Filipe Calasans Portugal de Oliveira

Proposta de Arquitetura de um Sistema em um Chip (SoC) para Fins Educacionais

São Carlos

2014

Filipe Calasans Portugal de Oliveira

Proposta de Arquitetura de um Sistema em um Chip (SoC) para Fins Educacionais

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos, da Universidade de São Paulo

Curso de Engenharia de Computação

Universidade de São Paulo - USP

Escola de Engenharia de São Carlos - EESC

Departamento de Engenharia Elétrica e de Computação - SEL

Orientador: Evandro L. L. Rodrigues

São Carlos

2014

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

O143p

Oliveira, Filipe Calasans Portugal de
Proposta de arquitetura de um sistema em um chip (SoC) para fins educacionais. / Filipe Calasans Portugal de Oliveira; orientador Evandro Luis Linhares Rodrigues. São Carlos, 2014.

Monografia (Graduação em Engenharia de Computação) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2014.

1. Sustema em um chip. 2. SoC. 3. FPGA. 4. Hdl. 5. Nexys3. I. Título.

FOLHA DE APROVAÇÃO

Nome: Filipe Calasans Portugal de Oliveira

Título: "Proposta de arquitetura de um sistema em um Chip (SoC) para fins educacionais"

Trabalho de Conclusão de Curso defendido em 10/06/2014

Comissão Julgadora:	**Resultado:**
Prof. Associado Evandro Luís Linhari Rodrigues (Orientador) - SEL/EESC/USP	Aprovado.
Prof. Dr. Vanderlei Bonato SSC/ICMC/USP	APROVADO
Prof. Dr. Maximiliam Luppe SEL/EESC/USP	Aprovado

Coordenador do Curso Interunidades - Engenharia de Computação:

Prof. Associado Evandro Luís Linhari Rodrigues

Dedico este trabalho a minha família, Renato, Solange, Renata e Tito. Grandes incentivadores das minhas realizações acadêmicas.

Agradecimentos

Agradeço ao Prof. Dr. Evandro Linhari pela orientação competente, pela paciência e compreensão nos momentos mais difíceis deste projeto.

Agradeço ao amigos de sala e república Tira Gosto e Um Dia A Casa Cai, que partilharam momentos preciosos durante a minha vida acadêmica.

Aos professores Mr. Tramel da California State University Long Beach e Dr. Luca Carloni da Columbia University in the City of New York, com os quais trabalhei em 2012/2013, e puderam agregar mais conhecimentos e despertar mais paixão científica pela área de projeto de sistemas em um chip (SoC).

Resumo

A utilização de *chips* complexos compostos por diversos processadores e periféricos em uma mesma pastilha se tornou realidade em projetos atuais de dispositivos eletrônicos devido ao baixo custo de processos de fabricação e disponibilidades de ferramentas de projeto. Neste contexto, a formação de um engenheiro capacitado ao desenvolvimento dessas novas tecnologias perpassa pela assimilação, e aplicação de conceitos fundamentais na teoria de sistemas digitais. Dentre eles: arquitetura de computadores e seu fluxo de dados, aplicação de máquinas de estados, arquiteturas de comunicação em *chip*, análise de tempo em circuitos digitais e utilização de linguagens de descrição de *hardware* (HDL). Neste contexto, este trabalho apresenta os fundamentos de desenvolvimento de um sistema em chip (SoC) simples de propósito educacional que possa ser implementado em um semestre letivo, que seja de baixo custo, e que o projeto contemple a concepção de uma arquitetura de *hardware* e de desenvolvimento de um *software* a ser executado pelo microcontrolador. A fim de tornar o processo o mais didático possível, este trabalho aborda o desenvolvimento completo de um módulo UART em Verilog HDL, seguido de sua integração utilizando uma arquitetura de barramento simples customizada que permita a integração do módulo UART criado, com o microcontrolador Xilinx Picoblaze KCPSM6 e interface de memória compatível com o *chip* de memória 16Mbyte CelullarRAM 1.5 MT45W8MW16BGX. Em seguida apresenta-se o desenvolvimento de um *Firmware* que utiliza todos os módulos implementados com a finalidade de verificar seu correto funcionamento. Ao fim, o projeto é portado para uma FPGA modelo Spartan 6 contido na placa de desenvolvimento Digilent Nexys 3.

Palavras-chaves: Sistema em um chip, SoC, FPGA, HDL, Nexys 3.

Abstract

Use of complex chips with several cores and peripherals on a same die has become a trend on current electronic devices, due to low cost of fabrication processes and availability of CAD tools. So, current projects have demanded high qualified engineers with fully understanding of a set of fundamental topics on digital design. Including, computer architecture and its data flow, state machine applications, on-chip communication architectures, time analysis and hardware description languages. In this context, this work presents the design fundamentals of a low cost system-on-chip (SoC) to educational purpose, that can be implemented in one semester. The project includes a hardware architecture design and firmware design. Including, the fully implementation of a custom UART module using Verilog HDL, followed by description of how to interface the UART module developed and a memory interface compatible with the chip CelullarRAM 1.5 MT45W8MW16BGX 16Mbytes to the microcontroller Xilinx Picoblaze KCPSM6 on a shared bus environment. Next, we present a firmware implementation able to use all modules implemented, aiming to verify if the devices are working properly. At the end of the project, we describe how to implement the design on the board Digilent Nexys 3.

Key-words: System on Chip, SoC, FPGA, Verilog, Nexys 3.

Lista de ilustrações

Figura 1 – Diagrama de blocos de um SoC de aplicação em multimídia (Sudeep Pasrich; Nikil Dutt, 2008). 22
Figura 2 – Diagrama funcional da arquitetura do microcontrolador KCPSM6 (CHAPMAN, 2013). 23
Figura 3 – Esquemático exemplificando como integrar a funcionalidade de porta de entrada a um projeto (CHAPMAN, 2013). 24
Figura 4 – Esquemático exemplificando como integrar a funcionalidade de porta de saída a um projeto (CHAPMAN, 2013). 24
Figura 5 – Diagrama funcional da arquitetura da memória CelullarRAM 1.5 MT45W8MW16BGX (MICRON, 2004). 25
Figura 6 – Sinais de controle, dado e endereço durante operação de leitura em memória (MICRON, 2004). 26
Figura 7 – Sinais de controle, dado e endereço durante operação de escrita em memória (MICRON, 2004). 26
Figura 8 – Requisitos de tempo durante a operação de leitura em memória (MICRON, 2004). 27
Figura 9 – Requisitos de tempo durante a operação de escrita em memória (MICRON, 2004). 28
Figura 10 – Ordem dos bits segundo o protocolo UART. 29
Figura 11 – Modelo de representação hierárquico utilizando Verilog. 31
Figura 12 – Arquitetura de uma FPGA (Stephen Brown, 2000). 32
Figura 13 – Diferença entre fluxos de projeto visando FPGA e ASIC (XILINX, 2014). 33
Figura 14 – Placa de desenvolvimento Digilent Nexys 3. 35
Figura 15 – Diagrama de bloco funcionais do SoC implementado. 36
Figura 16 – Gerador de *clock* da UART. 37
Figura 17 – Arquitetura do módulo UART. 38
Figura 18 – Máquina de estados do dispositivo de transmissão (TX). 39
Figura 19 – Máquina de estados do dispositivo de recepção (RX). 39
Figura 20 – Registradores do módulo de interface com a memória. 40
Figura 21 – Sinais de controle durante a operação de leitura. 41
Figura 22 – Sinais de controle durante a operação de escrita. 41
Figura 23 – Máquina de estados de Moore utilizada na geração dos sinais de controle da memória. 42
Figura 24 – *Buffer* bidirecional IOBUF e sua tabela verdade. 43
Figura 25 – Exemplo de execução do montador utilizado no projeto. 46

Figura 26 – Exemplo de como a memória de programa deve ser instanciada e integrada ao projeto (CHAPMAN, 2013). 46
Figura 27 – Circuito sincronizador utilizado no projeto(ERUSALA, 2011). 48
Figura 28 – Módulo top-level do SoC. 48
Figura 29 – Resultado da simulação da UART e microcontrolador durante a recepção seguida de transmissão de dados. 49
Figura 30 – Momento de transição entre a recepção de um dado e a transmissão de um dado pelo microcontrolador 50
Figura 31 – Tempo de bit para taxa BAUD 9600 verificado durante simulação. . . . 50
Figura 32 – Sinais de controle da memória gerados pela controladora projetada durante operação de escrita. 51
Figura 33 – Sinais de controle da memória gerados pela controladora projetada durante operação de leitura. 51
Figura 34 – Exemplo de execução à 9600 bauds do terminal implementado no SoC. 52

Lista de tabelas

Tabela 1 – Tempo de bit para cada um dos valores de BAUD comumente utilizado em projetos. 29
Tabela 2 – Taxas de BAUD para *clock* de 100MHz. 37
Tabela 3 – Mapeamento dos dispositivos de E/S utilizando a saída *port ID* do microcontrolador Picoblazer. 44

Sumário

1 INTRODUÇÃO **17**
1.1 Contextualização **17**
1.2 Objetivos **18**
1.2.1 Objetivos Específicos 18
1.3 Estrutura da Monografia **19**

2 FUNDAMENTAÇÃO TEÓRICA **21**
2.1 Sistema em um *Chip* (SoC) **21**
2.2 Microcontrolador Xilinx Picoblaze KCPSM6 8 bits **22**
2.3 Memória CelullarRAM 1.5 MT45W8MW16BGX 16 Mbytes **24**
2.3.1 Requisitos de tempo durante as operações de leitura e escrita assíncronas . 26
2.4 UART (*Universal Asynchronous Receiver/Transmitter)* **28**
2.5 Linguagens de Descrição de *Hardware* (HDL) **29**
2.5.1 Linguagem de Descrição Verilog 31
2.6 FPGA (*Field Programmable Gate Array*) **31**

3 MATERIAIS E MÉTODOS **35**
3.1 Placa Nexys 3 e ambiente de desenvolvimento **35**
3.2 Blocos desenvolvidos em Verilog **35**
3.2.1 Arquitetura do SoC 36
3.2.2 Módulo UART 37
3.2.3 Módulo *Memory Interface* 40
3.2.4 Módulo *BUS Control* 43
3.3 *Firmware* **45**
3.4 Prototipação em FPGA **47**
3.4.1 Sincronização, circuito de *reset* 47
3.4.2 Mapeamento dos pinos da FPGA nas portas do SoC. 48

4 RESULTADOS **49**

5 PROPOSTA DE CRONOGRAMA **53**

6 CONCLUSÃO **55**

7 TRABALHOS FUTUROS **57**

Referências **59**

APÊNDICES 61

APÊNDICE A – *FIRMWARE* DO TERMINAL *ECHO* 63

APÊNDICE B – ARQUIVO UCF UTILIZADO NO MAPEAMENTO DAS PORTAS DO SOC 69

APÊNDICE C – CASO DE TESTE UTILIZADO NA SIMULAÇÃO DO MÓDULO UART 71

APÊNDICE D – CÓDIGOS DESENVOLVIDOS EM VERILOG . . . 73

ANEXOS 99

ANEXO A – REQUISITOS DE TEMPO DE LEITURA ASSÍNCRONA EM MEMÓRIA 101

ANEXO B – REQUISITOS DE TEMPO DE ESCRITA ASSÍNCRONA EM MEMÓRIA 103

1 Introdução

1.1 Contextualização

Dispositivos eletrônicos, tais como câmeras digitais, roteadores, videogames, celulares e receptores de televisão digital fazem o uso de um ou mais Circuitos Integrados (CI), os quais são compostos por componentes como microprocessadores, memórias, interfaces de comunicação e *hardwares* customizados de processamento. A convergência digital, o desenvolvimento de ferramentas de projeto, e o aprimoramento de processos de fabricação permitiram que a integração destes diversos elementos passasse do nível de placa de circuito impresso para uma escala de um único *chip*, passando a serem denominados de Sistemas em um *chip* (SoC, do inglês *System on Chip*).

O fluxo de desenvolvimento de um SoC envolve o projeto do *hardware* como descrito acima, e do *software* responsável por controle do DSP, microprocessador ou microcontrolador que compõe a arquitetura do SoC, etapas que são geralmente desenvolvidos simultaneamente. A maioria dos SoC é composta por diversos módulos de *hardware*, denominados IPs (*Intellectual Property*), reutilizados e que seguem protocolos pré definidos por instituições de normas técnicas, tais como USB, UART, SPI e I2C. Estes diversos componentes de *hardware* em certa etapa de projeto são descritos em alguma Linguagem de Descrição de Hardware (HDL), a partir da qual é possível através de ferramentas CAD (Computer Aided Design) verificar seu correto funcionamento através de simulações. Na etapa de projeto em que uma versão funcional da descrição em HDL da arquitetura e um *software* estão disponíveis, implementasse um protótipo em FPGA (*Field Programmable Gate Array*) para verificação conjunta de *hardware* e *software* antes da produção física do *chip*.

As diversas etapas de desenvolvimento de um SoC envolve que diversos conceitos básicos em sistemas digitais estejam consolidados, e sejam aplicados em diversos níveis na concepção de um projeto. Desta forma, no momento do ensino dos conceitos de projeto de SoCs o aluno da área de computação deve ter consolidado em sua formação tópicos como arquitetura de computadores, elementos de lógica digital e programação de computadores, para que o conhecimento seja então aplicado em forma de um projeto completo de um SoC. Percebe-se no entanto, que neste ponto muitas vezes existe uma lacuna de projetos bem documentados de SoC que sejam de fácil entendimento ao estudante e de fácil implementação em sala de aula, uma vez que muitas vezes as ferramentas de desenvolvimento possuem diversas restrições quanto a licença e suporte, apresentam documentação muitas vezes complexa para um estudante nessa etapa de formação e muito extensa para ser tratada em uma disciplina semestral.

Devido isso, e a partir da experiência acadêmica pessoal durante o curso de Engenharia de Computação da Universidade de São Paulo e experiência de intercâmbio acadêmico na *California State University Long Beach*, onde tive contato com a estrutura curricular de cursos na área de projeto de sistemas digitais, especialmente nas disciplinas *Computer Logic Design II* e *System on Chip Design*, pude perceber a necessidade da concepção de um projeto didático que pudesse se encaixar na realidade dos alunos do curso de Engenharia de Computação da USP São Carlos.

1.2 Objetivos

O principal objetivo deste trabalho é apresentar de forma sistemática a implementação de uma arquitetura de um SoC simples para fins didáticos. Para que através dela, o estudante possa ter o primeiro contato pós disciplinas básicas de computação com os conceitos de implementação de um sistema embarcado completo, desde a concepção da arquitetura de *hardware* ao desenvolvimento de *firmware* e implementação em placa de desenvolvimento.

Ao fim deste projeto, o estudante terá confiança para desenvolver arquiteturas mais complexas, que envolvam componentes mais modernos e robustos, com um grau maior de automatização de projeto utilizando ferramentas CADs mais complexas, utilização de IPs fornecidas por diferentes fabricantes.

1.2.1 Objetivos Específicos

Encontram-se no escopo deste projeto os seguintes objetivos específicos:

- Desenvolvimento completo de um módulo UART configurável, com suporte a paridade e capacidade de comunicação de taxas de até 115200 bits por segundo.
- Desenvolvimento de módulo de interface de memória que permita a escrita e leitura assíncrona da memória CelullarRAM 1.5 MT45W8MW16BGX 16Mbyte, contida na PCB Digilent Nexys 3.
- Integração do microcontrolador Xilinx KPCSM6 aos dispositivos desenvolvidos através de um barramento desenvolvido para a arquitetura proposta.
- Desenvolvimento de um *firmware* que explore os dispositivos periféricos da arquitetura.
- Simulação dos principais módulos implementados em Verilog HDL.
- Implementação da arquitetura na placa de desenvolvimento Digilent Nexys 3.

- Proposta de um cronograma para implementação deste projeto em uma disciplina semestral.

1.3 Estrutura da Monografia

A estrutura da monografia foi dividida entre os capítulos Introdução, Fundamentação Teórica, Materiais e Métodos, Resultados, e ao final foram acrescentados anexos e apêndices pertinentes ao projeto.

2 Fundamentação Teórica

Neste capítulo são apresentados os fundamentos teóricos de sistemas em um chip (SoC), arquitetura do microcontrolador Xilinx Picoblaze KCPM6, memória Cellram 8Mb, protocolo de comunicação UART, linguagens de descrição de hardware, conceitos da linguagem Verilog e FPGA.

2.1 Sistema em um *Chip* (SoC)

Os avanços na área da microeletrônica permitiram avanços nos processos de fabricação de *chips* com capacidade na ordem de bilhões de transistores, como por exemplo o processador Intel Core i7-3960X com 2,3 bilhões de transistores (ANGELINI, 2011). Isso tem permitido o desenvolvimento de circuitos integrados (CIs) cada vez mais complexos e com diversas funcionalidades, tais como memórias em *chip*, dispositivos analógicos, CPUs, DSPs, decodificadores/codificadores MPEG, etc. O que configura-os como sistemas em um *chip* (SoC). Em (Grant Martin, 1999) define-se um SoC como "Um CI complexo que integra a maioria dos elementos funcionais de um produto final completo em um único *chip* ou *chipset*", ainda podemos encontrar que um SoC se caracteriza por um ou mais núcleos programáveis, conectados através de barramentos com processadores de sinais, sejam eles digitais ou analógicos, podendo incluir memória em *chip*. Na Figura 1 é apresentada a arquitetura de um SoC de aplicação em multimídia com dois núcleos de processamento ARM9, memórias, controlador DMA, periféricos como controlador de interrupções e interfaces de comunicação USB e Ethernet, interconectados através de barramentos compartilhados.

A disponibilidade de diversos dispositivos em um único *chip* permitiu o desenvolvimento de aplicações mais completas, que envolvem uma combinação de *software* e *hardware*, muitas vezes complexa por consistir de diversos subcomponentes heterogêneos (Bashir M. Al-Hashimi, 2006). Devido a isso, engenheiros têm desenvolvido novas metodologias e técnicas para gerenciar a crescente complexidade dos *chips* atuais (Resve Saleh, 2006). Uma das técnicas utilizadas, na qual blocos pré-projetados e pré-testados, denominados *Intellectual Property* (IP) são obtidos de fontes internas, ou de terceiros e combinados em um único *chip* (Resve Saleh, 2006). Essas IPs reutilizáveis podem consistir em microprocessadores desde 8-bits à 64-bits RISC, sistemas de memórias, controladores de I/O, incluindo PCI, Ethernet, ADs, DAs, conversores eletro-óticos e decodificadores de vídeo MPEG, ASF ou AVI (Pierre Bricaud, 2002). Portanto, o fluxo de projeto de um SoC envolve a reutilização de diversos blocos IP interconectados por uma topologia de comunicação em barramentos que atendam requisitos de projeto, os quais incluem:

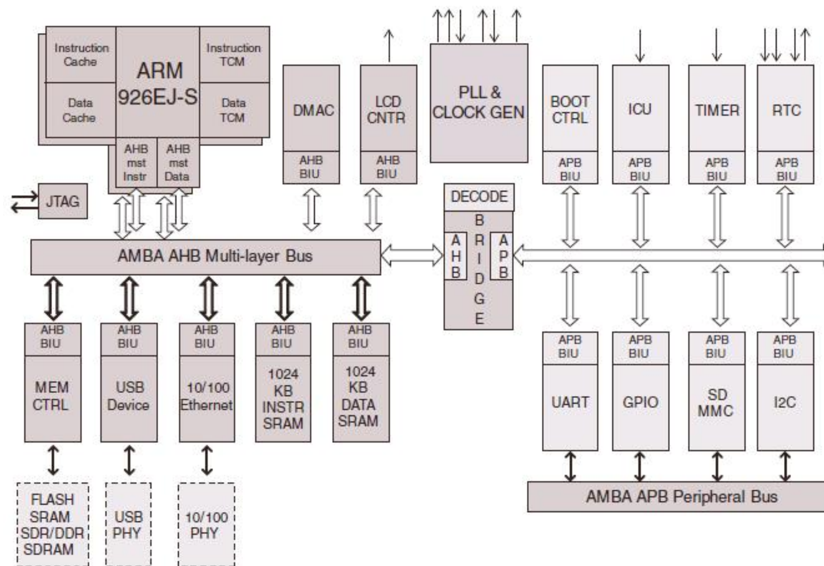


Figura 1 – Diagrama de blocos de um SoC de aplicação em multimídia (Sudeep Pasrich; Nikil Dutt, 2008).

desempenho, *power/energy/temperature*, custos, confiabilidade e *time-to-market* (Sudeep Pasrich; Nikil Dutt, 2008).

2.2 Microcontrolador Xilinx Picoblaze KCPSM6 8 bits

O KCPSM6 é uma *soft macro* – IP com maior grau de flexibilidade, uma vez que se trata de RTL sintetizável e não é específica ao processo de fabricação do *chip* – que define um microcontrolador 8 bits, o qual pode ser instanciado uma ou mais vezes à um projeto destinado às plataformas Xilinx Spartan-6. O microcontrolador KCPSM6 é uma IP que utiliza poucos recursos de FPGA, utilizando apenas 4,6% da FPGA Xilinx XC6SLX4 e somente 0,11% do modelo mais avançado Xilinx XC6SLX150T (CHAPMAN, 2013).Além disso, a IP possui licença gratuita e com bastante documentação de fácil compreensão por um aluno de graduação. Somado isso, a facilidade de integração da IP à projetos de SoCs customizados são aspectos que corroboram para a utilização do mesmo no projeto de um SoC para fins educacionais.

