56                  printf("Regulator [%s] = %.4f V\n",adc_2[1], count_to_volt(adc_value,
    1));
57                  fprintf(fout, "%.4f\n", count_to_volt(adc_value,1));
58              }
59
60
61              fclose(file);
62          }
63      }
64
65  }
66  fclose(fout);
67  }
68
69
70
```

```
#!/bin/bash

while :
do

    value=$(</sys/bus/i2c/devices/1-0048/temp1_input)
    timestamp=`date +"%F %T"`
    echo "$timestamp,$value"
    sleep 10m
done
```

Apêndice C – Verificação dos protótipos

Com o fim do processo de montagem dos componentes na placa, foi realizada a montagem mecânica dos módulos para a verificação dos furos de fixação e espaçadores das placas, uma foto do conjunto montado pode ser visto na Figura 1.

Figura 1 - Placa de Validação e módulo ECS-CBPF

Por medida de segurança a alimentação das placas foi realizada em separado. A primeira placa a ser alimentada foi a placa de validação, após a energização, percebeu-se que um dos capacitores da entrada dos reguladores, mais especificamente o C96, sofreu uma deterioração por superaquecimento, como pode ser visto na Figura 2. Após uma inspeção visual foi constatado que devido a um erro de montagem a polaridade dos capacitores C96 e C106 estavam invertidas. Os capacitores foram removidos e substituídos por componentes novos com a polarização correta. O segundo passo no processo de verificação foi a realização da medida dos níveis de tensões na saída dos reguladores. Foi constatado que todos estavam com os níveis de tensão corretos. Os resistores de proteção do circuito (R26, R28, R29 e R31), que tem a funcionalidade de isolar as fontes de tensão do

resto do circuito, foram soldado individualmente e a cada inclusão uma nova medida no nível de tensão foi realizada. O regulador de tensão (U30), cujo nível de tensão especificado é de 3.3 V, apresentou em sua saída 0 V o que indicou que após a inclusão do resistor de proteção algum circuito por este alimentado apresentava um curto circuito. Uma varredura no esquemático de todos os circuitos alimentados por este regulador foi realizada e um erro na alimentação do circuito integrado I²C PCF8574 (U18 e U19) foi diagnosticado. Em um primeiro momento ambos os circuitos integrados foram removidos da placa e uma nova medida no nível de tensão deste regulador foi realizada. Foi constatado que estes circuitos integrados eram o problema. Para a correção deste problema os pinos de alimentação foram invertidos com o uso de fios, como pode ser observado na Figura 3.

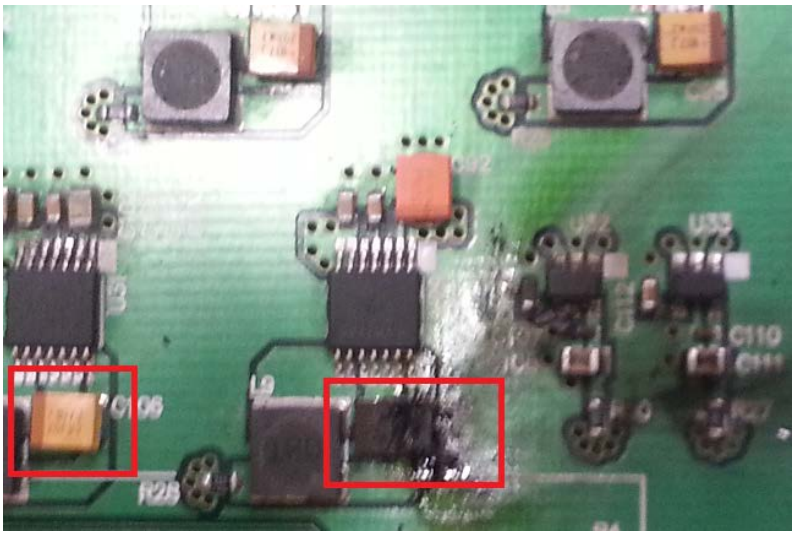


Figura 2 - Foto do capacitor C96 após problema

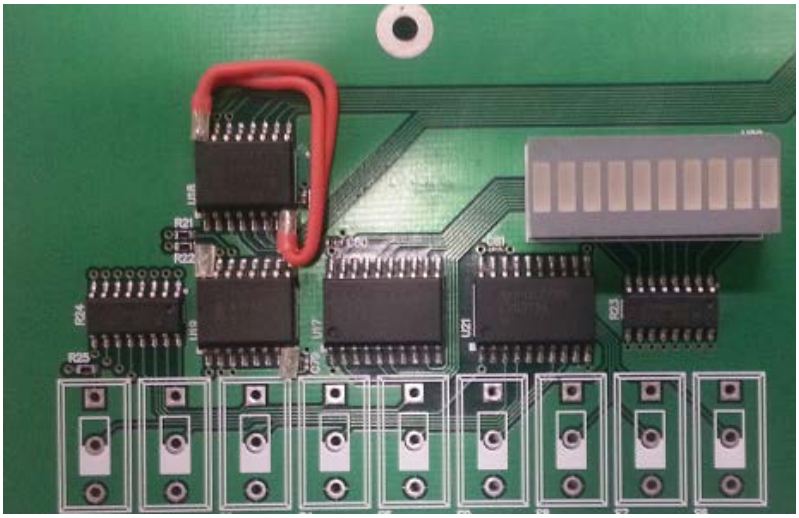


Figura 573 - Solução do problema dos circuitos U18 e U19

Após os problemas encontrados na parte de alimentação da placa resolvidos, foram iniciados alguns testes dos subsistemas da placa de validação. O primeiro teste a ser realizado, foi o carregamento de um código simples em Verilog no FPGA através do conector para programação externa P1 e o uso do programador USB Blaster. O código foi carregado com sucesso. Outro teste realizado foi o acesso dos dispositivos I²C através de um controlador I²C externo ligado no conector P3. Com exceção do circuito integrado LM75 (U24), todos os dispositivos responderam de forma satisfatória ao teste. O sensor de temperatura LM75 apresentava as conexões de *clock* e dados invertidos e devido à limitação física do tamanho do componente, não foi possível a correção do problema ocasionando a remoção do mesmo da placa.

