Document information:

Project:			
Abstract:	Documentação e organização do Charrua		
Created on:	15/12/2015	**Created by**	Mauricio Carvalho
Modified on:		**Modified by**	
Scope:	[X] NSCAD	[ ] CI Brasil	[ ] Public

1 Introdução

Este relatório dá uma visão geral sobre o projeto do processador de 16-bits, denominado Charrua. Ele foi o benchmark escolhido como caso de estudo para avaliar a biblioteca de células CTC06ST. Relata-se neste relatório a arquitetura do circuito e seu funcionamento, a estrutura organizacional das pastas e arquivos inerentes ao seu desenvolvimento bem como a sua simulação (front-end e back-end), estratégia de testes adotada, dados sobre síntese lógica e física, compilador e programas assembly de teste.

2 Arquitetura

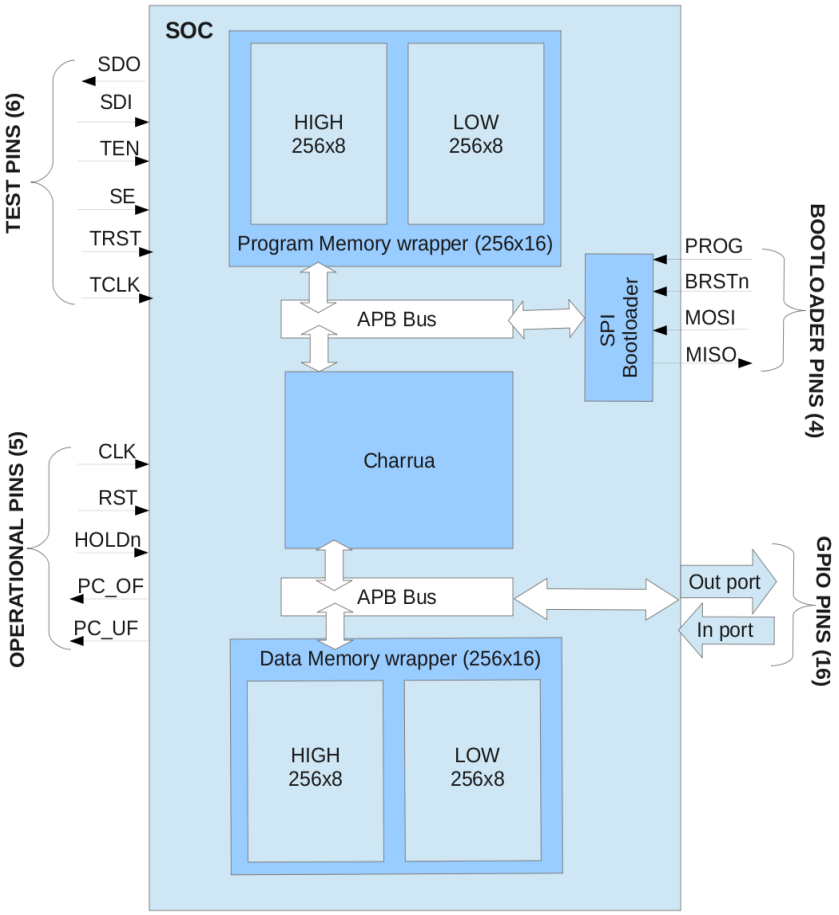

Figura 1: Arquitetura Top-level do Microprocessador

A arquitetura top-level ilustrada na figura 1 foi denominada de System-On-Chip (SOC) porque embarca o processador Charrua, o bootloader, a memória de instruções e também a de dados. Ambas as unidades de memória, são constituídas por duas instâncias da memória RAM fornecida pela XFAB entitulada Single Port Random Access Memory (SPRAM) que contém 256 palavras de 8 bits. Assim, utilizou-se um wrapper para unir as duas instâncias (HIGH e LOW) e apresentar ao processador como uma única unidade de memória de 256 x 16 bits, tanto para a memória de instruções (PMEM) como para a de dados (GMEM). A comunicação entre o processador Charrua (CPU) e as memórias dá-se através do bus com protocolo APB. Outro módulo importante para o funcionamento do SOC é o Serial Peripheral Interface (SPI) bootloader que possibilita carregar um programa na PMEM. Além disso, o SOC conta com 31 pinos que manipulam o processador e estão distribuídos em 4 grupos: Operacional, Bootloader, General Purpose Input/Output (GPIO) e de Testes. Esses grupos serão explicados em mais detalhes ao longo deste relatório.

