ESCOLA POLITÉCNICA DA

UNIVERSIDADE DE SÃO PAULO

Relatório de implementação do projeto da disciplina de Organização e Arquitetura de Computadores II (PCS3422)

Tema: Boot de Linux

Integrantes do grupo

Eric Koji Yang Imai Nº USP 9373712

Fábio Fusimoto Pires Nº USP 9853294

Gabriel Roberti Passini Nº USP 9868071

Júlia Fernandes Moraes Nº USP 8988104

Renato de Oliveira Freitas Nº USP 9837708

São Paulo
2019

SUMÁRIO

1 RESUMO

O projeto proposto pela disciplina PCS3422, Organização e Arquiteturas de Computadores II, foi o boot de Linux. Este projeto se inicia com a escolha de um Kernel Linux minimalista, a ser carregado num sistema que tem como processador o LEGv8 e do estudo do funcionamento do bootloader e da importância das device trees no contexto de inicialização de um sistema embarcado.

Posteriormente, foi gerado a imagem do Kernel de Linux utilzando as ferramentas disponíveis no software Buildroot, configurado para ter a arquitetura ARMv8 como target. A imagem gerada foi testada usando o emulador QEMU, responsável por rodar o sistema operacional na máquina em utilização para simular testes iniciais de funcionamento, antes do Deploy em FPGA.

Por fim, estudou-se como é feito o pareamento entre a placa FPGA e o computador (através de uma porta USB) para realizar o boot do Linux que estaria carregado num Micro SD Card (devidamente conectado a uma placa Cyclone V). Ao fim das atividades descritas, obtém-se uma máquina com um SO Linux funcional.

2 INTRODUÇÃO

A disciplina PCS3422, Organização e Arquitetura de Computadores II, busca apresentar conceitos avançados de arquitetura de computadores e técnicas de projeto de elementos arquiteturais e processadores. Visto este objetivo, foi fornecido aos alunos aulas teóricas a fim de capacitá-los no assunto, entretanto, apenas o conhecimento do conteúdo não possibilita o pleno exercício do aprendido. Dessa forma, a fim de preencher esta lacuna foi proposto um projeto, de temáticas sugeridas pelo professor da disciplina, a fim de pôr em prática o aprendido em sala de aula, no caso o projeto de boot de Linux em um sistema embarcado.

O diretório onde estão disponíveis os arquivos utilizados podem ser acessados pela seguinte url:

https://github.com/FabioFusimoto/PCS32422_2019S2

3 SISTEMA OPERACIONAL LINUX - VISÃO GERAL

Linux é um sistema operacional, open-source, disponível em diferentes distribuições (Debian, Ubuntu, Fedora e outros). Como um sistema operacional, é responsável pela interface entre o usuário e o hardware do computador, gerenciando os recursos do sistema, encaminhando cada informação necessária para a execução das instruções para o hardware, garantindo a integridade e proteção do sistema.

O sistema operacional Linux é composto por diferentes partes, dentre elas, o kernel constitui um componente essencial pois é o software que faz a comunicação entre todo o software sendo executado no computador e o hardware, ele é o primeiro conjunto de programas que é carregado na memória principal ao iniciar o sistema.

3.1 O KERNEL

O kernel do Linux é monolítico, o que significa que ele gerencia a CPU, a memória, a comunicação entre processos, os drivers de dispositivos, o sistema de arquivo e as chamadas do sistema.

Com relação ao gerenciamento da CPU, o kernel é responsável por gerenciar o seu compartilhamento, definindo qual processo deverá utilizar o processador em cada instante e quanto tempo o processo deve utilizá-lo antes de ser substituído por outro.

Tratando-se da memória, o kernel aloca um espaço de endereçamento da memória para cada programa, garante que esse espaço não será acessado por outros processos que não tenham a devida permissão e administra a quantidade de memória disponível para garantir que os programas possam ser executados. Todo o endereçamento virtual é controlado pelo kernel, dessa forma ele torna possível que um programa seja executado como se fosse o único em execução e como se a memória disponível fosse maior do que realmente é (pois nem todos os dados ficam armazenado na memória principal, mas também em uma unidade de armazenamento secundário, como um disco rígido).