O processo de alimentação do módulo ECS-CBPF foi realizado somente depois de verificado o funcionamento da placa de validação, pois a mesma alimenta o módulo através do conector J1. Devido a um erro na produção do *layout* o regulador de tensão (U10) de 5 V não foi montado na placa, pois o *footprint* estava espelhado em relação ao componente. Como solução a alimentação dos 5 V externamente.

A verificação da alimentação de tensão foi realizada de forma similar a placa de validação. Os níveis de tensão nos reguladores foram medidos, sendo constatado que em nenhum deles o nível de tensão na saída era o esperado pela alimentação do módulo, como citado anteriormente. A tensão de 5 V alimenta dois reguladores responsáveis pela geração de níveis de tensão mais baixos. O circuito integrado TPS75003 deveria gerar as tensões de 3.3 V, 2.5 V e 1.2 V este regulador possui duas saídas chaveadas, que necessitam de um circuito auxiliar para o seu funcionamento, e uma saída linear. Com o uso dos jumpers J9, J10 e J11 foi possível habilitar uma saída por vez para o diagnóstico do problema. A primeira saída a ser inspecionada foi a do regulador linear (2.5 V) a medida de tensão nos resistores R97 e R98 não foram as esperadas e se iniciou uma inspeção visual, foi notado que os componentes foram montados de forma errada, como mostra na Figura 4. Após seu correto posicionamento foi obtida a tensão de 2.5V na saída deste regulador. A segunda saída a ser inspecionada foi a saída de um dos reguladores chaveados (3.3 V). A medida nos níveis de tensões no dreno, fonte e porta do transistor FET Q2 apresentaram valores incoerentes com os valores esperados. Após uma consulta no *datasheet* do componente e a verificação do modelo utilizado, foi deduzido que a disposição dos pinos do *footprint* na placa não

condiziam com os esperados. Através de uma rotação no componente, como mostrado na Figura 49, foi possível a correção do problema e o mesmo foi realizado no transistor FET Q1 responsável pela tensão de 1.2 V. Neste caso porém, o regulador apresentou um problema intermitente, supostamente devido a polarização errada do transistor. Com a substituição do componente este passou a funcionar corretamente.

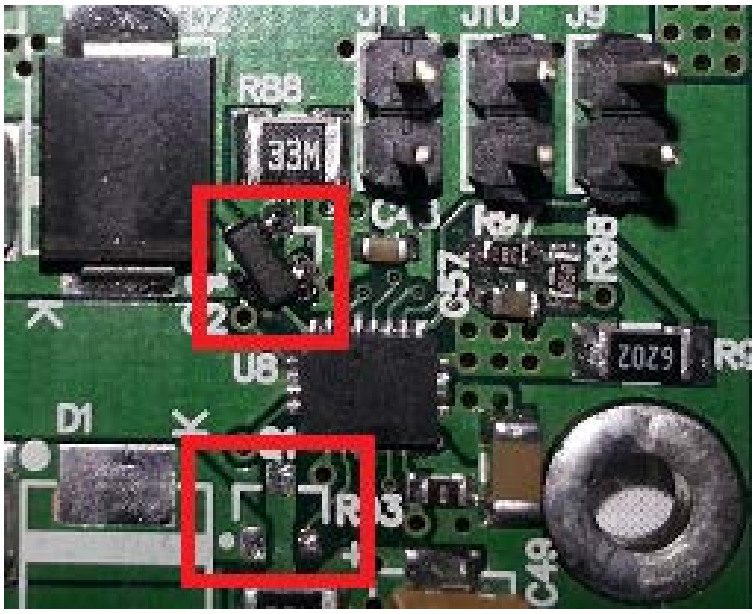


Figura 4 - Correção do posicionamento dos transistores Q1 e Q2

O regulador LTC3568 (U7) responsável pela geração da tensão de 1.0 V não apresentou a tensão esperada na sua saída. Após uma leitura detalhada no *datasheet* do componente, foi observado que dois parâmetros importantes foram deixados de lado no cálculo dos componentes externos ao regulador. O primeiro consiste na frequência máxima de operação em relação à tensão de entrada e saída, ajustadas pelo resistor R84. Após os cálculos necessários indicados no *datasheet*, o resistor de 130K foi substituído por um de 324k. O outro parâmetro importante é a corrente máxima na malha de realimentação na saída do regulador. Os resistores R82, R83 e R86 foram substituídos respectivamente pelos valores de 130K, 536K e 0. Com as modificações necessárias o regulador apresentou em sua saída a tensão esperada.

