TÉCNICO LISBOA

Embedded Software Development Environment for the OpenRisc Processor (Bare-Metal)

Carlos Alexandre Antunes Rodrigues

Dissertação para a obtenção de Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Nome do Presidente
Orientador:	Nome do Orientador
Vogais:	Nome do Vogal 1
	Nome do Vogal 2

Abril de 2014

Dedicated to someone special...

Agradecimentos

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: OpenRISC, Sistema em um chip,...

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: OpenRISC, System On Chip,...

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Acrônimos

ACK Acknowledge

Capítulo 1

Introdução

O sistema num chip em inglês System On Chip *System on Chip* é um chip integrado que tem integrado todas as funcionalidades de computador ou de um sistema electrónico, num único chip. Um SoC é constituído tipicamente por vários elementos interligados entre si por um barramento. Os elementos podem ser separados por grupos conformes os seus principais funcionalidades, os principais grupos são os seguintes: processamento, memoria, fontes de timing, periféricos, interfaces externas, interfaces analógicas e gestores de energia. Na figura 1.1 pode se um exemplo de um diagrama de blocos de um SoC, onde se pode o barramento ligação dos elementos.

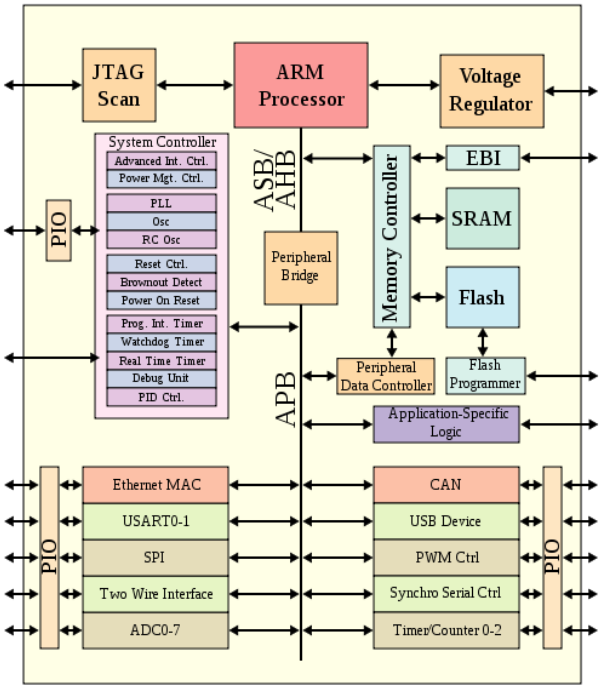


Figura 1.1: Diagrama de blocos de um soc ARM

Os SoC têm uma vasta possibilidade de utilização desde um simples relógio até no mais avançado tecnologicamente como no desenvolvimentos de módulos para satélites, passando pela industria automóvel e com o aparecimento do arduino cada vez mais os SoC são utilizados em projetos de pequena

escala desenvolvidos em casa, porque veio permitir a pessoas de com pouco ou nenhum conhecimento na área programar e efectuar debug no seu projecto com o SoC. A sua utilização traz vantagens com preço baixo, baixo consumo de energia e dimensões pequenas, como tudo o que existe também tem desvantagens sendo neste caso o baixo poder de calculo comparado com um computador.

Com uma área tão vasta de aplicações não é de estranhar que existam várias empresas a muito tempo e varias startup no mercado a desenvolver SoC para fins totalmente distintos, cada empresa optimizando o seu para que foi desenvolvido.

1.1 Contexto

Uma startup no desenvolvimento de um projecto/produto necessita de um SoC para poder comercializar o seu produto. Na pesquisa do SoC ideal para o seu projecto encontram diferentes possibilidades, mas nenhuma preenchia todos os critérios pretendidos para o projecto. Como não foi encontrada uma solução ideal dos vários SoC disponíveis no mercado, a solução possível serias desenvolver o seu próprio SoC desenvolvido a medidas para o projecto.

1.2 Motivação

desenvolver um sistema necessário para uma startup, o sistema vai ser desenvolvido a medida com o pretendido com a startup.

1.3 Objectivos

A startup no desenvolvimento de um projeto/produto necessita de um SoC, como se pode ver na figura 1.2 que tem de ter a capacidade de processamento necessária para efetuar a descodificação e codificação de dados que irá receber e enviar. O SoC desenvolvido tem de permitir que sejas ligado mais dois modelos assíncronos permitindo a recolha de dados no local para posterior envio dos dados pelo modelo de comunicação. O produto é para ser usado em locais de difíceis acessos este terá de ter um baixo consumo, permitindo que a bateria funcione o maior tempo possível. Para alem do mencionado anteriormente também necessita de duas interfaces de comunicação *Universal Asynchronous Receiver/Transmitter* e *General Purpose Input/Output*.

1.4 Desafios

Os principais desafios desta dissertação consiste no desenvolvimento de um sistema sintetizável que preencha todas as necessidades mencionadas pela startup.

Desenvolver uma interface assíncrona necessária para a comunicação entre o SoC e os modelos de comunicação e de recolha de dados a ser desenvolvido pela startup.

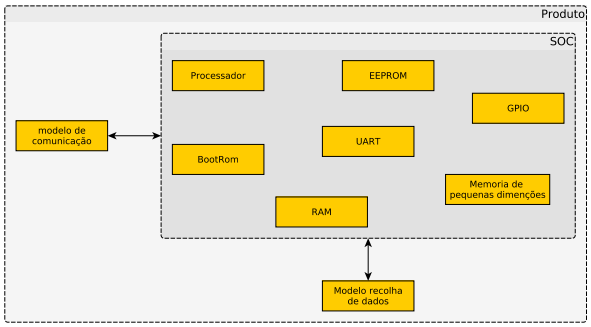


Figura 1.2: SOC pretendido pela Startup

A criação de uma bootrom que carregua para a memoria principal o programa que se encontra na EEPROM. Ainda testa o mau funcionamento da EEPROM ou da memoria principal notificando o utilizador.

Capítulo 2

Estado-da-arte

Existem duas maneiras de desenvolver o SoC desenvolvendo um sistema por completo incluindo processador, bus de comunicação, protocolos de comunicação necessários, que torna complicado a idealizar e desenvolver no tempo disponível para a sua execução. Ou juntamo-nos a uma das várias comunidades Open Hardware (open source de hardware) já existentes onde já tem algumas das funcionalidades idealizadas ou desenvolvidas, ainda permitindo contribuir para a comunidade escolhida com alguns melhoramentos ou resoluções de problemas existem nos seus projectos.

Das maiores comunidades de open source de hardware é a OpenCores, com aproximadamente 800 projectos e 95 mil utilizadores registados em 2010. Os projectos desenvolvidos por esta comunidade por serem no desenvolvimento de hardware usam uma linguagem de descrição de hardware na maioria dos projectos da comunidade são desenvolvidos em Verilog uma das linguagens mais utilizadas na descrição de hardware.

2.1 OpenRISC

O principal projecto inicial da comunidade é o projecto OpenRISC, tem como objectivo desenvolver uma serie de processadores com a arquitetura *Reduced Instruction Set Computing*.

A primeira discrição arquitectura é para o projecto OpenRISC1000, sendo um processador de 32 ou 64 bits com a opção de ponto flutuante e suporte para um processamento vectorial. A equipa OpenCores fez a primeira aplicação de um SoC com um processador deste tipo dando-lhe o nome *OpenRISC12000* este é escrito em verilog sintetizavel e tem um capacidade de processamento semelhante ao processador ARM10. Varias empresas desenvolvem os seus processadores com base neste processador entre as mais conhecidas estão a Samsung e a Nasa.

A discrição do segundo projecto de processadores com o nome OpenRISC2000 pretende ser um sucessor do anterior sendo as características principais compatível com o projeto anterior, projeto modular, adaptado para o uso de multicores, destinado para o embebido ou seja para dados de 16 e 32 bits sem suporte para 64 bits e desenvolvido para pequenos e médios processadores de *Field Programmable Gate Array*

2.1.1 Arquitectura

Na arquitectura desde processador como se pode ver na figura 2.1 o processador tem disponível unidade de debug permitindo um realizar-se debug em tempo real onde é ligado um de *Joint Test Action Group*, tick timer de alta resolução, controlador de interrupções programáveis, gestor de energia e uma cache e uma unidade de gestão de memoria tanto para instruções como para dados. Estas unidades mencionadas têm a hipótese de serem facilmente removidas caso não sejam necessárias permitindo assim que o processador tem uma tamanho menor e um consumo menor de energia. Para alem desta unidade opcionais existem duas outras unidas designadas Wishbone que pertencer a arquitectura base do processador, uma pertence a interface de instruções e a outra a interface de dados. Estas duas unidades fazem a conversão da interface interna RISC para a interface Wishbone que é recebida pelos periféricos.

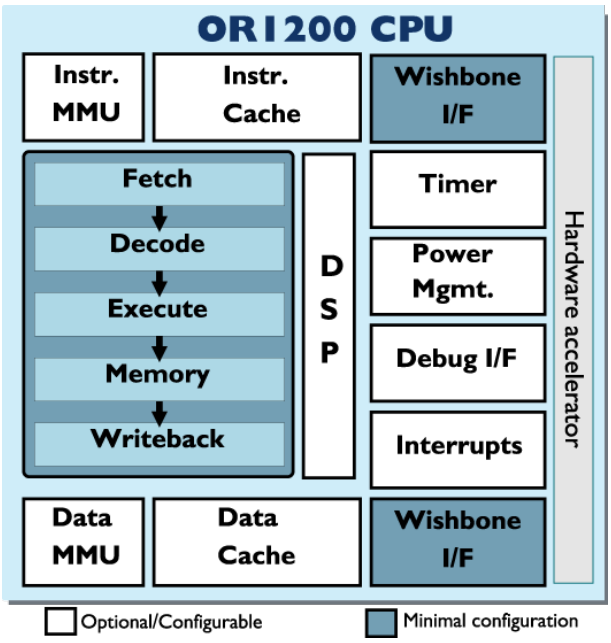


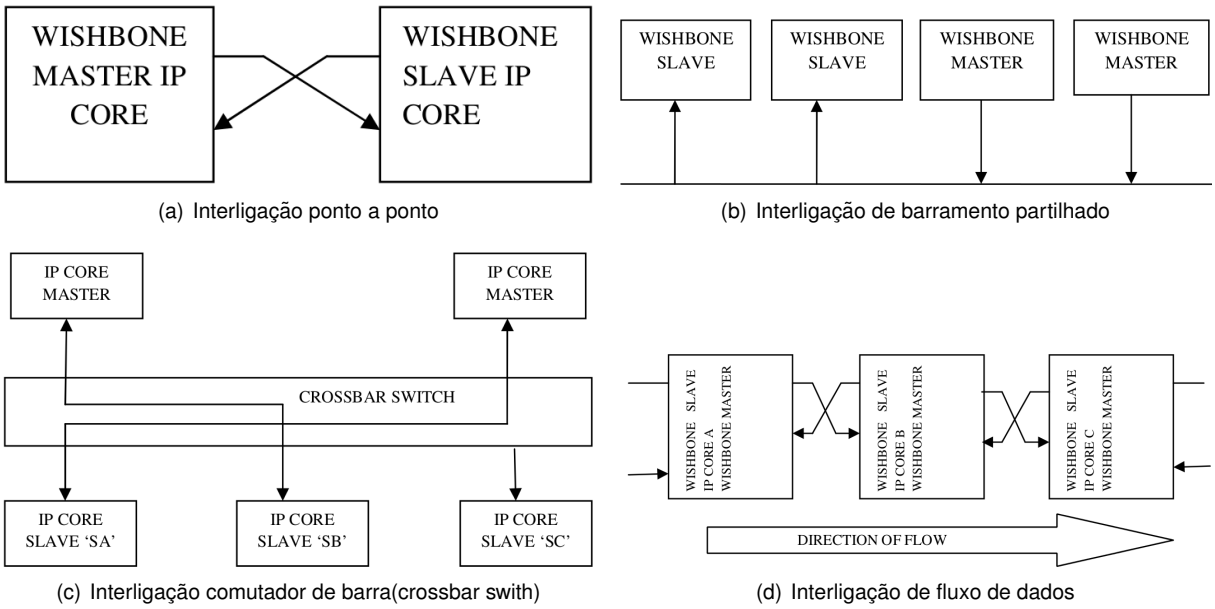
Figura 2.1: Diagrama de blocos do processador OpenRISC1200

2.1.2 Arquitectura Wishbone

A interface de barramento wishbone é hardware opensource, permitindo comunicação entre varias partes de um circuito integrado com o objetivos de ligar diferentes cores dentro de um chip. A interface é bastante utilizada em CPU e periféricos opensource, onde se destacam muitos dos projectos da comunidade OpenCores. A comunidade recomenda que todos os cores tenham disponível uma interface wishbone. O barramento wishbone foi desenvolvido pela silicore corporation em 1999 disponibilizando para domínio publico uma biblioteca em VDHL, a partir de 2002 a comunidade OpenCores tornou-se também sponsors do wishbone tendo uma pagina dedicada onde estão disponível novas revisões.

Existem várias interfaces possíveis com a arquitectuta wisbone os 4 tipos mais habituais podem ser visto na figura 2.2. A interligação ponto a ponto, na figura 2.2(a), permite apenas uma ligação de um periférico, este tipo de ligação não normalmente utilizada em SoC por estes normalmente serem

constituídos por vários periféricos. A interface de barramento partilhado, na figura 2.2(b), permite a utilização de vários mestres e vários escravos, visto que o barramento é partilhado o quanto um mestre utiliza o barramento o outro escravo tem de esperar que fique disponível o controlo é feito por um arbitro que decidi que arbrito controlo o barramento naquele momento. No caso do comutador de barra , na figura 2.2(c),é utilizado para uma tipologia multi-core. Este permite uma comunicação entre dois mestres e os escravos ao mesmo tempo desde que estejam a aceder a escravos diferentes, é semelhante ao barramento partilhado com uma elevada taxa de transferência de dados. Por ultimo a interligação de fluxos de dados, na figura 2.2(d), a informação flui de periférico para periférico, todos os periféricos têm de ter a interface de escravo e de mestre.