O KCPSM6 é um microcontrolador de 8 bits de dados com memória de programa capaz de armazenar até 4 mil instruções. Todas as instruções são de tamanho de 16-bits e necessitam de 2 ciclos de *clock* para serem executadas. A IP disponível suporta *clock* máximo do FPGA ao qual a arquitetura dá suporte. No caso deste projeto, utilizou-se o FPGA Xilinx Spartan-6 com *clock* de 100MHz, permitindo que microcontrolador atinja 50MIPS.

KCPSM6 possui dois bancos de 16 registradores de propósito geral, os quais fazem parte do fluxo principal de dados na arquitetura, como pode ser visualizado na Figura 2.

O fluxo de dados comumente verificado na arquitetura do KCPSM6 se inicia com a leitura de dados através da porta de entrada (*in port*) e armazenamento em um dos registradores disponíveis, seguida de alguma operação aritmética utilizando a *ALU* (*Aritimetic Logic Unit*) e armazenamento do resultado em algum registrador. O dado operado pode ser transferido para algum dispositivo externo através da porta de saída (*out port*).

A ALU do microcontrolador é capaz de realizar uma série de operações matemáticas, as quais incluem soma, subtração, comparação, algumas variações de deslocamento e rotação. As operações são realizadas sobre dados armazenados nos bancos de registradores ou valores constantes definidos durante a programação e armazenados na memória de programa. O resultado das operações atualiza as flags de zero (Z) e *carry* (C). Para uma compreensão detalhada do conjunto de instruções do microcontrolador, aconselha-se a leitura de (CHAPMAN, 2013).

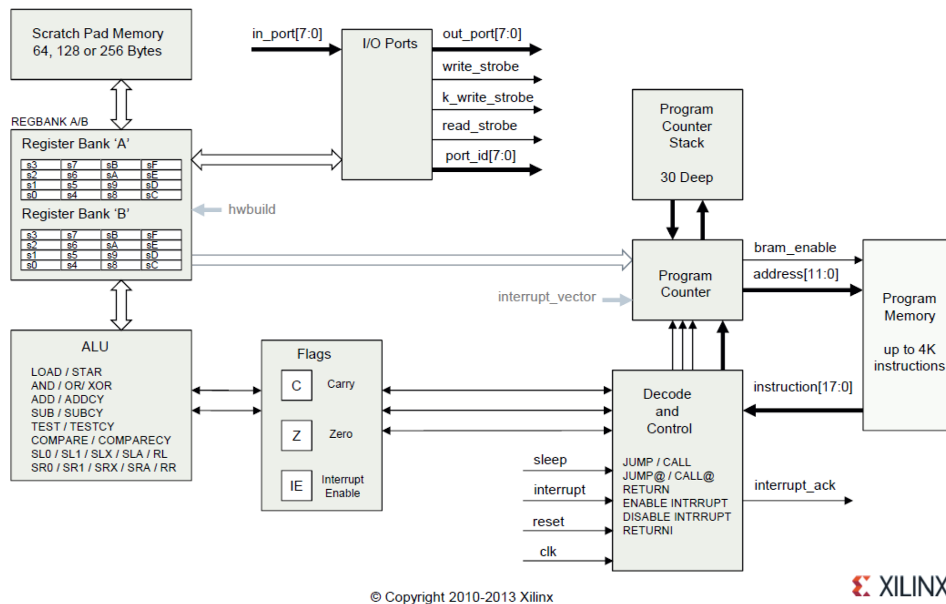


Figura 2 – Diagrama funcional da arquitetura do microcontrolador KCPSM6 (CHAPMAN, 2013).

A arquitetura ainda disponibiliza a *Scratch Pad Memory*, que consiste em uma memória de acesso aleatório de 64Kb (por padrão), 128Kb ou 256 Kb, que pode ser utilizada em casos de salvamento de contexto. Essa funcionalidade é útil quando a aplicação utiliza muitos registradores do banco e necessita que o contexto seja salvo quando há uma chamada de função.

A Comunicação entre o microcontrolador e o mundo externo se dá através do módulo *IO Ports*, como pode ser visto no diagrama de blocos. Com comprimento de 8 bits, o barramento de endereço *port_id* pode endereçar até 256 posições diferentes de escrita e leitura em dispositivos. A integração de dispositivos de entrada e saída deve ser feita conforme as ilustrações das Figuras 3 e 4 respectivamente. A partir das Figuras,

é possível observar que os dispositivos compartilham o barramento de endereço para operações de escrita e leitura no módulo *IO Ports*, por tanto deve-se prever isso no projeto do barramento. A transferência de dados nas portas de E/S(Entrada/Saída) é controlada utilizando as instruções "*INPUT*"e "*OUTPUT*"do microcontrolador. Ainda, o mecanismo é controlado pelo sinal *read_strobe* e *write_strobe*, que são pulsos de valor '1' que sinalizam a ocorrência de uma operação de escrita ou de leitura.

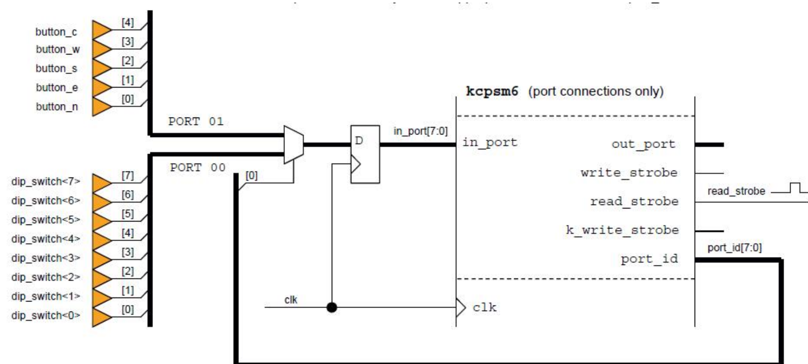


Figura 3 – Esquemático exemplificando como integrar a funcionalidade de porta de entrada a um projeto (CHAPMAN, 2013).

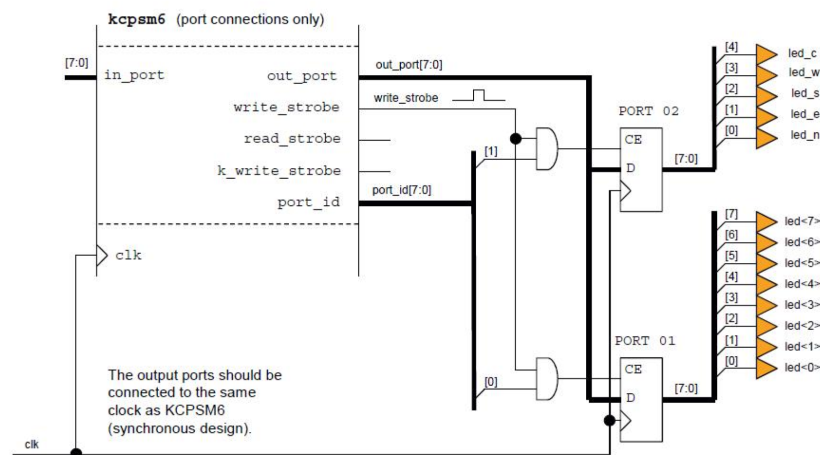


Figura 4 – Esquemático exemplificando como integrar a funcionalidade de porta de saída a um projeto (CHAPMAN, 2013).

2.3 Memória CelullarRAM 1.5 MT45W8MW16BGX 16 Mbytes

A memória CelullarRAM 1.5 é uma memória CMOS de alta velocidade desenvolvida para aplicações de baixa potência. A memória possui 16Mb DRAM, organizados como 8Mb x 16 bits, disponibilizada na placa de desenvolvimento Nexys 3 utilizada neste projeto, e integrada ao FPGA Spartan 6 que a compõe. A arquitetura da MT45W8MW16BGX é organizada segundo o diagrama funcional apresentado na Figura 5. A memória possui dois

registradores acessíveis ao usuário, que controlam o funcionamento do dispositivo. O RCR (*Refresh Configuration Register*), responsável por controlar como a operação de *refresh* é realizada, e o registrador BCR (*Bus Configuration Register*), com função de definir como o dispositivo interage com o barramento, o que inclui a definição do modo de operação entre assíncrono, paginada ou *burst* e outros parâmetros de configuração pertinentes ao modo de operação selecionado. Esse dispositivo possui barramento de endereço de 23 bits e barramento de dados de 16 bits.

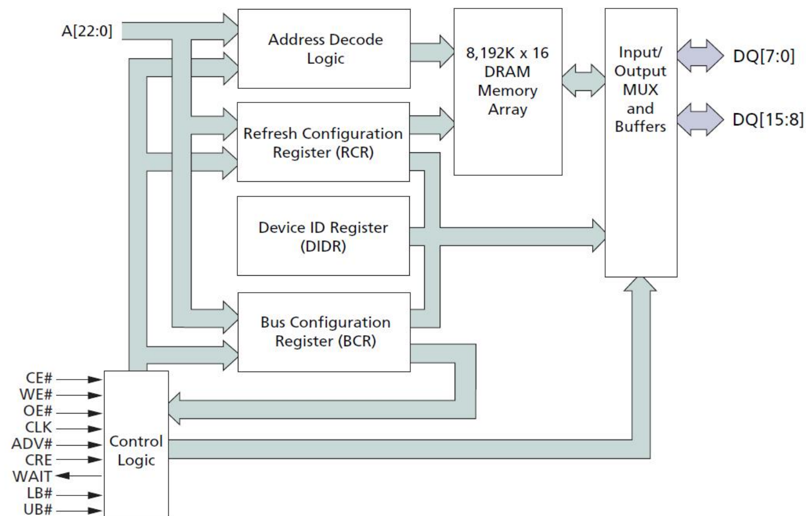


Figura 5 – Diagrama funcional da arquitetura da memória CelullarRAM 1.5 MT45W8MW16BGX (MICRON, 2004).

Neste projeto o modo de operação escolhido para e memória foi o assíncrono, uma vez que não existe requisitos de desempenho que justifiquem a implementação dos outros modos. Por este motivo, desenvolve-se a partir deste ponto apenas as fundamentações teóricas pertinentes a esse modo de operação.

Esse dispositivo quando inicializado é configurado em modo assíncrono por padrão. Esse modo utiliza os sinais de controle padrão da indústria para controle de barramento de memórias SRAM, que incluem os sinais CE, WE, OE, LB, UB apresentados na Figura 5. A operação de leitura, apresentada na Figura 6, é inicializada levando os sinais CE, OE e LB/UB à zero enquanto mantem-se WE em nível lógico um. O dado requisitado na operação é mantido disponível até que um dos sinais CE ou OE volte para o valor lógico um. Por outro lado, a operação de escrita é realizada mantendo-se os sinais CE, WE em zero, como é exemplificado na Figura 7, sendo que o tempo que o sinal WE deve permanecer em zero lógico não deve exceder o tempo tCEM - tempo máximo que o sinal WE pode permanecer em nível lógico zero - definido no manual do fabricante.

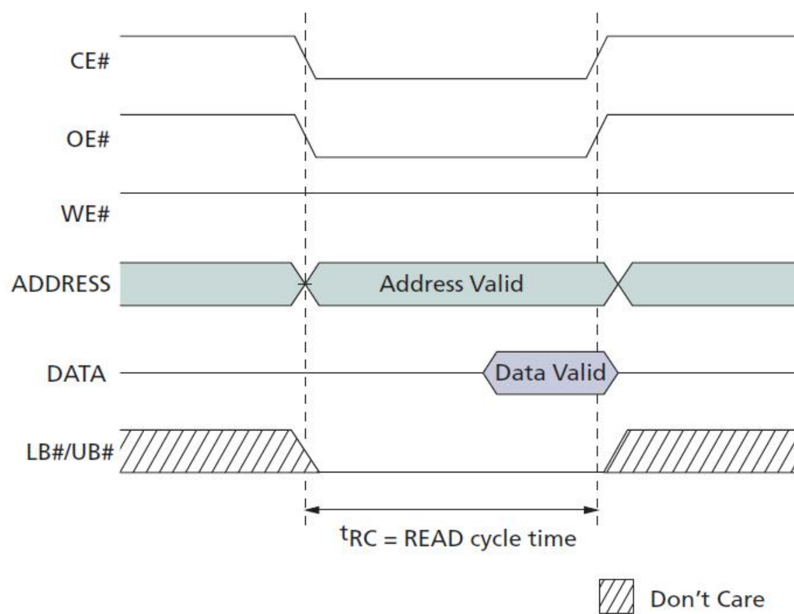


Figura 6 – Sinais de controle, dado e endereço durante operação de leitura em memória (MICRON, 2004).

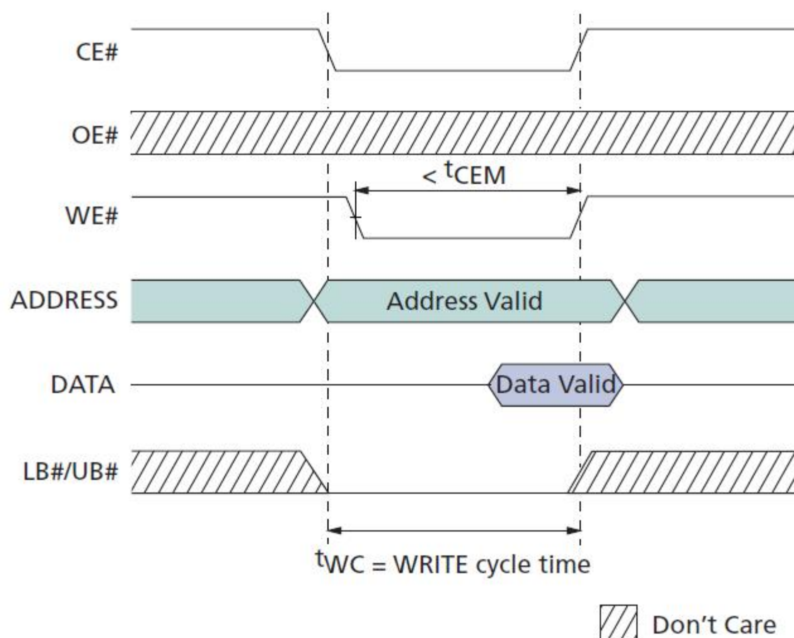


Figura 7 – Sinais de controle, dado e endereço durante operação de escrita em memória (MICRON, 2004).

2.3.1 Requisitos de tempo durante as operações de leitura e escrita assíncronas

Na seção anterior foram apresentadas as condições de nível lógico dos sinais de controle para as operações de leitura e escrita em memória. Tais condições são necessárias, mas não suficientes para que as operações tenham o resultado esperado. Além destas condições, é necessário que os sinais de controle respeitem condições impostas por requisitos

de tempo definidos pelo fabricante, os quais podem ser encontrados em anexo e que são disponibilizados nos manuais do fabricante da memória.

Quando da implementação de um módulo de interface para memória, deve-se projetar um módulo que seja capaz de produzir os sinais de controle necessários à efetivação da operação requerida. Para a operação de leitura apresentada na Figura 8, observa-se que tAA (tempo entre a identificação de um endereço válido e a disponibilização do dado) é de no máximo 85 ns, e o tRC (tempo de um ciclo de leitura) é de no mínimo 85 ns.

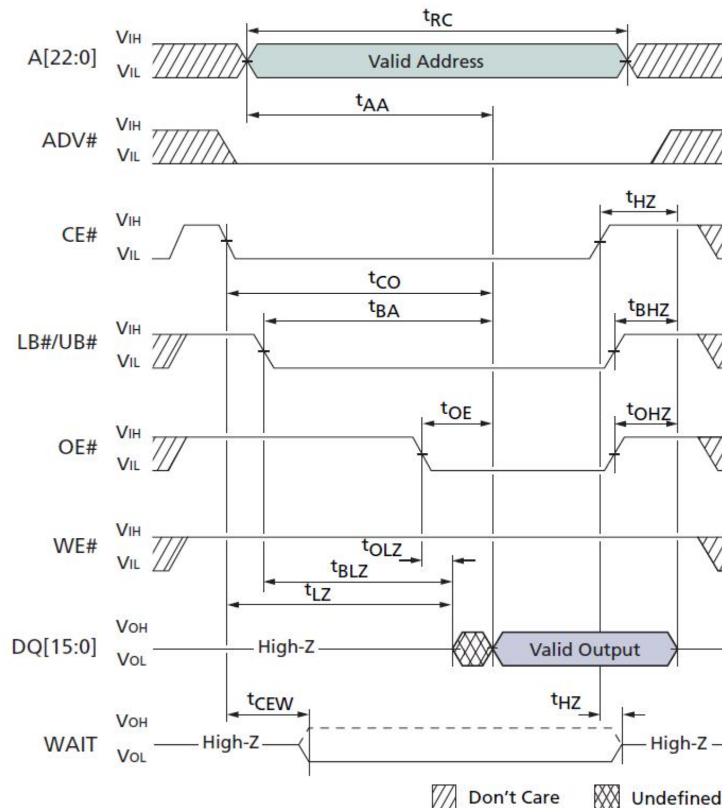


Figura 8 – Requisitos de tempo durante a operação de leitura em memória (MICRON, 2004).

Os requisitos de tempo para a operação de escrita são apresentados na Figura 9. Destaca-se neste caso a importância do tempo tWPH (tempo do pulso WE em nível lógico um), neste caso o valor é de no mínimo 10ns como consta na tabela em anexo, o tempo tWP (tempo do pulso WE em nível lógico zero), neste caso de no mínimo 45ns, e o tempo tWC (tempo de um ciclo de escrita), neste caso de no mínimo 85ns como consta na tabela em anexo.

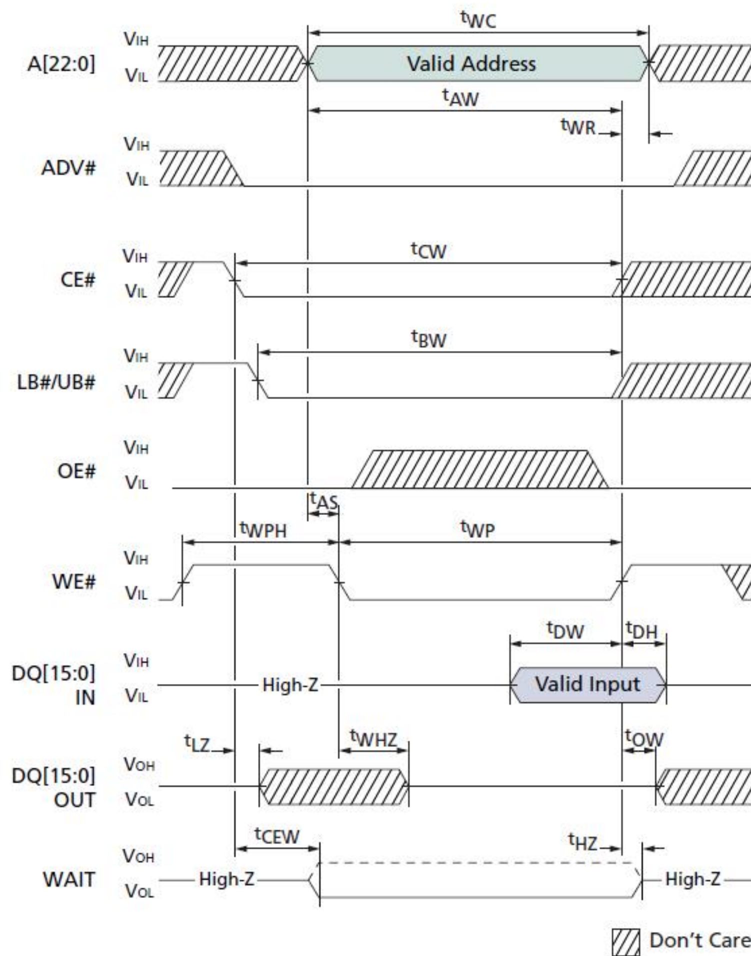


Figura 9 – Requisitos de tempo durante a operação de escrita em memória (MICRON, 2004).

2.4 UART (*Universal Asynchronous Receiver/Transmitter*)

UART se refere à *Universal Assynchonus Receiver/Transmitter*, que consiste em um módulo de comunicação que transmite e recebe dados de forma seqüencial. O protocolo UART recebe dados organizados em bytes e transmite cada um dos bits de forma sequencial. Os bits são recebidos do outro lado do canal de comunicação e reconvertidos em bytes. A conversão entre a forma paralela para a forma seqüencial, também chamada de serial, é realizada por registradores com operação de *shift* localizados em cada uma das extremidades do canal de comunicação.

UART define apenas como os dados devem ser transmitidos logicamente. Podendo fisicamente os sinais serem transmitidos segundo diversos padrões, alguns exemplos são os padrões RS-232, utilizado neste projeto, RS-422 e RS-485, os quais são definidos pela EIA (*Electronic Industries Alliance*).

A seqüência de bits transmitidos pela UART segue o seguinte padrão: quando não

se está transmitindo dados no canal, o mesmo deve ser mantido em nível logico um. Uma transação de dados se inicia com um bit de início com valor lógico zero, seguido dos bits de dados na ordem do menos para o mais significativo, seguido de um bit opcional de paridade e de um ou dois bits de parada de valor lógico um, como mostrado na Figura 10.

