Linguagem Assembly

Danrlei Almeida Araujo[1], Evelyn Suzarte Fernandes[2], Gustavo Boanerges[3], Igor Soares[4]

Engenharia da Computação – Universidade Estadual de Feira de Santana (UEFS)
Avenida Transnordestina s/n – 44036-900 – Feira de Santana – BA – Brasil

danrleiaraujo@gmail.com, evelynsuzarte@hotmail.com,
g.boanerges2@gmail.com[3], ifs5544@gmail.com

Resumo. *Este relatório tem como objetivo descrever como foi desenvolvido o código-fonte de configuração da UART (determinando o baud rate, paridade, stop bits e quantidade de bits de mensagem) de um Raspberry Pi Zero, utilizando a linguagem de programação Assembly para um processador ARM.*

1. Introdução

Desde a criação do computador até os dias atuais, sua arquitetura foi muito desenvolvida, assim como a sua forma de programação. Antigamente para programar um computador era necessário cartões de papelão que eram perfurados, criando códigos. Somente na década de 50 que as primeiras linguagens modernas surgiram, aumentando cada vez mais, o que chamamos de "nível de programação", onde sua abstração está cada vez mais longe da "linguagem de máquina" e consequentemente tendo menos acesso direto ao *Hardware*. Hoje, programadores estão acostumados com linguagens como *JAVA, JavaScript, Python ou PHP,* porém somente a linguagem *Assembly* que temos o controle absoluto da máquina, essa linguagem tem uma grande vantagem, sua consistência, pois existe desde os primórdios dos sistemas computacionais e dificilmente deixará de existir no futuro, já que através dela, conseguimos acessar todos os recursos do computador.

Visando o mercado de Internet das coisas (*IoT*) que movimentará mais de 30 bilhões de dólares na América Latina, fomos contratados para implementar um protótipo de um sistema digital baseado em um processador ARM, recebendo informações de sensores através de sua UART. Sendo simples e energeticamente eficiente, sem abrir mão da funcionalidade e elegância na solução proposta, utilizando um Raspberry Pi Zero, programado em linguagem assembly.

2. Fundamentação teórica

Para a produção do código-fonte para a configuração da UART foi necessário o conhecimento em alguns assuntos:

- Raspberry
- Configuração da UART
- Mapeamento
- *File Descriptor*

A Raspberry Pi é um computador de placa única do tamanho de um cartão de crédito (ou até menor, dependendo do modelo). Quando ela é conectada a um monitor, mouse e teclado, a RPi funciona como um computador desktop. O modelo utilizado foi o Raspberry Pi Zero com processador BCM 2835 e arquitetura ARMv6.

A UART (*Universal Asynchronous Receiver/Transmitter*) é um componente receptor e transmissor de dados, ele recebe os bytes de dados e transmite os bits de forma sequencial, em seguida, em um segundo UART, os *bits* são reunidos em *bytes* completos (FreeBSD Documentation Portal, 2021).

Para o nosso projeto, é utilizada a transmissão de dados de forma assíncrona, nessa forma de transmissão os dados são enviados de uma forma pré-programada a partir das configurações feitas na UART, sem que seja necessário enviar um sinal de *clock* ao receptor antes do envio da mensagem. O sistema criado deve permitir configurar o *baud rate*, paridade, *stop bits* e *bits* de mensagem da UART.

O *baud rate* é a velocidade da transmissão da mensagem, onde o receptor e o transmissor devem estar na mesma velocidade de *bits* por segundo.

O bit de paridade serve para verificar se ocorreu um erro na mensagem ao ser enviada. "Quando toda a palavra de dados foi enviada, o transmissor pode adicionar um bit de paridade que o transmissor gera. O *bit* de paridade pode ser usado pelo receptor para executar uma verificação de erros simples. Então pelo menos um *stop bit* é enviado pelo transmissor." (FreeBSD Documentation Portal, 2021).

Segundo BERTULUCCI SILVEIRA (2016), *stop bit* é o último *bit* que colocamos, pensando nele como se fosse um ponto final da mensagem.

As instruções utilizadas no projeto foram:

- LDR: é uma instrução utilizada para carregar conteúdo da memória para um registrador
- MOV: a instrução permite copiar um valor para dentro de um registrador
- SVC: essa instrução causa uma exceção. Isso significa que o processador muda para o modo supervisor e executa a ramificação de execução para o SVC
- BPL: é uma instrução que ramifica para o endereço especificado se o sinal negativo estiver limpo.
- MOVS: É um pseudocódigo que copia o conteúdo de um registrador para outro
- STR: é uma instrução usada para armazenar o conteúdo de uma registrador em um endereço de memória
- TST: é uma instrução que executa uma operação AND bit a bit entre dois valores e atualiza os sinalizadores de condições no resultado, mas não coloca o resultado em nenhum registro.
- BNE: a instrução verifica se os valores de dois registradores não são iguais, se não for o programa é desvio para a label especificada na instrução
- LSL: a instrução executa um deslocamento de bits a esquerda
- BIC: é uma instrução que é utilizada para limpar os bits (zerar os bits).