(a) Interligação ponto a ponto

(b) Interligação de barramento partilhado

(c) Interligação comutador de barra(crossbar swith)

(d) Interligação de fluxo de dados

Figura 2.2: Interfaces Wishbone

A interligação utilizada no desenvolvimento de SoC com apenas uma unidade de processamento é o barramento partilhado, porque tem vários periféricos disponíveis e por ser de simples implementação. Onde é feita toda a gestão da interface wishbone é dado o nome Intercon, este é constituído por vários elementos como multiplexer e árbitros wishbone. Como se pode ver na figura 2.3 ambos os modelos de wishbone do processador mencionados na figura 2.1 estão ligados cada um deles a um multiplexer um de dados e outro de instruções, estes enviam os dados para o periférico correspondente conforme o escalonamento atribuído a cada periférico. Existe uma ligação de ambos os multiplexeres ao árbitro este faz o controlo dos acessos à memoria principal, pois é possível aceder a memoria principal por necessitar de novas instruções como de necessitar de dados lá existentes. Os sinais da interface wishbone encontram-se descritos na tabela2.1, na tabela a direcção dos sinais mencionados visto do periférico escravo.

Para adicionar um novo periférico ao SoC que tem disponível uma interface wishbone, a interface do escravo é ligada ao multiplexer de wishbone de dados à semelhança dos outros periféricos ligados na figura 2.3, tem de ser atribuído ao periférico um conjunto de endereços disponíveis.

A interface wishbone é constituida por 12 sinais distintos que se encontram descritos na tabela 2.1,

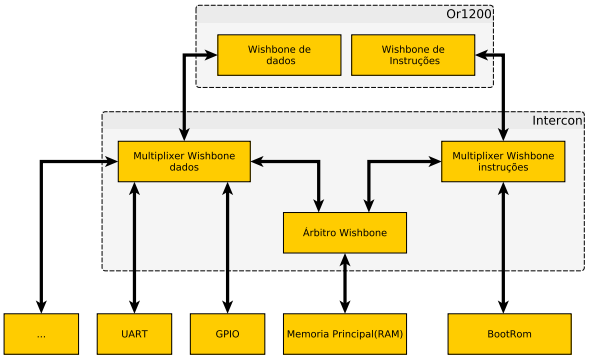


Figura 2.3: Diagrama de blocos da arquitectura wishbone.por esta figura mais completa ou uma como estava originalmente, sem bootrom

a maior parte dos sinais com a excepção de wbm_adr_i, wbm_dat_i, wbm_dat_o e wbm_sel_i são de apenas um bit e não variam o tamanho de bits conforme a quantidade de bits do SoC. na tabela mostra os sinais visto do lado do periférico.

Nome	Direcção	Tamanho(bits)	Descrição
wbm_cki_i	Input	1	Clock do sistema para a interface Wishbone.
wbm_rst_i	Input	1	Sinal de Reset (activo com valor lógico '1').
wbm_cyc_i	Input	1	Validação da informação no Bus.
wbm_adr_i	Input	32	Endereço para escrita ou leitura no periférico.
wbm_dat_i	Input	32	Dados enviados para o periférico.
wbm_dat_o	Output	32	Dados enviados pelo periférico.
wbm_sel_i	Input	4	Seleciona Byte para escrever ou ler.
wbm_ack_o	Output	1	Sinal de acknowledgment.
wbm_err_o	Output	1	Indica um ciclo anormal ocorreu um encerramento.
wbm_we_i	Input	1	sinal de leitura ou escrita, se tiver logico '1' escreve.
wbm_stb_i	Input	1	Valida os dados transmitidos.
wbm_rty_0	Output	1	

Tabela 2.1: Tabela dos sinais da interface Wishbone.

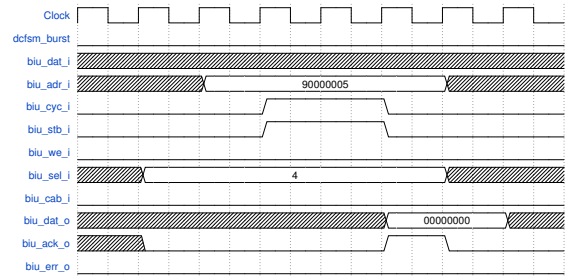
A interface Wishbone mestre é controlada pela elemento Data ou Instr. Cache, dependendo se se trata da interface de dados ou de instruções respectivamente, que pode ser visto na figura 2.1. Ai a leitura e as escritas para podem ser simples, leitura de apenas uma posição de memoria, ou brust, uma sequencia de posições de memória, em brust são feitas 4 acessos a memorias sequenciais, isto é definido está definido no código e corresponde ao tamanho do MMU correspondente. Na figura 2.4 podem-se ver vários diagrama temporais de leituras e escritas do elemento Cache.

Nas figuras 2.4(a) e 2.4(b) corresponde a diagramas temporais lidos no elemento Data Cache, o sinal dcfsm_burst aciona uma leitura ou escrita em burst. Na primeira figura temos uma leitura simples, como se pode ver o sinal de burst está com o valor lógico de 'zero' e o sinal biu_sel_i tambem se encontra com o valor logico de 'zero' indicando que é uma leitura. Também se pode ver que o sinal biu_sel_i é o primeiro a ser definido e tem o valor 4 em hexadecimal, indicando que é para ser lido apenas 2 Bytes e são para ser coculados nos 2 Bytes mais significativos do sinal biu_dat_o.

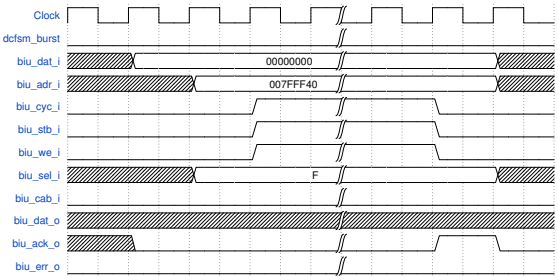
Na figura 2.4(b) temos uma escrita simples, neste caso ainda temos o sinal dcfsm_burst com o valor

logico de 'zero', mas biu_we_i já tem o valor lógico de 'um' acionado ao mesmo tempo que os sinais biu_cyc_i e biu_stb_i. Já neste caso o sinal biu_sel_i com o valor F em hexadecimal indica que todos os bytes de biu_dat_i são validos.

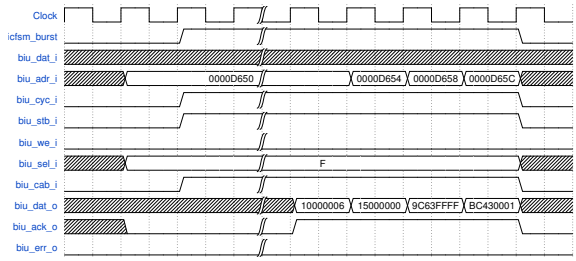
Por ultimo na figura 2.4(c) é um diagrama temporal do elemento Instr. Cache, onde é representado uma leitura em burst. Neste caso podemos ver que o sinal icfsm_burst tem o valor logico 'um', também se pode ver no sinal biu_adr_i que o endereço se mantem até receber o primeiro ACK a partir dai é incrementado de 4 em 4 posições de memoria em cada flanco ascendente visto que o periférico disponibiliza a palavra e que ao fim de receber 4 palavras o sinal de burst fica com o valor logico de 'zero'.



(a) Diagrama temporal do ciclo de leitura simples



(b) Diagrama temporal do ciclo de escrita simples



(c) Diagrama temporal do ciclo de leitura burst

Figura 2.4: Diagrama temporais Wishbone

2.1.3 Toolchain

Uma Toolchain é um conjunto de ferramentas de programação que permitem criar programas, vulgarmente uma toolchain simples disponível um compilador, um linker para fazer a linkagem do código compilado para um programa executável, bibliotecas que fornecem interface com o sistemas operativo e um debugguer. Uma das toolchain mais utilizadas para desenvolver programas em C é a toolchian da GNU sendo vital para o desenvolvimento de linux, alguns sistemas BSD e software para sistemas embebidos. A toolchain da GNU disponibliza mais algumas ferramentas do que uma simples toochain, como uma ferramenta para compilação automática vulgarmente conhecida por Make.

Por ser uma toolchain bastante utilizada em desenvolvimento de software a comunidade utiliza a toolchain da gnu para desenvolver software, mas o processador ainda não é suportado pela toolchain. A comunidade adicionou o seu processador em duas bibliotecas na newlib e na uClibc, a newlib é uma biblioteca já testada e utilizada desde a versão 1.18.0 com suporte de placas, sendo uma pequena e simples biblioteca de C do que uClibc e é a melhor para o desenvolvimento de aplicações em bare-

metal ou seja sem sistema operativo. uClibc é um biblioteca de C para sistemas embebidos onde foi removido algumas partes do padrão de C, mas ainda dispoen de todas a funcionalidades necessarias para um sistema operativo, é ideal para sistema embebidos suportando ARM, amd64,i386.

A biblioteca newlib é utilizada no desenvolvimento de aplicação em bare-metal por isso é necessário indicar ao compilador para fazer a linkagem. Para esse efeito existe a flag -mboard"onde se indica qual é a placa onde o o programa irá correr. Existem já algumas placas predefinidas como or1ksim, simulador or1ksim sem *Universal Asynchronous Receiver/Transmitter*, or1ksim-uart, simulador or1ksim com UART, de0_nano, placa FPGA de0 nano da Terasic. Quando indicamos com a flag qual é a placa que utilizamos o compilador irá buscar um ficheiro com o mesmo nome já précompilado que contem informações importante da placa que são frequencia de clock, endereço base da memoria principal, tamanho da memoria principal, endereço base da UART, buad rate da UART e o numero IRQ da UART. É possivel criar um ficheiro com as propriedades da placa que pretender para isso tem de criar um ficheiro com o nome da placa com a extensão .S.

2.1.4 OrpSoc

No desenvolvimento do SoC a comunidade OpenRISC percebeu-se da necessidade de uma plataforma de desenvolvimento fácil e modelar. Por esses motivos desenvolveram o *OpenRISC Reference Platform System-on-Chip* destinando-se a desenvolver um ambiente de verificação e de desenvolvimento de Cores IP ou SoC. Para alem dos principais objectivos teria de ser simples de usar tanto por utilizadores experimentes como por utilizadores novos, permitindo este simular e sintetizar o seu projeto facilmente. A plataforma encontrasse separada do repositório onde se encontra os sistemas e os cores, permitindo assim que a plataforma seja utilizada com SoC totalmente diferentes e por outras identidades.

O repositório onde se encontram os sistemas e os cores têm uma distribuição como se encontra na figura 2.5, no caso da OpenRISC o repositório tem o nome de Orpsoc-cores. Dentro do repositório existe mais dois com os nomes de cores e systems, dentro do repositório systems estão todos os SoC desenvolvidos ou em desenvolvimento cada um com o teu repositório especifico, dentro de cada SoC existem vários ficheiros sendo 2 deles bastante importantes que têm de ter o nome do sistema com as extensões .core e .system, por exemplo de0_nano.core e de0_nano.system no caso que seja o sistema de0_nano. O ficheiro .system tem descrição sobre o sistema e a localização dos ficheiros necessários para sintetizar o sistemas para uma determinada FPGA. No caso do ficheiro .core contem toda as dependências do sistema em relação aos cores, tendo também secção das várias ferramentas de simulação onde cada uma descrimina informações necessárias para a sua compilação, como ficheiros de testbench, flags de compilação e o ficheiro principal.

Quanto ao repositórios cores contem vários cores onde cada um tem o seu repositório onde é obrigatório ter o ficheiro com a extensão .core que contem a uma descrição do core, dependência de outros cores e os ficheiros de descrição do core. É possível que os ficheiros de discrição não estejam no repositório, nesse caso o ficheiro também contem um secção que indica a sua localização no servidor de Subversion da comunidade e qual a revisão, neste caso a plataforma *OpenRISC Reference Platform*

System-on-Chip também tem a responsabilidade de fazer uma copia deste core.

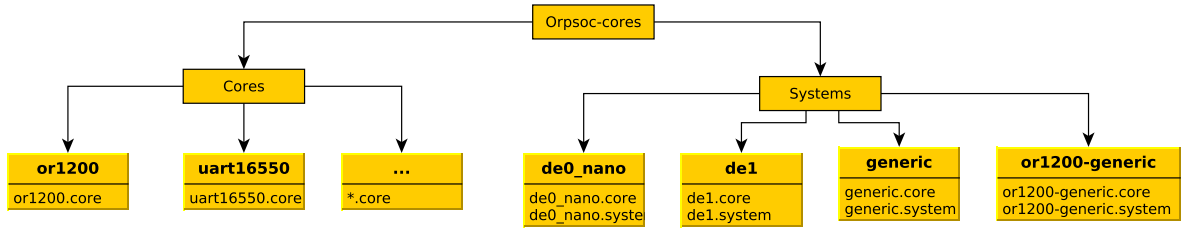


Figura 2.5: Organização do sistemas e cores da OpenRisc.

Cada sistema que se encontra no repositório systems onde pode ser visto na figura 2.5 é constituídos por vários cores que se encontram no repositório cores. Cada core pode depender ou não de um ou mais cores. Um core descreve como é um elemento, como processador ou periférico. Um sistemas descreve como esses cores estão interligados entre si. Tornando assim a criação de novos sistemas que podem usar o mesmo cores que outro sistema, não sendo necessário ter duas copias do mesmo core e sendo mais fácil manter os cores actualizados.