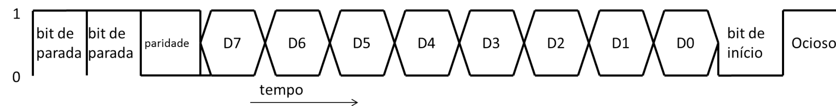


Figura 10 – Ordem dos bits segundo o protocolo UART.

Para o funcionamento correto da UART, é necessário que o transmissor e receptor estejam configurados com os mesmos parâmetros. Isso implica que ambos devem estabelecer o número de bits que serão transmitidos em cada pacote de dados e o tempo de duração de cada bit. O tamanho do pacote de dados pode variar de acordo com a necessidade de transmissão do bit de paridade, se são um ou dois bits de parada e qual o número de bits de dados configurado pelo usuário, que pode variar geralmente entre sete e oito. A duração que o transmissor deve manter o valor do bit inalterado para que o receptor seja capaz de amostrar o valor correto do bit, é chamado de tempo de bit, o qual depende da taxa de bits por segundos (BAUD) desejada. O tempo de bit pode ser calculado como o inverso da taxa de BAUD (bits por segundo). As taxas de BAUD são apresentadas na Tabela 1 juntamente com os seus respectivos tempos de bit.

Tabela 1 – Tempo de bit para cada um dos valores de BAUD comumente utilizado em projetos.

BAUD	**Tempo de Bit**
300	3,3333 ms
1200	833,33 us
2400	416,66 us
4800	208,33 us
9600	104,16 us
19200	52,083 us
38400	26,041 us
57600	17,361 us
115200	8,6806 us

2.5 Linguagens de Descrição de *Hardware* (HDL)

Linguagens de descrição de *hardware* (HDLs) compõem uma classe de linguagens que são usadas na área de eletrônica com a finalidade de descrever o comportamento de um circuito elétrico, comumente utilizadas na área de circuitos eletrônicos digitais. Elas

podem são utilizadas no processo de descrição do comportamento do circuito, geração de estímulos utilizados em etapas de simulação e verificação do circuito e geração de *netlists* utilizadas na descrição da implementação física do circuito.

HDLs apresentam uma estrutura similar, à primeira vista, a uma linguagem estrutural. Porém, apresentam características fundamentais que são indispensáveis à funcionalidade para qual foram criadas. Uma dessas características se refere à possibilidade de simulação da descrição de *hardware* em ambiente de simulação que permite ao projetista simular o comportamento concorrente do *hardware* ao longo do tempo. Isso é possível, graças aos programas de sínteses que são responsáveis por gerar o *netlist* utilizado pelos simuladores.

HDLs podem ser utilizadas em diversos níveis de projeto. Um projeto de um circuito digital se inicia com a modelagem comportamental em diversos níveis hierárquicos do *hardware* utilizando uma HDL, o resultado dessa etapa é colocado sob diversos testes de simulação descritos também em HDLs (chamados de casos de teste) e com a finalidade de verificação do seu correto funcionamento. Após essa etapa, a descrição em HDL em conjunto com uma biblioteca de componentes é usada por uma ferramenta de síntese para a geração automática de síntese. Além disto, essas ferramentas incluem uma etapa de otimização da lógica interna do circuito gerado, antes da geração das estruturas internas de armazenamento, da lógica combinacional e da estrutura de conexão dos componentes (*netlist*) (MIDORIKAWA, 2007).

Algumas vantagens da utilização de HDLs incluem:

- Projetistas podem desenvolver em um nível alto de abstração, sem a necessidade de se preocuparem com uma determinada tecnologia que deva ser utilizada na implementação do *hardware*. Com isso, se mudanças forem necessárias na lógica, ou se novas tecnologias forem lançadas não é necessário projetar a descrição do *hardware* novamente. Sendo necessário apenas, que o projetista passe a descrição pela ferramenta de síntese novamente e especifique a biblioteca dos componentes da nova tecnologia que deve ser utilizada no processo (PALNITKAR, 2003).

- A utilização de HDLs permite que simulações e verificações sejam feitas em etapas cada vez mais cedo do projeto. Desta forma, *bugs* são identificados mais cedo o que leva a um diminuição de custos e tempo de projeto.

- HDLs são linguagens textuais que são de mais fácil entendimento do que outros tipos de representação, como por exemplo esquemáticos. O que as torna de mais fácil manutenção.

2.5.1 Linguagem de Descrição Verilog

O desenvolvimento da linguagem Verilog se iniciou na companhia *Gateway Design Automation* por volta de 1984. A linguagem foi desenvolvida visando possuir as características da HDL HiLo, bastante popular na época, e da linguagem estrutural C. O primeiro simulador vendido pela *Gateway Design Automation* foi desenvolvido em 1984 com subseqüentes modificações até o ano de 1987 .

Em 1990 a *Cadence Design System* adquiriu a *Gateway Design System* e tornou a linguagem de domínio público, criando a Open Verilog International (OVI) com o temor da indústria aderir em massa à linguagem VHDL. A linguagem se tornou um padrão IEEE em 1995, já tendo as seguintes extensões desde então: Verilog-95 (padrão IEEE 1364-1995), Verilog 2001 (IEE 1364-2001) e Verilog 2005 (IEE1364-2005) (MIDORIKAWA, 2007).

Verilog permite ao usuário representar o *hardware* de forma hierárquica assim como apresentado na Figura 11. Os elementos básicos da linguagem que permitem isso são os módulos (*module*) e as portas (*ports*). Os módulos são as unidades básicas do modelo do *hardware*, os quais podem instanciar quantos módulos forem necessários. As portas são as entidades que permitem que os módulos instanciados troquem informações com outros módulos, as quais podem ser do tipo de *output*, *input* e *inout*.

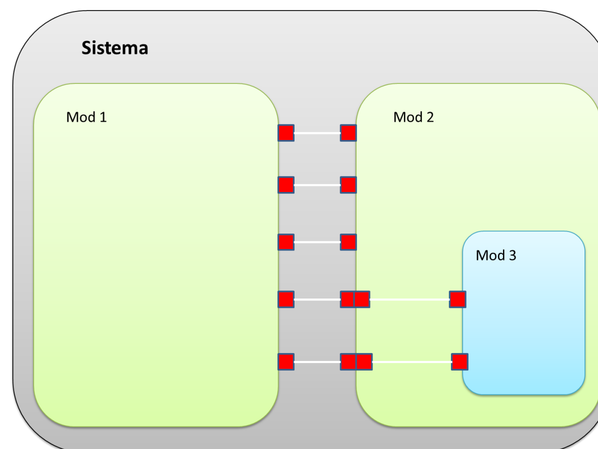


Figura 11 – Modelo de representação hierárquico utilizando Verilog.

2.6 FPGA (*Field Programmable Gate Array*)

Field Programmable Gate Arrays (FPGAs) são *chips* semicondutores que são baseado em uma matriz de blocos lógicos configuráveis (CLBs) conectados por conexões reconfiguráveis (Stephen Brown, 2000) como apresentado na Figura 12. Essa característica permite que as FPGAs possam ser reprogramadas de acordo com a funcionalidade requerida pela aplicação, ao contrário dos *Application Specific Integrated Circuits* (ASICs) que são projetados com a finalidade de uma aplicação específica.

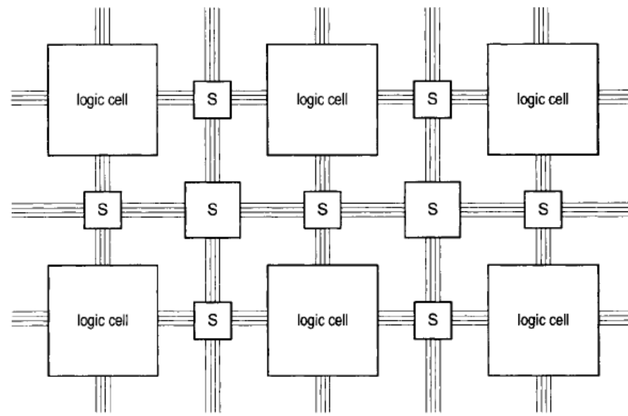


Figura 12 – Arquitetura de uma FPGA (Stephen Brown, 2000).

CLBs são as unidades básicas que compõem uma FPGA. Eles são compostos geralmente por um número pequeno de LUTs (*Look up Tables)*, alguns circuitos selecionadores (MUX), e *flip-flops.* As LUTs são reconfiguráveis e podem implementar diferentes lógicas combinacionais (CHU, 2008). As interconexões entre os diversos CLBs são flexíveis e permitem que sejam reconfiguradas dependendo da aplicação, sendo as regras de roteamento podendo ser geradas automaticamente por ferramentas de síntese, reduzindo desta forma a complexidade do projeto. Ainda, as FPGAs atuais disponibilizam blocos com funcionalidade específicas, os quais são geralmente fabricados em nível de transistores e suas funcionalidades complementam os blocos lógicos genéricos. Esse blocos incluem bancos de registrados de I/O, memória em *chip*, multiplicadores e circuitos gerenciadores de *clock* (CHU, 2008).

O fluxo de projeto visando a utilização de FPGA se difere em alguns aspectos em relação a de um visando a implementação de um ASIC. O primeiro elimina complexidade em etapas que envolvem o processo fabril do *chip*, uma vez que a partir de um modelo em HDL e em conjunto com ferramentas automáticas de síntese o projeto pode ser implementado em um dispositivo FPGA previamente disponibilizado no mercado. Apesar da etapa de síntese permitir, muitas vezes, total automatização, os fabricantes permitem que diversos parâmetro possam ser configurados com intuito de permitir a otimização de desempenho do projeto. A Figura 13 apresenta as diferenças entre os dois fluxos de projeto, constata-se que o tempo de projeto se encurta utilizando FPGAs uma vez que não se faz necessário etapas relacionadas à fabricação da pastilha.

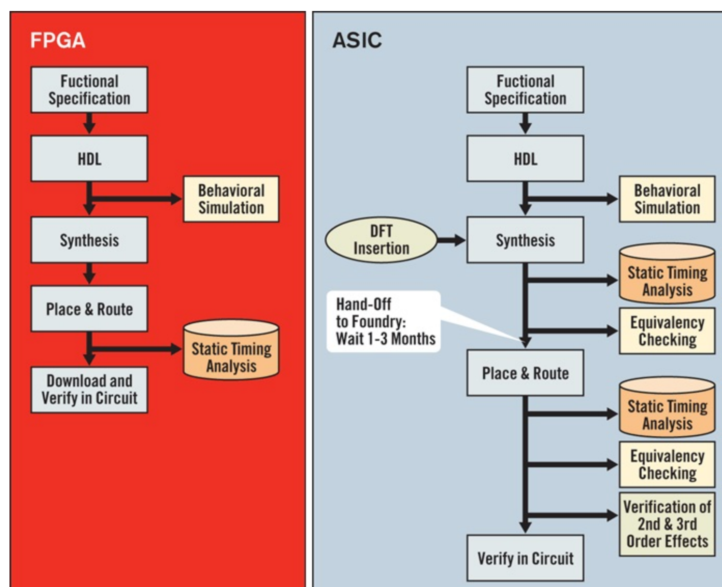


Figura 13 – Diferença entre fluxos de projeto visando FPGA e ASIC (XILINX, 2014).

3 Materiais e Métodos

Este capítulo apresenta o ambiente de desenvolvimento, dispositivos utilizados no projeto, além de como os diversos blocos funcionais presentes na arquitetura do SoC foram implementados e os detalhes necessários para implementação do protótipo.

3.1 Placa Nexys 3 e ambiente de desenvolvimento

Este projeto foi desenvolvido visando a implementação na placa de desenvolvimento Digilent Nexys 3 apresentada na Figura 14. A escolha dessa placa se deve ao fato da mesma ser de baixo custo e ser compatível com as ferramentas de desenvolvimento ISE da Xilinx sob a licença *WebPack* gratuita. A placa possui a FPGA Xilinx Spartan 6 XC6LX16-CS324, memória Micron Celullar RAM 16MBytes, USB-UART, porta USB utilizada para alimentação, programação e transferência de dados, chaves, botões , *display* de sete segmentos, porta Ethernet e conector VGA. A placa possui um oscilador de freqüência de 100MHz na qual o SoC operará.

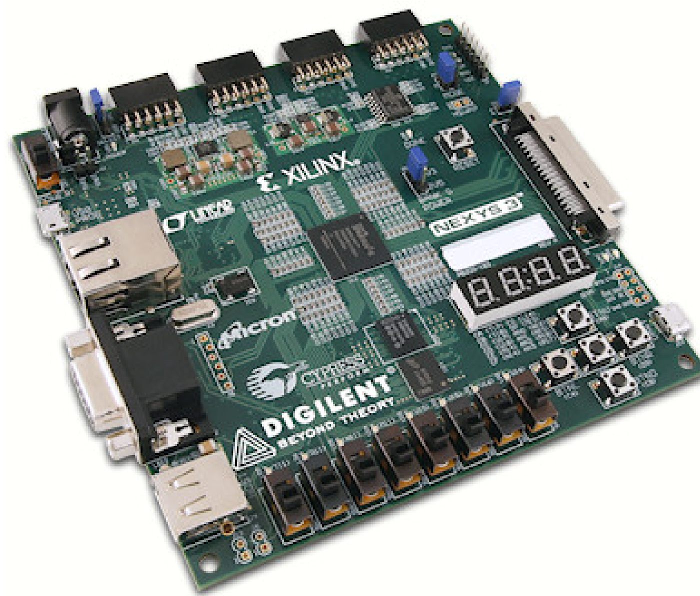


Figura 14 – Placa de desenvolvimento Digilent Nexys 3.

3.2 Blocos desenvolvidos em Verilog

Essa seção apresenta a descrição dos blocos funcionais que foram implementados em Verilog HDL e implementados na FPGA Xilinx Spartan 6. Devido à flexibilidade de

projeto de *hardware* utilizando HDLs, os códigos podem ser portados para outras FPGAs, inclusive de fabricantes diferentes, sendo exceção apenas o microcontrolador utilizado, que é compatível apenas com os modelos de FPGAs da série 6 da Xilinx. Isso ocorre devido ao fato de que o processador é otimizado para esses dispositivos e utilizam bibliotecas específicas do dispositivo a ser implementado. Apesar disso, os conceitos envolvidos nessa seção são aplicáveis em qualquer projeto que se venha a se desenvolvido utilizando outras tecnologias.

3.2.1 Arquitetura do SoC

A arquitetura do SoC desenvolvido neste projeto é composta pelos quatro grandes blocos funcionais: Picoblaze, *Bus Control*, *Memory Interface* e UART, como apresentado na Figura 15. A módulo *memory* é apresentado no diagrama para melhor compreensão do sistema, uma vez que o mesmo não representa um módulo implementado em HDL, mas sim um *chip* de memória externo à FPGA.

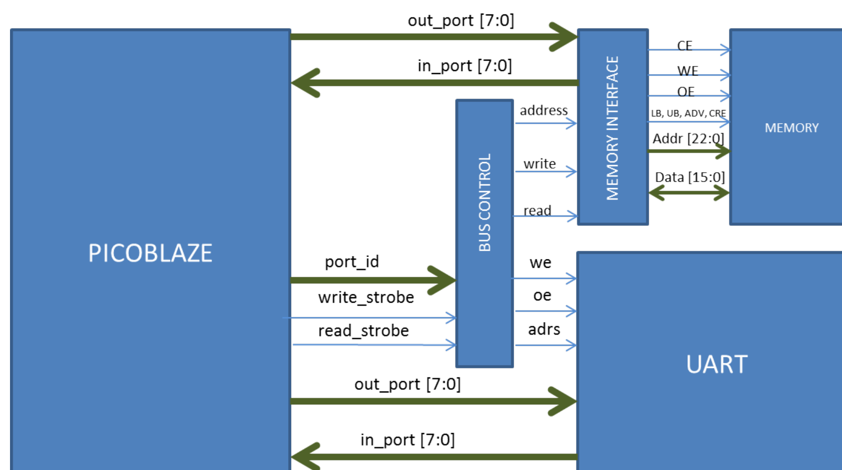


Figura 15 – Diagrama de bloco funcionais do SoC implementado.

O módulo Picoblaze corresponde ao microcontrolador 8 bits KCPSM6 disponibilizado em Verilog pela Xilinx e descrito na seção 2.2. A UART corresponde à uma implementação de um módulo contendo transmissor/receptor. *Memory Interface* é um módulo implementado com a finalidade de permitir que o microcontrolador de 8 bits seja capaz de escrever e ler dados na memória de barramento de endereço de 23 bits e barramento de dados de 16 bits. Para que isso seja possível, o módulo desenvolvido possui registradores de 8 bits endereçáveis um de cada vez para escrita e leitura. O módulo *BUS Control* é responsável por decodificar os endereços das portas (*port id*) do Picoblaze e permitir a comunicação do microcontrolador com o dispositivo designado por esse endereço, configurando-se como um barramento bastante simples.

3.2.2 Módulo UART

Esse módulo performa a conversão de serial para paralelo dos dados recebidos de um dispositivo externo, e converte dados recebidos pelo microcontrolador da forma paralela para a forma serial. O Status da UART pode ser lido a qualquer momento pela CPU. Esse módulo têm suporte a transmissão de dados de oito ou sete bits, e transmissão com paridade opcional. Esse método não inclui FIFO o que exige que a CPU mantenha verificação periódica da situação do dispositivo para transmissão. Este módulo têm suporte para a transmissão de dados nas taxas apresentadas na Tabela 1 da seção 2.4.

A Figura 16 apresenta um gerador de *clock* conceitual da UART. O módulo gera a partir do *clock* um sinal denominado BCLK, de freqüência igual à 16 vezes a freqüência de BAUD, a partir da contagem de valores previamente tabelados no módulo *BAUD TABLE* e calculados segundo a freqüência de operação de 100MHZ. Quando a UART está recebendo dados, o bit é amostrado no oitavo ciclo do BCLK. Para isso, o módulo possui um contador de 8 e 16 que sinaliza com um pulso o fim da sua contagem. A Tabela 2 apresenta os valores de taxas de BAUD efetivamente geradas utilizando a técnica descrita acima.

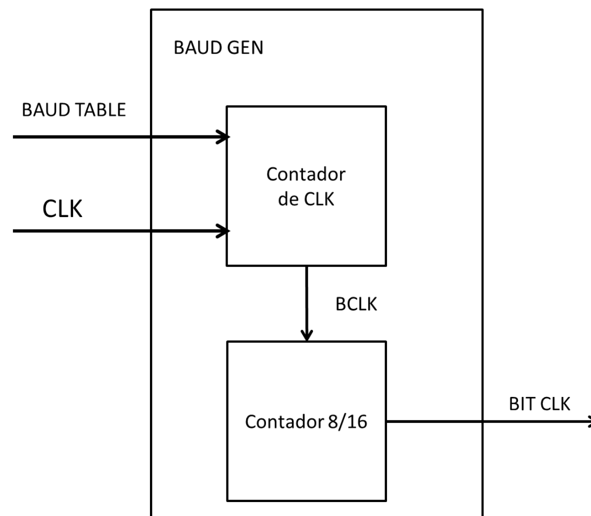


Figura 16 – Gerador de *clock* da UART.

Tabela 2 – Taxas de BAUD para *clock* de 100MHz.

BAUD	Tempo de Bit	Contador de CLK	Erro (%)	BAUD Efetiva
300	3.3333ms	20833	-0.000030	300.003
1200	833.33us	5208	-0.000048	1200.005
2400	416.66us	2604	0.000019	2399.998
4800	208.33us	1302	-0.000768	4800.077
9600	104.16us	651	0.003072	9599.693
19200	52.083us	326	-0.001229	19200.123
38400	26.041us	163	0.152311	38384.769
57600	17.361us	109	-0.003686	57600.369
115200	8.6806us	54	0.005898	115199.410

A Figura 17 apresenta a arquitetura desenvolvida para a UART. Os módulos TX FSM e RX FSM são máquinas de estados de Moore responsáveis por controlarem o processo. O dispositivo TX é responsável pela transmissão dos dados, enquanto que o RX pela recepção dos dados. Cada um dos dispositivos possui um contador de bits e um registrador com operação de *shift* responsável pela conversão serial/paralela. O Status da UART pode ser acessado a qualquer momento utilizando o seu endereço correspondente (0x01). Dentre os bits de *status* disponíveis temos o de erro de paridade, erro de *overflow*, TX disponível (*Txrdy*) e dado disponível no RX (*Rxrdy*).