Com a correção de todos os problemas nos circuitos de alimentação do módulo, foi iniciado a verificação dos subsistemas presentes na placa. Foi notado que dois circuitos integrados sensíveis, o FPGA (U3) e o Switch PCIe (U12), apresentaram um aquecimento não esperado o que levantou a suspeita que os dois foram danificados pela polarização errada

do transistor[72]. A leitura do *datasheet* dos componentes demonstrou que o valor máximo absoluto nas tensões de entrada foram ultrapassados e que estes componentes deveriam ser substituídos. Após a remoção de ambos os componentes, foi observado que a máscara de solda da placa foi comprometida, possivelmente pelo superaquecimento do componente e/ou pelo processo de remoção. Além disso, um plano de terra com grande área de dissipação térmica foi inserida no *layout* nos componentes com encapsulamento BGA (U3 e U12), o qual prejudica o processo de soldagem do componente. Estes fatores levaram a decisão de incluir em um primeiro momento apenas o FPGA e posteriormente, após comprovado o funcionamento do FPGA, o Switch PCIe.

Os demais circuitos integrados do módulo, de acordo com os *datasheets*, toleraram um valor máximo absoluto de tensão superior à aplicada. Com isso foi possível à realização do teste do CCPC e seus circuitos auxiliares antes da inserção do FPGA na placa. O módulo do CCPC foi inserido no conector J1 da placa ECS-CBPF, o conector DVI inserido no conector J8 e o teclado na porta USB. Após a alimentação da placa foi possível visualizar a tela de boot do CCPC, porém com a imagem esbranquiçada. O teclado respondeu aos comandos após o pressionamento da tecla para a inicialização do sistema de configuração da BIOS. Uma medida de continuidade foi feita nas linhas D+ e D- da porta USB apresentando a não continuidade de ambas as linhas. Com a realização de uma inspeção visual foi verificado que o filtro para prevenir interferências eletromagnéticas estava com seu *footprint* errado, como pode ser visto na Figura 5. A correção foi feita através da remoção do componente e a solda direta de fios nos terminais do filtro. Após uma nova alimentação o teclado passou a funcionar e com o acesso a configuração da BIOS foi possível corrigir o problema da tela, mudando a opção de saída de vídeo para 24 bits ao invés de 18 bits definido como padrão. Um boot completo com o sistema operacional foi realizado através do procedimento Ethernet boot, onde se pode verificar o funcionamento da interface de rede e o completo funcionamento do CCPC e seus circuitos auxiliares.

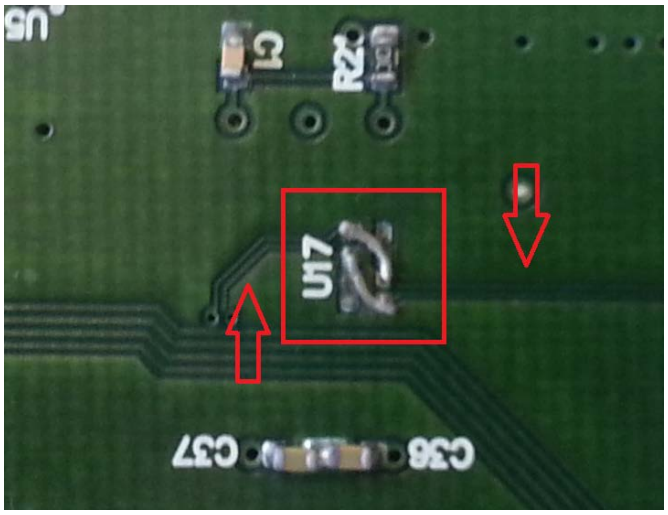


Figura 5 - Problema nos sinais de dados do USB

Com a validação do CCPC, um novo FPGA foi inserido na placa. A verificação do funcionamento do novo componente inserido foi feito através do procedimento de programação de *firmware.* Este procedimento apresentou resultados intermitentes. Estes fatores foram importantes para a dedução que os problemas anteriormente encontrados, com a máscara de solda e plano de terra influenciaram no não-funcionamento do novo dispositivo. Este fator inviabilizou a realização da verificação do sistema completo utilizando o módulo ECS-CBPF para a extração dos resultados.