A figura 2 mostra a arquitetura interna do Charrua. Nesta versão do processador Charrua, incluiu-se um registrador INDR para a implementação de ponteiros, ou seja, possibilita que um programa faça operações com os endereços da GMEM. Assim, consegue-se fazer operações com vetores utilizando poucas instruções. Além disso, adicionou-se ao processador dois blocos que verificam se Program Counter (PC) estrapolou os limites da memória PMEM, tanto para baixo (pc underflow) quanto para cima (pc overflow). Caso isso aconteça, uma das duas flags PC_OF ou PC_UF será setada pelo processador, indicando que a instrução não se encontra mais na PMEM. Desta maneira, um circuito externo consegue parar o processador (HOLDn=0) e re-programar a memória com novas instruções através do bootloader, permitindo executar programas maiores que 255 instruções que são limitados pelo tamanho físico da PMEM. Fez-se necessário incluir essas duas funcionalidades no Charrua para poder elaborar programas maiores tanto para testar a lógica interna quanto para testar as memórias.

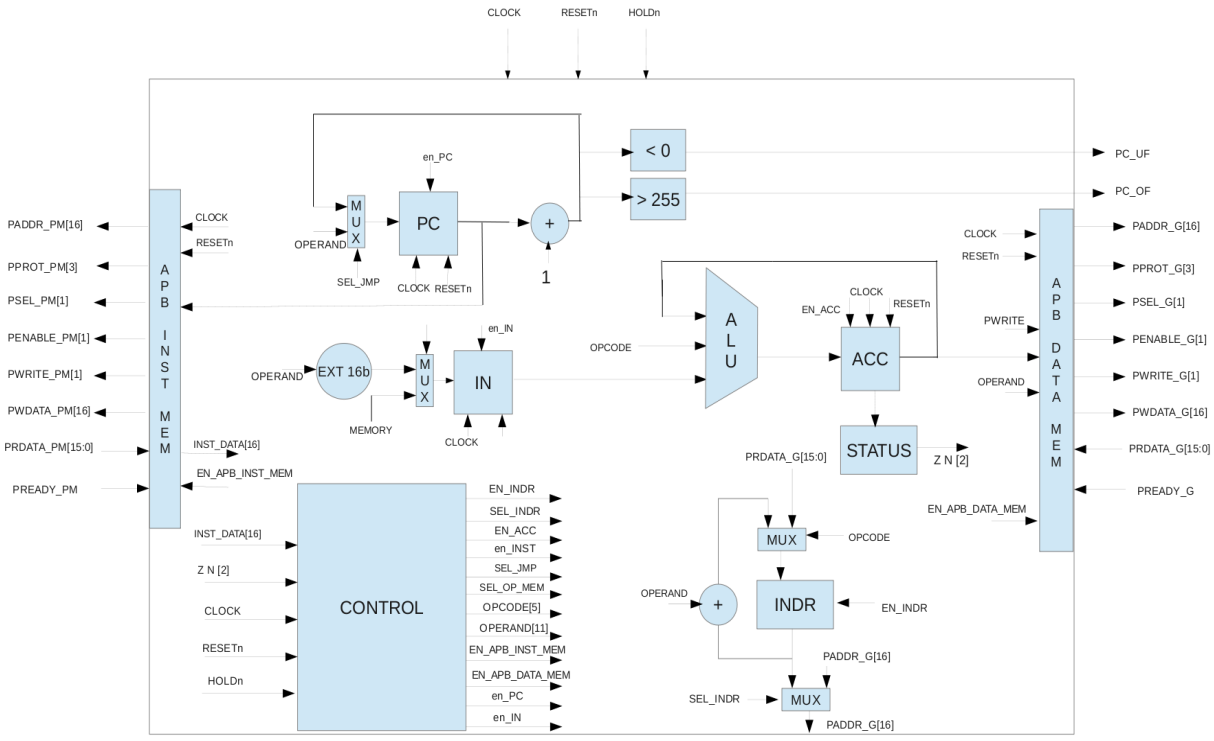

Figura 2: Arquitetura da CPU Charrua

3 Especificações Funcionais

3.1 Conjunto de Instruções

A Tabela I mostra o conjunto de instruções que o Charrua executa. Embora o processador tenha apenas essas instruções implementadas, ele teria a capacidade de executar mais instruções caso o circuito atual fosse minimamente modificando. As 26 instruções são simples e variam desde operações de memória (Load e Store) até gerenciamento de vetores. Além disso, modificou-se o Charrua original para que as instruções de saltos condicionais (BEQ até BLE) ocorram em relação ao PC, ou seja, o operando (em vermelho) contém a quantidade de instruções que se deseja saltar a partir do PC atual. Neste caso, o operando de salto representado em complemento 2 pode variar entre -255 instruções a +254 a partir do PC.

00001	STO	Memory[operand] ← ACC	Store