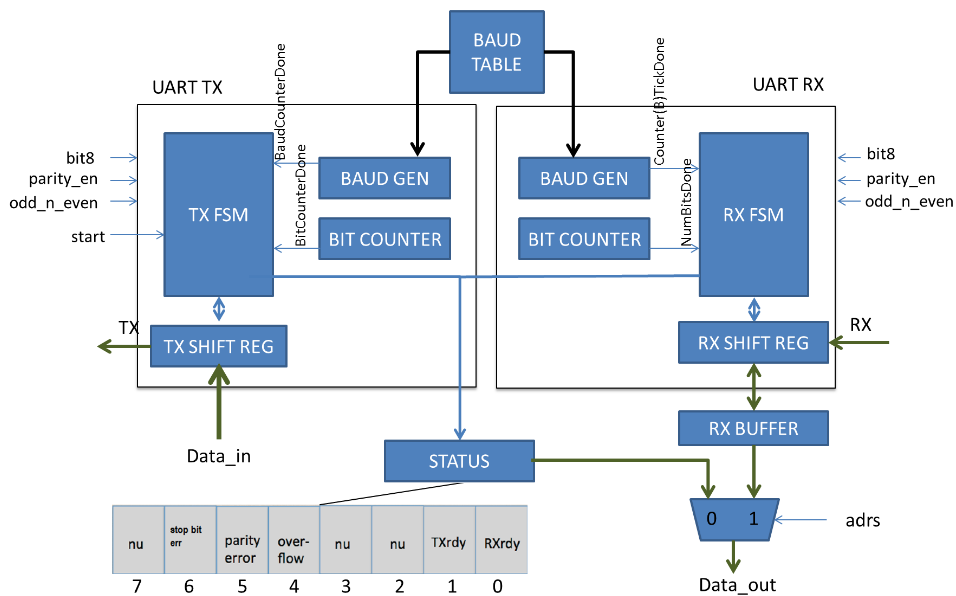


Figura 17 – Arquitetura do módulo UART.

O módulo TX FSM controla o processo de transmissão dos bits. A máquina de estados utilizada é apresentada na Figura 18. Ao iniciar o sistema (*reset*), a máquina de estados permanece em estado ocioso esperando por dados a serem transmitidos. A transmissão se inicia quando o sinal *start* vai para o valor lógico um, em seguida os dados são carregados em um registrador com operação de *shift*. Por fim, os bits são gerados utilizando o auxilio dos módulos BAUD GEN, para o controle de tempo de Bit, e o módulo BIT COUNTER, para controle de número de bits transmitidos.

O Módulo RX FSM controla o processo de recepção dos bits. A máquina de estados implementada é apresentada na Figura 19. Ao iniciar o sistema, a máquina de estados permanece em estado ocioso a espera que a recepção seja iniciada. A recepção se inicia quando o bit de início de valor lógico zero é identificado. A amostragem dos bits é feita sempre na metade do tempo de bit. Por tanto, espera-se metade do tempo de bit no primeiro bit recebido e então espera-se um tempo de bit antes da próxima amostragem para os demais bits. A amostragem é feita realizando-se a operação de *shift* no momento

adequado. Ao fim do processo, o dado é armazenado em um registrador denominado *Buffer RX*, onde permanece até que seja sobrescrito por um novo byte recebido.

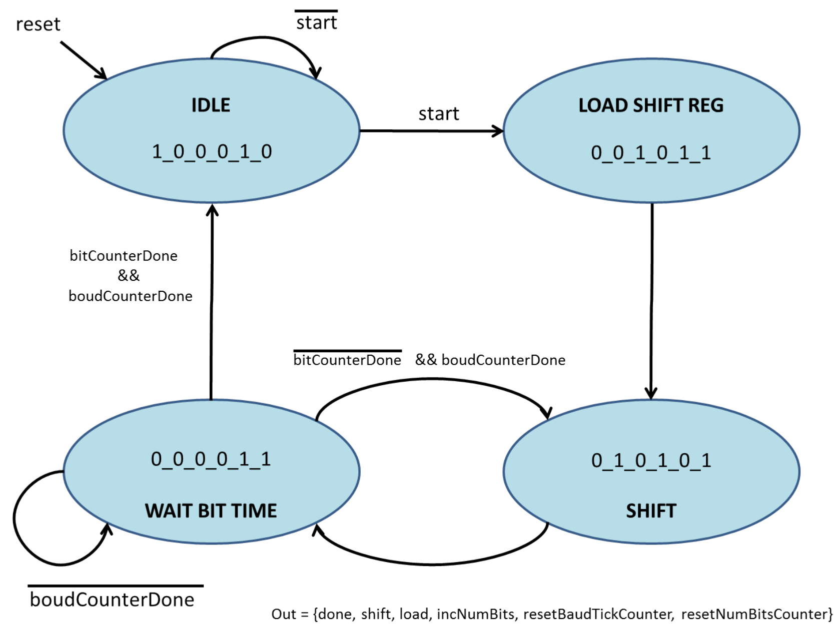


Figura 18 – Máquina de estados do dispositivo de transmissão (TX).

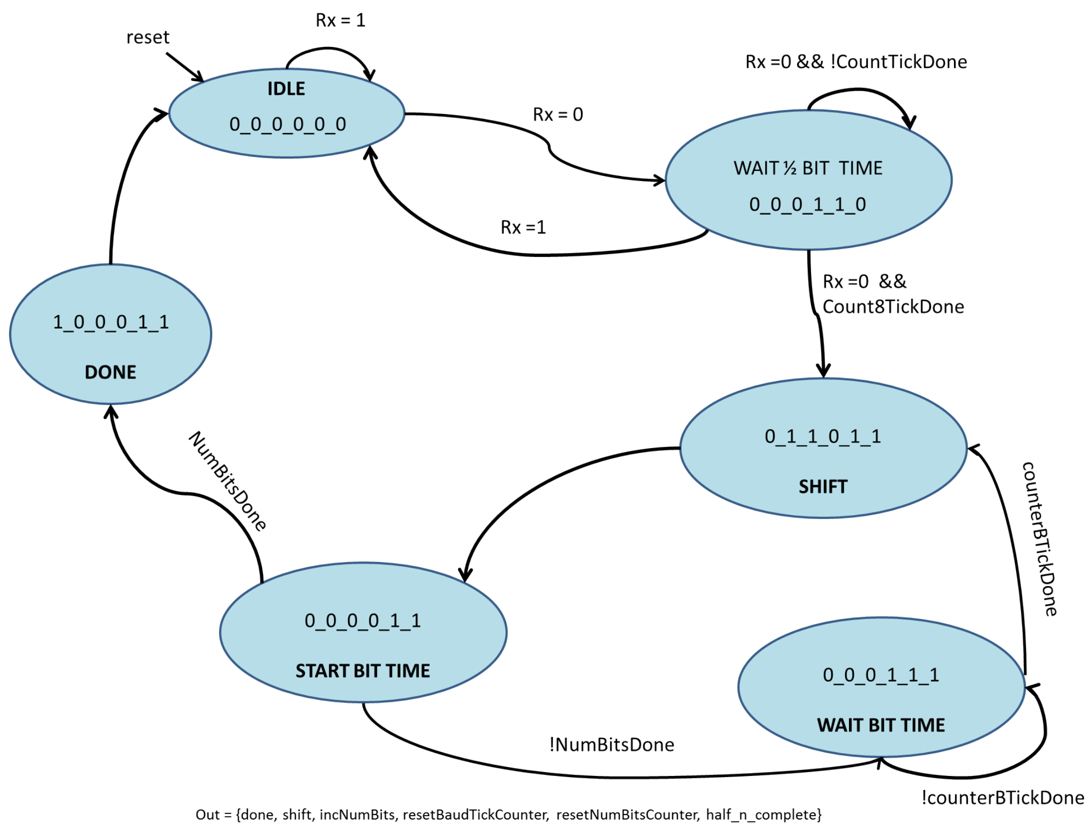


Figura 19 – Máquina de estados do dispositivo de recepção (RX).

3.2.3 Módulo *Memory Interface*

O módulo *Memory Interface* possui duas funcionalidades: a primeira consiste em permitir a comunicação do microcontrolador de 8 bits com a memória, a qual possui barramento de endereço de 23 bits e barramento de dados de 16 bits, e a segunda consiste em gerar os sinais de controle necessários às operações de leitura e escrita e que respeitem os requisitos de tempo apresentados na seção 2.3.1.

Como apresentado na Figura 20, o módulo possui três registradores de 8 bits que compõem o endereço de memória, dois conjuntos de dois registradores de 8 bits responsáveis pelo armazenamento dos dados lidos ou que serão escritos em memória e um registrador de status que tem a função de notificar se o módulo está ocupado em alguma operação. As operações de leitura e escrita nos registradores desse módulo e a ativação das operações de escrita e leitura em memória são feitas utilizando os endereços apresentados na Tabela 3 da seção 3.2.4.

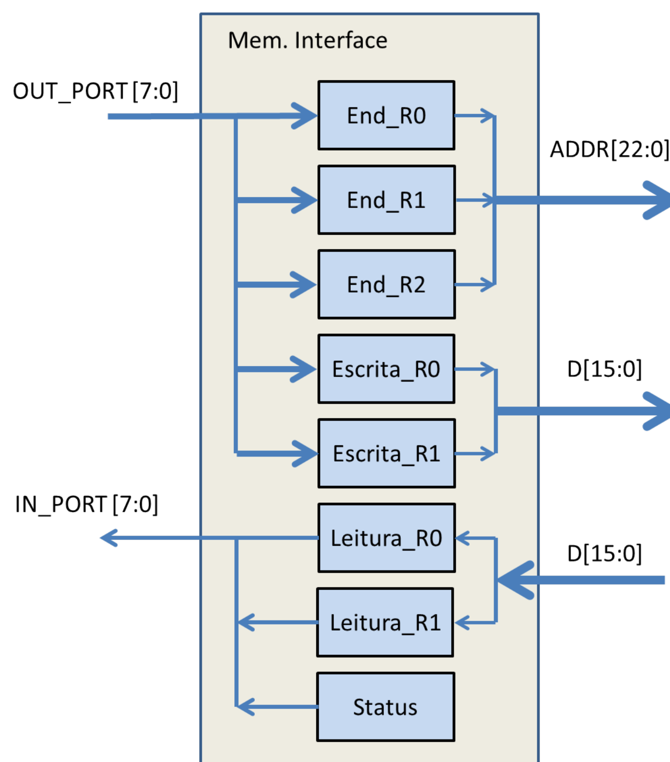


Figura 20 – Registradores do módulo de interface com a memória.

A interface de memória (*Memory Interface*) deve gerar os sinais de controle que respeitem os requisitos de tempo de cada uma das operações da memória. Os requisitos de tempo de leitura e escrita foram apresentados na seção 2.3.1. A partir dos gráficos de tempo das operações de escrita e leitura, encontrou-se condições que aplicada aos sinais de controle satisfizessem os requisitos de tempo das operações. Analisando os gráficos chegamos às seguintes conclusões:

- Requisitos de tempo para operação de leitura: Observa-se que levando CE, OE e LB/UB para zero após a disponibilização do endereço no barramento, e mantendo os sinais nesse nível por 90ns, os requisitos de tempo são satisfeitos e a leitura do dado pode ser feita corretamente. A solução é apresentada na Figura 21;

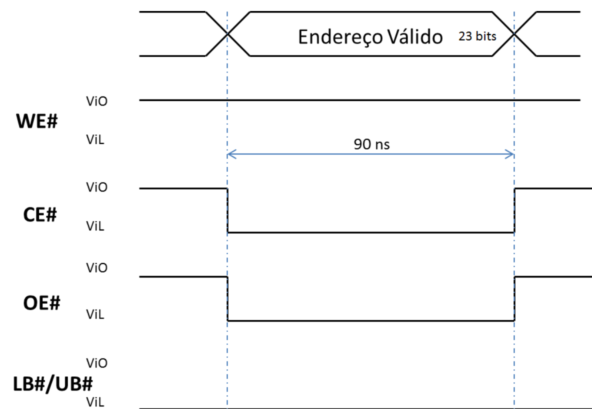


Figura 21 – Sinais de controle durante a operação de leitura.

- Requisitos de tempo para operação de escrita: Observa-se que garantindo que ao iniciarmos uma operação de escrita o sinal WE permaneça 10 ns em nível lógico um, enquanto CE e OE permaneçam em zero, seguido de WE 80ns em nível lógico zero e CE e OE sem alteração, que o endereço e o dado permaneçam disponíveis por 90ns e que LB/UB permaneçam em zero por toda operação, garantimos que os requisitos de tempo sejam satisfeitos para a operação de escrita. Essa solução é apresentada na Figura 22.

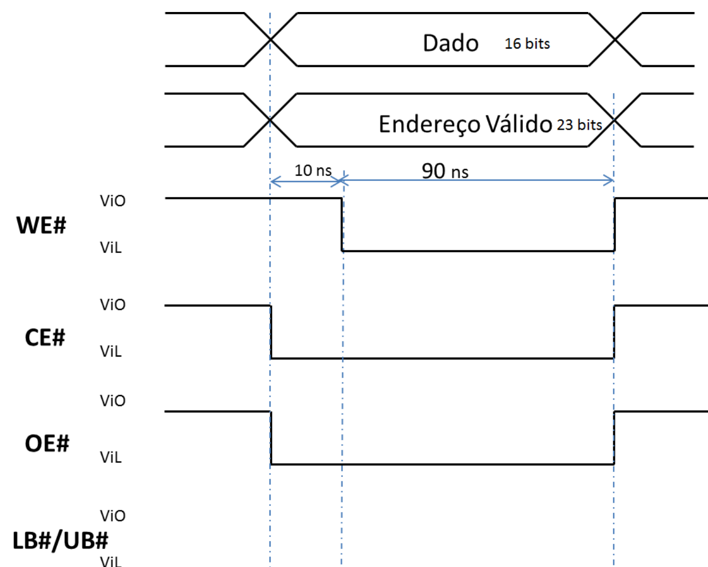


Figura 22 – Sinais de controle durante a operação de escrita.

A fim de implementar as observações citadas, foi projetada uma máquina de estados finitos (FSM) do tipo Moore para gerar os sinais de controle. A máquina de estados apresentada na Figura 23 possui duas ramificações. A ramificação superior se destina a geração dos sinais de controle da operação de escrita e a inferior ao sinais de leitura. Ao se iniciar o sistema (*reset*) a FSM espera o tempo de 15μs referente ao tempo de inicialização da memória definido no manual do fabricante, após essa etapa a interface permanece em estado ocioso até que uma operação de escrita ou leitura seja iniciada. A condição para que uma operação de leitura seja iniciada é que uma operação de escrita seja feita no endereço 0x09, isso é possível utilizando-se a instrução "OUTPUT"do microcontrolador no endereço 0x09, enquanto que para a operação de leitura utiliza-se o mesmo princípio aplicado ao endereço 0x08. Observa-se na figura que a contagem do contador *cont* está subtraído de um, tanto para escrita quanto para leitura. Isso ocorre devido ao fato da variável de saída da FSM ser responsável por iniciar a contagem, tendo efeito apenas no *clock* seguinte, portanto acrescentando um atraso de um período de *clock* (10 ns).

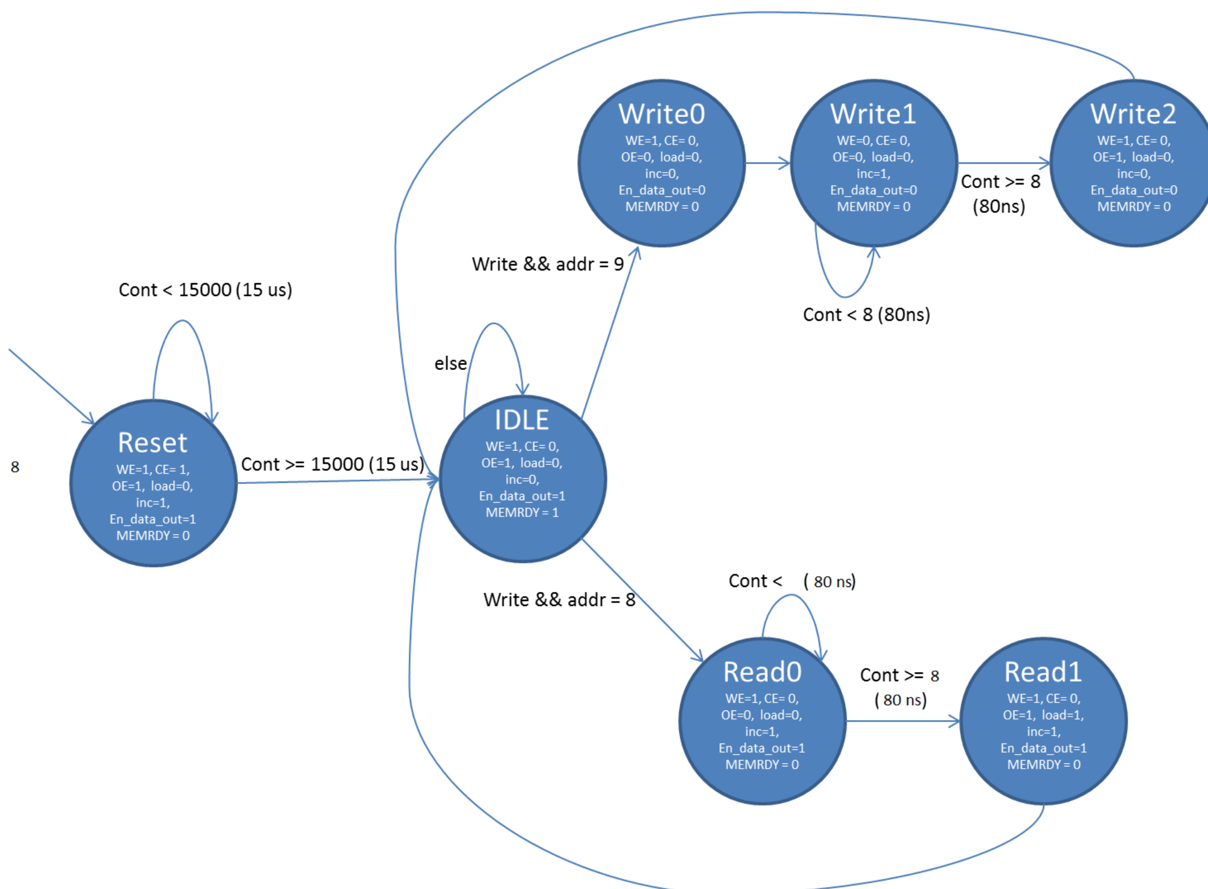
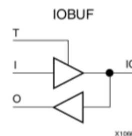


Figura 23 – Máquina de estados de Moore utilizada na geração dos sinais de controle da memória.

A integração entre a FPGA e o dispositivo de memória externa necessita que as entradas e saídas utilizem blocos TSI (*Technology Specific Instantiations*), que são

células disponibilizadas em bibliotecas do fabricante do *chip* da FPGA, que permitem que as portas de E/S respeitem requisitos elétricos e de tempo do dispositivo externo ao qual se pretende conectar. Nesse contexto, as linhas de controle e as de endereçamento são unidirecionais da FPGA para o dispositivo de memória na placa PCB, por isso sua integração é feita utilizando o módulo OBUF da biblioteca de primitivas da FPGA Xilinx Spartan 6E utilizada no projeto. As linhas de dados são bidirecionais, ou seja, algumas vezes elas são utilizadas para leitura e em outras para escrita. Devido a isso e ao fato de que dentro da FPGA os barramentos de dados são sempre unidirecionais, faz-se necessário a utilização de um *Buffer* bidirecional. Neste projeto utilizou-se o *Buffer* bidirecional IOBUF da biblioteca de primitivas da Xilinx, apresentado na Figura 24 juntamente com sua tabela verdade.



Inputs		Bidirectional	Outputs
T	**I**	**IO**	**O**
1	X	Z	IO
0	1	1	1
0	0	0	0

Figura 24 – *Buffer* bidirecional IOBUF e sua tabela verdade.

3.2.4 Módulo *BUS Control*

O módulo *BUS Control* tem uma funcionalidade de gerenciar qual dispositivo utilizará os recursos do barramento. Uma peculiaridade da arquitetura é o fato de possuir um único núcleo, portanto há apenas uma requisição por vez para uso do barramento, e possuir um único dispositivo mestre, não fazendo necessária a implementação de nenhum algoritmo de escalonamento. Apesar de existirem diversos padrões de barramento prontos, e ser possível implementar um desses padrões, o intuito da abordagem tomada, desenvolvendo uma barramento simples e funcional apenas para este projeto, teve como objetivo demostrar como se dá o mapeamento de dispositivo.

O módulo implementado recebe como entrada o endereço da porta com a qual o microcontrolador quer trocar dados e ele força os sinais de controle aos valores necessários para que a operação seja realizada com sucesso. O módulo identifica que existe uma operação em curso entre o processador e um dispositivo, quando o sinal *read_strobe* ou *write_strobe* possuem valores lógico um. Utilizando estes princípios, os dispositivos foram mapeados segundo os endereços apresentados na Tabela 3. Percebe-se que os endereços referenciam registradores pertencentes aos dispositivos periféricos da arquitetura, com exceção dos endereços 0x08 e 0x09 que são unicamente utilizados para iniciar as operações

de escrita e leitura em memória. Além disso, a Tabela 3 sinaliza a direção da operação que pode ser efetuada em cada endereço, no caso escrita apenas, leitura apenas ou ambas.

Tabela 3 – Mapeamento dos dispositivos de E/S utilizando a saída *port ID* do microcontrolador Picoblazer.

