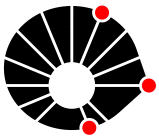

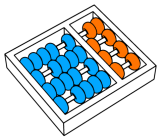

ReonV: Implementação de um processador Leon3 com Instruções Risc-V

Monitor de depuração RVMON

Ricardo Zaideman Charf

Relatório Técnico - IC-PFG-18-36
Projeto Final de Graduação
2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

RVMON, um monitor ReonV
IC-PFG-18-36

Ricardo Zaideman Charf
Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

Resumo

O Instituto de Computação iniciou um projeto para adaptar o processador de amplo suporte Leon3 de arquitetura SPARC, para a arquitetura RISC-V, objetivando aumentar a gama de suporte à ela. Este projeto foi batizado de ReonV.

O projeto relatado nesse documento constituiu na continuidade desse projeto, que começou através de uma Iniciação Científica porém mais focado em algumas tarefas em específico: A criação de um substituto para o software proprietário GRMON, tipicamente usado para depuração no Leon3 mas que não suporta Risc-V.

Durante a pesquisa do projeto, foi encontrado uma forma de adicionar um periférico que funciona como boot ROM à ambas arquiteturas. Foi, também, corrigido um bug do Leon3 que impedia a utilização correta do periférico.

Além disso, foi criado com sucesso um substituo aberto para o GRMON, batizado de RVMON, que consegue se comunicar, ler, escrever e executar código no ReonV, se comportando como uma versão do GRMON adaptada à Risc-V.

Relatório sobre projeto de conclusão de curso

Engenharia de Computação - Curso 34
Instituto de Computação UNICAMP
Dezembro 2018

Sumário

1 Introdução

Este documento detalha a execução e resultados do Projeto Final de Graduação do referido aluno, sob a orientação do Prof. Dr. Rodolfo Jardim de Azevedo.

O ReonV tem origem através de um ramo de pesquisa do IC sobre supervisão do Prof. Rodolfo, para desenvolver uma versão do processador RISC-V que tenha maior suporte a kits de desenvolvimento de FPGA, utilizando como base o processador Leon3.

O RISC-V é um conjunto de instruções (do inglês ISA, Instruction Set Architecture) aberto, desenvolvido pela University of California para uso livre de qualquer propósito, focado nos princípios RISC (reduced instruction set computing), por esses motivos acabou sendo muito adotado pela área de pesquisa e hoje é amplamente utilizado, inclusive no Instituto de Computação da UNICAMP. Já o Leon3 é um processador SPARC desenvolvido pela Sun Microsystems.

Infelizmente as implementações do RISC-V não suportam muitas placas FPGAs, ao contrário do Leon3 que possui uma diversidade bem maior. Assim, para sanar este problema, foi realizada a alteração do conjunto de instruções do Leon3 para suportar as instruções do RISC-V, permitindo rodar programas RISC-V nas placas suportadas pelo Leon3. Este projeto foi batizado de ReonV (uma mistura de ambos os nomes) e foi iniciado através de uma Iniciação Científica, pelo aluno Lucas Castro. O projeto ReonV se encontra no repositório https://github.com/lcbcFoo/ReonV.

2 Objetivo

Dado que boa parte do ramo de pesquisa mencionado está dependendo do ReonV, o objetivo desse Projeto Final de Graduação consiste em retomar e acelerar o desenvolvimento dele, em continuidade da iniciação científica.

O Objetivo deste projeto é, ao final, ter contribuído de forma significativa no avanço do ReonV com foco em software, criando um substituto para o software proprietário de assistência à depuração em sistemas Leon3 GRMON, que não está preparado para ser usado com as instruções RISC-V. Ou a criação de um boot code que execute no RiscV e possibilite essa mesma funcionalidade, apesar da primeira opção ser preferível.

3 Desenvolvimento

3.1 Hardware utilizado

Para esse projeto, foi usada a placa Nexys4ddr da Digilent.