As interrupções geradas pelos dispositivos de entrada e saída também são tratadas pelo núcleo, isto é feito a partir dos drivers dos dispositivos. Para isso, ele

possui uma lista de dispositivos disponíveis que pode ser gerada previamente no programa do kernel, pode ser configurada pelo usuário ou detectadas pelo sistema como no caso de dispositivos plug and play.

O kernel também gerencia as chamadas de sistema, que são interrupções geradas pelos programas ou softwares de dispositivos para requisitar algo ao sistema ou alertá-lo sobre algum evento.

3.2 O PROCESSO DE BOOT NOS SISTEMAS LINUX

O boot de Linux se inicia com a execução de um programa localizado em uma posição fixa da memória, que é responsável por inicializar o hardware – essa etapa é independente do sistema operacional que será utilizado. Em computadores convencionais esse programa é chamado de BIOS (Basic I/O System), em sistemas embarcados é um programa localizado em uma posição fixa da memória flash. Ele executa o POST (Power On Self Test), que verifica o funcionamento do hardware, caso seja identificado algum problema o processo de boot pode ser interrompido. Após essa checagem, ele busca por dispositivos inicializáveis, carrega na memória RAM o primeiro setor com dados para boot encontrado e executa o código carregado. Normalmente, esse setor encontra-se no Master Boot Record (MBR), que é um setor de 512 bytes encontrado no primeiro setor do disco utilizado para inicialização.

Nesse ponto já se inicia o estágio controlado pelo bootloader, que é o software que gerencia o processo de boot do computador. Ele possui como objetivo carregar o kernel do sistema operacional e seus componentes. A maioria das distribuições de Linux utilizam o bootloader GRUB. As atividades realizadas pelo bootloader podem ser divididas em 3 etapas.

O primeiro estágio começa quando o bootloader está carregado na memória e basicamente consiste em localizar e carregar o setor que contém o código do estágio seguinte (comumente chamado de estágio 1.5), que é responsável pela estrutura do sistema de arquivos. Esse segundo estágio possui como função iniciar a execução dos drivers do sistema de arquivos necessários para localizar os dados para o último estágio e carregar os drivers essenciais para continuar o boot. O estágio 1.5 é importante pois o setor que contém o boot record carregado pela etapa 1 tem um

tamanho limitado, por isso não possui informações sobre sistemas de arquivos, sendo essencial que uma etapa seguinte possa carregar essa estrutura para o sistema.

Todos os arquivos utilizados no último estágio estão localizados no diretório /boot, sua função é localizar e carregar o kernel do Linux na memória RAM e transferir o controle do computador para o kernel. Nessa etapa também é carregado o initrd, que é um sistema de arquivos temporário, que possui programas que possibilitam localizar e montar o verdadeiro sistema de arquivos quando o kernel é de fato inicializado. O initrd pode armazenar também módulos necessários para interface com periféricos, o que permite que o kernel possa ser pequeno sem deixar de possuir suporte a diferentes configurações possíveis de hardware.

O bootloader também é responsável por carregar o device tree blob que consiste de um arquivo com a descrição do hardware do sistema, que é utilizado para localizar e registrar dispositivos. De forma geral, o device tree é uma estrutura de dados que caracteriza por exemplo, o número e tipo de processadores existentes, endereços e tamanho da memória RAM, conexão com dispositivos periféricos e outras informações relacionadas ao hardware. O bootloader passa o device tree para o kernel, que é capaz de seguir com as configurações necessárias para o hardware.

Quando o kernel está carregado na memória, ele primeiramente precisa ser descompactado. Em seguida, ele carrega o systemd e passa o controle para ele. O systemd é um software responsável por configurar um estado do Linux onde outros processos possam rodar. Algumas de suas funções são: estruturar o sistema de arquivos, resolver as dependências para configurar o sistema tornando-o apto a executar suas tarefas e iniciar processos que rodam em segundo plano.