00010	LD	ACC ← Memory[operand]	Load
00011	LDI	ACC ← operand	Load
00100	ADD	ACC ← ACC + Memory[operand]	Arithmetic
00101	ADDI	ACC ← ACC + operand	Arithmetic
00110	SUB	ACC ← ACC - Memory[operand]	Arithmetic
00111	SUBI	ACC ← ACC - operand	Arithmetic
01000	BEQ	STATUS.Z ? PC ← PC+**operand** : PC ← PC + 1	Conditional Branch
01001	BNE	~STATUS.Z ? PC ← PC+**operand** : PC ← PC + 1	Conditional Branch
01010	BGT	(~STATUS.Z) && (~STATUS.N) ? PC ← PC+**operand** : PC ← PC + 1	Conditional Branch
01011	BGE	~STATUS.N ? PC ← PC+**operand** : PC ← PC + 1	Conditional Branch
01100	BLT	STATUS.N ? PC ← PC+**operand** : PC ← PC + 1	Conditional Branch
01101	BLE	(STATUS.Z) || (STATUS.N) ? PC ← PC+**operand** : PC ← PC + 1	Conditional Branch
01110	JMP	PC ← operand	Inconditional Branch
01111	NOT	ACC ← ~ACC	Logic
10000	AND	ACC ← ACC & Memory[operand]	Logic
10001	ANDI	ACC ← ACC & operand	Logic
10010	OR	ACC ← ACC | Memory[operand]	Logic
10011	ORI	ACC ← ACC | operand	Logic
10100	XOR	ACC ← ACC ^ Memory[operand]	Logic
10101	XORI	ACC ← ACC ^ operand	Logic
10110	SLL	ACC ← ACC << operand	Logic
10111	SRL	ACC ← ACC >> operand	Logic
11101	INDWR	INDR ← Memory[operand]	Vector Management
11000	STOV	$INDR_N$ ← INDR + operand; Memory[$INDR_N$] ← ACC	Vector Management
11001	LDV	$INDR_N$ ← INDR + operand; ACC ← Memory[$INDR_N$]	Vector Management

Tabela 1: Conjunto de Instruções do Charrua

3.2 Unidade de controle

A unidade de controle do Charrua contém uma máquina de estados (FSM) de 5 estágios, porém cada instrução necessita de 4 ciclos de clock para ser executada. A FSM executa na ordem pre-fetch, fetch, decode e execute ou idle.

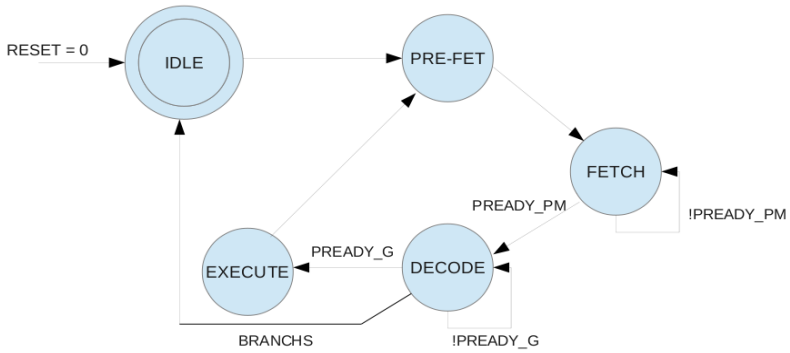

Figura 3: Máquina de Estados do Charrua

3.3 Programando o Charrua

Para programar o Charrua é necessário obedecer a comunicação SPI determinada abaixo conforme as formas de onda. O pinos que devem ser manipulados são os que pertencem aos grupos de pinos de operação e de bootloader como mostra na Figura 1.