Mapeamento E/S		
Funcionalidade	**ID Porta**	**Direção da Operação (Picoblaze)**
Escreve Reg. Endereço Memória 0 (LSB)	0x01	Saída
Escreve Reg. Endereço Memória 1	0x02	Saída
Escreve Reg. Endereço Memória 2 (MSB)	0x03	Saída
Escreve Reg. Memória Dado 0 (LSB)	0x04	Saída
Escreve Reg. Memória Dado 1 (MSB)	0x05	Saída
Lê Reg. Memória Dado 0 (LSB)	0x06	Entrada
Lê Reg. Memória Dado 1 (MSB)	0x07	Entrada
Escreve Dado dos Registradores de Interface p/ Memória	0x08	Saída
Lê Dado da Memória para os Registradores de Interface	0x09	Saída
Dado UART	0x0A	Entrada/Saída
Lê Reg. Status UART	0x0B	Entrada
Lê Reg. Status da Interface de Memória	0x0C	Entrada

A implementação do módulo proposto utilizou os comando condicionais *case* e *if* da linguagem Verilog. Com os quais implementou-se um módulo com apenas lógica combinacional que permitisse que dependendo do valor do *Port ID* a lógica produzisse um certo conjunto de valores para os sinais de controle. Abaixo é apresentado um pseudo código que transmite a ideia utilizada para o desenvolvimento da lógica principal deste módulo. Ressalvas devem ser feitas, uma vez que HDLs não são linguagens estruturais e por tanto não seguem o mesmo fluxo de execução delas, por isso quando da análise de pseudo códigos deve-se compreender que se trata apenas de um esboço para o melhor entendimento do leitor e o mesmo não consegue representar os conceitos de orientação à eventos característicos das HDLS.

A idéia de um barramento compreende o compartilhamento de linhas de dados e endereços por diversos dispositivos. Isso implica que pode haver apenas uma operação por vês no barramento, para que não haja conflitos e perda de dados. Devido a isso, ao projetar o barramento tomou-se cuidado em observar que uma e apenas uma das condições lógicas da estrutura de controle fosse satisfeita por operação.

Algoritmo 1 Estrutura de controle de barramento.

Dados Entrada: Port ID, *write_strobe*, *read_strobe*
Resultado: Sinais controle dos diversos periféricos

```
if PortID = 0x01 then
  if read_strobe then
    sinaisControle ← controleReadEnd_0x01
  end if
  if write_strobe then
    sinaisControle ← controleWriteEnd_0x01
  end if
else if PortID = 0x02 then
  if read_strobe then
    sinaisControle ← controleWriteEnd_0x02
  end if
  if write_strobe then
    sinaisControle ← controleReadEnd_0x02
  end if
  .
  .
  .
end if
```

3.3 *Firmware*

O *firmware* desenvolvido consiste em um terminal *echo*, que recebe um conjunto de caracteres ASCII e o retorna através do mesmo canal UART. O *firmware* armazena em memória o histórico de caracteres recebidos, e os envia um a um via UART quando recebe um caractere '*'. O Terminal possui limite de 28 caracteres por linha, forçando quebra de linha quando o limite é atingido. O *firmware* desenvolvido pode ser encontrado nos apêndices.

O *firmware* é um código *assembly* escrito segundo a sintaxe do microcontrolador KCPSM6. O montador pode ser encontrado no site da Xilinx e é executado utilizando a interface gráfica fornecida pelo fabricante, como demonstrado na Figura 25. O montador gera um arquivo de mesmo nome do código fonte, porém com extensão ".v", o qual corresponde à memória de programa em Verilog, que deve ser instanciada no projeto e conectada ao microcontrolador KCPSM6 conforme exemplificado na Figura 26.

```
kcpsm6.exe
KCPSM6 Assembler v2.46
  Ken Chapman - Xilinx Ltd - 18th February 2013

 Enter name of PSM file: your_program.psm

 Reading top level PSM file...
   C:\Data\chapman\PicoBlaze_Designs\your_program.psm

 A total of 28 lines of PSM code have been read

 Checking line labels
 Checking CONSTANT directives
 Checking STRING directives
 Checking TABLE directives
 Checking instructions

 Writing formatted PSM file...
   C:\Data\chapman\PicoBlaze_Designs\your_program.fmt

 Expanding text strings
 Expanding tables
 Resolving addresses and Assembling Instructions
   Last occupied address: 004 hex
   Nominal program memory size: 1K (1024)    address(9:0)
   Occupied memory locations: 5
   Assembly completed successfully

 Writing LOG file...
   C:\Data\chapman\PicoBlaze_Designs\your_program.log
 Writing HEX file...
   C:\Data\chapman\PicoBlaze_Designs\your_program.hex
 Writing VHDL file...
   C:\Data\chapman\PicoBlaze_Designs\your_program.vhd

 KCPSM6 Options.....
         R - Repeat assembly with 'your_program.psm'
         N - Assemble new file.
         Q - Quit
```

Figura 25 – Exemplo de execução do montador utilizado no projeto.

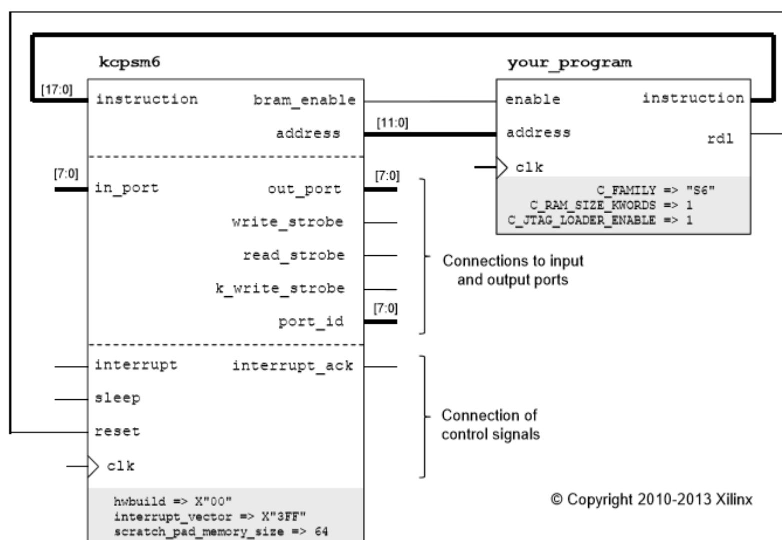


Figura 26 – Exemplo de como a memória de programa deve ser instanciada e integrada ao projeto (CHAPMAN, 2013).

3.4 Prototipação em FPGA

Esta seção apresenta os fundamento utilizados para a implementação da arquitetura no dispositivo FPGA.

3.4.1 Sincronização, circuito de *reset*

O problema de meta-estabilidade se configura por ser um estado em que um circuito eletrônico digital não consegue definir o seu estado lógico nem como '0', nem como '1', permanecendo desta forma em um estado instável de equilíbrio. Como resultado, o circuito digital pode agir de forma imprevisível, e de forma ao qual não foi projetado. Problemas de meta-estabilidade são problemas comumente tratados em domínios de circuitos assíncronos, ou quando dois circuitos de domínios de *clock* diferentes precisam se comunicar. Uma das soluções para eventuais problemas de meta-estabilidade, é utilizar mecanismos de projeto de circuitos digitais para certificar que o sinal meta estável só seja utilizado quando essa condição for resolvida, e não em condição de erro.

No caso deste projeto, decidimos utilizar um botão de pressão (*press-switch*) da placa de desenvolvimento com função de *reset* assíncrono. Pelo fato de estarmos utilizando uma sinal de entrada assíncrono para um sistema síncrono, problemas de meta-estabilidade podem ocorrer se o sinal de *reset* não respeitar os tempos de *setup* e de *hold* dos *flip-flops* a que ele está conectado. Isso se deve ao fato da transição do sinal de *reset* de um para zero ser imprevisível, ou seja, não temos como garantir que ao sair da condição de *reset*, momento em que o botão é liberado, o sinal terá respeitado os requisitos de tempo dos *flip-flops*. Desta forma, decidiu-se por implementar um módulo sincronizador para solucionar este problema.

A Figura 27 apresenta o sincronizador utilizado neste projeto. Trata-se de uma solução tradicional, frequentemente citada por diversos autores. Nessa solução, o sinal *rst_pin* é o sinal advindo do botão da placa de desenvolvimento (ativo em '1') e o *clk_a* é o mesmo *clock* de 100MHz do SoC. Uma análise rápida, nos faz concluir que o sinal intermediário entre os dois *flip-flops* pode apresentar meta-estabilidade devido as razões citadas anteriormente, porém esse sinal se resolve antes da próxima borda de subida do *clock* do próximo *flip-flop*, garantindo desta forma a integridade do sinal *rst_clk_a*, que é utilizado como *reset* do circuito. O *flip-flop* desenvolvido para este caso apresenta uma particularidade, por levar a saída do dispositivo para um quando o *reset* é acionado, uma vez que o microcontrolador KCPSM6 possui *reset* ativo em um.

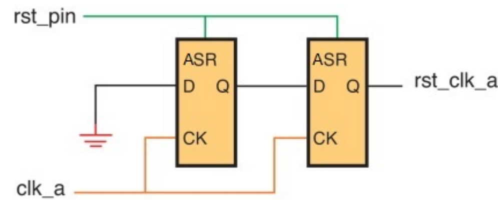


Figura 27 – Circuito sincronizador utilizado no projeto(ERUSALA, 2011).

3.4.2 Mapeamento dos pinos da FPGA nas portas do SoC.

O mapeamento dos pinos da FPGA nas respectivas portas do SoC é a última etapa de desenvolvimento antes da sintetização do HDL para a geração do *bitstream* a ser gravado da FPGA. Essa etapa consiste em fornecer um arquivo de extensão ".ucf"que especifica o mapeamento. O arquivo desenvolvido para este projeto é apresentado nos apêndices.

As portas do SoC que devem ser mapeadas são as portas de E/S do módulo mais acima na hierarquia, também chamado de *top-level*. A Figura 28 apresenta o módulo *top-level* do SoC. Nessa implementação, procurou-se mapear os *bits* de configuração da UART em chaves deslizantes (*slide switches*) da placa de desenvolvimento, para que se pudesse verificar diferentes configurações de maneira mais simples, sem dessa forma precisar modificar o *firmware*.

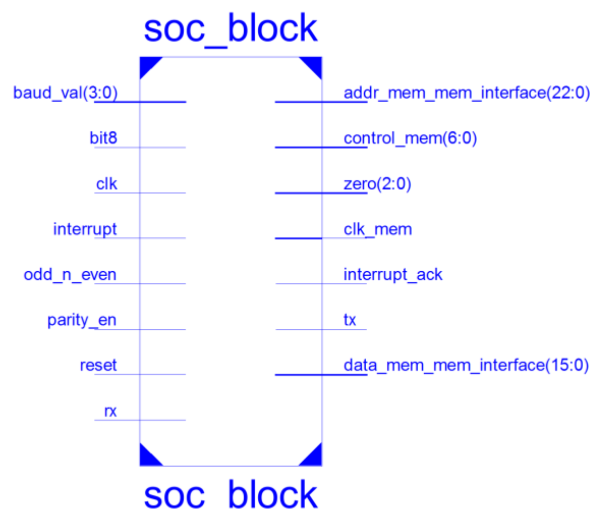


Figura 28 – Módulo top-level do SoC.

Por fim, sintetizamos o conjunto de arquivos HDL e o arquivo UCF, gerando o arquivo *bitstream*. Em seguida, utilizando o programa disponibilizado pela Digilent, realizamos a gravação do *bitstream* e pudemos verificar o correto funcionamento do projeto.

4 Resultados

A simulação apresentada na Figura 29 apresenta o funcionamento da UART e do microcontrolador KCPM6. O caso de teste desenvolvido para essa simulação pode ser encontrado no apêndice, e envolve a geração da seqüência "a,b,c,d", ou em ASCII 0x41, 0x42, 0x43 e 0x44 na Rx e observação da transmissão dos dados recebidos pelo microcontrolador e enviados pela Tx. Os bits foram gerados a uma taxa de 9600 bauds e sem paridade. Observando a Figura 29 pode-se notar o funcionamento do microcontrolador a partir das variações do endereço e instruções.

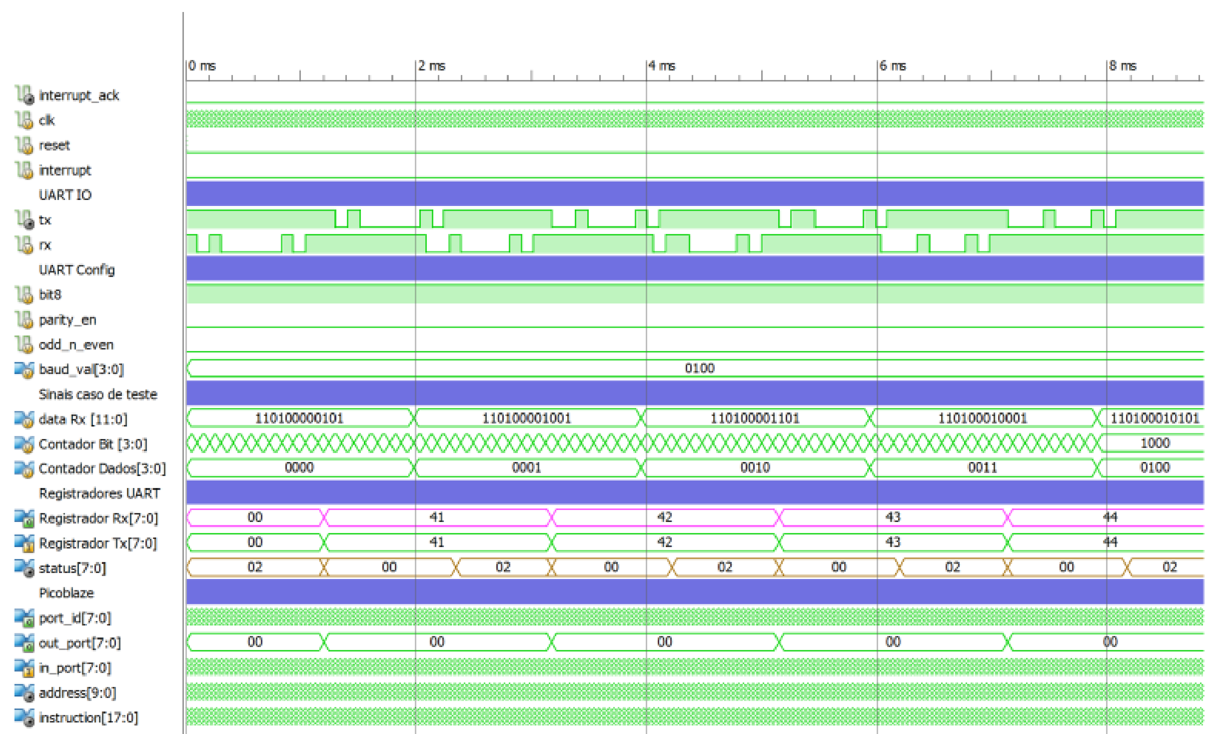


Figura 29 – Resultado da simulação da UART e microcontrolador durante a recepção seguida de transmissão de dados.

A Figura 30 apresenta a mesma simulação anterior, porém com ampliação no momento de transição entre o fim da recepção do dado no microcontrolador, momento mostrado pelo marcador azul, e o início do envio do dado, momento mostrado pelo marcador amarelo. Percebe-se o valor dos registrador de status: 0x02 ($rxrdy=0$, $txrdy=1$), 0x03 ($rxrdy=1$, $txrdy=1$), 0x02 ($rxrdy=0$, $txrdy=1$), 0x00 ($rxrdy=0$, $txrdy=0$).

A Figura 31 apresenta o tempo de bit medido durante a transmissão do dado. Verificou-se que o tempo de bit gerado pelo módulo desenvolvido foi de 104,20μs. O valor medido apresenta diferença inferior a 1% do valor teórico de 104,16μs.

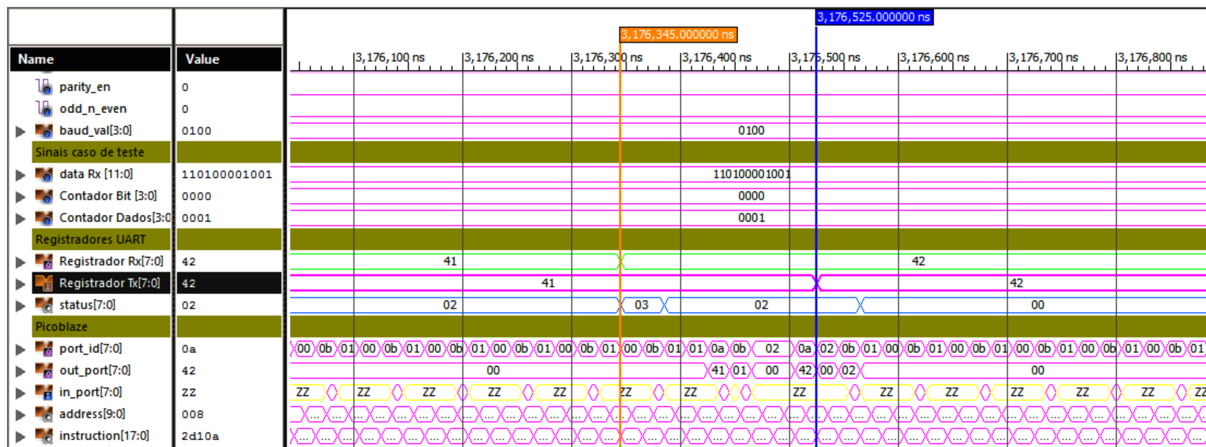


Figura 30 – Momento de transição entre a recepção de um dado e a transmissão de um dado pelo microcontrolador

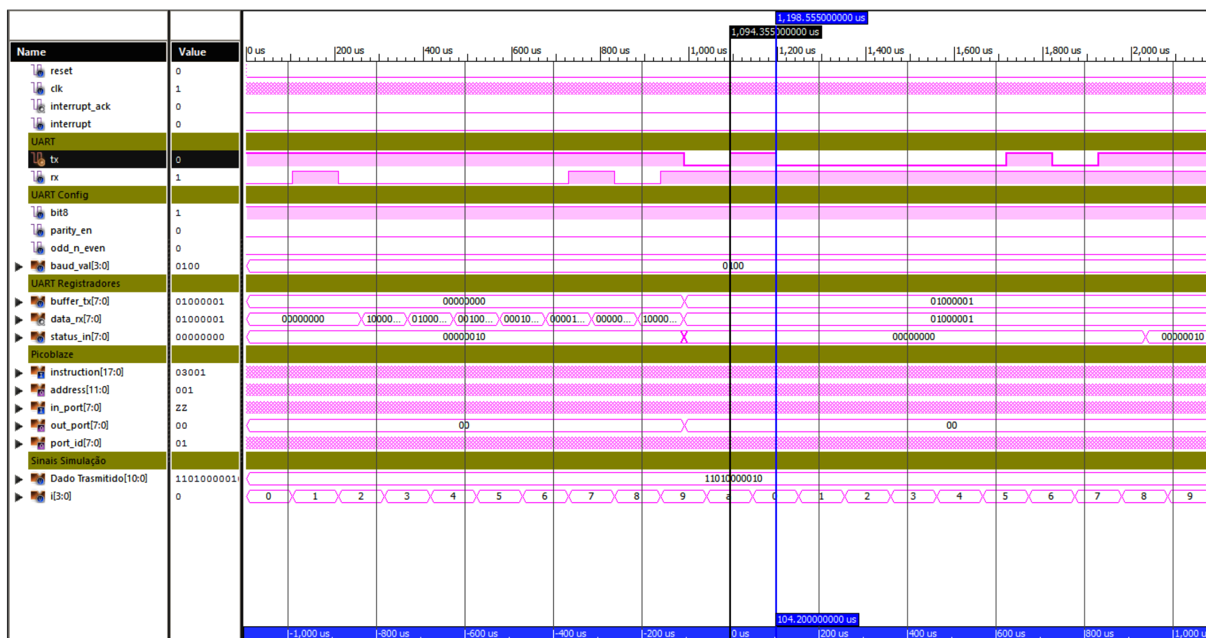


Figura 31 – Tempo de bit para taxa BAUD 9600 verificado durante simulação.

A simulação da controladora de memória teve como objetivo verificar se os sinais de controle estavam sendo gerados conforme projetado, porém não foi possível observar o fluxo de dados entre a memória e o microcontrolador, uma vez que não conseguimos um modelo para simulação da memória que funcionasse corretamente. A Figura 32 apresenta os sinais de controle de memória gerados na operação de escrita. Observa-se que todo o processo de escrita dura 105μs, sendo que o sinal WE permanece 95μs com valor zero, condições que são suficientes para satisfazer os requisitos de tempo descritos na seção 2.3.1.