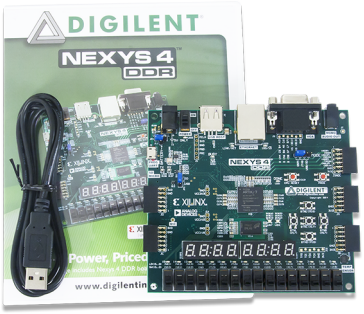

Figura 1: Foto da placa FPGA utilizada.

3.2 Ambiente

Todos projeto foi desenvolvido em um computador com sistema operacional Ubuntu versão 16.04.1.

3.3 Instalação de dependências

Lista de softwares utilizados:

- Xilinx Vivado versão 2017.1
- Modelsim - Simulador de FPGA
- Xilinx ISE 14.7
- RISC-V Toolchain
- BCC2 SPARC v8 compilation Toolchain
- GRMON3
- interceptty - ferramenta de sniffing em portas seriais

A partir disso, segui as intruções fornecidas no repositório do ReonV, nesse link para a sintetização do processador. Obs: As ferramentas e software são os mesmos para a sintetização de um Leon3 ou do ReonV.

Um exemplo script que auxilia a colocação dos paths corretos na variável de ambiente, pode ser encontrado no code listing 1:

```
1 export PATH="$PATH:$HOME/Xilinx/Vivado/2017.1/bin"
2 export PATH="$PATH:$HOME/Xilinx/SDK/2017.1/bin"
3 export PATH="$PATH:/opt/sparc-elf-4.4.2/bin"
4 export PATH="$PATH:$HOME/riscv-gcc/bin"
5 export PATH="$PATH:$HOME/intelFPGA_pro/18.1/modelsim_ase/bin"
6 export XILINX="$HOME/Xilinx/14.7/ISE_DS/ISE"
```

Listing 1: Exemplo de script para load de dependências nas variáveis de ambiente

3.4 Metodologia

A estratégia inicial era investigar o funcionamento do GRMON usando um método conhecido como "sniffing". Esse método consiste em interceptar as mensgaens trocadas entre a placa e o software do GRMON. Com isso, descobriria-se quais processos que ele executa e seria possível reproduzi-los no software que futuramente seria o monitor do ReonV.

Infelizmente, não foi possível executar esse método incialmente. Da maneira que o GRMON conectava-se com a placa por JTAG, não era possível interceptar as mensagens trocadas.

O GRMON era executado com as seguintes opções:

```
$ grmon -digilent -u -ni
```

Com esse impedimento, a estratégia teve que ser modificada. Criaria-se um código que rodasse em Risc-V e que forneçesse algum tipo de terminal como interface com o processador. Para isso, era necessesário o carregamento de código sem a necessidade do uso do GRMON. A alternativa foi criar uma boot ROM.

3.5 Boot ROM

Para criar um boot code, precisava ser possível permanência de dados na placa para que ele pudesse ser executado sem a necessidade de um software externo. A solução encontrada foi sintetizar uma ROM em hardware, onde o código de boot residiria. Por sorte, essa opção já existia nas configurações de sintetização do Leon3. Bastava ativa-la, como na figura 2:

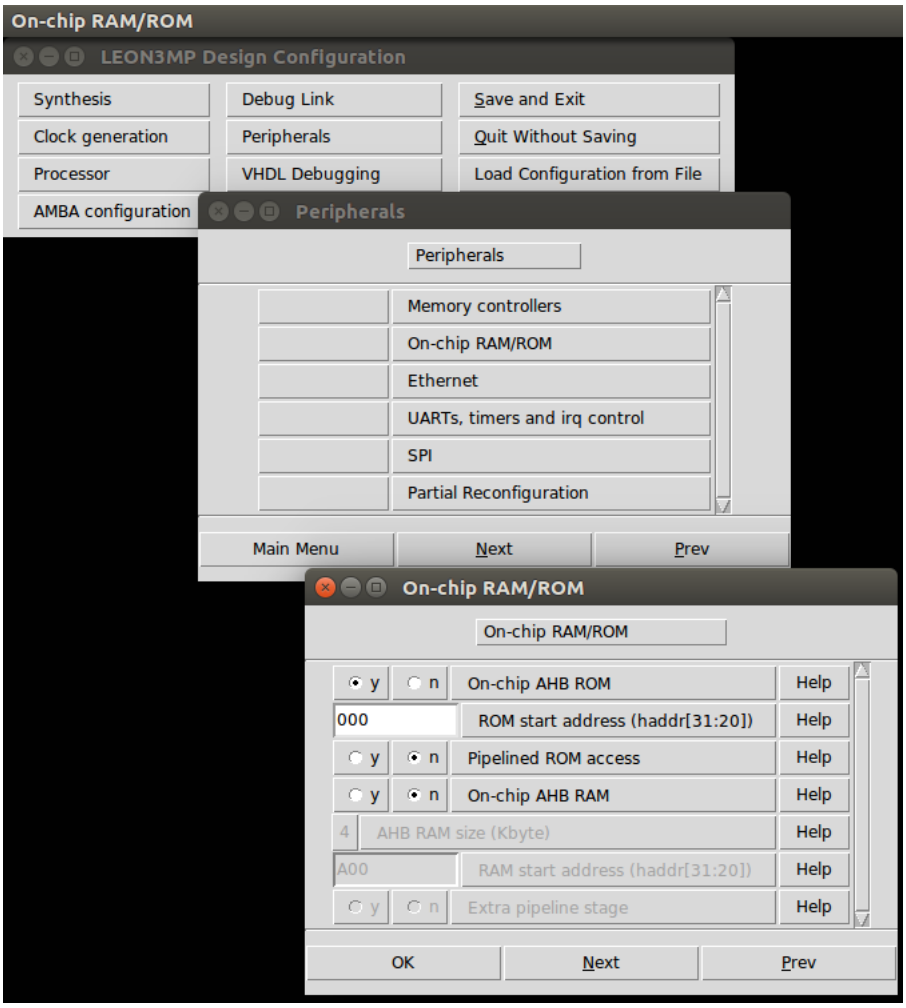

Figura 2: Configuração de sintetização para ativação da AHBROM

Com a AHBROM (componente em hardware ligado ao barramento AHB que contém código ROM) configurda para o endereço de memória 0x0, o processador deveria executar as instruções codificadas ali na sua inicialização, pois é o comportamento padrão da arquitetura.

Para testes, criei um pequeno código em SPARC que dispara o carácter '%'(0x25) em loop para a UART, e o compilei usando o toolchain fornecido no BCC2:

Arquivo *test.s:*

```
.global start, _start
_start:
start:
.global prog
set 0x25, %g5
set 0x80000100, %g6
prog:
st %g5, [ %g6 ]
ba prog
nop
```

Listing 2: Código envia o cracter "%"para a UART em loop

Arquivo *generate_and_substitute_ahbrom.sh:*

```
#compila código em sparc
sparc-elf-as test.s -o prog.o
sparc-elf-gcc -msoft-float -nostdlib prog.o -o prog.exe
#get program binary dump
sparc-elf-objdump -sj .text $$$1 | tail -n +5  > dump.prom.filtered
# generate AHBROM.vhd
python3 ahbrom_gen.py < dump.prom.filtered > ahbrom.vhd.gen
#substitute original AHBROM.vhd to custom one.
cp ahbrom.vhd.gen /grlib/designs/leon3-digilent-nexys4ddr/ahbrom.vhd
```

Listing 3: Script que mostra um processo de transformação de código em assembly até a ROM em vhdl

Os scripts *ahbrom_gen.py* e *generate_and_substitute_ahbrom.sh*, que são referênciados no código 3, podem ser encontrados no repositório do RVMON: https://github.com/barleto/RVMON/tree/master/helpers

Um exemplo de ahbrom.vhd customizada gerada com o código em assembly mostrado acima pode ser encontrado aqui: https://github.com/barleto/RVMON/blob/master/ahbrom_generated_example.vhd