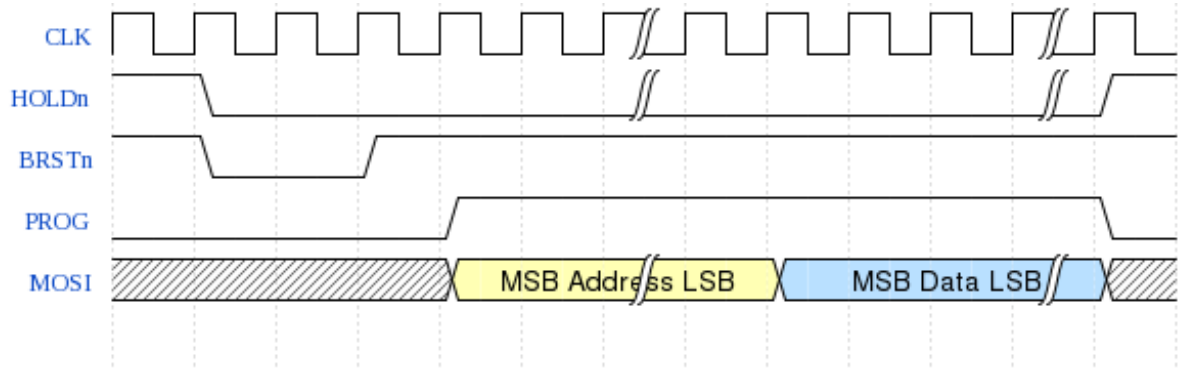

Figura 4: Programando a PMEM através do SPI bootloader

Primeiramente, é necessário parar o processador usando o HOLDn e resetar o bootloader através do BRSTn para sincronizar o início de uma transação de comunicação/programação. Então, seta-se o pino PROG e inicia-se a transação de 32 bits, sendo que os primeiros 16 bits são do endereço e os últimos 16 bits do dado que se deseja gravar na PMEM. É importante lembrar que os bits tanto do endereço como os do dado são inseridos no MOSI sempre iniciando pelo bit mais significativo (MSB). Isto quer dizer que Addr[15] vai primeiro e Addr[0] por último, assim como o de dado: Data[15] primeiro e Data[0] o último. Ainda, para facilitar a programação, consegue-se fazer uma gravação do tipo BURST na PMEM. Isto é, consegue-se gravar 255 instruções sequencialmente, sempre intercalando endereço e dado, como mostra na Figura 4 e mantendo PROG ativo até o último dado a ser escrito, sem a necessidade de resetar o bootloader a cada dado escrito.

Durante a programação, o clock pode ter uma frequência de até 3x mais que a frequência nominal de operação estipulada em 5MHz. Isso pode ser útil caso o programa que está sendo executado contém mais de 255 instruções e novas instruções precisam ser inseridas rapidamente na memória.