Na figura 2.6 representa o sistema de ficheiros existente no ORPSoC. Encontra-se dividido por utilidades ao nivel da do ORPSoC encontram-se ficheiros que disponiblizam ferramentas básicas, na repositório Build encontram-se ficheiros com ferramentas para a sintetização do SoC para a placa, no Simulator tem um ficheiro para cada tipo de simulador dentro de cada tipo temos o seu precedimentos necessários e no Provider destina-se a descarregar os cores necessários para o SoC que a sua descrição não se encontra localmente.

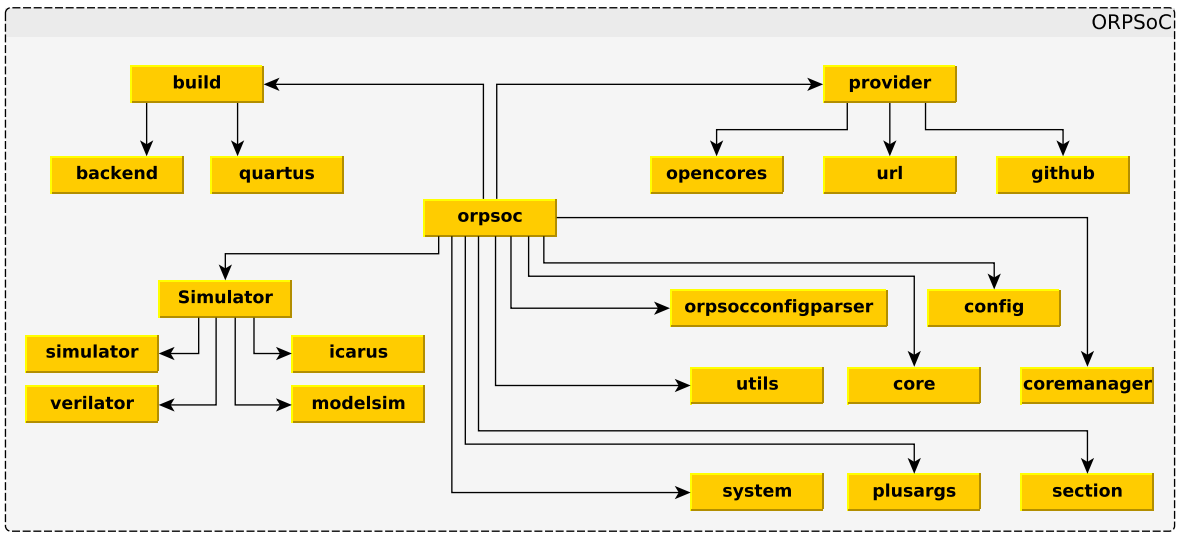


Figura 2.6: Diagrama de ficheiros do ORPSoC.

Durante o desenvolvimento desta tese a comunidade OpenRisc ponderou que esta ferramenta poderia ser utilizada no desenvolvimento de outros SoC sem ser especificamente os seus sistemas. Então a ferramenta tornou-se independente da comunidade e alteram o seu nome para FuseSoC.

2.2 Ferramentas

No desenvolvimento de SoC e de software quando o SoC ainda não se encontra fisicamente criado, é importante ter disponível várias ferramentas de desenvolvimento pois tornasse bastante dispendioso e demorado criar um SoC para desenvolver.

2.2.1 Or1ksim

Trata-se de um simulador de um simples SoC da arquitectara OpenRisc 1000, este é desenvolvido em C. Pretende-se que sejas autônomo, que permita umas simulação rápida permitindo analisar o código e avaliação de desempenho do SoC, que seja de fácil configuração de diferentes ambientes alterando o processador, alteração do tamanho das memorias e adicionando novos periféricos e permitir a utilização do debugger remoto. Porem tem a desvantagem do que se está a simular não ser exatamente o que se tem no sistemas descrito e implica quando se faz uma alteração no SoC esta alteração tenha de ser feita também no or1ksim. Mas é optimo para testar código em desenvolvimento por ser bastante rápido a executar.

2.2.2 Verilator

O verilator é uma ferramenta que converte o código que descreve o SoC em verilog e converte para objectos em C++ ou systemC. Esses objectos necessitam de ser linkados à testbench que pode ser escrita em C ou systemC. A testbench controla o sinal de clock, podendo também excitar os sinais de entrada e efectuar leitura tando nos sinais de saída como em qualquer final dentro do SoC, como pode ser visto na figura 2.7. O Verilator é um simulador de dois estados (0,1), é bastante mais lento que o 2.2.1 mas pode disponibilizar um ficheiro que permite visualizar o estado de todos os sinal do SoC a todos os instante da simulação, permitindo assim encontrar erros no hardware. Como é efectuado uma conversam do SoC a simulação efetuada é sobre o SoC desenvolvido.

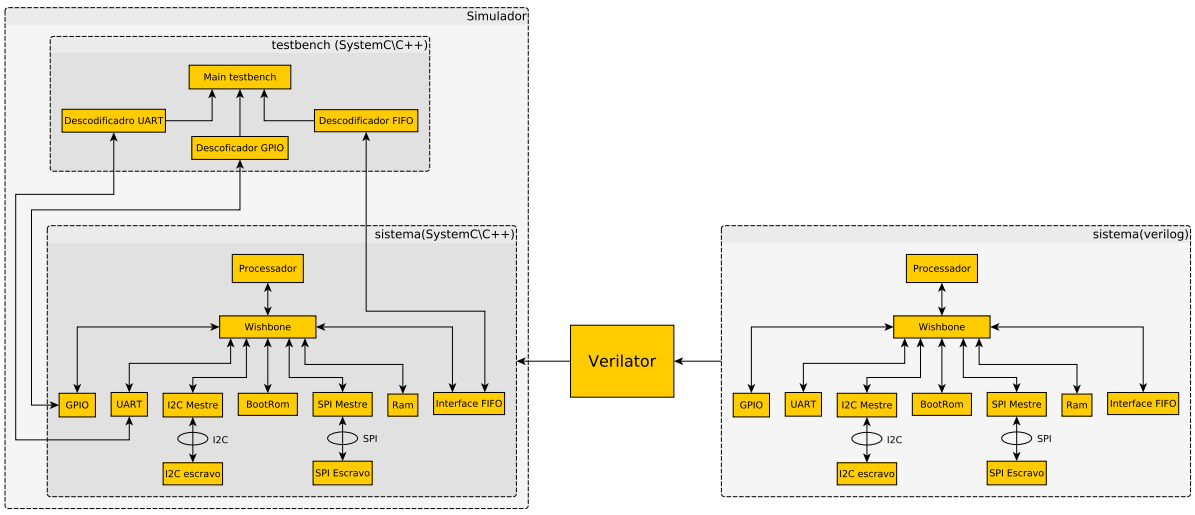


Figura 2.7: Diagrama do simulador verilator.

2.2.3 Icarus

Icarus Verilog conhecido por apenas Icarus é uma ferramenta de simulação e de sintetização de verilog. Funciona como um compilador, compila o codigo fonte me verilog na norma(IEEE-1364) e executa a simulação. Como de pode perceber pela figura 2.8 tal como o 2.2.2 também necessita de um testbench mas este em verilog. O Icarus é um simulador de três estado, alem dos dois valores lógicos (0,1) também simula o estado de alta impedância, sendo ainda mais próximo da realidade, tal como o 2.2.2 também disponibiliza o ficheiro do estado dos sinais do SoC. Mas o simulador Icarus é bastante mais lento que o 2.2.2.

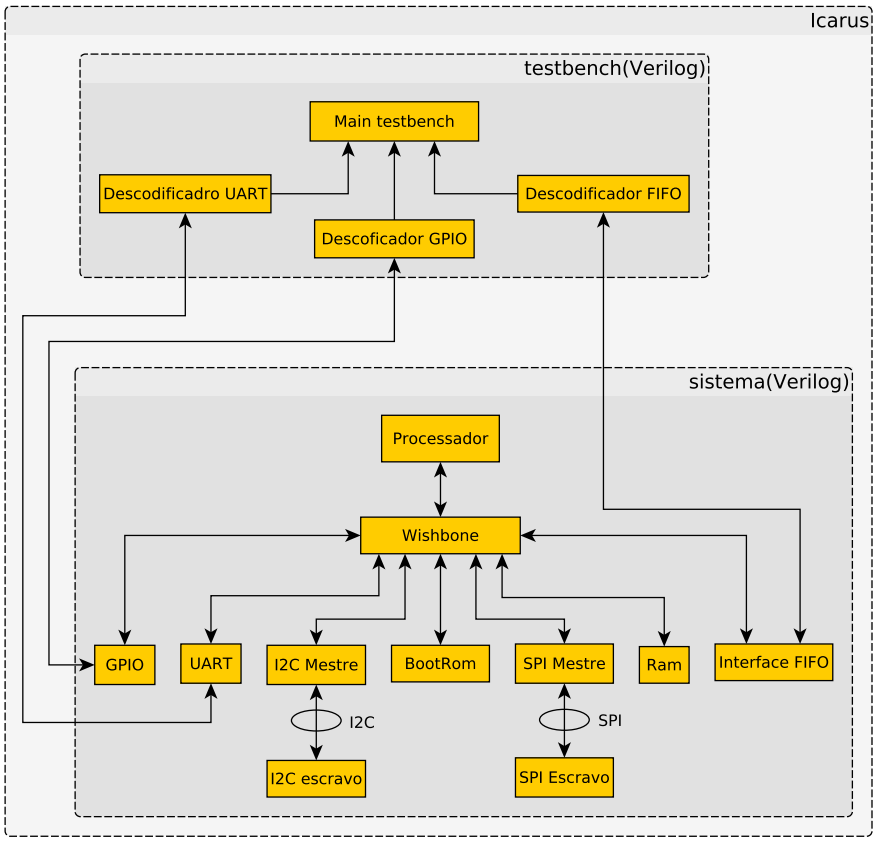


Figura 2.8: Diagrama do simulador icarus.

2.2.4 Placa com processador FPGA

O processadores *Field Programmable Gate Array* são dispositivos semicondutores que são baseados numa matriz de blocos lógicos configurável. As FPGA podem ser reprogramáveis após a fabricação para requisitos e funcionalidade distintas. permitindo programar recursos e funções, adaptar-se a novas normas, e reconfigurar hardware, mesmo depois do procuro estar aplicado instalado no local.

O primeiro dispositivo lógico programável foi a memória PROM(Programmable read only memory) sendo tanto programável na fabricação ou pelo utilizador, de onde evoluído o chip FPGA. Em 1985 a Xilinx desenvolve o primeiro chip em que é possível programar os blocos de logica como a interligação entre elas. Em 1987 foi proposto uma ideia de criar um novo chip que utilizava uma nova tecnologia

de matrizes de blocos programáveis usando software. a experiencia tinha 2 objetivos determinar uma forma de interligar os planos de matrizes e desenvolver um compilador capaz de programar funções para este chip.

Como foi descrito em cima e se pode ver na figura 2.9 o sintetizador é utilizado como um compilador que com toda a descrição do hardware converte essa descrição em um ficheiro de programação para essa FPGA. Para se programar a FPGA é utilizado um software disponibilizado pela marca da FPGA. Assim que é programado o hardware começa em funcionamento. Com um processador FPGA o hardware que estamos a testar é igual não existindo qualquer tipo diferença no código da descrição de hardware para desenvolver o hardware fisicamente, é o mais rápido dos simuladores mesmo que o próprio 2.2.1, não tem a possibilidade de se visualizar o estado dos sinais. Comparando com os outros sistemas de simulação descritos em cima 2.2.1, 2.2.2, 2.2.3 utilizar uma placa de FPGA seria o ideal sendo o mais rápido e o mais semelhante em hardware, mas tem o se não é bastante caro comprar uma placa de FPGA. Então utiliza-se cada uma destas ferramenta em situações diferentes o 2.2.1 para o desenvolvimento de software, 2.2.2 para testar o hardware e resolver alguns problema perspetiveis sem alta impedância, 2.2.3 para testar o hardware quando não é detectável com o 2.2.2, e a placa de FPGA para testar o hardware quando não é detectável com o 2.2.3 e os testes finais para se certificar o correcto funcionamento do do hardware com o software.

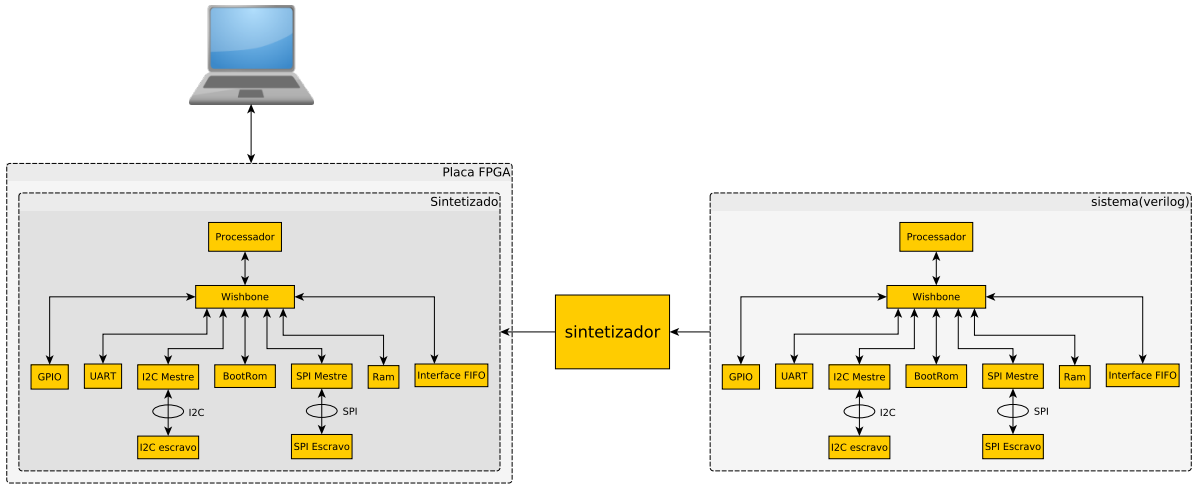


Figura 2.9: Diagrama do funcionamento da placa de FPGA.