De maneira inesperada, ao ler a memória da posição 0x0, ficou claro que os dados estavam incorretos. O comando "disassemble"do GRMON possibilitou esse memory dump, que pode ser visto na figura 3.

```
grmon3> disassemble 0x0
   0=> 0x00000000: 00000000  unimp
       0x00000004: 00000008  unimp
       0x00000008: 00000008  unimp
       0x0000000c: 00000000  unimp
       0x00000010: 00000000  unimp
       0x00000014: 00000000  unimp
       0x00000018: 00000000  unimp
       0x0000001c: 00000000  unimp
       0x00000020: 00000000  unimp
       0x00000024: 00000000  unimp
       0x00000028: 00000000  unimp
       0x0000002c: 00000000  unimp
       0x00000030: 00000000  unimp
       0x00000034: 00000000  unimp
       0x00000038: 00000000  unimp
       0x0000003c: 00000000  unimp
```

Figura 3: Disassemble de dados à partir do endereço 0x0

Após alguma investigação, a suspeita desse erro é que a AHBROM estava conflitando com algum outro periférico no mapeamento da memória para o barramento AHB. Para sanar essa dúvida, é possível executar o Leon3 no simulador que acusaria caso houvesse algum conflito de endereços.

```
# ahbctrl: slv6: Cobham Gaisler          Generic AHB ROM
# ahbctrl:       memory at 0x00000000, size 1 Mbyte, cacheable, prefetch
# ahbctrl: slv7: Cobham Gaisler          SPI Memory Controller
# ahbctrl:       I/O port at 0xfff70000, size 256 byte
# ahbctrl:       memory at 0x00000000, size 16 Mbyte, cacheable, prefetch
# ** Failure: AHB slave 7 bank 1 intersects with AHB slave 6 bank 0
#    Time: 2 ns  Iteration: 0  Process: /testbench/d3/ahb0/diag File: ../../lib/grlib/amba/ahbctrl.vhd
```

Figura 4: Output da simulação: Alerta de conflito de ampeamento entre SPI e AHBROM

O resultado da simulação (Figura 4) informou que o controlador de memória SPI estava conflitando com a ROM.

A busca nas configurações de sintetização por alguma opção para mudar o mapeamento do SPI controller não foi frutífera. Essa opção não existe. A solução foi buscar a instanciação do componente de SPI no arquivos VHDL do Leon e fazer a modificação manualmente.

Arquivo *leon3mp.vhd:*

```
1  [...]
2  ----------------------------------------------------------------------
3  ---  SPI Memory controller ------------------------------------------
4  ----------------------------------------------------------------------
5  spi_gen: if CFG_SPIMCTRL = 1 generate
6  -- OPTIONALY set the offset generic (only affect reads).
7  -- The first 4MB are used for loading the FPGA.
8  -- For dual ouptut: readcmd => 16#3B#, dualoutput => 1
9    spimctrl1 : spimctrl
10   generic map (hindex => 7, hirq => 7, faddr => 16#000#, fmask => 16#ff0#,
11     ioaddr => 16#700#, iomask => 16#fff#, spliten => CFG_SPLIT,
12     sdcard => CFG_SPIMCTRL_SDCARD, readcmd => CFG_SPIMCTRL_READCMD, dummybyte => CFG_SPIMCTRL_DUMMYBYTE,
13     dualoutput => CFG_SPIMCTRL_DUALOUTPUT, scaler => CFG_SPIMCTRL_SCALER, altscaler => CFG_SPIMCTRL_ASCALER)
14   port map (rstn, clkm, ahbsi, ahbso(7), spmi, spmo);
15
16 [...]
```

Listing 4: Instanciação do controlador SPI no arquivo leon3mp.vhd. Note, na linha 10 o parâmetro genérico faddr. É ele que define o início do mapeamento de memória desse componente e é ele que deve ser mudado.

É possível notar no generic map desse controlador (linha 10 do código 4) a configuração para o mapeamento:

```
faddr => 16#000#
```

Modificou-se o endereço para evitar colisão:

```
faddr => 16#300#
```

Dessa maneira, o controlador SPI seria mapeado à partir do endereço 0x30000000 evitando conflitos com a AHBROM.

Acredito que esse conflito é um erro por parte do Leon, que explicita a opção de sintetização da ROM mas que, ao ativá-la, não é possível usar por causa de conflitos entre endereços hardcoded.