4 Pastas e Arquivos

|-**digital_database**
|---gate {netlist_charrua.v, netlist_charrua_scan.v, scandef/charrua.scandef}
|---rtl {SoC.v, charrua.v, control_v2.v, alu.v, apb, apb_mc_no_clock.v, datapath.v}
|-----bootloader {bootloader.v, interface.v, spi.v}
|-**digital_workflow**
|---ATPG
|---logical_synth
|-----lec {charrua_{scan}_final_lec.do, charrua_{scan}_lec.do , charrua_{scan}_lec.v}
|-----reports {area.rpt, timing.rpt, gates.rpt, power.rpt, sequential.rpt, messages.rpt, summary.rpt }
|-----scripts {synth.sh, sort_chain.sh, adjust.awk, put_scan.sh, 1_RTL2Net_noscan.tcl, 2_Net2Net_wscan.tcl}
|-----cmd
|-----log
|---physical_synth
|-----encounter
|----data {charrua.conf, charrua.io, charrua.view, cts.ctstch}
|----logs
|----reports
|----scripts{floorplan.tcl, fixing.tcl, placement.tcl, qrc.tcl, route.tcl, validate_timing.tcl, cts.tcl, config.tcl}
|----work
|---{makefile}
|-----QRC
|-----virtuoso
|---simulation
|---cov_work
|---RESTORE {restore_charrua_rtl.tcl, restore_charrua_rtl.tcl.svcf, restore_charrua_netlist.tcl, restore_charrua_netlist.tcl.svcf}
|--{runRTL, runCharrua.tcl, runNetlist, runScan, sourceme, testbench_RTL.v, testbench_netlist.v, program.bin}
|---verification

|-**documents**
|---{structural_charrua_specification .odt, functional_charrua_specification.odt, PMUB_charrua.odt}

|-**lib**
|---lef {CTC06.lef}
|---liberty {CTC06ST_typ_3_3V_25C.lib}
|---verilog {CTC06ST.v}
|---doc {datasheet_st.pdf}
|-**mem**
|---SPRAM256X8 {SPRAM256X8.v, SPRAM256X16.v, SPRAM256X8.lib, SPRAM256X8.lef}

|-**software_devel**
|---bin {novocharrua2.o, adjust16.sh}
|---src {Makefile, novocharrua2.l, novocharrua2.ypp,novocharrua2.tab.cpp, novocharrua2.tab.hpp}
|---test_programs {individualAO.input, individual.asm, pagetest.asm, teste2.asm,teste.asm,vectest.asm}
|---compile.sh

Todos os arquivos (em vermelho) usados no desenvolvimento e na verificação do Charrua encontram-se distribuídos na hierarquia dos diretórios (em azul) listados acima.

5 Verificação do Charrua

Foram desenvolvidos 3 "*testbenches*" para verificar o Charrua que se encontram dentro da pasta ***simulation***. São eles: testbench_RTL.v, testbench_Netlist.v e testbench_scan.v. Seguindo a ordem e a sugestão dos nomes, o primeiro serve para verificar o funcionamento do Charrua em nível RTL, o segundo para verificar o funcionamento da netlist e o terceiro para verificar que a scan chain inserida funciona corretamente.

Para verificar que o Charrua executa um programa, é necessário gerar o binário das instruções no arquivo **program.bin** que deve estar dentro da pasta ***simulation***. Dentro da pasta **software_devel** existem alguns programas testes que poderão ser utilizados para verificar o funciomento do Charrua. Para gerar um **program.bin**, basta executar o seguinte comando dentro da pasta software_devel:

```
./compile.sh <nome_programa.asm>
```

É válido lembrar que já existem alguns programas de testes dentro da pasta **test_programs**:

pagetest.asm – Programa para testar quando a memória não contém a instrução que deve ser executada. Assim, o testbench pára o Charrua e re-programa a memória usando o bootloader

vectest.asm – É um programa para testar o uso de ponteiros, primeiro enchendo a memória de dados e depois trocando os dados do índice mais alto com os dados dos mais baixo da memória.

teste2.asm – Programa simples

teste.asm – Programa simples

individualAO.asm – Programa desenvolvido através de um Algoritmo Evolutivo para aumentar as métricas de cobertura funcional

Uma vez compilado o programa desejado, deve-se copiar o arquivo **program.bin** para ***simulation***. Dentro da pasta **simulation**, pode-se executar a simulação tanto RTL, como Netlist ou Scan usando os seguintes comandos:

```
1 - source sourceme
2a - irun -f runRTL -input RESTORE/restore_charrua_rtl.tcl
2b - irun -f runNetlist -input RESTORE/restore_charrua_netlist.tcl
2c - irun -f runScan -input RESTORE/scan.tcl
```

Os comandos acima executam uma simulação para o nível de abstração desejado e abrem as formas de ondas dos sinais do Charrua. Assim, fica fácil a visualização do funcionamento do charrua de forma organizada.