Quando um arquivo é aberto no sistema operacional é necessário identificá-lo, isso é feito pelo *File Descriptor* ou descritor de arquivos, em português, que é um número inteiro não negativo identificador do arquivo aberto no SO.

3. Metodologia

Para a resolução do problema foi-se necessário utilizar a linguagem assembly para a criação do protótipo requerido. A placa Raspberry Pi Zero presente no laboratório apresenta-se conectada por meio de uma protoboard com extensão dos pinos GPIO, e percebeu-se que havia o sistema operacional Linux para acessar a placa em si.

O SO intermedia o acesso à memória, pois ele reserva espaços para seus processos, então não é possível acessar os endereços encontrados no datasheet do processador. Assim, para acessar o dispositivo de comunicação do SBC, a UART, era requerido o emprego de uma função de mapear memória.

O primeiro passo em direção a solução do problema foi o mapeamento da memória física 0x20201000 em virtual, que consiste em abrir o arquivo "/dev/mem", que contém um ponteiro para que seja possível acessar os locais da memória que não tem-se acesso. Tendo os outros parâmetros necessários para abrir o "/dev/mem", a diretriz para abertura de arquivo do Linux é executada e o retorno é um File Descriptor, que será usado como parâmetro para a função mmap2.

Para utilizar a mmap2 são requeridos alguns parâmetros: o file descriptor - inteiro identificador do arquivo aberto, variáveis de proteção de memória, variável para deixar o SO escolher o endereço virtual e o principal que é o endereço físico que pretende-se mapear. Como a UART possui 2 pinos, Tx e Rx, reservados para a comunicação, não é necessário mapear a GPIO e ativar estes pinos.

Após receber o endereço mapeado da UART, precisa-se utilizar seus registradores para realizar a comunicação de fato. As primeiras etapas deste processo são desativar a UART, testar se há transmissão/recepção de dados acontecendo e esperar que termine, desativar FIFO no registrador Line Control, reprogramar as configurações necessárias para envio de dados, habilitar a UART novamente ativando os pinos de Tx, Rx e LE, transmissão, recepção e loopback habilitado - Tx ligado ao Rx da UART.

Agora com a UART devidamente configurada e novamente com o endereço mapeado, escreve-se uma mensagem binária no registrador de dados, que será adicionado ao FIFO de transmissão. Apesar do registrador possuir 32 bits, apenas os 8 bits menos significantes serão utilizados para a transmissão e 12 menos significantes para a recepção. Se o FIFO estiver vazio, a mensagem será transmitida imediatamente. Para a recepção, verifica-se o bit do flag register, que sinaliza se o FIFO de recepção está vazio, quando a FIFO possuir bits, pode-se examinar a mensagem recebida.