Sintetizei a placa e conferi que a ROM estava configurada corretamente com a ajuda do GRMON:

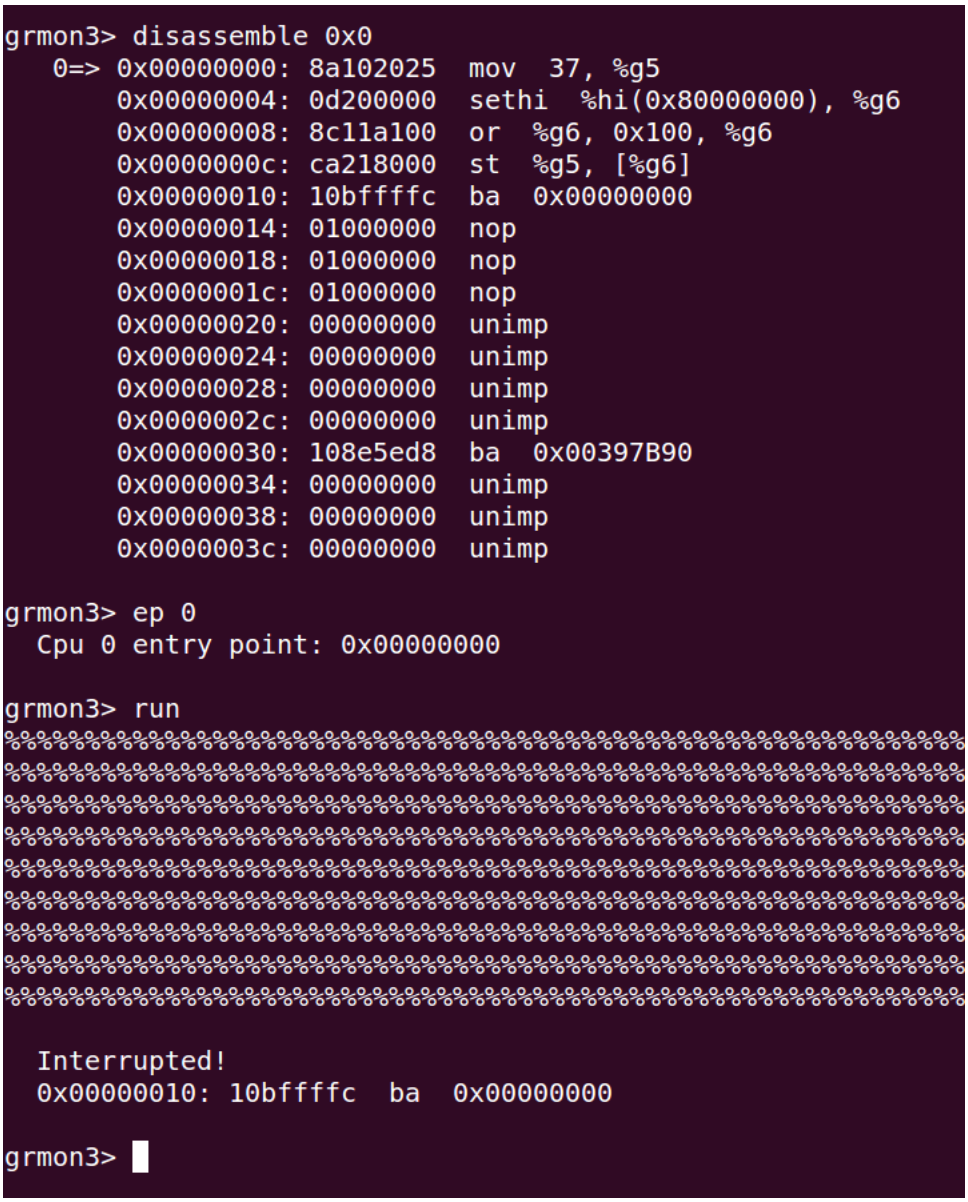

Figura 5: Screenshot de output da simulação: AHBROM correto. Foi possível rodar o código teste 2 com sucesso. A interrupção foi causada intencionalmente pelo usuário.