6 Estruturas de Teste

6.1 NAND tree

Aproveitou-se os PADs da XFAB que contêm portas lógicas do tipo NAND para executar os testes paramétricos e testar os PADS. Este teste usufrui da estrutura NAND tree que surge da interconexão serial entre os PADS de entrada, podendo então verificar se cada PAD de entrada funciona corretamente através do PAD de saída denominado PAD_TEST_OUT. Para realizar este teste, basta colocar todas as entradas do circuito em nível lógico '1' e observar o valor lógico da saída PAD_TEST_OUT. Então, cada entrada é posta em nível lógico '0' uma após a outra verificando toda vez que o PAD_TEST_OUT troca de nível. A Figura 5 mostra a configuração NAND tree que conecta os PADs.

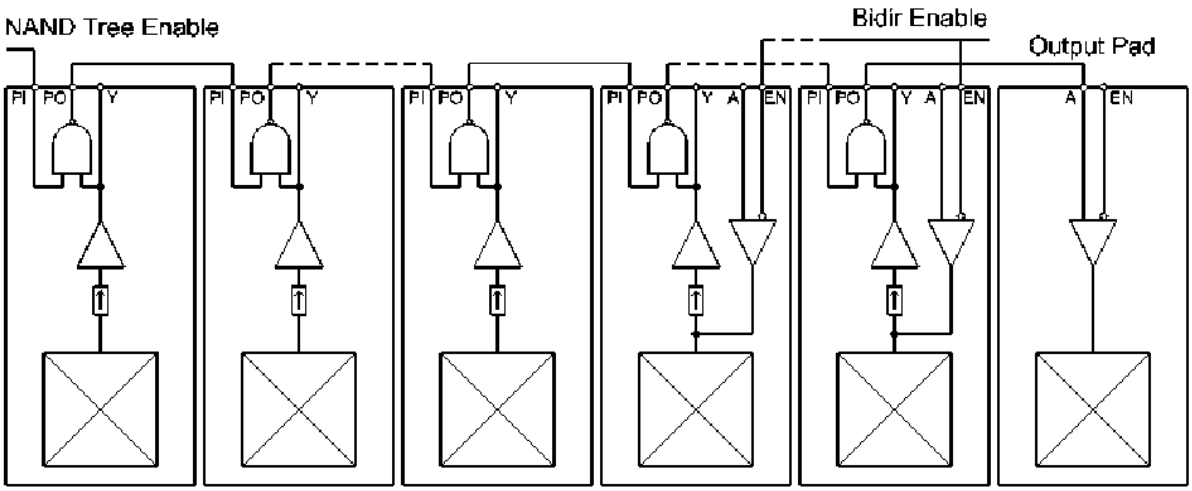

Figura 5: Configuração da NAND tree usada no teste paramétrico

6.2 Design para Testabilidade (DfT)

É evidente que deve-se ter cuidado ao utilizar uma biblioteca nova num projeto VLSI. Especialmente quando ela será utilizada para criar um benchmark de caso de estudo, como processador Charrua. Assim, adotou-se uma estratégia de testes peculiar possibilitando inserir DfT no circuito de forma transparente. Colocou-se uma scan-chain do tipo shadow, isto quer dizer, que os FFs originais da síntese do circuito não são substituídos pelos os FFs de scan. Ao invés, colocou-se FFs de scan paralelamente aos FFs normais compartilhando as entradas e multiplexando as saídas. Desta forma, se os FFs de scan estiverem com problema, eles não irão interferir no funcionamento normal do circuito. Por outro lado, se o circuito possuir células combinacionais com problemas (erro de design ou defeito de fabricação), a scan-chain poderá ser utilizada para fazer diagnose e detectar o defeito. A Figura 6 mostra como a scan-chain está inserida no circuito. Lembrando que o FF normal e o FF de scan têm clocks diferentes. O primeiro funciona com o clock nominal denominado CLK e o segundo funciona com o clock de teste, denominado TCLK. Essa configuração possibilita executar testes de **stuck-at, delay, path delay e debug** de programas sendo executados no modo funcional.