A Figura 33 apresenta os sinais de controle da memória gerados durante a operação

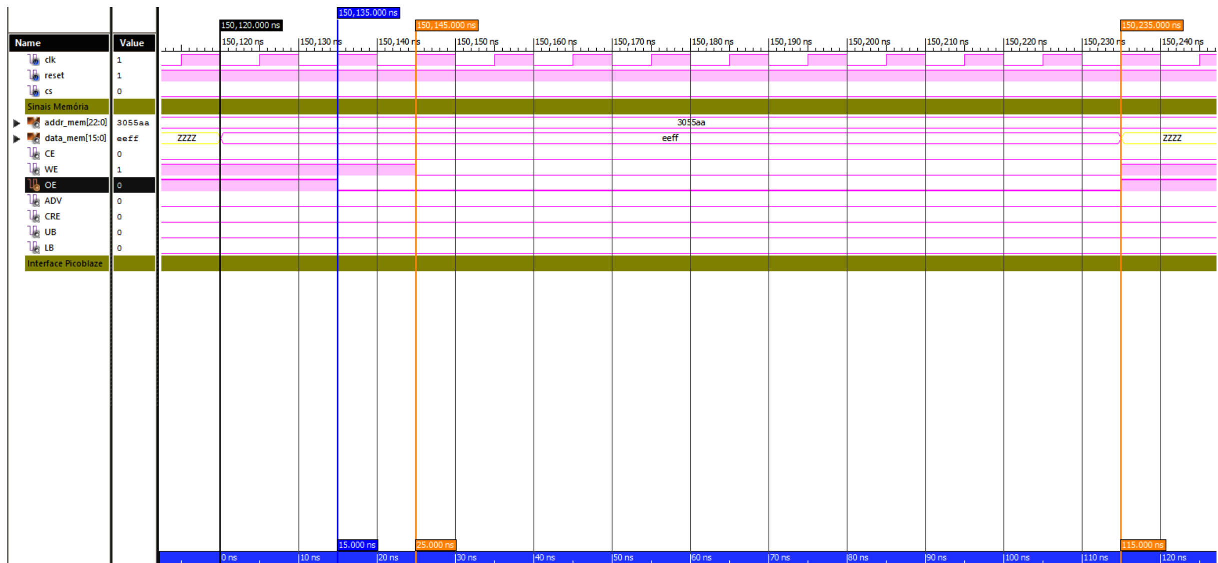


Figura 32 – Sinais de controle da memória gerados pela controladora projetada durante operação de escrita.

de leitura. A operação de leitura tem duração de 90μs como pode ser notado no gráfico.

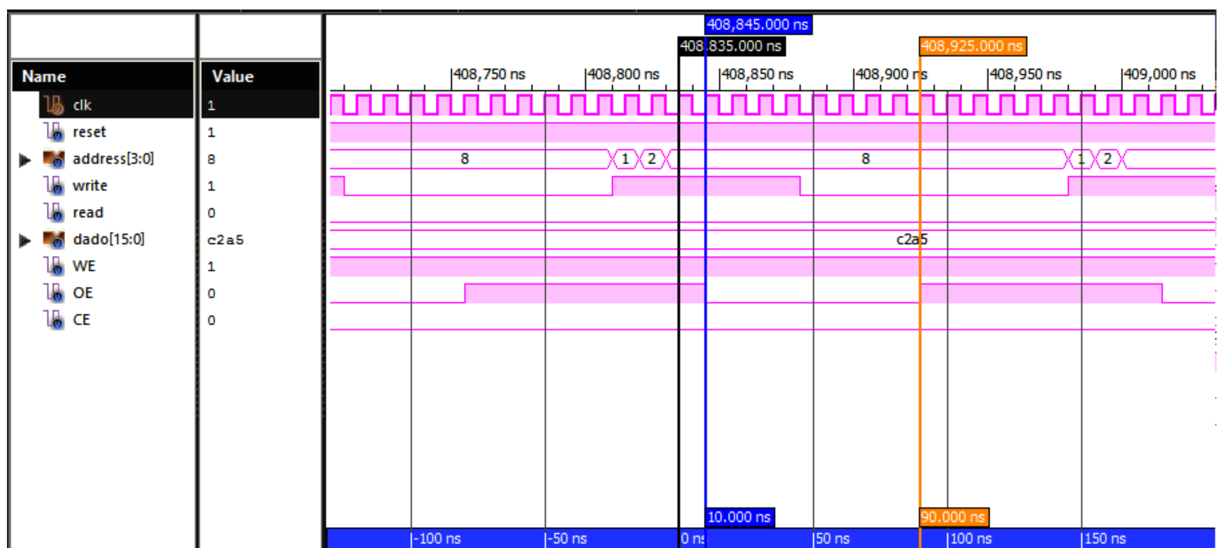


Figura 33 – Sinais de controle da memória gerados pela controladora projetada durante operação de leitura.

Ao fim das simulações, o sistema foi portado para a FPGA apresentada na seção 14, para verificação do seu funcionamento. O programa RealTerm que implementa um terminal capaz de comunicação em vários protocolos, incluindo UART, no computador foi utilizado para a comunicação com o SoC. Como os bits de configuração da UART foram direcionados às chaves deslizantes da placa, pudemos testar o sistema para diversas taxas de transmissão e paridade, podendo constatar o funcionamento correto do sistema

para taxas de transmissão de até 115.200 bits por segundo. Um exemplo de execução é apresentado na Figura 34.

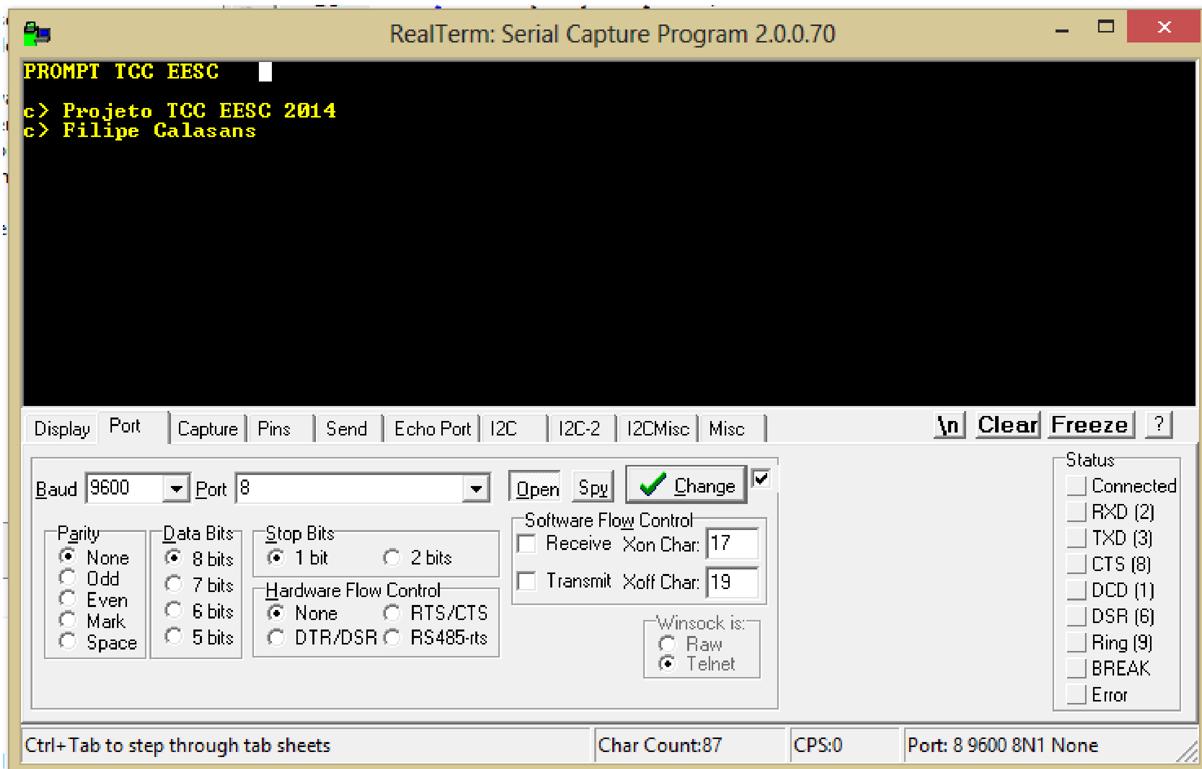


Figura 34 – Exemplo de execução à 9600 bauds do terminal implementado no SoC.

5 Proposta de Cronograma

Apresenta-se uma proposta de calendário para a implementação deste projeto em um semestre letivo, de forma que o estudante tenha a oportunidade de ter contato desde o desenvolvimento de uma IP customizada à sua integração numa arquitetura de um SoC, incluindo etapas de simulação, implementação de *firmware* e a prototipação em FPGA.

A seguir encontra-se o cronograma para desenvolvimento da arquitetura durante um semestre letivo:

- Semana 1 à 4: Desenvolvimento do mecanismo de transmissão da UART e do módulo sincronizador. Com o propósito de o estudante perceber o seu progresso, pede-se que ele simule e apresente uma versão executando em FPGA.
- Semana 5 à 8: Desenvolvimento do mecanismo de recepção da UART. Ao fim dessa etapa o estudante terá desenvolvido seu módulo customizado. Pede-se que os estudantes simulem e apresentem uma versão executando em FPGA.
- Semana 9 à 12: Integração do módulo UART com o Microcontrolador Picoblaze. Nessa etapa pode-se fornecer um *firmware* básico para teste, que receba e envie um caractere através da UART.
- Semana 13 à 16: Interfaceamento da memória com a arquitetura implementada, e desenvolvimento de um *firmware* que implemente o terminal *echo* apresentado nesse projeto. Pede-se que ao fim do projeto o estudante apresente o projeto implementado em FPGA.

6 Conclusão

Neste trabalho abordou-se o desenvolvimento de um SoC básico capaz de se comunicar com outros dispositivos através do protocolo UART e com capacidade de memória em *chip*. A implementação do SoC envolveu desafios no que tange à integração de IPs disponibilizadas por diferentes fabricantes, além de uma implementação de um módulo customizado de comunicação UART. O desenvolvimento deste projeto apresentou um caráter didático por consolidar um espectro grande de conhecimentos em sistemas digitais, dentre eles máquinas de estados, implementação de sistemas síncronos e arquiteturas de computadores.

O desenvolvimento deste projeto permitiu a avaliação da capacidade de integração do microcontrolador KCPSM6 em uma arquitetura de SoC de propósito educacional. Percebeu-se que o microcontrolador é bastante simples e apresenta o inconveniente de precisar ser programado em linguagem *assembly*, o que deixa o tempo de desenvolvimento e a manutenibilidade do *software* comprometida. Apesar disso, o microcontrolador é de fácil integração à um sistema mais complexo e apresenta bastante documentação disponível, o que o torna atrativo para fins de aprendizado e para aplicações simples.

A implementação do módulo customizado UART envolveu o desenvolvimento de duas máquinas de Moore. Porém, observa-se que outras abordagens podem ser exploradas no desenvolvimento deste módulo, incluindo utilização de máquina de Mealy ou até mesmo abdicando de uma solução que envolva um *template* de máquina de estados. Portanto, para fins educacionais em uma disciplinas, diversas abordagens podem ser utilizadas, e diferentes estudantes podem tomar diferentes rumos na elaboração da solução final.

O desenvolvimento do protótipo em FPGA foi uma etapa fundamental para a verificação do sistema final. Mostrou-se que a prototipação em FPGA é de fácil, rápida implementação e de baixo custo (120 dólares referente ao preço da placa de desenvolvimento) permitindo que diversas configurações pudessem ser experimentadas em um curto intervalo de tempo.

Ao fim, apresentou-se uma proposta de cronograma semestral para implementação do projeto em classe, o qual se baseou na implementação deste trabalho.

7 Trabalhos Futuros

Este projeto apresenta a implementação de uma plataforma didática que permite que o estudante tome diferentes abordagens na sua execução. Entre elas destacam-se diferentes abordagens de projeto da interface UART, entre elas máquinas de estados finitos do tipo Moore e Mealy, dentre outras possibilidades que não envolvam explicitamente máquinas de estados. Também é possível explorar durante a implementação do projeto a utilização de diferentes configurações de mapeamento dos dispositivos.

Uma das sugestões que pode ser apresentada aos estudantes durante a implementação deste projeto é a expansão da arquitetura. Para isso, pode-se implementar outros módulos que utilizem outros recursos da placa de desenvolvimento, tais como módulos de GPIO, módulos de varredura para *display* de 7 segmentos, SPI e controladora de vídeo VGA. Ao fim, esses módulos podem ser integrados à arquitetura desenvolvida, mapeando os seus registradores nos endereços disponíveis.

Além disso, é possível implementar uma variedade incontáveis de *firmwares* para o microcontrolador. Permitindo que o SoC desenvolvido possa ser incorporado em projetos maiores de sistemas embarcados.

Referências

ANGELINI, C. *Intel Core i7-3960X Review: Sandy Bridge-E And X79 Express*. 2011. Disponível em: <http://www.tomshardware.com/reviews/core-i7-3960x-x79-sandy-bridge-e,3071.html>.

Bashir M. Al-Hashimi. *System-On-Chip Next Generation Electronics Circuits, Devices and Systems*. [S.l.]: IET Circuits, Devices and Systems Series, 2006.

CHAPMAN, K. *PicoBlaze for Spartan-6, Virtex-6 and 7-Series (KCPSM6)*. [S.l.], 2013.

CHU, P. P. *FPGA Prototyping By Verilog Examples*. [S.l.]: A JOHN WILEY SONS, INC., PUBLICATION, 2008.

ERUSALA, S. *How do I reset my FPGA?* 2011. Disponível em: <http://www.eetimes.com/document.asp?doc_id=1278998>.

Grant Martin, H. L. M. *Surviving the SOC Revolution: A Guide to Platform-Based Design*. [S.l.]: Kluwer Academic Publisher, 1999.

MICRON. *Async/Page/Burst CelullarRAM 1.5 MT45W8MW16BGX*. [S.l.], 2004.

MIDORIKAWA, E. T. *Uma I,ntrodução às Linguagens de Descrição de Hardware*. [S.l.], 2007.

PALNITKAR, S. *Verilog HDL: A Guide to Digital Design and Synthesis*. [S.l.]: Prentice Hall PTR, 2003.

Pierre Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs*. [S.l.]: Springer, 2002.

Resve Saleh, S. System-on-chip: Reuse and integration. 2006.

Stephen Brown, J. Architecture of fpgas and cplds: A tutorial. 2000.

Sudeep Pasrich; Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect, System on Silicon*. [S.l.]: Morgan Kaufmann Publishers, 2008.

XILINX. *FPGA vs. ASIC*. 2014. Disponível em: <http://www.xilinx.com/fpga/asic.htm>.

Apêndices

APÊNDICE A – *Firmware* do terminal *echo*

Firmware desenvolvido para o microcontrolador Xilinx KCPSM6 que implementa um terminal com funcionalidade *echo* e opção de histórico de escrita.

```
;----------------------------------------------------------------
;- Port Constants
;----------------------------------------------------------------
CONSTANT nop, 00              ; port nop
CONSTANT war0, 01             ; port write memory address 0
CONSTANT war1, 02             ; port write memory address 1
CONSTANT war2, 03             ; port write memory address 2
CONSTANT wdr0, 04             ; port write memory data 0
CONSTANT wdr1, 05             ; port write memory data 1
CONSTANT rdir0, 06            ; port read memory data 0
CONSTANT rdir1, 07            ; port read memory data 1
CONSTANT do_mr, 08            ; port do memory read
CONSTANT do_mw, 09            ; port do memory write
CONSTANT uart_d, 0A           ; port uart data
CONSTANT uart_s, 0B           ; port uart status
CONSTANT mem_s, 0C            ; port read memory status
;----------------------------------------------------------------
CONSTANT TXRDY, 02            ;txrdy bit in status register
CONSTANT RXRDY, 01            ;rxrdy bit in status register
;----------------------------------------------------------------
CONSTANT DATA_UART, 00        ;Variable in the memroy
CONSTANT MEMRDY, 01           ;memory ready bit in status
    register
CONSTANT CR_ASC, 0D
CONSTANT LF_ASC, 0A
CONSTANT BS_ASC, 08
CONSTANT NULL_ASC, 00
;{sF,sE,sD} = Address counter for dump memory
;{sC,sB,sA} = number of elements
;s6 = number of elements on the current line, it must be less
    than 40
;----------------------------------------------------------------
ADDRESS 000
CALL init_message
CALL send_new_line
;Initializes mem[0000] = 0
LOAD s0, 00
LOAD s1, 00
LOAD s2, 00
CALL write_address
```

```
           LOAD s0 , 00
           LOAD s1 , 00
           CALL write_memory
initilize_pointer:
           LOAD sA , 00
           LOAD sB , 00
           LOAD sC , 00
           ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
           LOAD s5 , 00
           LOAD s6 , 00
main_loop: INPUT s0 , uart_s          ;ler uart status
           AND s0 , RXRDY          ;temos um byte?
           JUMP Z, main_loop
           INPUT s1 , uart_d          ;ler uart byte
           LOAD s5 , s1                ;move uart data para s5
wait: INPUT s0 , uart_s          ;ler uart status
           AND s0 , TXRDY             ;espera por txrdy
           JUMP Z, wait
           ;add s5 , 01
           COMPARE s5 , "*"           ;verifica dump character
           JUMP NZ, dont_dump
           CALL dump_memory
           JUMP reset_dump
dont_dump:
           COMPARE s5 , CR_ASC
           JUMP NZ, test_LF
           CALL send_new_line
           LOAD s6 , 00
           JUMP main_loop
test_LF: COMPARE s5 , LF_ASC
           JUMP NZ, test_BS
           CALL send_new_line
           LOAD s6 , 00
           JUMP main_loop
test_BS: COMPARE s5 , BS_ASC
           JUMP NZ, visible_character
           CALL handle_backspace
           COMPARE s6 , 00
           JUMP Z, main_loop
           SUB s6 , 01
           JUMP main_loop
visible_character:
           ADD s6 , 01
           COMPARE s6 , 28
           JUMP C, line_not_complete
           CALL send_new_line
           LOAD s6 , 01
```

```
line_not_complete:
            LOAD s0, sA
            LOAD s1, sB
            LOAD s2, sC
            CALL write_address
            LOAD s0, s5
            LOAD s1, 00
            CALL write_memory
            LOAD s0, 01
            ADD sA, s0
            ADDCY sB, 00
            ADDCY sC, 00
            LOAD s0, s5
            CALL write_uart
            JUMP main_loop
reset_dump:
            LOAD sA, 00
            LOAD sB, 00
            LOAD sC, 00
            LOAD s0, sA
            LOAD s1, sB
            LOAD s2, sC
            CALL write_address
            LOAD s0, 00
            LOAD s1, 00
            CALL write_memory
            JUMP main_loop
             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
dump_memory:
            CALL send_new_line
            LOAD sD, 00
            LOAD sE, 00
            LOAD sF, 00
            COMPARE s6, 00
            JUMP Z, dump_finished
loop_dump:
            LOAD s0, sD
            LOAD s1, sE
            LOAD s2, sF
            CALL write_address
            CALL read_memory
            CALL write_uart
            LOAD s0, 01
            ADD sD, s0
            ADDCY sE, 00
            ADDCY sF, 00
            COMPARE sA, sD
```

```
          JUMP NZ, loop_dump
          COMPARE sB, sE
          JUMP NZ, loop_dump
          COMPARE sC, sF
          JUMP NZ, loop_dump
          JUMP dump_finished       ; Fim do vetor
          JUMP loop_dump
dump_finished:
          RETURN
          ;;;;;;;;;;;;;;;;;;;;;;;;;;
write_uart: ;argument in s0
wait_uart: INPUT s1, uart_s          ;ler uart status
          AND s1, TXRDY              ;temos um byte?
          JUMP Z, wait_uart
          OUTPUT s0, uart_d
wait_uart2: INPUT s1, uart_s         ;ler uart status
          AND s1, TXRDY              ;temos um byte?
          JUMP Z, wait_uart2
          RETURN
          ;;;;;;;;;;;;;;;;;;;;;;;;;
init_message:
          LOAD s0, "P"
          CALL write_uart
          LOAD s0, "R"
          CALL write_uart
          LOAD s0, "O"
          CALL write_uart
          LOAD s0, "M"
          CALL write_uart
          LOAD s0, "P"
          CALL write_uart
          LOAD s0, "T"
          CALL write_uart
          LOAD s0, " "
          CALL write_uart
          LOAD s0, "T"
          CALL write_uart
          LOAD s0,"C"
          CALL write_uart
          LOAD s0, "C"
          CALL write_uart
          LOAD s0, ""
          CALL write_uart
          LOAD s0, "E"
          CALL write_uart
          LOAD s0, "E"
          CALL write_uart
```

```
            LOAD s0, "S"
            CALL write_uart
            LOAD s0, "C"
            CALL write_uart
            LOAD s0, CR_ASC
            CALL write_uart
            LOAD s0, LF_ASC
            CALL write_uart
            RETURN
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
write_address: ;{ad2,ad1,ad0} = {s2,s1,s3} registradores nao modificados
            OUTPUT s0, war0
            OUTPUT s1, war1
            OUTPUT s2, war2
            RETURN
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
write_memory: ;{d1,d0} = {s1,s0} s0 sera modificado apos essa funcao
wait_write_mem2:
            INPUT s2, mem_s
            AND s2, MEMRDY
            JUMP Z, wait_write_mem2
            OUTPUT s0, wdr0
            OUTPUT s1, wdr1
            OUTPUT s0, do_mw
wait_write_mem:
            INPUT s0, mem_s
            AND s0, MEMRDY
            JUMP Z, wait_write_mem
            RETURN
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
read_memory: ;{s1,s0} = {d1,d0}
wait_read_mem1:
            INPUT s0, mem_s
            AND s0, MEMRDY
            JUMP Z, wait_read_mem1
            OUTPUT s0, do_mr
wait_read_mem:
            INPUT s0, mem_s
            AND s0, MEMRDY
            JUMP Z, wait_read_mem
            INPUT s0, rdir0
            INPUT s1, rdir1
            RETURN
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
send_new_line:
            LOAD s0, CR_ASC
            CALL write_uart
```

```
            LOAD s0, LF_ASC
            CALL write_uart
            LOAD s0, "c"
            CALL write_uart
            LOAD s0, ">"
            CALL write_uart
            LOAD s0, " "
            CALL write_uart
            RETURN
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
handle_backspace:
            COMPARE s6, 00
            JUMP Z, end_bs
            LOAD s0, BS_ASC
            CALL write_uart
            LOAD s0, " "
            CALL write_uart
            LOAD s0, BS_ASC
           CALL write_uart
end_bs:
           RETURN
```