2.2.5 OpenOCD

O *Open On-Chip Degugger* começou inicialmente por uma tese de mestrado onde se pretendia desenvolver e implementar uma solução de debugger para sistemas embebidos baseado na família ARM7 e ARM9. Atualmente existe uma comunidade a trabalhar no OpenOCD suportando novos SoC, adaptadores de debugger e flash programáveis. O OpenOCD tem como objectivo disponibilizar debugger, programação em sistemas e testes boundary-scan para dispositivos embebidos. Para isso o OpenOCD necessita dum adaptadores de debugger alguns deles já se encontram integrados nos dispositivos de desenvolvimentos, não existindo uma uniformização nem por essa razão é possível encontrar de vários

tipos, JTAG entre outros, muito delas já com suporte.

Sem utilizar o OpenOCD tinha-se de desenvolver uma testbench onde se tinha de incluir um servidor de *Remote Serial Protocol* que estaria ligado por JTAG ao SoC, e disponha de uma ligação com o protocolo TCP/IP para ligar um o cliente de debugger para se efetuar o debuguer. No arranque do OpenOCD é necessário identificar qual é a placa alvo e qual é a placa de interface que é utilizada para comunicar com a placa. Depois de se iniciar ficar disponível para ligação em tres portos cada um para o ser protocolo como se pode ver na figura 2.10, e ligando-se a placa de interface por USB. Vai processando as instruções que vai recebendo conforme o seu alvo e a placa de interface. Enviando-as por USB para a placa de interface que converte a informação recebida no procolo de comunicação neste caso JTAG, que é recebido pelo SoC pelo seu modelo de JTAG que acaba por enviar para o modelo de ADS. Assim o SoC não necessita de uma testbench onde estaria disponivel o servidor de RSP, o que este faria é feito pelo o OpenOCD.

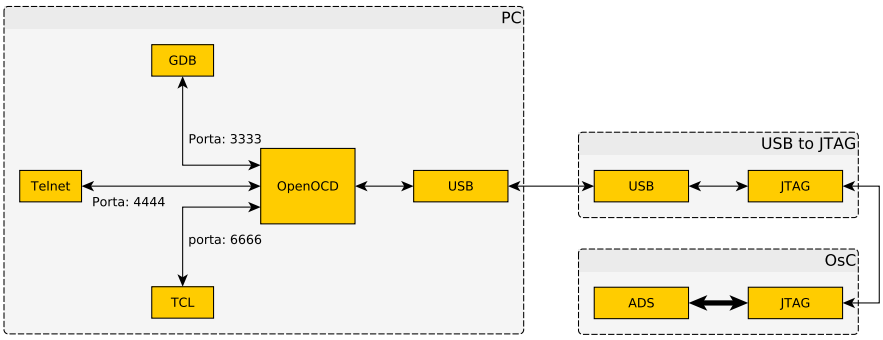


Figura 2.10: Diagrama do OpenOCD.

2.3 Periféricos

Os Periféricos são modelos com funcionalidades especificas que são acrescidas ao processador, estes comunicam utilizando um barramentos de comunicação neste caso a comunidade utiliza o barramento 2.1.2. Os periféricos são conectados ao barramentos como se pode ver na figura 2.3.

2.3.1 Bootrom

A bootrom é uma pequena memoria que tem um rutina de inicialização de sistema. A bootrom pode ser mais ou menos complexa podendo ser como por exemplo um *Basic Input Output System* de com computador ou um simples rotina que carregar o programa de uma memoria secundária para a memoria principal(RAM).

2.3.2 Uart

A UART é um modelo de comunicação assíncrono, o formato dos dados e a velocidade de transmissão de dados são configuráveis. Este normalmente fazem parte de um circuito integrado para comunicação

série com computadores ou dispositivos com porta série. O modelo de UART emissor recebe bytes de dados que os transmite sequencialmente cada bit, o modelo de UART receptor recebe bit sequenciais e agrupa-os numa byte. A comunicação pode ser simples em que cada modelo só pode fazer apenas de emissor ou receptor, Full duplex em que ambos os modelos podem receber dados ao mesmo tempo e half duplex em que os dois modelos podes receber e enviar mas intercalador.

Por motivos históricos a tinha encontra-se com o valor lógico '1' quando não se encontra uma comunicação a decorrer para mostrar que a linha não foi danificada. Como se pode ver na Figura 2.11, para o caso de envio de 8 bits, o emissor envia inicialmente um bit com valor lógico '0' para informação ao receptor que vai receber um conjunto de bits que pode ser configurável ente 5 a 9 bits, tipicamente é usado 8 bits, seguido sendo opcional de um bit de paridade, se não forem enviados 9 bits, por fim o bit de paragem que tem o valor lógico '1'. Existem vários modelos de chips UART sendo que a comunidade optou por utilizar no seu SoC a uart16550. O parametros bastante importante é o *Baud rate* ou seja a taxa de transferência o emissor e o receptor tem de estar em acordo se não a transferência não trabalhará corretamente.

Cada modelo de uart é feita por duas linhas TX linha de transmissão e RX linha recepção, a ligação entre os dois modelos é cruzado, ou seja liga-se a linha de TX do primeiro modelo ou RX do segundo modelo e o Tx do segundo modelo ou RX do primeiro modelo.

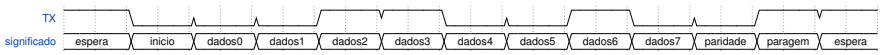


Figura 2.11: Diagrama temporal da UART

2.3.3 GPIO

O *General Purpose Input/Output* como o nome indica trata-se um de pinos de input e output de uso genérico, este pinos não têm uma função especifica. São apenas linhas disponíveis e controladas pelo utilizador em tempo de execução. A ideia é na construção de um SoC pode dar jeito ter disponível linhas de controlo digital e não necessitar de ter de organizar o chip por forma em as conseguir. Os pinos de GPIO podem ser activos ou desactivos, permitindo serem programados como input ou output. Os pinos que estão configurados por input permitem apenas leituras, normalmente de valores lógicos '0' ou '1', estes ainda por vezes são utilizados para criar interrupções no processador. Os pinos em output permitem leituras e escritas.

Os GPIO tem algumas aplicações tipicas como SoC pelas sua escassez de pinos disponíveis, portas multifunções como em gestão de energia, codecs de audio e placas de video e por aplicações em sistemas embebidos para comunicação com sensores, LCD's, LED's para estados. Alguns exemplos mais práticos dos pinos GPIO são alguns computadores portáteis da ACER usar o GPIO0 para ligar o amplificar da colunas internas, o chip Realtek ALC260 tem disponivel 8 pinos de GPIO que não são utilizados por omissão.

2.3.4 FIFO

A interface *Frist In First Out* destina-se a interligar um modelo assíncrono ao SoC com comunicação paralela de 32 bits por palavra, queria-se com esta interface que a comunicação seja rápida entre os dois modelos. Para uma comunicação paralelas com modelos assíncronos é utilizada uma memoria do tipo FIFO para cada sentido, ou seja uma FIFO para o SoC escrever e o modelo ler e outra para o oposto. Estas FIFO's ficaram ligadas a interface que 2.1.2 permitindo uma rápida comunicação com o processador. Para alem disso será possível efetuar ligação da FIFO de recepção ao interrupções, permitindo assim efetuar uma sub-rotina de interrupção permitindo assim que a resposta do processador seja mais ainda mais rápida e que se possa por em modos de poupança de energia.

2.3.5 I2C

O *Inter-Integrated Circuit* é um barramento desenvolvido pela Philips semicondutores agora conhecida como NXP semicondutores, é uma barramento serie, síncrono, multi-mestres, multi-escravos. Desenvolvido para conectar periféricos de baixa velocidade a computadores, sistemas embebidos e a telefones celulares. Este barramento utiliza apenas duas linhas de comunicação *Serial Data* para o envio de dados e *Serial Clock* sinal de clock que é controlado pelo o mestre, estas duas linhas são de dreno aberto. O I2C define tipo simples de comunicação o mestre lé dados do escravo, o mestre escreve dados para o escravo e combinado onde o mestre faz pelo menos duas leituras e/ou escritas, estas comunicação começam sempre por um sinal de começo(START) e acabando por um sinal de paragem(STOP). Todos os escravos têm um endereço de identificação, que é utilizado para diferencias com qual dos escravos o mestre quer comunicar.

Este barramento desde de Outubro de 2006 não necessita de licenciamento para ser utilizado. Tendo cada vez mais utilizações praticas como acesos a pequenas memorias de sistemas embebidos, comunicação com conversores digitai para analógico e analógico para digital, controlo pequeno monitores como para telemóveis são alguns dos exemplos.

2.3.6 SPI

O *Serial Peripheral Interface* é um barramento serie, síncrono, que permite apenas um único mestres e opera em *full-duplex*(ou seja permite receber e enviar dados ao mesmo tempo). O barramento utiliza pelo menos 4 linhas, isto porque necessita de uma linha para cada escravo, utiliza uma linha de clock comum a todos os escravos *Serial Clock*, tem duas linhas para dados uma que envia do mestre para o escravo *Master Out Slave In* e outra que envia dados do escravo para o mestre *Master In Slave Out*, por ultimo tem a linha que seleciona o escravo com que se encontra a comunicar *Slave Select* existem tantas linhas deste tipo no mestre como escravo estão ligados ao barramento sendo cada linha para cada escravo, esta linha é *active low* significa que o escravo só se encontra selecionado quando este linha se encontra num valor lógico '0'. Este barramento é utilizado para pequenas distancia em que o tempo é importante, como sistemas embebidos, sensores, lentes da Canon, LCD cartões SD.

ve se queres por aqui mais coisas.

Capítulo 3

SPI

O nosso SoC tem disponível uma memoria principal voatil ou seja sempre que a memoria é desligada da fonte de alimentação perde a informação contida nela. Então é necessário ter disponível no SoC uma outra memoria que não seja voatil como um cartão *Security Digital*, uma EEPROM, para se poder guardar o programa, que será carregado para a memoria principal sempre que o SoC for inicializado ou seja alimentado. Optou-se pela utilização de um cartão SD por permitir uma fácil substituição em qualquer local que esteja aplicado o SoC a trabalhar, estando planeado que estagia a trabalhar em locais de difíceis acessos. O cartão SD é possível ligar-se por barramento SPI, como se pode ver na tabela 3.1 e a correspondência entre os pinos na figura 3.1 que se encontra ao lado.

melhorar a posição disto

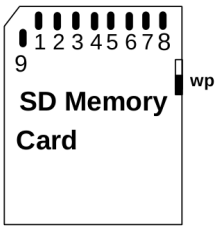


Figura 3.1: Cartao SD

Pinos SD	Nome(SD)	Nome (SPI)	Descrição
1	CS	SS	Selectiona o cartão
2	DI	MOSI	Entrada de dados
3	VSS	–	Tensão de alimentação(massa)
4	VDD	–	Tensão de alimentação
5	SCLK	SCLK	Clock
6	VSS	–	Tensão de alimentação(massa)
7	DO	MISO	Saida de dados
8	RSV	–	Reservado
9	RSV	–	Reservado

Tabela 3.1: Relação pinos do cartão SD SPI

Como foi dito 2.3.6 o SPI é um barramento de comunicação utilizado em casos onde é necessário uma comunicação com a velocidade de transferências elevada, como é neste caso. O barramento é constituído por 3 linhas de comunicação mais uma por cada escravo que esteja ligado a mestre como se pode ver na figura 3.2, neste caso como tem três escravos têm 3 linhas mais 3 linhas de selecção de escravo. A linha que seleciona o escravo é a linha SS esta é seleciona o escravo quando tem um valor lógico de '0'. Apenas o escravo que está selecionado lé e/ou escreve no barramento, como se trata de barramento full-duplex têm disponível uma linha de leitura para o escravo MOSI e outra de escrita

do escravo MISO. Ainda existe mais uma linha de sincronização com um sinal de clock SCLK. Todas as linhas do barramentos com as excepção da SS são partilhadas por todos os escravos, a linha SS como se trata de uma linha de selecção de escravo cada escravo tem a sua linha independente. Para se efetuar comunicação o mestre SPI seleciona com qual escravo quer comunicar, dependendo do escravo com que está a comunicar enviar uma ou mais palavras de 8 bits na linha MOSI sincronizado com o sinal de clock que se encontra na linha SCLK, posteriormente o mestre necessita de transmitir mais sinais de clock para o escravo, para este poder transmitir os dados pedidos pelos mestres na linha MISO.

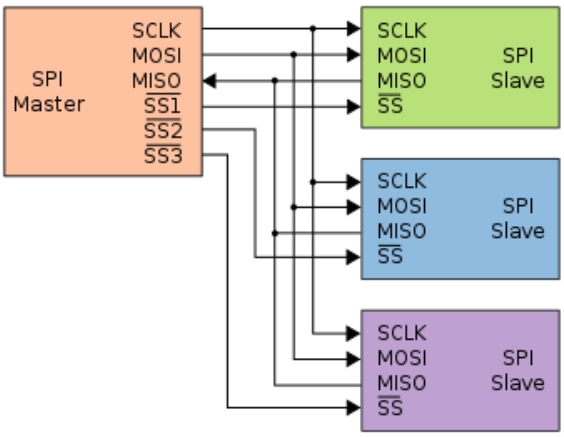


Figura 3.2: Esquematico do sistema de comunicação SPI, com 1 master e vários escravos"tirou do wiki".