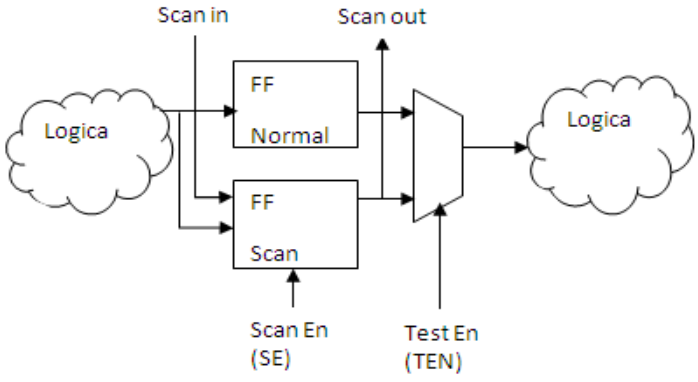

Figura 6: Shadow scan-chain

7 Síntese lógica

Para fazer a síntese lógica com a inserção da scan-chain shadow descrita na seção 6, deve-se ir à pasta **logical_synth/scripts** e executar o comando synth.sh, da seguinte maneira:

```
./synth.sh <nome_do_projeto>
```

Durante a execução deste script, será necessário modificar manualmente a primeira netlist em

digital_database/gate/<nome_do_projeto>_netlist_scan.v. Deve-se ajustar manualmente o cabeçalho da definição do módulo charrua inserindo as portas: SDI, SDO, TEN, TCLK, TRST e SE. Ainda, antes da diretiva **endmodule**, deve-se inserir a seguinte linha:

```
assign SDO = <Saída do último FF de scan> ;
```

Uma vez modificada a netlist, o script pedirá para apertar ENTER para continuar com a segunda parte da síntese. Caso tiver algum problema no ajuste da netlist, basta modificá-la novamente e executar o seguinte comando:

```
rc -f 2_Net2Net_wscan.tcl -ex 'set name <nome_do_projeto>' -cmdfile cmd/<nome_do_projeto>_scan.cmd
-logfile log/<nome_do_projeto>_scan.log
```

Ainda, é necessário incluir os PADS na netlist sintetizada. Caso deseja-se usar uma versão pronta, basta utilizar a netlist já ajustada localizada em: **digital_database/gate/netlist_charrua.v**

Os reports de área e performance estão dentro da pasta **logical_synth/reports**.

O report de área da síntese lógica incluindo a scan-chain acusa que o SoC utiliza uma área total de 2,36 mm^2

Instance	Cells	Cell Area	Net Area	Total Area
SoC	**1078**	**2100056**	**263194**	**2363250**

Além do mais, o circuito deverá funcionar numa frequência de até 20MHz no melhor caso, porém recomenda-se usar um clock de 5MHz.

8 Síntese Física

O resultado da síntese física está ilustrado na Figura 7 abaixo. Para realizar a síntese física sem erros, foi necessário usar uma área de 3 mm^2. Desta forma, consegue-se acomodar corretamente os pads e também o roteamento