APÊNDICE B – Arquivo UCF utilizado no mapeamento das portas do SoC

Os fabricantes das placas fornecem geralmente um arquivo padrão com as portas dos dispositivos da placa devidamente identificadas, sendo necessário ao usuário apenas mapear essas portas às portas do módulo *top-level*. Abaixo segue o arquivo UCF utilizado neste projeto.

```
NET "clk"                 LOC = "v10"  ;

NET "reset"               LOC = "c4"   ;

NET "baud_val<0>"         LOC = "t10"  ;
NET "baud_val<1>"         LOC = "t9"   ;
NET "baud_val<2>"         LOC = "v9"   ;
NET "baud_val<3>"         LOC = "m8"   ;

NET "bit8"                LOC = "t5"   ;
NET "odd_n_even"          LOC = "u8"   ;

NET "parity_en"           LOC = "v8"   ;
NET "rx"                  LOC = "N17";
NET "tx"                  LOC = "N18";

NET "interrupt"                   LOC = "c9";
NET "interrupt_ack"               LOC = "T11";

## onBoard Cellular RAM
NET "control_mem<4>"          LOC = "L18";
NET "control_mem<5>"          LOC = "M16";

NET "control_mem<3>"          LOC = "H18";
NET "control_mem<6>"          LOC = "L15";
NET "clk_mem"                 LOC = "R10";

NET "control_mem<2>"          LOC = "M18";
NET "control_mem<0>"          LOC = "K16";
NET "control_mem<1>"          LOC = "K15";

NET "addr_mem_mem_interface<0>"       LOC = "K18";
NET "addr_mem_mem_interface<1>"       LOC = "K17";
NET "addr_mem_mem_interface<2>"       LOC = "J18";
NET "addr_mem_mem_interface<3>"       LOC = "J16";
```

```
NET "addr_mem_mem_interface<4>"       LOC = "G18";
NET "addr_mem_mem_interface<5>"       LOC = "G16";
NET "addr_mem_mem_interface<6>"       LOC = "H16";
NET "addr_mem_mem_interface<7>"       LOC = "H15";
NET "addr_mem_mem_interface<8>"       LOC = "H14";
NET "addr_mem_mem_interface<9>"       LOC = "H13";
NET "addr_mem_mem_interface<10>"      LOC = "F18";
NET "addr_mem_mem_interface<11>"      LOC = "F17";
NET "addr_mem_mem_interface<12>"      LOC = "K13";
NET "addr_mem_mem_interface<13>"      LOC = "K12";
NET "addr_mem_mem_interface<14>"      LOC = "E18";
NET "addr_mem_mem_interface<15>"      LOC = "E16";
NET "addr_mem_mem_interface<16>"      LOC = "G13";
NET "addr_mem_mem_interface<17>"      LOC = "H12";
NET "addr_mem_mem_interface<18>"      LOC = "D18";
NET "addr_mem_mem_interface<19>"      LOC = "D17";
NET "addr_mem_mem_interface<20>"      LOC = "G14";
NET "addr_mem_mem_interface<21>"      LOC = "F14";
NET "addr_mem_mem_interface<22>"      LOC = "C18";
NET "zero<0>"                         LOC = "C17";
NET "zero<1>"                         LOC = "F16";
NET "zero<2>"                         LOC = "F15";

NET "data_mem_mem_interface<0>"        LOC = "R13";
NET "data_mem_mem_interface<1>"        LOC = "T14";
NET "data_mem_mem_interface<2>"        LOC = "V14";
NET "data_mem_mem_interface<3>"        LOC = "U5"  ;
NET "data_mem_mem_interface<4>"        LOC = "V5"  ;
NET "data_mem_mem_interface<5>"        LOC = "R3"  ;
NET "data_mem_mem_interface<6>"        LOC = "T3";
NET "data_mem_mem_interface<7>"        LOC = "R5";
NET "data_mem_mem_interface<8>"        LOC = "N5";
NET "data_mem_mem_interface<9>"        LOC = "P6";
NET "data_mem_mem_interface<10>"       LOC = "P12";
NET "data_mem_mem_interface<11>"       LOC = "U13";
NET "data_mem_mem_interface<12>"       LOC = "V13";
NET "data_mem_mem_interface<13>"       LOC = "U10";
NET "data_mem_mem_interface<14>"       LOC = "R8";
NET "data_mem_mem_interface<15>"       LOC = "T8";
```

APÊNDICE C – Caso de Teste Utilizado na Simulação do módulo UART

Abaixo encontra-se o caso de teste utilizado na simulação do módulo UART.

```
`timescale 1ns / 1ps

module tb_lb_soc_block;

	// Inputs
	reg clk;
	reg reset;
	reg rx;
	reg interrupt;
	reg bit8;
	reg parity_en;
	reg odd_n_even;
	reg [3:0] baud_val;

	// Outputs
	wire tx;
	wire interrupt_ack;

	//internal
	reg [11:0] char;
	reg [3:0] i;
	reg [3:0] j;

	// Instantiate the Unit Under Test (UUT)
	soc_block uut (
		.clk(clk),
		.reset(reset),
		.tx(tx),
		.rx(rx),
		.interrupt(interrupt),
		.interrupt_ack(interrupt_ack),
		.bit8(bit8),
		.parity_en(parity_en),
		.odd_n_even(odd_n_even),
		.baud_val(baud_val)
	);

	always
		#5 clk = ~clk;
```

```
        initial begin
                // Initialize Inputs
                clk = 0;
                reset = 1;
                rx = 1;
                interrupt = 0;
                bit8 = 1;
                parity_en = 0;
                odd_n_even = 0;
                baud_val = 4'b0100;
                char = 12'b1_1_01000001_01; //ASC A
                #10;
                reset =0;

        for(j=0; j<4; j=j+1) begin

                        for(i=0; i<11; i=i+1) begin
                                rx=char[i];
                                #104110;
                        end


                        for(i=0; i<8; i=i+1) begin
                                #104110;
                        end

                        char = char + 11'b00_00000001_00;

                        end


                #100;

        end

endmodule
```

APÊNDICE D – Códigos Desenvolvidos em Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////
// Module Name:     soc_block
//////////////////////////////////////////
module soc_block(clk, reset, tx, rx, interrupt, interrupt_ack, bit8,
    parity_en,
        odd_n_even, baud_val, addr_mem_mem_interface,
            data_mem_mem_interface, control_mem, clk_mem, zero
    );

        input clk, interrupt;
        input reset;
    input bit8;
    input parity_en;
    input odd_n_even;
    input [3:0] baud_val;
        input rx;
    output tx, interrupt_ack;
        output [22:0] addr_mem_mem_interface;
        inout [15:0] data_mem_mem_interface;
        output [6:0] control_mem;
        output clk_mem;
        output [2:0] zero;

        reg we, oe, adrs;
        wire [7:0] out_port, in_port;
        wire write_strobe, read_strobe;

        wire [7:0] port_id;

        wire clk_mem;
    reg write_mem_interface, read_mem_interface; //ok
    wire [7:0] data_in_mem_interface; //ok
    reg [3:0] address_mem_interface; //ok
    wire  [15:0] data_mem_mem_interface; //external output
        wire [7:0]data_out_mem_interface; // ok
    wire [22:0] addr_mem_mem_interface; //ok
    wire CE_out_mem_interface, WE_out_mem_interface, OE_out_mem_interface;
        wire ADV_mem_interface, CRE_mem_interface, UB_mem_interface,
            LB_mem_interface;
```

```
        wire [6:0] control_mem;
        wire [2:0] zero;

        assign zero = 0;
        assign clk_mem = 0;
        assign control_mem = {CE_out_mem_interface, WE_out_mem_interface,
            OE_out_mem_interface,
                                                                ADV_mem_interface,
                                                                    CRE_mem_interface
                                                                    ,
                                                                    UB_mem_interface
                                                                    ,
                                                                    LB_mem_interface
                                                                    };

        lb_reset uart_reset (
    .clk(clk),
    .resetb(reset),
    .cs(1'b0),
    .system_reset(u_reset)
    );

        // Instantiate the module
lb_reset_processor proc_reset (
    .clk(clk),
    .resetb(reset),
    .cs(1'b0),
    .system_reset(processor_reset)
    );


processor_top_level processor (
    .clk(clk),
    .port_id(port_id),
    .write_strobe(write_strobe),
    .out_port(out_port),
    .read_strobe(read_strobe),
    .interrupt(interrupt),
    .interrupt_ack(interrupt_ack),
    .reset(reset),
    .in_port(in_port),
    .k_write_strobe(k_write_strobe)
    );


lb_UART_toplevel uart (
    .clk(clk),
    .reset(u_reset),
```

```
    .cs(1'b0),
    .we(we),
    .oe(oe),
    .adrs(adrs),
    .data(out_port),
    .bit8(bit8),
    .parity_en(parity_en),
    .odd_n_even(odd_n_even),
    .baud_val(baud_val),
    .rx(rx),
    .tx(tx),
    .data_out(in_port)
    );

assign in_port = data_out_mem_interface;
assign data_in_mem_interface = out_port;

memory_interface mem_interface (
    .clk(clk),
    .reset(u_reset),
    .write(write_mem_interface),
    .read(read_mem_interface),
    .cs(1'b0),
    .data_in(data_in_mem_interface),
    .address(address_mem_interface),
    .data_mem(data_mem_mem_interface),
    .data_out(data_out_mem_interface),
    .addr_mem(addr_mem_mem_interface),
    .CE_out(CE_out_mem_interface),
    .WE_out(WE_out_mem_interface),
    .OE_out(OE_out_mem_interface),
    .ADV(ADV_mem_interface),
    .CRE(CRE_mem_interface),
    .UB(UB_mem_interface),
    .LB(LB_mem_interface)
    );

//Memory Interface Block
//CONSTANT nop,     00   ; nop
//CONSTANT war0,    01   ; port write memory address 0
//CONSTANT war1,    02   ; port write memory address 1
//CONSTANT war2,    03   ; port write memory address 2
//CONSTANT wdr0,    04   ; port write memory data 0
//CONSTANT wdr1,    05   ; port write memory data 1
//CONSTANT rdir0,   06   ; port read memory data 0
//CONSTANT rdir1,   07   ; port read memory data 1
//CONSTANT do_mr,   08   ; port do memory write
```

```
//CONSTANT do_mw,   09    ; port do memory read
//CONSTANT uart_d , 0A    ; port uart data
//CONSTANT uart_s , 0B    ; port uart status
//CONSTATN status   0C    ; port read status Memory



//BUS Control
always @ (*)
//Interface Block
        if(port_id==8'h0b || port_id == 8'h0a) begin
                write_mem_interface = 0;
                read_mem_interface = 0;

                if(read_strobe) begin
                        oe = 1; end
                else begin
                        oe = 0;
                        end
                if(port_id == 8'h0a) begin
                        adrs = 0;
                        if(write_strobe) begin
                                we = 1; end
                        else begin
                                we = 0; end
                        end
                else begin
                        we = 0;
                        adrs = 1;
                        end
                end
                else begin
                        oe=0;
                        we=0;
                        case(port_id)
                        1: begin
                                if (write_strobe) begin
                                        address_mem_interface = 4'h1;
                                        write_mem_interface = 1;
                                        read_mem_interface = 0;
                                end
                                else begin
                                        write_mem_interface = 0;
                                        read_mem_interface = 0;
                                end
                        end
                        2:begin
                                if (write_strobe) begin
```

```
                    address_mem_interface = 4'h2;
                    write_mem_interface = 1;
                    read_mem_interface = 0;
            end
            else begin
                    write_mem_interface = 0;
                    read_mem_interface = 0;
            end
    end
    3:begin
            if (write_strobe) begin
                    address_mem_interface = 4'h3;
                    write_mem_interface = 1;
                    read_mem_interface = 0;
            end
            else begin
                    write_mem_interface = 0;
                    read_mem_interface = 0;
            end
    end
    4:begin
            if (write_strobe) begin
                    address_mem_interface = 4'h4;
                    write_mem_interface = 1;
                    read_mem_interface = 0;
            end
            else begin
                    write_mem_interface = 0;
                    read_mem_interface = 0;
            end
    end
    5:begin
            if (write_strobe) begin
                    address_mem_interface = 4'h5;
                    write_mem_interface = 1;
                    read_mem_interface = 0;
            end
            else begin
                    write_mem_interface = 0;
                    read_mem_interface = 0;
            end
    end
    6:begin
            if (read_strobe) begin
                    address_mem_interface = 4'h6;
                    write_mem_interface = 0;
                    read_mem_interface = 1;
```

```
                end
                else begin
                        write_mem_interface = 0;
                        read_mem_interface = 0;
                end
        end
        7:begin
                if (read_strobe) begin
                        address_mem_interface = 4'h7;
                        write_mem_interface = 0;
                        read_mem_interface = 1;
                end
                else begin
                        write_mem_interface = 0;
                        read_mem_interface = 0;
                end
        end
        8:begin
                if (write_strobe) begin
                        address_mem_interface = 4'h8;
                        write_mem_interface = 1;
                        read_mem_interface = 0;
                end
                else begin
                        write_mem_interface = 0;
                        read_mem_interface = 0;
                end
        end
        9:begin
                if (write_strobe) begin
                        address_mem_interface = 4'h9;
                        write_mem_interface = 1;
                        read_mem_interface = 0;
                end
                else begin
                        write_mem_interface = 0;
                        read_mem_interface = 0;
                end
        end
        4'hC:begin
                if (read_strobe) begin
                        address_mem_interface = 4'hA;
                        write_mem_interface = 0;
                        read_mem_interface = 1;
                end
                else begin
                        write_mem_interface = 0;
```

```
                                        read_mem_interface = 0;
                                end
                        end
                        default: begin
                                write_mem_interface = 0;
                                read_mem_interface = 0;
                                end
                        endcase
                end