3.1 Mestre SPI

A comunidade OpenRISC já tinha disponível um core de mestre SPI com uma interface Wishbone disponível para ligação ao SoC. Porém este core apenas realizava corretamente escritas para o escravo, não efetuando qualquer leitura de dados enviada pelo escravo. Sendo uma parte indispensável para o objectivo que seria pretendido para o core de SPI, como foi dito em cima seria efetuar leituras de uma memoria de um cartão SD para efetuar o carregamento do programa para a memoria principal.

Na figura 3.3 pode ser visto o fluxo de dados dentro do core mestre de SPI. Quando se pretende enviar uma palavra para o escravo esta é recebida pelo cores vindo da interface Wishbone que se encontra ligada ao processador do SoC, é guardada na FIFO, com uma tamanho de 4 palavras, que contem as palavras que serão enviadas para o escravo, na figura tem o nome de FIFO IN. Quando o modelo que faz a serialização se encontra parado e tem disponível uma palavra para enviar na FIFO. O modelo vai buscar a palavra à FIFO e começa a gerar clocks ao mesmo tempo que serializa a palavra e a enviar bit a bit para o escravo pela linha MOSI.

Quando se pretende receber dados do escravo, o modelo de desserialização recebe essa informação e se a FIFO que recebe as palavras vindas do escravo, na figura com o nome de FIFO OUT, não estiver cheia, começa a gerar clock's. O modelo recebe bit a bit a palavra do escravo na linha MISO, quando receber a palavra toda deixa de gerar clocks e enviar a palavra para a FIFO. Em seguida a palavra será

enviada da FIFO para o processador pela interface Wishbone. Visto que se trata de um barramento full-duplex quando se encontra a enviar dados para o escravo, o mestre também lé a linha MISO e guarda a palavra da FIFO. No caso destes dados forem lixo é necessário ter em atenção se os dados não forem removidos da FIFO quando se for efectuar a primeira leitura podemos estar a ler lixo.

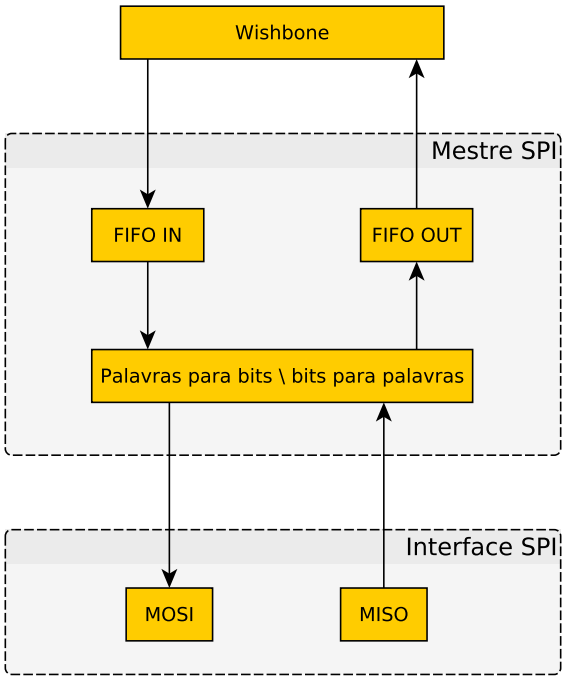


Figura 3.3: Fluxo dos dados dentro do core mestre SPI

Na tabela 3.2 temos todos os sinais do core mester de SPI com uma pequena descrição, os primeiros 10 sinais correspondem à interface Wishbone que é ligado ao multiplexer wishbone de dados á semelhança dos cores de uart e de gpio como se pode ver na figura 2.3. Os ultimos quatros sinais correspondem a barramento SPI para serem ligados ao escravo.

Interface	Nome	direcção	Descrição
Wishbone	clk_i	input	Clock, Recebido pelo Wishbone.
	rst_i	input	Reset, renicia o core quando se encontra no valor logic ”1”.
	cyc_i	input	
	stb_i	input	
	adr_i	input	Endereço do registo onde se pretende ler ou escrever no core.
	we_i	input	Bit de selecção de escrita.
	dat_i	input	Recepção de dados por parte do processador.
	dat_o	output	Envio de dados para o processador.
	ack_o	output	bit que informação o processador a recepção do comando pelo core.
	inta_o	output	bit de interrupção.
SPI	sck_o	output	Clock do protocolo de comunicação SPI.
	ss_o	output	Faz a selecção do escravo que se pretende ler ou escrever.
	mosi_o	output	envio de dados para o escravo (master Out Slave In).
	miso_i	input	recepecção de dados do escravo (master Out Slave In).

Tabela 3.2: Tabela de sinais da interface SPI master

A tabela 3.3 é uma descrição dos registos do mestre SPI com uma pequena descrição de cada registo e o seu Offset.

Nome	leitura/escrita	Descrição	Offset End.
Registo de controlo	escrita e leitura		0X00
Tipo de leitura	escrita	selectiona o tipo de leitura que é pretendida no SPI	0X01
Registo de estado	leitura	disponibliza várias informações do estado do core	0X01
Leitura de dados	leitura	leitura dos dados recebidos pelo SPI	0X02
Escrita de dados	escrita	escrita dos dados a enviar pelo SPI	0X02
Ext.registo de controlo	escrita e leitura		0X03
selecção do escravo	escrita e leitura	seleciona o escravo que se pretende escrever ou ler	0X04

Tabela 3.3: Tabela de registos da interface SPI master

O core mestre de SPI disponibiliza o modo de escrita que já se encontrava funcional, ficando a disponibilizar também o modo de leitura de dados que não se encontrava totalmente funcional. Esta leitura que a vou chamar de simples era feita apenas quando o era pedida pela processador ou seja, o processador pedia para ler uma palavra ao core, ai ele realizava a leitura no core. Este tipo de leitura perde-se bastante tempo porque o processador tinha de ficar a espera o core efectua-se a leitura ao escravo e que este o respondesse com a palavra, principalmente se pensarmos para que tipo de funcionalidade foi adicionado, para efetuar leitura sucessivas e posições de memorias sequenciais. Então pensei como poderia diminuir o tempo em que o processador tivesse a espera da resposta do core de forma transparante para ele. A solução encontrada é existir dois modos de leitura, um modo simples descrito em cima e outro modo em brust. Quando se efectua uma leitura é necessário identificar o tipo de leitura ao core, isto é feito no registo com o offset 0x01 como se pode ver na tabela 3.3 se escrevemos a palavra 0x02 é uma leitura simples, se for 0x01 é uma leitura brust. A principal diferença entre os dois modos de leitura é que o core na leitura simples vai pedir dados escravo após cada pedido pelo processador, na leitura em brust o core faz pedidos de dados sucessivos ao escravos até a FIFO OUT ficar cheia, ou seja não espera que o processador peça dados. Este tipo de leitura só mostra vantagem quando são leitura sucessivas de vários dados sucessivos como por exemplo a carregar o programa na memoria principal, como se pode ver nas figuras XXXX por imagem das ondas a comprar o tempo.

3.2 escravo SPI

dizer que foi desenhado desde o inicio.

por um diagrama de blocos de como circula a informação dentro do meu modelo de SPI. fazer uma descrição do funcionamento com uma descriçao do fluxo de dados.

por a tabela de sinais da interface de SPI (sinais de entrada e saida)

explicar o modo de funcionamento de esrita e de leitura.

explicar os 2 modos de funcionamento de esrita e de leitura que tem 2 modos.

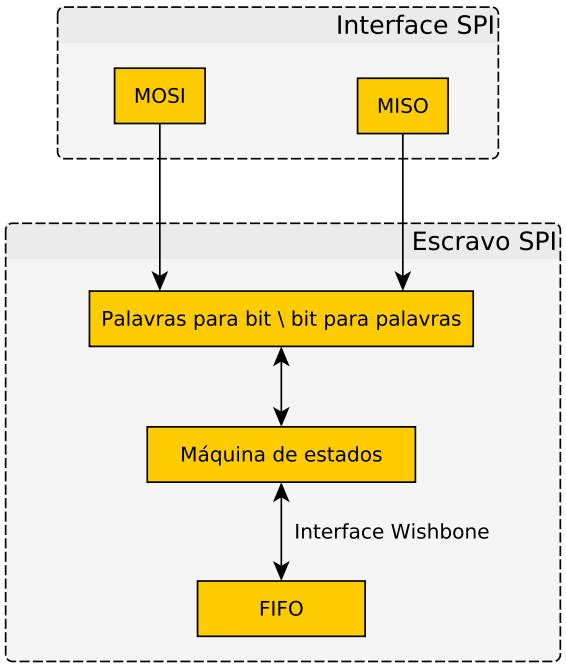


Figura 3.4: Fluxo dos dados dentro do core SPI escravo

Nome	direcção	Descrição
clk_i	input	Clock, Recebido pelo Wishbone.
rst_i	input	Reset, renicia o core quando se encontra no valor logic ”1”.
sck_o	input	Clock do protocolo de comunicação SPI.
ss_o	input	selectiona se é o escravo pretendido
mosi_o	input	recepecção de dados do master (masterOut SlaveIn).
miso_i	output	envio de dados para o master (masterOut SlaveIn).

Tabela 3.4: Tabela de sinais da interface SPI escravo

3.3 Diagramas temporais

Em seguida temos os diagramas temporais com uma breve explicação do que se encontra em cada diagrama.

3.3.1 Selecção de chip

Na figura 3.5 estão ligados dois escravos ao mestre de SPI, por essa razão existes dois sinais de SS cada um ligado apenas a uma escravo. Como foi dito em cima o sinal de SS é ativo em baixo, significa que o escravo está selecionado quando o seu sinal tem o valor lógico de ’0’. No primeiro flanco positivo da figura 3.5 não se encontra qualquer um dos escravos selecionado, o escravo um é selecionado no primeiro flanco negativo e assim se mantem até ao segundo flaco negativo, o segundo escravo é selecionado no terceiro flanco negativo e desselecionado no quarto flanco negativo. Apenas o segundo e o quarto flanco positivo têm um escravo selecionado, o primeiro e o segundo escravo correspondentemente.

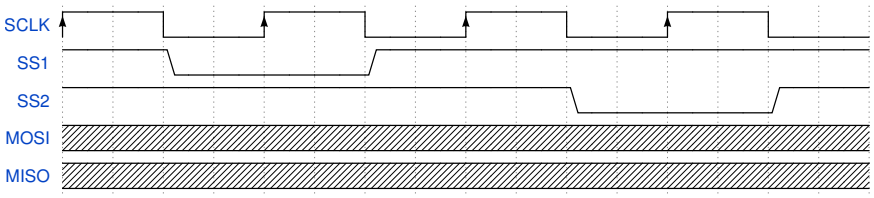


Figura 3.5: Diagrama temporal da selecção do escravo.

3.3.2 Dados

Os dados transmitidos tanto pelo mestre ou pelo escravo são lidos nos flancos positivo.

Envio de dados pelo mestre

No caso do envio dos dados do mestre para o escravo estes são feitos pelo sinal MOSI como se pode ver na figura 3.6. O escravo apenas vai guardar os dados que forem enviados depois de ser selectionado, que so acontece após o segundo flaco positivo. Após o escravo ser selecionado os dados são lidos no flanco positivo do sinal de SCLK, nesta figura por exemplo as palavras são de 6 bits, a palavra recebida pelo escravo no exemplo da figura é 110100 na base binária.

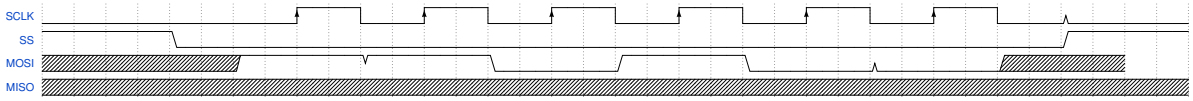


Figura 3.6: Diagrama temporal do envio de dados pelo mestre.

Envio de dados pelo escravo

Para o escravo enviar dados para o mestre, o mestre necessita de criar o sinal de clock em SCLK e o escravo só pode usar o sinal MISO quando se encontra selecionado. Como no envio de dados do mestre para o escravo, neste caso do envio de dados do escravo para o mestre os dados são lidos nos flancos positivos do sinal de clock gerado pelo mestre. A caso de exemplo a figura 3.7 a palavra envida pelo escarvo é de apenas 6 bits, neste exemplo a palavra transferida pela o escravo é 010110 na base binária.

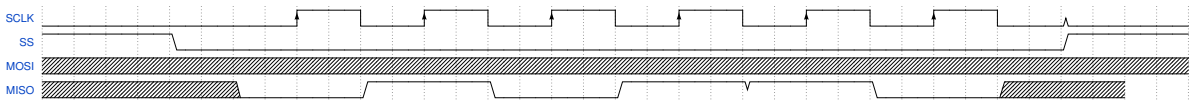


Figura 3.7: Diagrama temporal do envio de dados pelo escravo.