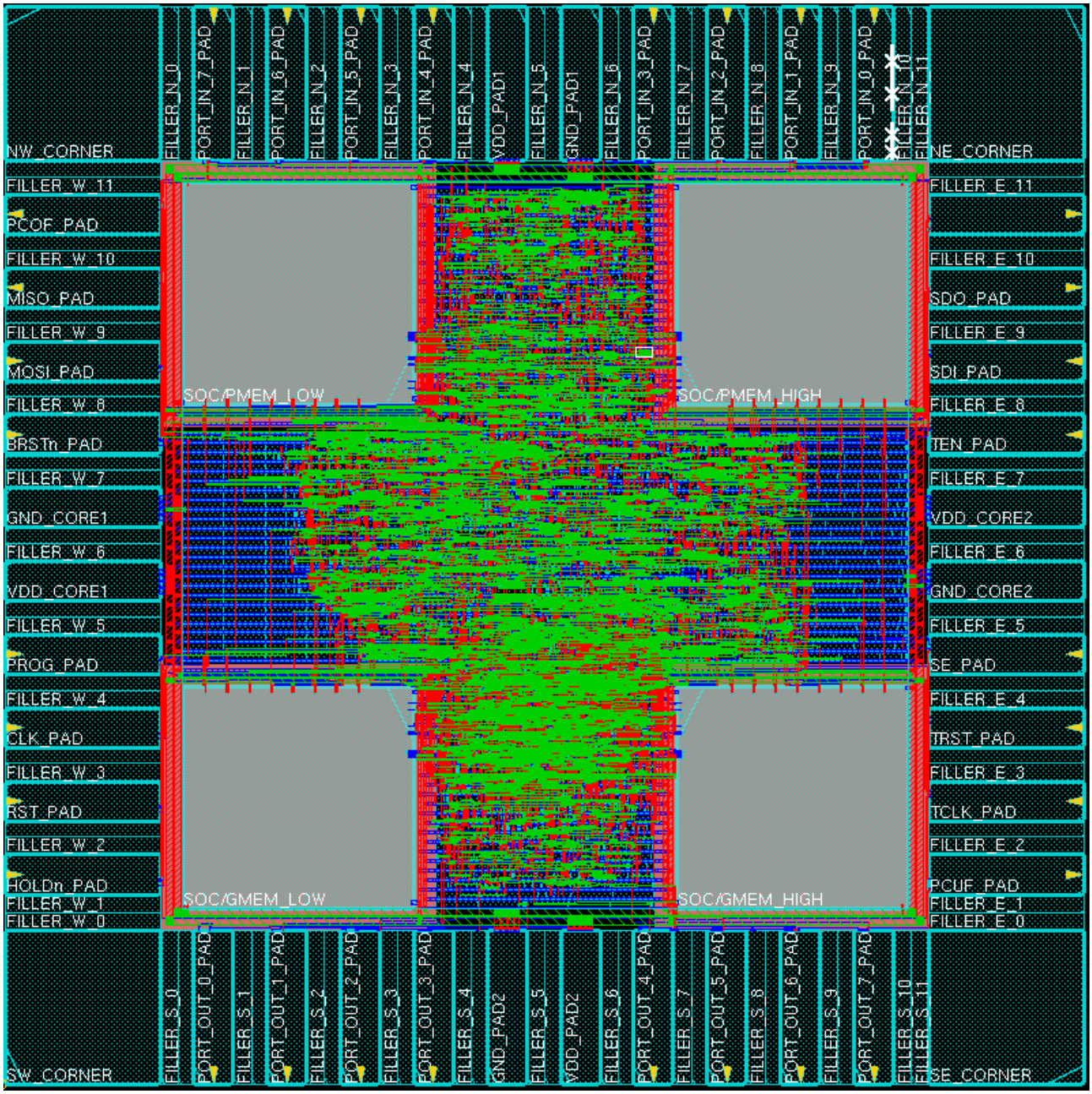

Figura 7: Síntese física do Charrua

Todos os arquivos realizados na síntese física estão localizados na pasta **digital_workflow/physical_synth/encounter**. A síntese física foi executada dentro da pasta work obedecendo o script **charrua_boot_backend.tcl.** Caso queira-se executar o script, basta ir à pasta work, iniciar o encounter e executar as linhas contidas no script.

Ainda, pode-se executar o Makefile localizado na pasta **encounter** para re-fazer todos os passos de forma detalhada e salvando o projeto à medida que os scripts são executados. Executa-se o comando da seguinte forma:

```
#> make floorplan
#> make placement
```

```
#> make cts
#> make route
```

Ainda, deve-se executar os comandos de extração de parasitas para uma simulação mais detalhada e de equivalência lógica entre o circuito sintetizado no encounter e a netlist elaborada no rtl compiler. Executa-se os seguintes comandos:

```
#> make qrc
#> make lec
```

A síntese final não acusa erros graves de DRC e conexões estranhas, porém acusa erroneamente uma violação da falta de conexão do VDD e GND no PAD PORT_IN[0].

Virtuoso ...

9 Plano de Testes

Assim que as amostras estiverem prontas, deve-se aplicar os seguintes testes:

9.1 Inspeção visual

9.2 Teste de continuidade

9.3 Teste de alimentação controlada e corrente de leakage

9.4 Teste paramétrico

9.5 Teste estrutural

9.6 Teste das memórias

9.7 Teste de delay

9.8 Teste de path delay

9.9 Teste funcional

9.10 Teste de performance funcional