Como observado na imagem 5, nota-se que os dados da AHBROM estão disponíveis no endereço 0x0 e estão corretos. A execução à partir do endereço zero comprova que o código funciona. Com essa correção é possível rodar código na inicialização do Leon3.

Porém, somente isso, não é suficiente. É necessário ser possível modificar a memória, necessitando de acesso à algum tipo de writable memory, como a DDR2 presente na placa. Nesse momento, não estava claro como configurar a DDR2 sem a ajuda do GRMON. Por essa razão, voltou-se a investigação do GRMON, e uma nova opção foi encontrada. Com essa opção, é possível conectar a placa por uma conexão UART. E, portanto, possível voltar a investigar como o GRMON funciona

por sniffing:

```
$ grmon -uart /dev/ttyUSBV -u -ni
```

Resgatou-se a estratégia inicial.

3.6 GRMON sniffing

Utilizando a ferramenta *interceptty* é possível criar um device falso "ttyUSBV", e conectar o GRMON à ele. Com isso, o *interceptty* consegue exibir na saída padrão o tráfego entre o monitor e a placa.

No código 5, observa-se a saída da execução do interceptty com o GRMON rodando em outro terminal:

```
$ sudo interceptty /dev/ttyUSB1 /dev/ttyUSBV | interceptty-nicedump
<  55 55 55 55 80 ff ff ff f0    | UUUU
>  00 00 10 82                   |
<  80 ff ff ff f0               |
>  00 00 10 82                   |
<  87 ff ff f0 00               |
>  01 00 30 60 00 00 00 00 00 00 |   0`
>  00 00 00 00 00 00 00 00 00 00 |
>  00 00 00 00 00 00 00 00 00 00 |
>  00 00                         |
<  87 ff ff f0 20               |
```

Listing 5: Ouput do comando *interceptty*. Caracteres ≪" representa dados sendo enviados e ≫" dados sendo recebidos. Após "|", temos a representação dos dados em ascii daquela linha

Com isso, foi possível entender os processos que o GRMON executava em startup e nos comandos, lendo os dados enviados à placa que seguiam um protocolo de transmissão. Formato esse explicado no manual IP Cores da grlib, disponível no site da Gaisler.

3.6.1 UART Transmission protocol

A interface suporta um protocolo simples, o qual consiste em um byte de controle seguido de um endereço de 32 bits. Acesso de escrita não retorna reposta, enquanto um acesso de leitura somente retorna os dados lidos.

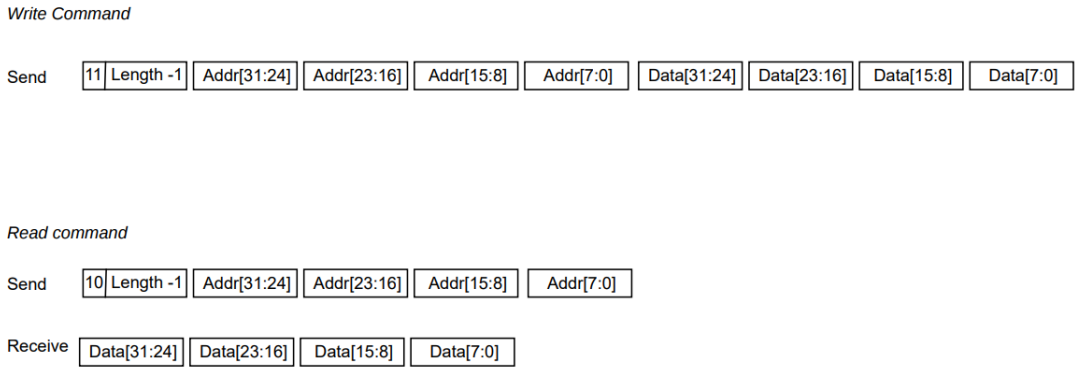

Figura 6: Diagrama do protocolo retirado do manual de IP cores da Gaisler para a AHBUART.

Transferências em bloco podem ser feitas ajustando o campo de Length para n-1, sendo que n denota o número de palavras transferidas. Para um acesso de escrita, o byte de controle e endereço é enviado uma vez só, seguido pelos dados a serem enviados. Para leitura, é envido o byte de controle seguido do endereço. Nos dois casos, os endereços são incrementados sequencialmente para a quantidade de palavras requeridas no campo Length do byte de controle.