Capítulo 4

I2C

Explicar o protocolo i2c. como funciona e as linhas que tem. explicar em que acasos é mais usado

O SoC utilizam uma memoria de pequenas dimensões, não voatil, que permita guardar alguns dados em caso de algum problema com o SoC, como falta de energia. Permitindo assim que o SoC saiba que horas eram quando faltou a energia. Estas memorias utilizam para comunicar com o SoC o barramento de I2C. Éste barramento utiliza apenas duas linhas para comunicação, como é um barramento síncrono uma das linas é o sinal clock SCL, a outra linha SDA é utilizada para transferência de dados entre o mestre e o periférico. Apesar de ser um barramento relativamente lendo, comparado com o 3, de transferência de dados, é um barramento bastante utilizado pela sua simplicidades de adicionar mais periféricos aos já existentes, como se pode ver na figura **??**, para adicionar um periférico é necessário apenas ligar as duas linhas do barramento ao periféricos. Em vez de utiliza uma linha para selecionar o periférico com que quer comunicar como no 3, todos os periféricos tem o seu endereço de sete bits, em alguns periféricos permitem a alteração dos bits menos significativos. Como se pode ver na Figura XXXX o mestre envia um endereço de oito bits, sendo os sete bits mais significativos os que definem o escravo e o bit menos significativo é utilizado pelo mestre para informar ao escravo se pretende escrever, com o bit a zero, ou ler, com o bit a 1. Como existem apenas duas linhas de comunicação o barramento utiliza vários sinais interpretado pelos escravo e pelo mestre para poderem comunicar entre si, mais a frente no subcapítulo 4.3 serão explicados os vários sinais existentes. Cada operação começa com um sinal de start bit 4.3.1 seguido de 8 bits correspondente ao endereço do escravo que se pretende comunicar, com a informar se é uma leitura ou uma escrita. De seguida o escravo responde com um ACK 4.3.5. No caso de ser uma leitura em seguida o escravo envia os dados para o mestes e este responde com um ACK 4.3.5, este processo do escravo enviar dados e o mestre enviar ACK 4.3.5 é repetido continuamente até o mestre ter os dados necessários e responder com um NACK 4.3.6 seguido de um sinal de 4.3.2. No caso de ser um escrita a diferença para o anterior é quem envia os dados e os ACK 4.3.5, passado o mestre a enviar os dados e os escravo e enviar os ACK 4.3.5, e quando mestre já tiver enviado os dados todos este em vez de enviar dados envia um sinal de Stop 4.3.2, este processo pode se visto de maneira sucinta na figura XXXX.

por figura com o protocolo dos endereços

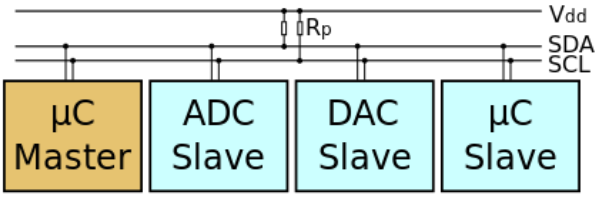


Figura 4.1: Esquematico do sistema de comunicação I2C, com 1 master e vários escravos"tirou do wiki".

4.1 Mestre I2C

A openRisc tem disponivel um core I2C mestre com uma ligação Wishbone para se ligar se ligar ao processdor do SoC. Apesar deste cores estar em desenvolvimento ainda não estava a trabalhar correctamente, tanto na leitura como na escrita de dados no escravo. A utilização deste barramento tem como objectivo poder guardar dados caso seja existe alguma problema com o SoC, por isso é necessário que este trabalhe correctamente na leitura como na escrita de dados.

O core mestre I2C está subdividido por várias camadas, como se pode ver na figura 4.2. O barramento Wishbone comunica directamente com a camada Master I2C top, que faz toda a gestão do protocolo I2C com o barramento Wishbone de entrada e saida de dados do core. Caso se pretenda efetuar uma escrita para o escravo o Master I2C top excita o I2C Byte e envia a palavra de 8 bits que se pretende enviar para o escravo. Neste nível da camada a palavra é subdividida e envida para a ultima camada I2C bit conforme esta for solicitando. Nesta ultima camada é gerada a onda de clock ao mesmo tempo que é enviado o bit para a linha de dados *Serial Data*. No barramento I2C ao contrario do barramento 3 não é possível efetuar uma leitura e escrita simultaneamente. Por isso corresponde a uma processo novo quando se pretende fazer uma leitura, começando por haver uma troca de informação entre camadas , da mais alta para a mais baixa, a indicar que se pretende fazer uma leitura. Quando a camada mais baixa recebe essas informação ela começa a gerar cloks que são recebidos pelo escravo, e este envia os dados que tem para enviar. Em simultâneo a camada I2C Byte fica em estado de prontidão a espera que a camada mais baixa I2C bit lhe envie um bit para ser guardado numa palavra de 8 bits. Quando são recebidos os 8 bits de dados do escravo a palavra é enviada para a camada mais alta Master I2C top, que por sua vez envia para o processador pelo barramento Wishbone.

O mestre I2C tem disponivel dois barramentos os Wishbone para transmitir dados para o processador e o barramento I2C para comunicação com os escravos. Todos os sinais estão descritos na tabela 4.1. Como pode ver na tabela a parte do barramento I2C tem 3 vezes mais sinais que o barramento de I2C explicado anteriormente. Isto deve-se ao por o core não ser tri-state ou seja não tem três estado, o estado 1, 0 e alta indução, então essa gestão é feita fora de cada core. Sendo as linhas de comunicação entre os dois cores o scl_pad_i é o sinal de clock e sda_pad_i para a transmissão de dados.

por a tabela de sinais da interface de i2c (sinais de entrada e saida)

Na gestão do barramento tri-state é feito foi duas linhas para cada sinal no caso do clock temos o sinal scl_pad_o, que não é mais que um sinal que tem sempre o valor lógico '0'. O outro sinal é o scl_padoen_o este sinal vai variando conforme como um clock comutando o sinal de clock scl_pad_i

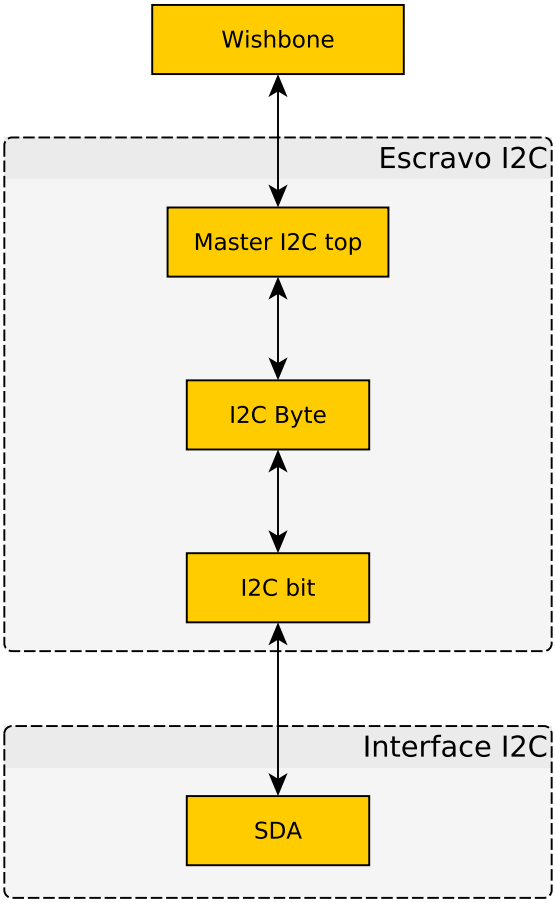


Figura 4.2: Fluxo de dados do core master I2C

Interface	Nome	direcção	Descrição
Wishbone	wb_clk_i	input	Clock, Recebido pelo Wishbone.
	wb_rst_i	input	Reset, renicia o core quando se encontra no valor logico ”1”.
	arst_i	input	Reset assincrono, renicia o core quando se encontra no valor logico ”1”.
	wb_adr_i	input	Endereço do registo onde se pretende ler ou escrever no core.
	wb_dat_i	input	Recepecção de dados por parte do processador.
	wb_dat_o	output	Envio de dados para o processador.
	wb_we_i	input	Bit de selecção de escrita.
	wb_stb_i	input	
	wb_cyc_i	input	
	wb_ack_o	output	bit que informaça o processador a recepecção do comando pelo core.
	wb_inta_o	output	bit de interrupção.
I2C	scl_pad_i	input	sinal de clock
	scl_pad_o	output	sinal de clock .
	scl_padoen_o	output	habilitador do sinal de clock .
	sda_pad_i	input	sinal de dados
	sda_pad_o	output	sinal de dados .
	sda_padoen_o	output	habilitador do sinal de dados .

Tabela 4.1: Tabela de sinais da interface I2C master

entre o valor lógico ’0’ e o valor de alta impedância. Aparentemente o sinal de clock estaria a varias entre o valor lógico ’0’ e o de alta impedância que não corresponde a nenhum estado logico possível

de determinar. Mas o sinal de clock encontra-se ligado por uma resistência a uma tensão igual ao valor lógico ’1’, como pode ver na figura XXXXXX. Então sempre que se encontrar no estado de alta impedância é como se tivesse no valor lógico ’1’, mas permitindo que ambos os cores coloquem o valor lógico ’0’ sempre que pretendam, bastando por o sinal de clock a ’0’. A lógica para o sinal de dados é igual ao de sinal de clock sendo o sinal sda_pad_o o que se encontra sempre no valor lógico ’0’ e o sinal sda_padoen_o vai variando conforme a informação que necessita enviar.

por uma imagens para explicar o tri-state XXXXXX

Na tabela 4.2 podemos ver os registos existentes no mestre I2C. Estes registos são utilizados para configurar o core mestre, como configurar a frequência de clock, ver o estado do core da reseccao e da transmição dos dadospara o core escravo.

tabela de registos (endereços do registos).

Nome	leitura/escrita	Descrição	Offset End.
Ajuste do clock	escrita e leitura	ajuste da frequência de clock 8bits menos significativos	0X00
Ajuste do clock	escrita e leitura	ajuste da frequência de clock 8bits mais significativos	0X01
Registo de controlo	escrita e leitura	disponibiliza várias informações do estado do core	0X02
Leitura de dados	leitura	leitura dos dados recebidos pelo I2C	0X03
Escrita de dados	escrita	escrita dos dados a enviar pelo I2C	0X03
Registo de estado	leitura	indica o estado do core	0X04
Registo de Debug	leitura	envio o que foi escrito pelo processador para o core.	0X05
Registo de estado	leitura	indica o estado do core, que comando se encontra a realizar.	0X06
Endereço do escravo	escrita e leitura	endereço do escravo onde que se encontra a escrever ou a ler.	0X07

Tabela 4.2: Tabela de registos da interface I2C master

O core mestre de I2C tem disponível o processo de leitura e escrita. No processo de escrita do mestre é necessário configurar o mestre antes, com a informação da frequência do sinal de clock do protocolo I2C. apos a configuração é necessário enviar para o mestre o endereço do escravo que pretendemos comunicar, este endereço já vem com o bit menos significativo com o valor logico 1 ou 0 conforme pretendemos escrever ou ler do escravo como foi explicado anteriormente. Por fim efetuamos uma leitura ou escrita ao endereço de Offset 0X03 o core mestre que irá enviar os dados recebidos para o escravo ou irá pedidos os dados ao escravo e erá enviar para o processador.

4.2 escravo I2C

A comunidade OpenRISC ainda não tinha desenvolvido nenhum core I2C escravo. No processo do seu desenvolvimento constantei que o core I2C mestre sem a parte que faz a gestão do barramento Wishbone, e com uma nova camada de alto nível que realizasse a gestão com uma memoria ou com outro tipo de periférico com wishbone, teríamos um core de I2C escravo. na construção desde cores foi utilizado as duas camadas mais baixas do mestres, ou seja os ficheiros I2C bit e o I2C byte. Para

o funcionamento foi preciso criar uma maquina de estados no nivel mais alto, que efectua o controlo entre o protocolo I2C e a memoria, como se pode ver na figura 4.3. O Fluxo de informação no core começa pelo pela camada mais baixa ou seja no I2C bit que recebe do sinal SDA e envia para a camada superior I2C byte onde agrupo os bits em palavras de 8 bits e envia para a maquina de estado onde está conforme o estado que esteja envia para a memoria essa informação para ser guardada do endereço definido, isto no caso que se esteja a efectuar uma escrita. No caso de uma leitura a maquina de estado recebe o endereço de leitura realiza uma leitura a memoria e envia para o I2C byte, onde vai separar a palavra de 8 bits por bits e vai ser sendo enviado conforme este for pedindo pelo I2C bit que vai enviado para o I2C mestre.

por uma imagem que mostra que tirei o codigo do mestre e criei o escravo.

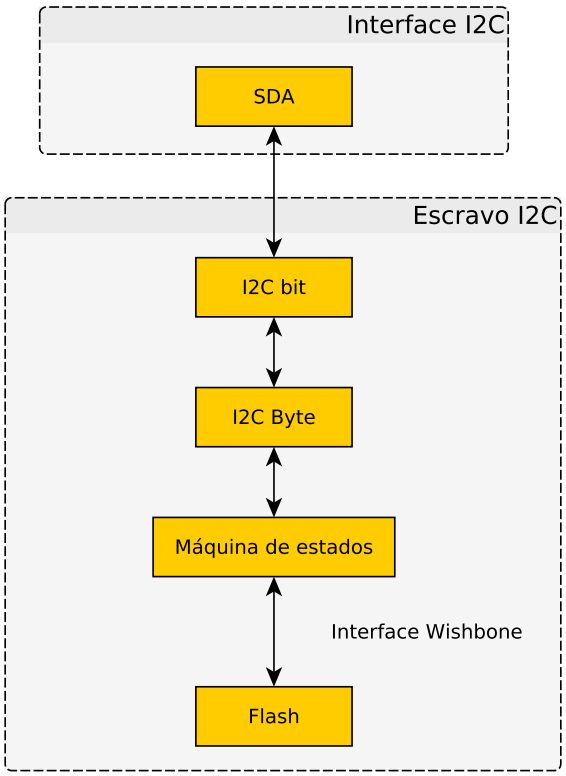


Figura 4.3: Fluxo de dados do core escravo I2C

Na tabela 4.3 tem uma breve descrição das ligações que o escravo tem disponível. O escravo conta com um conjunto de ligação de I2C iguais ao do mestre para controlar o as duas linhas da ligação de I2C. Para alem dessas ligaçoes conta com um sinal de Clock que é gerado pelo sistema, um sinal de reset que é utilizado para por as definições default do core e um sinal de interrupção que é utilizado se for pretendido receber informação do core, como por exemplo de algo correu mal do processo que está a decorrer.