//assign adrs = (port_id == 8'h0a) ? 0 : ((port_id==8'h0b) ? 1 : 0);
//assign oe = ((port_id == 8'h0a) | (port_id == 8'h0b) ) ? ((read_strobe) ?
    1 : 0) : 0;
//assign we = (port_id == 8'h0a) ? ((write_strobe) ? 1 : 0) : 0;
endmodule
```

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////
// Module Name:    top_level
//////////////////////////////////////////////////////
module processor_top_level(clk, port_id, write_strobe, out_port,
   read_strobe,
        interrupt, interrupt_ack, reset, in_port, k_write_strobe
    );

        output  [7:0]   port_id ;
        output          write_strobe, read_strobe, interrupt_ack,
            k_write_strobe;
        output  [7:0]   out_port ;
        input   [7:0]   in_port ;
        input           interrupt, reset, clk ;


        wire    [11:0]  address ;
        wire    [17:0]  instruction ;
        wire    bram_enable;


kcpsm6 processor (
    .address(address),
    .instruction(instruction),
    .bram_enable(bram_enable),
    .in_port(in_port),
    .out_port(out_port),
    .port_id(port_id),
    .write_strobe(write_strobe),
    .k_write_strobe(k_write_strobe),
```

```
    .read_strobe(read_strobe),
    .interrupt(interrupt),
    .interrupt_ack(interrupt_ack),
    .sleep(1'b0),
    .reset(reset|rdl),
    .clk(clk)
    );

uart_test_mem_pre_presentation program (
    .address(address),
    .instruction(instruction),
    .enable(bram_enable),
    .rdl(rdl),
    .clk(clk)
    );

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////
// Create Date:    05:35:32 02/27/2013
// Design Name:
// Module Name:    lb_UART_Tx_Module
/////////////////////////////////////////
module lb_UART_Tx_Core(
    input clk,
    input reset,
    input [7:0] data,
    input start,
    input [19:0] baud_value,
    input bit8,
    input parity_en,
    input odd_n_even,
        input cs,
    output tx,
    output tx_done
    );

        reg [2:0] missingBits;
        wire shift, load;

        lb_UART_Tx_ControlUnit tx_cu (
                .clk(clk),
                .reset(reset),
                .start(start),
                .bit8(bit8),
                .parity_en(parity_en),
```

```
                .baudPrescale(baud_value),
                .cs(cs),
                .done(tx_done),
                .load(load),
                .shift(shift)
                );

        lb_11bitsShiftRegister tx_shift_reg (
                .clk(clk),
                .reset(reset),
                .load(load),
                .shift(shift),
                .data_in({missingBits,data[6:0],1'b0,1'b1}),
                .cs(cs),
                .data_out(tx)
                );

        //Missing Bits Combo Logic !
        always @ (*)
                case ({bit8, parity_en,odd_n_even})
                        3'b000: missingBits = {1'b1, 1'b1, 1'b1};
                        3'b001: missingBits = {1'b1, 1'b1, 1'b1};
                        3'b010: missingBits = {1'b1, 1'b1, (^data[6:0])};
                        3'b011: missingBits = {1'b1, 1'b1, ~(^(data[6:0]))
                            };
                        3'b100: missingBits = {1'b1, 1'b1, data[7]};
                        3'b101: missingBits = {1'b1, 1'b1, data[7]};
                        3'b110: missingBits = {1'b1, ^(data[7:0]), data
                            [7]};
                        3'b111: missingBits = {1'b1, ~(^(data[7:0])), data
                            [7]};
                endcase

endmodule
```

```
`timescale 1ps / 1fs
////////////////////////////////////////////
// Create Date:    01:37:04 02/27/2013
// Design Name:
// Module Name:    lb_UART_Tx_FSM
////////////////////////////////////////////
module lb_UART_Tx_FSM(
    input clk,
    input reset,
    input start,
    input baudTickCounterDone,
    input bitCounterDone,
```

```
        input cs,
    output reg done,
    output reg shift ,
    output reg load,
    output reg incNumBits,
    output reg resetBaudTickCounter ,
        output reg resetNumBitsCounter
    );

        localparam IDLE                   = 2'b00,
                   LOAD_SHIFT_REG = 2'b01,
                   SHIFT_REG          = 2'b10,
                   WAIT_BIT_TIME   = 2'b11;

        reg nxt_done, nxt_shift , nxt_load, nxt_incNumBits,
           nxt_resetBaudTickCounter;
        reg nxt_resetNumBitsCounter;

        reg [1:0] nq;
        reg [1:0] q;

/**************************************************
**  Present State assignment
**
**************************************************/

        always @ (posedge clk, negedge reset)
                //if(!cs) begin
                        if(!reset)
                                        q <= IDLE;
                        else
                                        q <= nq;
                //end

/**************************************************
**  Present output assignment
**
**************************************************/

        always @ (posedge clk, negedge reset)
                //if(!cs) begin
                        if(!reset)
                                {done, shift , load, incNumBits,
                                            resetBaudTickCounter ,
                                                resetNumBitsCounter} <=
                                                6'b1_0_0_0_1_1;
                        else begin
```

```
                        done <= nxt_done;
                        shift <= nxt_shift;
                        load <= nxt_load;
                        incNumBits <= nxt_incNumBits;
                        resetBaudTickCounter <=
                            nxt_resetBaudTickCounter;
                        resetNumBitsCounter <=
                            nxt_resetNumBitsCounter;
                    end
            //end

/**********************************************
** State Transition Logic and
**
**********************************************/

        always @ (*)
            case (q)
                IDLE: begin
                        {nxt_done, nxt_shift, nxt_load,
                            nxt_incNumBits, nxt_resetBaudTickCounter
                            , nxt_resetNumBitsCounter} = 6'
                            b1_0_0_0_1_0;
                        if(start)
                                nq = LOAD_SHIFT_REG;
                        else
                                nq = IDLE;
                end
                LOAD_SHIFT_REG: begin
                        {nxt_done, nxt_shift, nxt_load,
                            nxt_incNumBits, nxt_resetBaudTickCounter
                            , nxt_resetNumBitsCounter} = 6'
                            b0_0_1_0_1_1;
                        nq = SHIFT_REG;
                end
                SHIFT_REG: begin
                        {nxt_done, nxt_shift, nxt_load,
                            nxt_incNumBits, nxt_resetBaudTickCounter
                            , nxt_resetNumBitsCounter} = 6'
                            b0_1_0_1_0_1;
                        nq = WAIT_BIT_TIME;
                end
                WAIT_BIT_TIME: begin
                        {nxt_done, nxt_shift, nxt_load,
                            nxt_incNumBits, nxt_resetBaudTickCounter
                            , nxt_resetNumBitsCounter} = 6'
                            b0_0_0_0_1_1;
```

```
                                        if(!baudTickCounterDone)
                                                nq = WAIT_BIT_TIME;
                                        else begin
                                                if (!bitCounterDone)
                                                        nq = SHIFT_REG;
                                                else
                                                        nq = IDLE;
                                        end
                                end
                        endcase

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////
// Module Name:     lb_UART_Tx_ControlUnit
// Project Name:
///////////////////////////////////////////////
module lb_UART_Tx_ControlUnit(
    input clk,
    input reset,
    input start,
    input bit8,
    input parity_en,
    input [19:0] baudPrescale,
        input cs,
    output done,
    output load,
    output shift
    );

        wire baudTickCounterDone, resetBaudTickCounter, bitCounterDone;
        wire incNumBits, resetNumBitsCounter;

        //assign startCounterNumBits  = resetNumBitsCounter & reset;
        //assign startBaudTickCounter = resetBaudTickCounter & reset;

        lb_16BaudTickCounter baudTickCounter (
                .clk(clk),
                .reset(reset),
                .prescale(baudPrescale),
                .cs(cs),
                .load(resetBaudTickCounter),
                .done(baudTickCounterDone)
                );

        lb_UART_Tx_FSM fsm (
```

```
                .clk(clk),
                .reset(reset),
                .start(start),
                .baudTickCounterDone(baudTickCounterDone),
                .bitCounterDone(bitCounterDone),
                .cs(cs),
                .done(done),
                .shift(shift),
                .load(load),
                .incNumBits(incNumBits),
                .resetBaudTickCounter(resetBaudTickCounter),
                .resetNumBitsCounter(resetNumBitsCounter)
                );

        lb_contNumBits counterNumBits (
                .clk(clk),
                .reset(reset),
                .parity_en(parity_en),
                .bit8(bit8),
                .inc(incNumBits),
                .cs(cs),
                .load(resetNumBitsCounter),
                .done(bitCounterDone)
        );


endmodule
```

```
`timescale 1ps / 1fs
////////////////////////////////////////
// Module Name:       lb_UART_toplevel
////////////////////////////////////////
module lb_UART_toplevel(
    input clk,
    input reset,
    input cs,
    input we,
        input oe,
        input adrs,
    input [7:0] data,
    input bit8,
    input parity_en,
    input odd_n_even,
    input [3:0] baud_val,
        input rx,
    output tx,
        output [7:0] data_out
```

```
        );

        wire [19:0] baud;
        wire system_reset;
        wire next_tx, next_txrdy;
        reg [7:0] status_in;
        wire [7:0] status_out;
        wire [7:0] data_rx;
        wire [7:0] data_buff;
        assign tx = (cs) ? next_tx : next_tx;
        assign txrdy = (cs) ? next_txrdy : next_txrdy;

        reg [7:0] buffer_tx;

        lb_UART_Tx_Core uart_tx(
    .clk(clk),
    .reset(reset),
    //.data(data),
    .data(buffer_tx),
        .start(we&~cs),
    .baud_value(baud),
    //.baud_value(20'b10),
        .bit8(bit8),
    .parity_en(parity_en),
    .odd_n_even(odd_n_even),
        .cs(cs),
    .tx(next_tx),
    .tx_done(next_txrdy)
    );

        lb_BAUD_Table baudMux (
    .baudSelect(baud_val),
    .counterValue(baud)
    );

        lb_UART_Rx_Core uart_rx (
                .clk(clk),
                .reset(reset),
                .baud_value(baud),
                .bit8(bit8),
                .parity_en(parity_en),
                .odd_n_even(odd_n_even),
                .cs(cs),
                .rx(rx),
                .data_out(data_rx),
                .done(donerx),
                .parity_err(parity_err),
```

```
            .stopErr(stopErr)
            );


    lb_8bitReg statusReg(
            .clk(clk),
            .reset(reset),
            .d_in(status_in),
            .load(donerx|(oe&(~cs))),
            .d_out(status_out)
            );

    lb_8bitReg dataRxBuff(
            .clk(clk),
            .reset(reset),
            .d_in(data_rx),
            .load(donerx),
            .d_out(data_buff)
            );

    //as the txrdy is already driven to a register before to come out
        the fsm,
    //we decided not store the txrdy into the status register avoiding
        delay.
    assign data_out = oe ? ((adrs) ? {status_out[7:2],txrdy,status_out
        [0]} : data_buff) : 8'hzzzzzzzz;

    always @ (*)
            //;
            if((oe&(~cs)) && !donerx) begin
                    status_in[6:4] = 3'b000;
                    status_in[0] = 1'b0;
                    status_in[1] = txrdy;//ignore
                    {status_in[7],status_in[3:2]} = 3'b000;
                    end
            else if(!(oe&(~cs)) && donerx) begin
                    status_in[6:4] = {stopErr, parity_err, donerx &
                        status_out[0]};
                    status_in[0] = donerx;
                    status_in[1] = txrdy; //ignore
                    {status_in[7],status_in[3:2]} = 3'b000;
                    end
            else if((oe&(~cs)) && donerx) begin
                    status_in[6:4] = {stopErr, parity_err, 1'b0};
                    status_in[0] = donerx;
                    status_in[1] = txrdy; //ignore
                    {status_in[7],status_in[3:2]} = 3'b000;
```

```
                        end

        always @ (posedge clk, negedge reset)
                if(!reset)
                        buffer_tx <= 0;
                else if(we)
                        buffer_tx <= data;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////
// Module Name:     lb_UART_Rx_Core
//////////////////////////////////////
module lb_UART_Rx_Core(
    input clk,
    input reset,
    input [19:0] baud_value,
    input bit8,
    input parity_en,
    input odd_n_even,
        input cs,
        input rx,
    output [7:0] data_out,
    output done,
        output reg parity_err,
        output reg stopErr //stopbit error
        );

        wire shift;
        reg [10:0] data_received;
        wire [11:0] data_tobe_modified;

        reg parity;

        lb_UART_Rx_ControlUnit rx_core (
                .clk(clk),
                .reset(reset),
                .bit8(bit8),
                .parity_en(parity_en),
                .baudPrescale(baud_value),
                //.baudPrescale(20'b11),
                .cs(cs),
                .rx(rx),
                .done(done),
                .shift(shift)
                );
```

```
lb_shiftreg_rx shift_reg (
          .clk(clk),
          .reset(reset),
          .data_in(rx),
          .shift(shift),
          .data_out(data_tobe_modified)
          );

always @ (*)
          case({parity_en, bit8})
          0: data_received = {2'b11,data_tobe_modified[9:1]};
          1: data_received = {1'b1,data_tobe_modified[10:1]};
          2: data_received = {1'b1,data_tobe_modified[10:1]};
          3: data_received = data_tobe_modified;
          endcase

assign data_out = (bit8) ? data_received[8:1] : {1'b0,data_received
    [7:1]};

//even = par
//odd = impar
//Parity Check
always @(*)
          if(parity_en) begin

                    if(bit8) begin
                              if(odd_n_even) begin
                                        if (^data_received[9:1] == 0)
                                                  parity_err = 1'b0;
                                        else
                                                  parity_err = 1'b1; //
                                                      parity even error
                                                      detected
                              end
                              else begin
                                        if (^data_received[9:1] == 1)
                                                  parity_err = 1'b0;
                                        else
                                                  parity_err = 1'b1; //
                                                      parity odd error
                                                      detected
                              end
                    end

                    else begin
                              if(odd_n_even) begin
```

```
                                        if (^data_received[8:1] == 0)
                                                parity_err = 1'b0;
                                        else
                                                parity_err = 1'b1; //
                                                    parity even error
                                                    detected
                                end
                                else begin
                                        if (^data_received[8:1] == 1)
                                                parity_err = 1'b1;
                                        else
                                                parity_err = 1'b0; //
                                                    parity odd error
                                                    detected
                                end
                        end
                end
                else
                        parity_err = 0;


        //stopBit Error
        always @ (*)
                case ({parity_en, bit8})
                        0: begin
                                if(data_received[8] != 1)
                                        stopErr = 1;
                                else
                                        stopErr = 0;
                        end
                        1:begin
                                if(data_received[9] != 1)
                                        stopErr = 1;
                                else
                                        stopErr = 0;
                        end
                        2:begin
                                if(data_received[9] != 1)
                                        stopErr = 1;
                                else
                                        stopErr = 0;
                        end
                        3:begin
                                if(data_received[10] != 1)
                                        stopErr = 1;
                                else
                                        stopErr = 0;
```

```
                        end
                        endcase
endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////
// Module Name:      lb_UART_Rx_ControlUnit
///////////////////////////////////////////
module lb_UART_Rx_ControlUnit(
    input clk,
    input reset,
    input bit8,
    input parity_en,
    input [19:0] baudPrescale,
        input cs,
    input rx,
        output done,
    output shift
    );

        wire baudTickCounterDone, resetBaudTickCounter, bitCounterDone;
        wire incNumBits, resetNumBitsCounter, half_n_complete;

        lb_UART_Rx_fsm rx_fsm (
                .clk(clk),
                .reset(reset),
                .baudTickCounterDone(baudTickCounterDone),
                .bitCounterDone(bitCounterDone),
                .rx(rx),
                .done(done),
                .shift(shift),
                .half_n_complete(half_n_complete),
                .incNumBits(incNumBits),
                .resetBaudTickCounter(resetBaudTickCounter),
                .resetNumBitsCounter(resetNumBitsCounter)
                );



        lb_16_n_8BaudTickCounter btickcounter (
                .clk(clk),
                .reset(reset),
                .prescale(baudPrescale),
                ._16_or_8_ticks(half_n_complete),
                .cs(cs),
                .load(resetBaudTickCounter),
                .done(baudTickCounterDone)
```

```
                );

        lb_contNumBits bitCounterRx (
                .clk(clk),
                .reset(reset),
                .parity_en(parity_en),
                .bit8(bit8),
                .inc(incNumBits),
                .cs(cs),
                .load(resetNumBitsCounter),
                .done(bitCounterDone)
                );

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////
// Module Name:    lb_reset_processor
//////////////////////////////////////
module lb_reset_processor(
        input clk,                              //system clock
    input resetb,                               //reset button signal
        input cs,
    output wire system_reset    //system reset signal assynch-in, synch-out
        .
    );

        wire qd0_dd1;
        wire logic_one;

        assign logic_one = 1'b0;
        //              D,    clk,   reset,   cs, Q
        lb_dff_processor d0(logic_one, clk, resetb, cs, qd0_dd1),
                    d1(qd0_dd1, clk, resetb, cs, system_reset);



endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////
// Module Name:    lb_pulseMakerNegEdge
////////////////////////////////////////
module lb_pulseMakerNegEdge(
    input clk,                  //clock
    input signal_in,    //signal which the module is going to identifie the
         transition
        input reset,        //reset (negedge)
```

```
        input cs,                         //chip select
    output wire pulse     //pulse generated.
    );

        wire qd0_dd0;
        wire qd1_and;
        //                 D,     clk,   reset,   Q
        lb_dff d0(signal_in, clk, reset, cs, qd0_dd0),
                   d1(qd0_dd0, clk, reset, cs, qd1_and);

        assign pulse = (qd1_and & (~qd0_dd0));

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////
// Module Name:     lb_dff_processor
//////////////////////////////////////
module lb_dff_processor(
        input D,                    //Data input
    input clk,            //Clock
    input reset ,         //Reset
        input cs,
        output reg Q      //Data output
    );

always @ (posedge clk, posedge reset)
        //if(!cs) begin
                if(reset)
                        Q <= 1'b1;
                else
                        Q <= D;
        //end


endmodule
```

```
`timescale 1ps / 100fs
////////////////////////////
// Module Name:     lb_dff
///////////////////////////
module lb_dff(
    input D,              //Data input
    input clk,            //Clock
    input reset ,         //Reset
        input cs,
        output reg Q      //Data output
    );
```

```
always @ (posedge clk , negedge reset)
        //if(!cs) begin
                if(!reset)
                        Q <= 1'b0;
                else
                        Q <= D;
        //end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////
// Module Name:    lb_contNumBits
////////////////////////////////
module lb_contNumBits(
    input clk,
    input reset,
    input parity_en,
    input bit8,
        input inc,
        input cs,
        input load,
    output done
    );

        reg [3:0] value;

        lb_counter4Bits counter4Bits (
                .clk(clk),
                .reset(reset),
                .inc(inc),
                .value(value),
                .cs(cs),
                .load(load),
                .done(done)
                );

        always @ (*)
                case({parity_en, bit8})
                        2'b00: value = 10;
                        2'b01: value = 11;
                        2'b10: value = 11;
                        2'b11: value = 12;
                endcase

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////
// Module Name:     lb_clockCounter
////////////////////////////////////
module lb_clockCounter(
    input clk,
        input reset,
        input [N-1:0] value,
        input cs,
        input load,
        output wire done
    );

        localparam N = 20;
        reg [N-1:0] n_bits;
        reg [N-1:0] cont;
        wire [N-1:0] n_cont;

        assign n_cont = ((done) ? 0 : (cont+1));
        assign done = (cont==n_bits) ? 1 : 0;

        always @(posedge clk, negedge reset)
                //if(!cs) begin
                        if(!reset) begin
                                cont <= 0;
                                n_bits <= value;
                        end
                        else begin
                                if(!load) begin
                                        cont <= 0;
                                        n_bits <= value;
                                end
                                else
                                        cont <= n_cont;
                        end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////
// Module Name:     lb_bitCounter
//////////////////////////////////////
module lb_counter5Bits(
    input clk,
        input reset,
        input inc,
        input [N-1:0] value,
```

```
        output wire done
    );

        localparam N = 4;
        reg [N-1:0] n_bits;
        reg [N-1:0] cont;
        wire [N-1:0] n_cont;

        assign n_cont = (inc) ? ((done) ? 0 : (cont+1)) : cont;
        assign done = (cont==n_bits) ? 1 : 0;

        always @(posedge clk, negedge reset)
                if(!reset) begin
                        cont <= 0;
                        n_bits <= value;
                end
                else cont <= n_cont;

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////
// Module Name:     lb_BAUD_Table
///////////////////////////////////////////
module lb_BAUD_Table(
    input [3:0] baudSelect,
    output reg [19:0] counterValue
    );

        always @ (*)
                case(baudSelect)
                        4'b0000: counterValue = 20'd20832; //300
                        4'b0001: counterValue = 20'd5207;
                        4'b0010: counterValue = 20'd2603;
                        4'b0011: counterValue = 20'd1301;
                        4'b0100: counterValue = 20'd650; //9600 BAUD
                        4'b0101: counterValue = 20'd324;
                        4'b0110: counterValue = 20'd162;
                        4'b0111: counterValue = 20'd107;
                        4'b1000: counterValue = 20'd53;
                        4'b1001: counterValue = 20'd27;
                        4'b1010: counterValue = 20'd14;
                        4'b1011: counterValue = 20'd7;    //921600 BAUD
                        default: counterValue = 20'd650; //9600 BAUD
                endcase
/*
        always @ (*)
```

```
                case(baudSelect)
                        4'b0000: counterValue = 20'd166665; //300
                        4'b0001: counterValue = 20'd41667;
                        4'b0010: counterValue = 20'd20833;
                        4'b0011: counterValue = 20'd10417;
                        4'b0100: counterValue = 20'd5209; //9600 BAUD
                        4'b0101: counterValue = 20'd2604;
                        4'b0110: counterValue = 20'd1303;
                        4'b0111: counterValue = 20'd868;
                        4'b1000: counterValue = 20'd434;
                        4'b1001: counterValue = 20'd217;
                        4'b1010: counterValue = 20'd109;
                        4'b1011: counterValue = 20'd54;    //921600 BAUD
                        default: counterValue = 20'd5209; //9600 BAUD
                endcase
*/

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////
// Module Name:     lb_16_n_8BaudTickCounter
//////////////////////////////////////////
module lb_16_n_8BaudTickCounter(
     input clk,
    input reset,
    input [N-1:0] prescale,
        input _16_or_8_ticks, //16 = 1, 8 = 0
        input cs,
        input load,
        output done
    );

        localparam N = 20;

        wire inc, doneCounterPulseMaker;
        wire [3:0] countValue;

        assign countValue = (_16_or_8_ticks) ? 4'hF : 4'h7;

        lb_pulseMakerNegEdge pulseMakerNegEdge (
                .clk(clk),
                .signal_in(doneCounterPulseMaker),
                .reset(reset),
                .cs(cs),
                .pulse(done)
                );
```

```
        lb_counter4Bits counter4Bits (
                .clk(clk),
                .reset(reset),
                .inc(inc),
                .value(countValue), //16 or 8 Clock ticks each BTICK
                .cs(cs),
                .load(load),
                .done(doneCounterPulseMaker)
                );

        lb_clockCounter clockCounter (
                .clk(clk),
                .reset(reset),
                .value(prescale),
                //.value(20'b11),
                .cs(cs),
                .load(load),
                .done(inc)
                );


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////
// Module Name:     lb_8bitReg
/////////////////////////////////////////////
module lb_8bitReg(
    input clk,
    input reset,
    input [7:0] d_in,
    input load,
    output reg [7:0] d_out
    );

        always @ (posedge clk, negedge reset)
                if(!reset)
                        d_out <= 8'b0000_0000;
                else if(load)
                        d_out <= d_in;
                        else
                        d_out <= d_out;
endmodule
```

Anexos

ANEXO A – Requisitos de Tempo de Leitura Assíncrona em Memória

Tabela de valores fornecida pelo fabricante da memória para os requisitos de tempo apresentados na Figura 8.

Parameter	Symbol	70ns		85ns		Unit	Notes
		Min	Max	Min	Max		
Address access time	^{t}AA		70		85	ns	
ADV# access time	^{t}AADV		70		85	ns	
Page access time	^{t}APA		20		25	ns	
Address hold from ADV# HIGH	^{t}AVH	2		2		ns	
Address setup to ADV# HIGH	^{t}AVS	5		5		ns	
LB#/UB# access time	^{t}BA		70		85	ns	
LB#/UB# disable to DQ High-Z output	^{t}BHZ		8		8	ns	1
LB#/UB# enable to Low-Z output	^{t}BLZ	10		10		ns	2
Maximum CE# pulse width	^{t}CEM		4		4	µs	3
CE# LOW to WAIT valid	^{t}CEW	1	7.5	1	7.5	ns	
Chip select access time	^{t}CO		70		85	ns	
CE# LOW to ADV# HIGH	^{t}CVS	7		7		ns	
Chip disable to DQ and WAIT High-Z output	^{t}HZ		8		8	ns	1
Chip enable to Low-Z output	^{t}LZ	10		10		ns	2
Output enable to valid output	^{t}OE		20		20	ns	
Output hold from address change	^{t}OH	5		5		ns	
Output disable to DQ High-Z output	^{t}OHZ		8		8	ns	1
Output enable to Low-Z output	^{t}OLZ	3		3		ns	2
Page READ cycle time	^{t}PC	20		25		ns	
READ cycle time	^{t}RC	70		85		ns	
ADV# pulse width LOW	^{t}VP	5		7		ns	

Notes:
1. Low-Z to High-Z timings are tested with the circuit shown in Figure 27 on page 36. The High-Z timings measure a 100mV transition from either V_{OH} or V_{OL} toward $V_{CC}Q/2$.
2. High-Z to Low-Z timings are tested with the circuit shown in Figure 27 on page 36. The Low-Z timings measure a 100mV transition away from the High-Z ($V_{CC}Q/2$) level toward either V_{OH} or V_{OL}.
3. Page mode enabled only.

ANEXO B – Requisitos de Tempo de Escrita Assíncrona em Memória

Tabela de valores fornecida pelo fabricante da memória para os requisitos de tempo apresentados na Figura 9.

Parameter	Symbol	70ns		85ns		Unit	Notes
		Min	Max	Min	Max		
Address and ADV# LOW setup time	^tAS	0		0		ns	
Address HOLD from ADV# going HIGH	^tAVH	2		2		ns	
Address setup to ADV# going HIGH	^tAVS	5		5		ns	
Address valid to end of WRITE	^tAW	70		85		ns	
LB#/UB# select to end of WRITE	^tBW	70		85		ns	
CE# LOW to WAIT valid	^tCEW	1	7.5	1	7.5	ns	
CE# HIGH between subsequent async operations	^tCPH	5		5		ns	
CE# LOW to ADV# HIGH	^tCVS	7		7		ns	
Chip enable to end of WRITE	^tCW	70		85		ns	
Data HOLD from WRITE time	^tDH	0		0		ns	
Data WRITE setup time	^tDW	20		20		ns	
Chip disable to WAIT High-Z output	^tHZ		8		8	ns	1
Chip enable to Low-Z output	^tLZ	10		10		ns	2
End WRITE to Low-Z output	^tOW	5		5		ns	2
ADV# pulse width	^tVP	5		7		ns	
ADV# setup to end of WRITE	^tVS	70		85		ns	
WRITE cycle time	^tWC	70		85		ns	
WRITE to DQ High-Z output	^tWHZ		8		8	ns	1
WRITE pulse width	^tWP	45		55		ns	3
WRITE pulse width HIGH	^tWPH	10		10		ns	
WRITE recovery time	^tWR	0		0		ns	

Notes:
1. Low-Z to High-Z timings are tested with the circuit shown in Figure 27 on page 36. The High-Z timings measure a 100mV transition from either V_{OH} or V_{OL} toward $V_{CC}Q/2$.
2. High-Z to Low-Z timings are tested with the circuit shown in Figure 27 on page 36. The Low-Z timings measure a 100mV transition away from the High-Z ($V_{CC}Q/2$) level toward either V_{OH} or V_{OL}.
3. WE# LOW time must be limited to ^tCEM (4µs).