Dessa maneira, é possível interpretar a comunicação com o hardware, como no exemplo a seguir, do code listing 6:

```
80fffffff0    - Leitura de 1 palavra no endereço 0xfffffff0
    00001082  - conteúdo endereço 0xfffffff0
84fffff000 - Leitura de 4 palavras a partir de 0xfffff000
    01003060  - conteúdo endereço 0xfffff000
    00000000  - conteúdo endereço 0xfffff004
    00000000  - conteúdo endereço 0xfffff008
    00000000  - assim por diante ....
    00000000
c140001000 - escrita de 2 palavras começando no endereço 0x40001000
    001100bb - dado escrito em 0x40001000
    80000043 - dado escrito em 0x40001004, e assim por diante
[...]
[...]
```

Listing 6: Saída do comando interceptty formatada e comentada. Um exemplo de transmissão de dados protocolada entre o GRMON e o Leon3.

Há mais um exemplo de troca de mensagens do GRMON. Foi registrada a comunicação do comando *run*. Esse registro documentado pode ser encontrado nesse link. Obs: Nesse exemplo os retornos dos comandos read foram retirados.

3.7 RVMON

Sabendo quais endereços que o GRMON lia/escrevia, foi possível descobrir os periféricos que ele acessava e configurava. Seguindo o mapeamento de registradores na memória descrito no manual dos IP cores e as informações da área de plug&play, foi possível reproduzir os passos e até acrescentar configurações em periféricos.

Entendido o protocolo de transmissão, é possível reproduzi-lo em um software novo e aberto que permita seu uso sem restrições com o ReonV. Batizou-se esse software de RVMON (ReonV MONitor).

A implementação do RVMON foi feita em python3, utilizando o modulo CMD para criar uma interface de linha de comando. O resultado desse software pode ser encontrado no seguinte repositório com as instruções de instalação e execução: https://github.com/barleto/RVMON

Não vou descrever o funcionamento interno do RVMON nesse relatório. Seu funcionamento, comandos e configurções estão bem documentadas em seu repositório, caso queira-se saber mais sobre o software.

Inicialmente, o RVMON foi configurado e testado em um processador Leon3 SPARC. Quando todas as features estavam funcionando normalmente, troquei para o processador em Risc-V. Houve algumas adaptações para que funcionasse direito, e a criação de um novo comando que pudesse carregar um arquivo no formato ELF dentro do Reon. Também foi criado o comando *regs*, que printa na tela os registrador do ReonV, mapeados da forma correta. Pra tal, escrevi um pequeno código em Risc-V assembly que escreve no registrador de número n o valor n, descrito no código 7.

Arquivo *registerTest.s:*

```
.section .text
.globl _start
_start:
                addi  x0, x0, 0
                addi  x1, x1, 1
                addi  x2, x2, 2
                addi  x3, x3, 3
                addi  x4, x4, 4
                addi  x5, x5, 5
                addi  x6, x6, 6
                addi  x7, x7, 7
                addi  x8, x8, 8
                addi  x9, x9, 9
                addi  x10, x10, 10
                addi  x11, x11, 11
                addi  x12, x12, 12
                addi  x13, x13, 13
                addi  x14, x14, 14
                addi  x15, x15, 15
                addi  x16, x16, 16
                addi  x17, x17, 17
                addi  x18, x18, 18
                addi  x19, x19, 19
                addi  x20, x20, 20
                addi  x21, x21, 21
                addi  x22, x22, 22
                addi  x23, x23, 23
                addi  x24, x24, 24
                addi  x25, x25, 25
                addi  x26, x26, 26
                addi  x27, x27, 27
                addi  x28, x28, 28
                addi  x29, x29, 29
                addi  x30, x30, 30
                addi  x31, x31, 31
```