O modo de funcionamento do core escravo de I2C, inicialmente permanece a escuta na linha dos dados e ficando a espera do Start bit. quando assim acontece o core fica a espera de receber o meu endereço enviado pelo mestre. Quando é recebido o seu endereço o escravo identifica pelo ultimo bit se o mestre pretende efetuar uma leitura ou uma escrita. Conforme esse precessão o escreve evolui

Nome	direcção	Descrição
clk_i	input	Clock, Recebido do sistema.
rst_i	input	Reset, renicia o core quando se encontra no valor logic ”1”.
scl_i	input	sinal de clock
scl_padoen_o	output	habilitador do sinal de clock .
scl_pad_o	output	sinal de clock .
sda_i	input	sinal de dados
sda_padoen_o	output	habilitador do sinal de dados .
sda_pad_o	output	sinal de dados .
irq_o	output	interrupcção .

Tabela 4.3: Tabela de sinais da interface I2C slave

para o estado correspondente dentro da maquina de estados interna. De seguida o mestre envia o 3 conjuntos de 8 bits que é o endereço onde se vai efetuar a escrita ou a leitura. No caso de uma escrita, o escravo fica a espera que o mestre gere sinais de clock efetua leitura na linha de dados quando tiver recebido os 8 bits da palavra responde com um ACK. Como este core escravo é uma memoria grava essa palavra no endereço da memoria especificado pelo mestre, e incrementa uma posição ao endereço. voltando a ficar a escuta de novos dados por parte do mestre, até que mestre envie o stop bit depois do escravo o ter enviado. No caso de ser de leitura depois de receber o endereço onde que o mestre pretende ler, efectua uma leitura a posição de memoria, e incrementa uma posição ao endereço de memoria. Ficando a espera que o mestre gere sinais de clock para poder a palavra lida da memoria, depois de enviar voltar ler e volta a ficar a espera que o mestre gere o sinal de clock. Até que o mestre responda com NACK e de seguinda envie um Stop Bit. O escravo responde ao mestre a todas as palavras recebidas com um ACK.

4.3 diagrama temporal

o protocomo de comunicação de I2C usa várias combinações temporais permitindo um intendimento entre o mestre e os vários escravos existentes no barramento de comunicação. resumindo a explicação feita anteriormente do protocolo de comunicação, qualquer comunicação entre os dois cores começa com o 4.3.1 e é finalizado sempre com um 4.3.2. Sempre que um core recebe oito bits tem de enviar um 4.3.5, o 4.3.6 é enviado automaticamente caso o cores não responda com o com o ACK. Visto que a linha de dados tem o valor logico ’1’ caso ninguém o force ao valor logico ’0’ como foi explicado anteriormente.

4.3.1 Start bit

O sinal Star bit utilizado desperta os cores escravos que vai começar a ser enviado um dados. O sinal de dados quando se encontras no valor de alta impedância o mestre força o valor lógico de ’1’ quando a onda de clock está no lado negativo da onda. Quando o clock está no lado positivo da onda o mestre força os dados a ’0’, até que o clock volte a estar no lado negativo da onda, como se pode ver na figura

4.4.

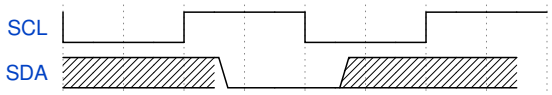


Figura 4.4: Bit de inicialização

4.3.2 Stop bit

O sinal Stop bit que informa o core escravo que toda a informação pretendida já foi enviada ou recebida. Quando a onda de clock se encontra no lado negativo o mestre forca o estado a onda de dados no valor lógico '0', quando o clock passa para o lado positivo da onda o mestre escreve no sinal de dados o valor lógico de '1'. ficando neste estado até que o sinal de clock esteja no lado negativo da onda. esta onda pdoe ser vista de força simplificada na figura 4.5.

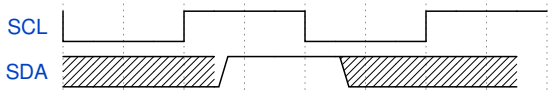


Figura 4.5: Bit de paragem

4.3.3 Restart bit

O sinal de restart bit não passa de um conjunto dos dois sinais anteriores, um sinal de stop bit seguido de um sinal de start bit, como se pode perceber na figura 4.6.

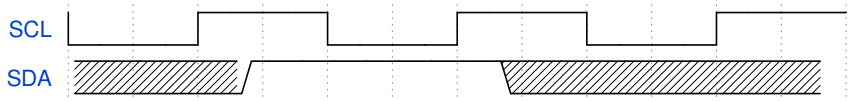


Figura 4.6: Bit de recomeço

4.3.4 Dados

Depois de se pode iniciar e parar uma comunicação é importante saber transmitir dados. Nos próximos dois capitulos temos uma breve explicação de como é transmitido um bit com um valor '1' e valor '0'. Quando se envia um bit, esse dados mantem-se no sinal de dados durante um ciclo de clock completo e se for necessario o valor do sinal dos dados é alterado quando o sinla de clock se encontra com o valor negativo.

Valor logico '1'

O core que quer enviar o bit com o valor 1 forca o sinal de dados ao valor lógico 1 quando o clock tem o valor negativo. Mantem esse valor durante um periodo de clock, libertando o sinal de dados para escrever outro dados ou para o outro core responder.

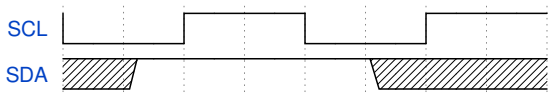


Figura 4.7: Bit de dados com valor logico ’1’

Valor logico ’0’

O core que está a enviar a informação quando quer enviar um bit a 0 força o sinal de dados SDA no valor lógico ’0’, durante um periodo de clock começando quando o clock tem um valor lógico baixo, seminhante ao caso anterior.

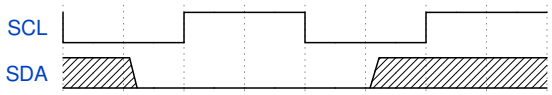


Figura 4.8: Bit de dados com valor logico ’0’

4.3.5 ACK

Sempre que o core que está a receber os dados recebe 8 bits de dados, este tem de responder que recebeu a informação com sucesso. Para isso o core utiliza o sinal ACK. o envio deste sinal é feito no ciclo de clock seguinte a ter recebido os 8 bits. Como se pode ver na figura YYYY o assim que é acaba de receber o ultimo bit o core que está a receber os dados força o sinal de dados SDA a um valor lógico ’0’.

POR FIGURA YYYY

4.3.6 NACK

Caso contrario se recessão de dados não foi efetuada com sucesso no sinal dos 8 bits o cores que está a receber força o sinal de dados SDA com o valor lógico ’1’.

POR FIGURA XXXX

Capítulo 5

Interface FIFO

Nas Acknowledge (ACK) comdições iniciais do projecto seria o systema construido ser capaz de comunicar com dois modelos que seria acrescentados posteriormente ao sistema. Os dois modelos que serão ligados poderam ter frequencia de clock totalmente dististas do sistemas construido. Os modelos que se pretendem ligar são modelos simples que não têm disponível um processador, por essas razão este modelos querem escrever para o sistema que estamos a desenvolver sempre que têm dados, e ler sempre que são imformados que têm dados de leitura. Por essa razão opetou-se por utilizar um sistema de comunicação simples e que trabalho-se de forma assíncrona. A comunidade Opensource ainda não tinha disponivel qualque cores desenvolvido ou mesmo começado. Por ser uma comunicação de forma simples, assincrona e onde era importante uma elevada de taxa de transferência de dados. Ponderou-se a utilização de duas memorias do tipo FIFO para cada modelo que se pretencia comunicar, onde seria utilizado uma fifo para cada sentido. Como pretendemos uma comunicação rápida entres o sistema e os modelos os dados serão enviados em paralelo permitindo assim uma redução bastante grande do tempo de envio de dados.

Como se pode ver na figura 5.1 este core tem disponível uma interface wishbone para comunicação com o processador do sistema e uma outra interface que utiliza 8 ligações de comunicação, sendo 2 delas de dados utilizando uma para cada sentido. As restantes linhas destinam-se para controlo informando o modelo lá ligado do estado do core.

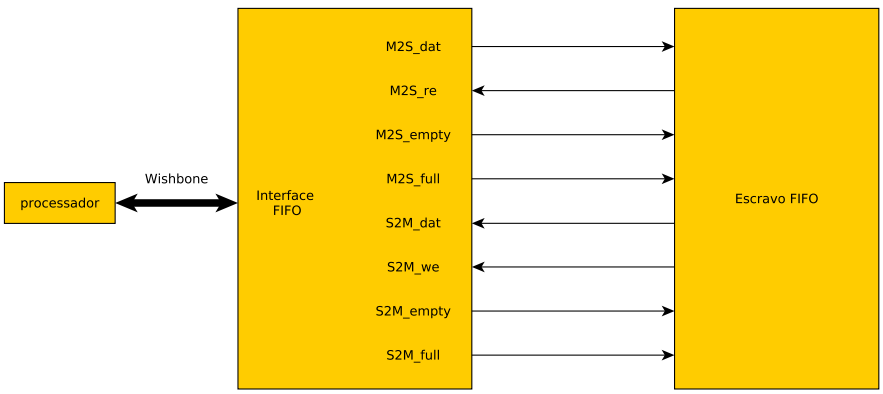


Figura 5.1: Esquematico do sistema de comunicação FIFO.

Como já foi dito antes o core utilizada duas memorias FIFO para a gestão de dados para cada sentido como se pode ver na figura 5.2. Os dados que são enviado pelo wishbone são guardados na FIFO até o modelo que se encontra do outro lado pedir para o ler. Apenas assim o dados é enviado da memoria FIFO identificada na figura por M2S_FIFO e envida para o modelo. Quando o modelo pretende enviar dados para o sistemas este envia os dados para a memoria FIFO na figura com o nome de S2M_FIFO, os dados seram guardado na memorias até o processador pedir mais dados desta interface. O processador antes de enviar ou ler dados da interface deverá ver os estados da memoria correspondente, para o caso de querer enviar dados de verificar se a memoria numerada na figura por M2S_FIFO não se encontra cheia, para não escrever em cima de dados validos. No caso de pretender ler deverá verificar se a memoria para esse efeito se têm dados, para não ler dados que não fazem sentido. Este procedimento também deverá ser feito pelos modelos que utilizam esta interface, para não haver problemas querencia.

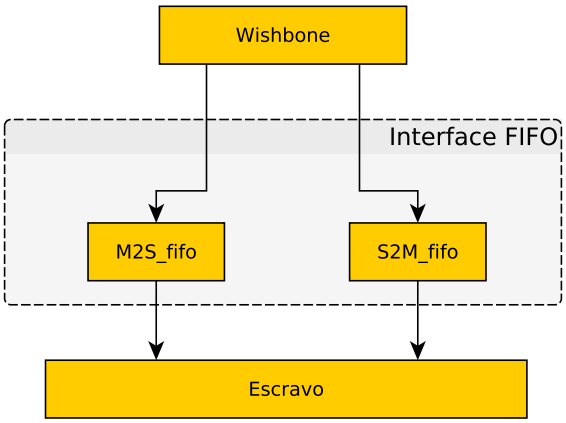


Figura 5.2: Fluxo de dados do core Interface FIFO

O core tem dois grandes grupos de linhas de comunicação como se pode ver na tabela 5.1, tem disponível todas as linhas de comunicação wishbone que se encontra ligada ao ao e a parte de ligações relacionada com o controlo de dados enviados e recebidos pelo outro modelo. O processador pela interface Wishbone tem acesso a informações importantes sobre o cores relacionadas com o estado das duas memorias fifo, permitindo assim este saber quando poder ler dados validos e quando pode enviar dados sem perder dados. O conjunto de ligações da interface FIFO, onde seram ligados os modelos com que se pretendem comunicar.

Como se pode ver na tabela as primeiras quatro ligações são utilizadas para a recepçao e gestão dos dados do sistemas que seram enviados para o outro modelo. As outro quatro ligação são utilizadas para o sentido oposto ou seja para a comunicação e gestão do dados enviados do modelos para para o sistemas que estamos a desenvolver. As linhas identificadas por o sufixo ”dat_i”e ”dat_o”destinasse as ligações para a troca de dados, uma para para entrada de dados e outra para o envio de dados respectivamento, como esta interface é paralela esta duas linhas representam um conjunto de ligações que será do tamanho igual ao tamanho das palavras de dados que pretendemos enviar. Ou seja se pretendemos enviar palavras de 32 bits cada uma destas linhas tem um tamanho de 32 ligações.As

linhas identificadas na figura pela sufixo empty e full, empty para vazia e full para cheia, são utilizadas para informar o outro modelo do estado das memorias FIFO's. A linha identificada na figura por M2S_re é utilizada pelo outro modelo para informar que está a ler a informação disponível da linha de dados. Por ultimo a linha na figura com o nome M2S_we informa que os dados que estão na linha de dados de leitura são dados validos para guardar na memoria FIFO.