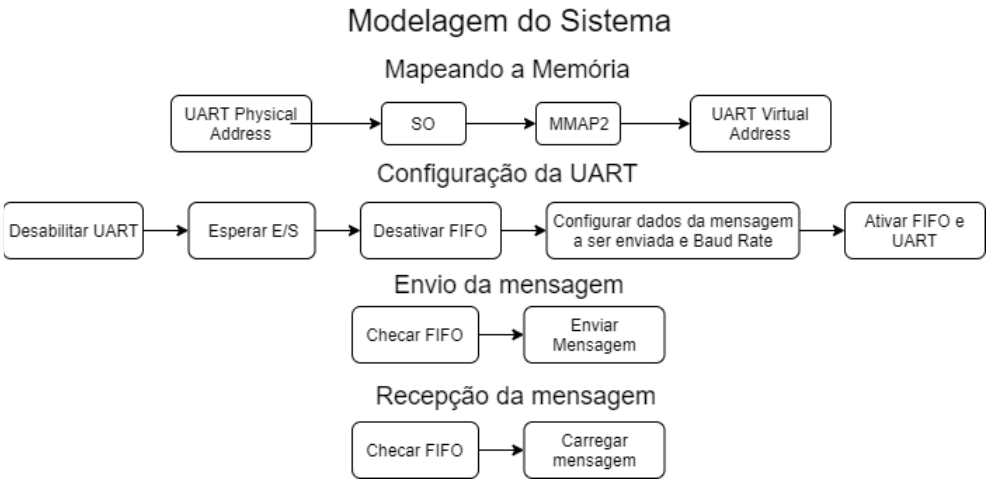

Figura 1. Modelo da solução

4. Resultados e discussão

Para realizar a comunicação via UART é necessário utilizar seus registradores. Dos 18 registradores, 6 são no mínimo necessários para a resolução do problema proposto. Estes 6 registradores são: Data Register (DR), Control Register (CR). Flag Register (FR), Line Control Register (LCRH), Integer Baud Rate Divisor (IBRD) e Fractional Baud Rate Divisor (FBRD).

Já possuindo o endereço virtual da UART, é necessário desligá-la escrevendo o valor 0 no primeiro bit do Control Register.

Verificar se o FIFO está transmitindo ou recebendo dados, carregando o valor da posição da memória base da UART com o offset do Flag Register, e testando se o bit BUSY do FR está igual a 1 - transmitindo byte.

Desabilitar o FIFO escrevendo o número 16 em binário, 0b10000, no Line Control Register, o quinto bit de nome FEN - FIFO Enable.

Configurar o Line Control Register com os seguintes fatores: paridade ativa, paridade par/ímpar, 1 ou 2 bits de parada e o tamanho da mensagem a ser enviada, 00 para 5 bits até 11 para 8 bits. Possuindo os valores da baud rate, escrever no IBRD a parte inteira e no FBRD a parte fracionária.

Finalizando a configuração da UART, precisa-se ativar o FIFO no LCRH, escrever 1 nas posições 9, 8, 7 e 0 do CR, respectivamente RXE - ativa o pino receptor, TXE - ativa o pino transmissor, LBE - ativa o loopback em que o Tx é alimentado ao Rx, UARTEN - habilita a UART novamente.

Para enviar um dado é necessário verificar se o FIFO transmissor está cheio, se não estiver pode começar a transmitir, porque os dados são enviados e recebidos um bit por vez. Da mesma forma, verificar se o FIFO receptor está cheio.

O produto desenvolvido não foi capaz de realizar o envio/recepção de dados. Por conta disso os testes se baseiam na utilização do GDB para apurar os valores nos registradores que estavam sendo manipulados, então foi-se possível constatar que o

mapeamento do endereço físico em virtual da UART estava sendo realizado corretamente.

O *Makefile* utilizado consiste nas linhas abaixo, dentro de um arquivo sem extensão com o nome build, e para compilar -— criar o executável — utiliza-se o comando 'chmod +x build' e executar com o './build'. Para diminuir os comandos pode-se utilizar o 'e comercial' duplo (&&) juntando os comandos anteriores em uma só linha, ficando 'chmod +x build && ./build', dessa forma o ./build só será efetuado se o chmod também for. Então, se não houver erros execute o programa de nome uart, com o comando 'sudo ./uart'. Aqui utiliza-se 'sudo' para obter privilégios de administrador, pois o arquivo /dev/mem é protegido. Abaixo temos os códigos para compilar e gerar o executável:

```
as -g -o uart.o uart.s

ld -o uart uart.o
```

5. Conclusões

O sistema desenvolvido não alcançou todos os requisitos solicitados. Não foi possível efetuar o teste de loopback e verificar a transmissão de dados por meio do osciloscópio. Porém foi possível executar o mapeamento da memória da UART, além configurar a UART inserindo o baud rate, selecionando a existência de paridade, o tipo de paridade (par ou ímpar), o tamanho da mensagem a ser transmitida e a quantidade de bits para o stop bit.

Por fim, a partir da construção deste sistema podemos compreender o processo de desenvolvimento em linguagem de baixo nível, além de perceber as particularidades que uma arquitetura de processador dispõe.

Referências

BROADCOM CORPORATION. **BCM2835 ARM Peripherals**. Cambridge, 2012

GUSE, Rosana. **O que é Raspberry Pi?.** 2020. Disponível em: https://www.filipeflop.com/blog/o-que-e-raspberry-pi/. Acesso em: 20 abr. 2022.

PYEATT, Larry D.. **Modern Assembly Language Programming with the ARM Processor**. Newnes, 2016.

SMITH, Stephen. **Raspberry Pi Assembly Language Programming**: arm processor coding. Colúmbia Britânica: Apress, 2019.

SILVEIRA, Cristiano Bertulucci. **Desvendando a Comunicação RS232**. 2016. Disponível em: https://www.citisystems.com.br/rs232/. Acesso em: 20 abr. 2022.