Listing 7: Código envia o cracter "%"para a UART em loop

Após executar o programa 7, usei o RVMON para ler os registradores:

```
$ rvmon /dev/ttyUSBV 115200 -u
```

```
Starting rvmon...
Try to connect to /dev/ttyUSB1 @ 115200 baudrate
Connected.
Entering processor debug mode...
Redirection UART to rvmon.
rvmon> regs

     00:  0x00000000   08: 0x00000008   16: 0x00000010   24: 0x00000018
     01:  0x00000001   09: 0x00000009   17: 0x00000011   25: 0x00000019
     02:  0x00000002   10: 0x0000000a   18: 0x00000012   26: 0x0000001a
     03:  0x00000003   11: 0x0000000b   19: 0x00000013   27: 0x0000001b
     04:  0x00000004   12: 0x0000000c   20: 0x00000014   28: 0x0000001c
     05:  0x00000005   13: 0x0000000d   21: 0x00000015   29: 0x0000001d
     06:  0x00000006   14: 0x0000000e   22: 0x00000016   30: 0x0000001e
     07:  0x00000007   15: 0x0000000f   23: 0x00000017   31: 0x0000001f


   psr: 0xf30000e0   wim: 0x00000002   tbr: 0x40001000   Y: 0x00000025

   pc:   0x40001080
   npc:  0x40001084

rvmon>
```

Figura 7: Resultado do comando *regs* no RVMON

No output da execução do RVMON (Figura 7), os registradores x0 até x31 contém os valores corretos.

4 Resultados

Foi criado com sucesso um substituto para o GRMON. RVMON é um software completo, com o necessário para poder se comunicar, ler, escrever e executar código em um processador ReonV. Ele se comporta da mesma maneira que o GRMON configurado pelo device UART.

Considero, portanto, projeto concluído com sucesso. O resultado pode ser encontrado no repositório https://github.com/barleto/RVMON.

Além disso, foi possível sintetizar o processador com uma boot ROM. Durante o desenvolvimento do projeto, um script foi criado facilitando esse processo que gera uma ROM a partir de um executável ELF de Risc-V ou SPARC. Por consequência, resolvi um bug de conflito de endereços entre periféricos que impedia o uso da boot ROM. Com isso, é possível criar um software, futuramente, que inicialize o processador de forma correta e até forneça uma interface terminal para sua programação.

5 Próximos passos e melhorias

Na versão atual, o RVMON funciona perfeitamente para a placa que usei durante o projeto. Para diferentes controladores de memórias e placas FPGA, é possível que não funcione corretamente por

não saber como inicializar os periféricos de forma correta. Uma melhoria é deixar o RVMON mais genérico para aceitar toda a gama de hardwares que o GRMON aceita.

Além disso, é possível adicionar melhorias ao código e deixá-lo mais customizável pelo usuário. Adicionar novas funcionalidades e comandos. Melhorar o tratamento de erros e exceções. No geral, tornar o software mais robusto.

Pode-se, tmabém, trabalhar sobre a conexão JTAG com o argumento *-digilent* no GRMON. O RVMON se comporta de forma igual ao GRMON em conexão UART. Mas o GRMON na conexão JTAG é mais estável e rápido. Vale tentar descobrir o porque e modificar RVMON com esse conhecimento.

Por fim, como comentado anteriormente, é possível implementar um software de boot para o Reon-V.

Referências

[1] Gaisler. Grmon3 user's manual https://www.gaisler.com/doc/grmon3.pdf, 2018.

[2] Gaisler. https://www.gaisler.com/products/grlib/grip.pdf grlib ip core user's manual, 2018.

[3] Gaisler. https://www.gaisler.com/products/grlib/grlib.pdf grlib ip library user's manual, 2018.

[4] Gaisler. https://www.gaisler.com/products/grlib/guide.pdf leon/grlib guide - configuration and development guide, 2018.

[5] SPARC International Inc. The sparc architecture manual version 8 https://www.gaisler.com/doc/sparcv8.pdf.

[6] RISC-V Instruction Set Reference. rv8 risc-v simulator for x86-64 https://rv8.io/isa.html.

[7] Inc The Santa Cruz Operation. System v application binary interface https://www.gaisler.com/doc/sparc-abi.pdf.