Interface	Nome	direcção	Descrição
Wishbone	clk_i	input	Clock, Recebido pelo Wishbone.
	rst_i	input	Reset, renicia o core quando se encontra no valor logic "1".
	cyc_i	input	
	stb_i	input	
	adr_i	input	Endereço do registo onde se pretende ler ou escrever no core.
	we_i	input	Bit de selecção de escrita.
	dat_i	input	Recepecção de dados por parte do processador.
	dat_o	output	Envio de dados para o processador.
	ack_o	output	bit que informaça o processador a recepecção do comando pelo core.
FIFO	fifo_s2m_dat_i	input	sinal de entrada de dados do escravo para o mestre.
	fifo_s2m_we	input	sinal de escrita do escravo.
	fifo_s2m_empty	output	sinal da fifo que envia os dados do escravo para mestre se encontra vazi
	fifo_s2m_full	output	sinal da fifo que envia os dados do escravo para mestre se encontra chei
	fifo_m2s_dat_o	output	sinal de saida de dados do mestre para o escravo
	fifo_m2s_re	input	sinal de leitura do escravo.
	fifo_m2s_empty	output	sinal da fifo que envia os dados do mestre para escravo se encontra vazi
	fifo_m2s_full	output	sinal da fifo que envia os dados do mestre para escravo se encontra chei

Tabela 5.1: Tabela de sinais da interface FIFO

O core necessita de ter do lado do wishbone uma endereço como de pode ver na tabela 5.2 esta tem apenas 2 endereço. o primeiro com um OFFset de 0X00 sendo o endereço de escrita de de leitura do dados para enviar e enviados pelo modelo. O segundo endereço com o OFFset de 0X02 é um endereço apenas de leitura é onde contem os dados das FIFO.

Nome	leitura/escrita	Descrição	Offset End.
Leitura e escrita de dados	escrita e leitura	Envio ou rececção dos dados	0X00
Estado fifos	leitura	leitura dos estados das fifos	0X02

Tabela 5.2: Tabela de registos da interface FIFO

Sempre e por qualquer das duas interfaces deverá ser feitar uma verificação dos dados na fifo. No caso da leitura da memoria FIFO deverá verificar-se que a memorias não está vazia e no caso da escrita verificar que esta não está cheia. Evitando-se assim no primeiro caso a ler-se dados errados e no segundo caso estar-se a escrever em cima de dados importantes e estes serem perdidos.

Capítulo 6

Bootrom

Quando um sistema arranca ou seja é dada energia ao sistema, este faz resite a todos os cores até mesmo ao processador. Depois do resite o processador vai ao endereço que é atribuido pela pessoas que desenvolver o sistema e carrega e carrega essa informação para o registo para começar a desenvolver o processo que está descrito a partir dessa posição de memoria. A partir daqui existe duas grande hipoteses, a primeira hipotese é o programa estar estar carregado na memoria de instruções partida. Tendo apenas o endereço de restart ser o endereço da memoria de instruções e ser o endereço onde começara o programa, este hipotese tem a vantagem de ser mais simples e de ter um restart mais rápido, começando o programa a correr assim que o sistema recebe energia, mas tem a desvantagem de o programa não poder ser alterado facilmente. a segunda Hipoetese é ...

explicar o que é uma bootrom

o que se tem de fazer para correr a partir da bootrom.

explicar o que se pretende com a bootrom.

por um fluxograma do fluxo da bootrom

6.1 Testes de verificação

para que servem estes testes?

6.1.1 teste de leitura SPI

explicar como é feito o teste de leitura

como mostra o erro tanto com o GPIO como na uart.

6.1.2 teste de escrita na memoria principal

explicar como é feito o teste de escrita na memoria principal

como mostra o erro tanto com o GPIO como na uart.

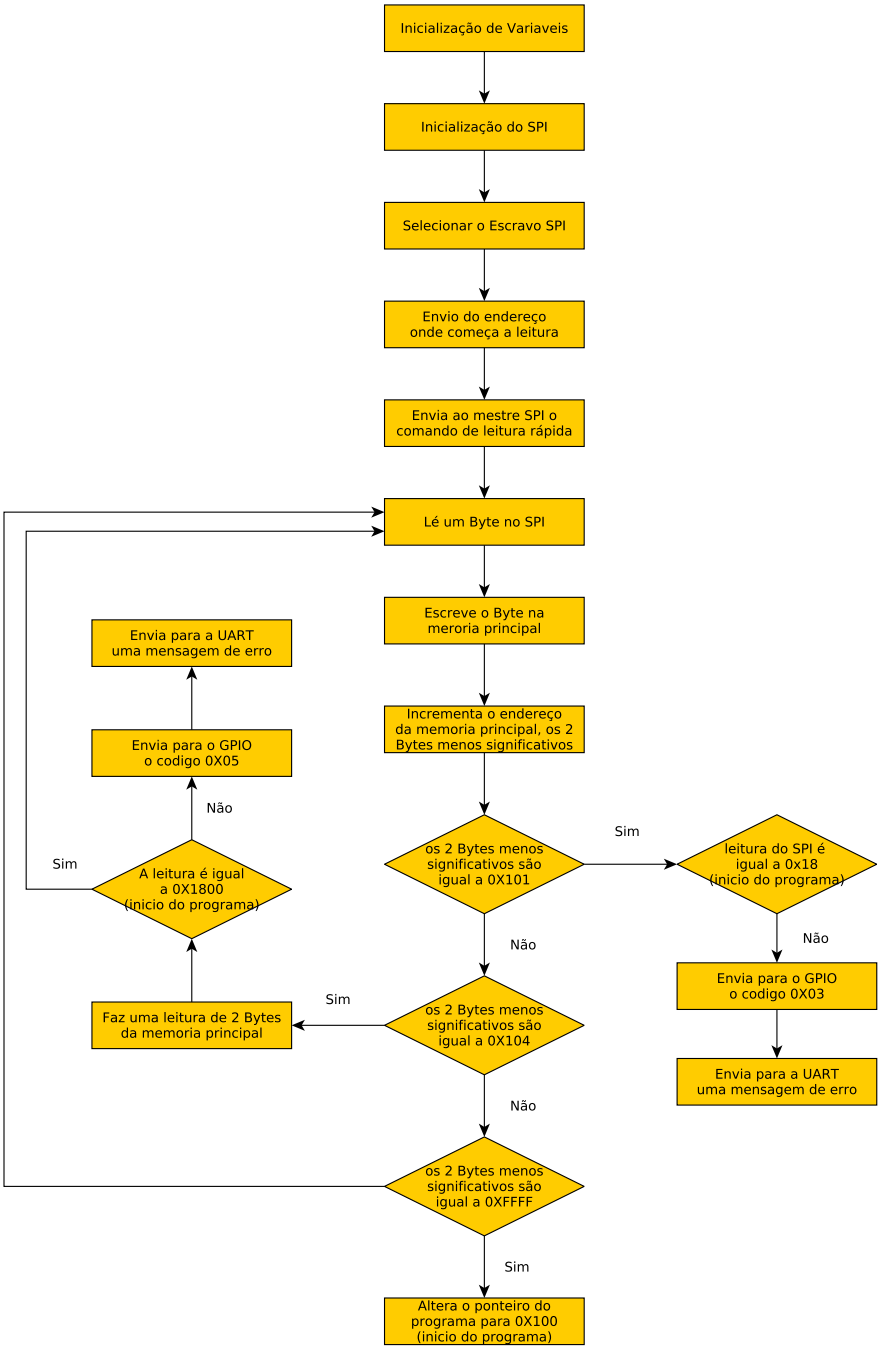


Figura 6.1: Fluxograma da Bootrom.

Capítulo 7

Testes de funcionamento

explicar para que quero os testes.

como foi implementado no Orpsoc.

adicionar uma imagem explicativa como foi implementado.

falar sobre as flags do teste

parametro	Descrição	Default
–test	Corre apenas o teste indicado	Corre todos os testes existentes na pasta.
–mode	Indica com que ferramenta se pretende fazer os testes (verilator,icarus,board).	Corre os teste com o verilator.
–folder	Selectionar uma pasta alternativa com os testes, a partir da raiz.	Selectiona a pasta ’./orpsoc/orpsoc/test/’.
system	Selecionar qual dos sistemas se pretende testar.	Parametro obrigatorio.

Tabela 7.1: Tabela com os parametros existentes na função de correr testes

como posso ver os resultados dos teste

7.1 adicionar novos testes

fazer novos testes.

7.2 Testes desenvolvidos

para mostrar que o sistema funciona corremante foi desenvolvido estes testes

7.2.1 SPI

explicar como é testado

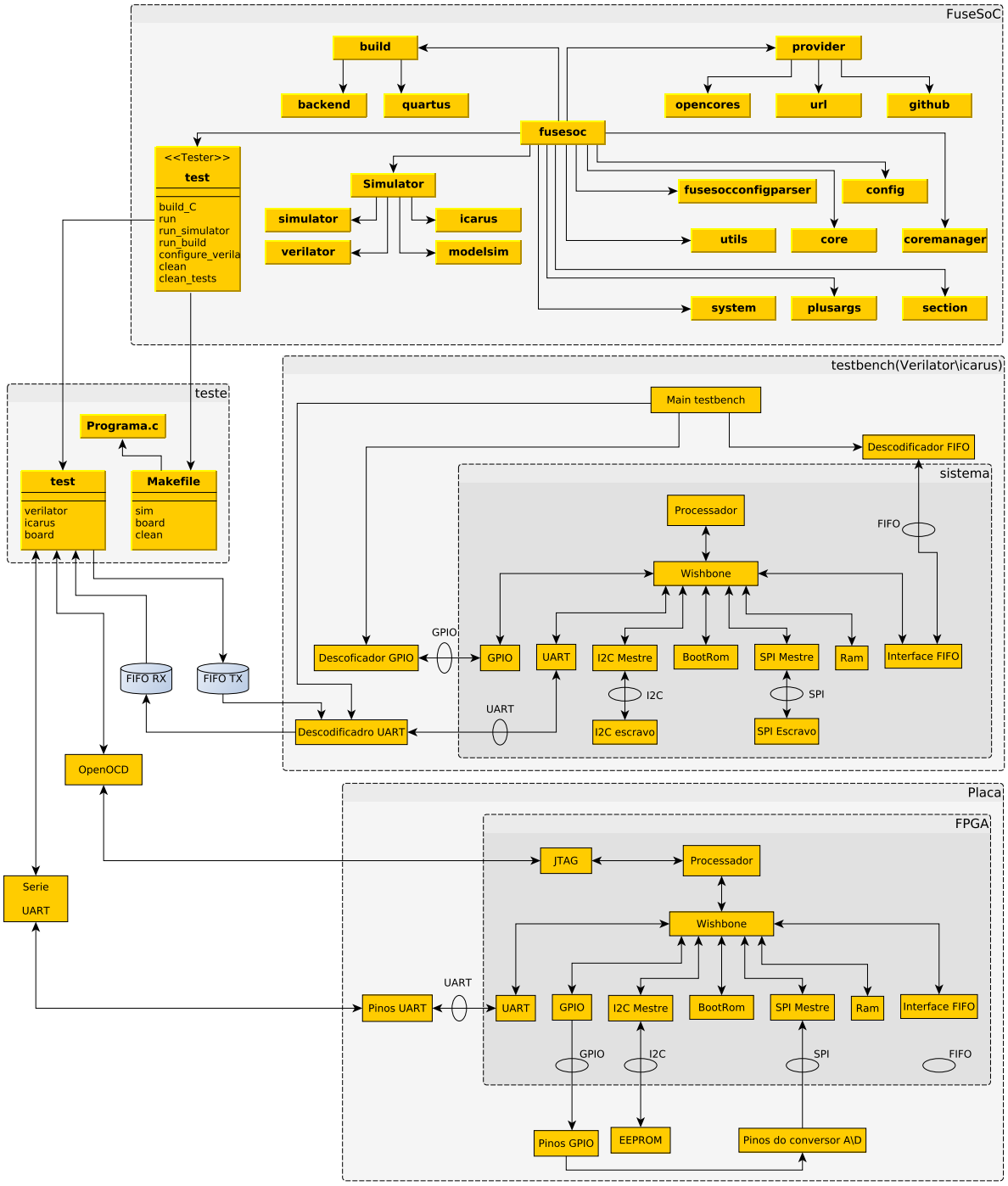


Figura 7.1: Diagrama da plataforma de testes usando o Fusesoc.

simulador verilator

Board FPGA

7.2.2 I2C

explicar como é testado

simulador verilator

Board FPGA

7.2.3 Interface FIFO

explicar como é testado

simulador verilator

Board FPGA

7.2.4 Bootrom

explicar como é testado

simulador verilator

Board FPGA

Capítulo 8

Resultados

8.1 Analise da área utilizado pelo systema

8.2 Analise da frequencia de trabalho do sistema

Capítulo 9

Conclusão

Insert your chapter material here...

9.1 Achievements

The major achievements of the present work...

9.2 Trabalho Futuro

desenvolver testes para os cores existentes no na Opencores.

Anexo A

Vector calculus

In case an appendix if deemed necessary, the document cannot exceed a total of 100 pages...

Some definitions and vector identities are listed in the section below.

A.1 Vector identities

$$\nabla \times (\nabla \phi) = 0 \tag{A.1}$$

$$\nabla \cdot (\nabla \times \mathbf{u}) = 0 \tag{A.2}$$

Bibliografia

Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.0, Argonne National Laboratory, 2004.

Antony Jameson, Niles A. Pierce, and Luigi Martinelli. Optimum aerodynamic design using the Navier–Stokes equations. In *Theoretical and Computational Fluid Dynamics*, volume 10, pages 213–237. Springer-Verlag GmbH, January 1998.

A. C. Marta, C. A. Mader, J. R. R. A. Martins, E. van der Weide, and J. J. Alonso. A methodology for the development of discrete adjoint solvers using automatic differentiation tools. *International Journal of Computational Fluid Dynamics*, 21(9–10):307–327, October 2007.

Andre C. Marta, Sriram Shankaran, D. Graham Holmes, and Alexander Stein. Development of adjoint solvers for engineering gradient-based turbomachinery design applications. In *Proceedings of the ASME Turbo Expo 2009: Power for Land, Sea and Air*, number GT2009-59297, June 2009.

Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, May 2004.

Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer, 1999.