4 O EMULADOR QEMU

O QEMU é um emulador e virtualizador de hardware genérico e open source. Emulador é um hardware ou software que permite que um sistema rode softwares e controle periféricos feitos para outro sistema. Em geral, ambos o sistema operacional e o software serão interpretados, ou seja, a execução de instruções é realizada diretamente na linguagem de programação, sem a necessidade de serem previamente compilados e traduzidos para linguagem de máquina. Já virtualização é a mudança de percepção das características físicas da plataforma de hardware do computador por outras características desejadas. Os programas que controlam a virtualização são chamados de ”Control Program” (termo ultrapassado), “Hypervisor” ou “Virtual Machine Monitor”.

Como emulador, o QEMU consegue rodar sistemas operacionais feitas para uma máquina (ARM, por exemplo) em outra (o computador sendo utilizado). Ao usar tradução dinâmica, ele alcança uma boa performance. Tradução dinâmica é uma técnica em que uma pequena sequência de código é traduzida de uma *Instruction Set* de origem para outra de destino e salva. O código só é traduzido quando é descoberto e quando for possível traduzir, e instruções de *branch* são feitas para apontar para códigos já traduzidos e salvos.

Como virtualizador, ele alcança performance que rivalizam o hardware nativo ao executar o código diretamente na CPU do computador.

Neste projeto, a sua função de emulador será utilizada para testes iniciais de funcionamento antes do *boot* do Linux ser realizado na placa FPGA com o processador completo LEGv8 realizado pelo grupo responsável.

Existem dois modos de operação de emulação: emulação de sistema completo e emulação de modo de usuário.

No primeiro, o QEMU emula um sistema completo (por exemplo um PC), incluindo um ou muitos processadores e vários periféricos. Pode ser usado para rodar diferentes sistemas operacionais sem fazer *reboot* do computador ou para realizar *debug* de um código do sistema.

No segundo, o QEMU pode rodar um único processo (ou programa) compilado para uma CPU em outra CPU diferente, *cross-compiling* e *cross-debugging* rápidos são a maior vantagem e objetivo desse modo.

Será utilizado o modo de emulação de sistema completo no projeto, como especificado na seção 6 através do parâmetro -softmmu.

O QEMU consegue emular diversos sistemas operacionais (GNU/Linux, Windows, Mac OS X, *BSD) para diferentes arquiteturas (x86-64, MIPS64, ARM, RISC-V, entre outros).

Ao rodar a simulação (ver seção 6) pode-se alterar diversos parâmetros para parametrizar a execução. As opções de interesse estão explicadas brevemente a seguir:

- *-machine [type=]name*: seleciona a máquina emulada por *name*;
- *-cpu model*: seleciona o modelo de CPU por *model*;
- *-nographic*: esta opção desativa qualquer saída gráfica do QEMU, transformando-o em uma simples aplicação de linhas de comando (quando estiverem disponíveis uma comunicação serial e uma placa de vídeo, esta opção pode ser ignorada);
- *-kernel bzImage*: usa *bzImage* como a imagem de kernel (opção específica de *boot* de Linux);
- *-initrd file*: seleciona *file* como disco RAM inicial (opção específica de *boot* de Linux).

machine é a especificação do PC que será utilizado, por exemplo pc-i440fx-2.8 e pc-q35-2.8 da arquitetura x86-64. Para arquiteturas ARM, o QEMU providencia uma máquina especial chamada de “virt”, feita especificamente para emulações e virtualizações. Esta máquina contém as funções básicas:

- núcleos de CPU configuráveis;
- U-Boot carregado e executando a partir do endereço 0x0;

- *device tree blob* (DTB) generado colocado no início da memória RAM;
- memória RAM com tamanho configurável, descrevido pelo DTB;
- porta serial PL011, determinável via DTB;
- timer arquitetado pela arquitetura ARMv7/ARMv8.

cpu especifica qual o processador utilizado, no caso desta simulação, será o cortex-a53, uma implementação da ISA ARMv8.

5 PREPARANDO A IMAGEM DO LINUX

Estar em posse do código fonte do Kernel do Linux e dos arquivos compõe o *root filesystem* é apenas o primeiro passo para se obter uma gerar uma versão bootável deste sistema operacional. É necessário, em seguida, utilizar um compilador que gera código de máquina adequado à execução em ambiente simulado. Todas as etapas do processo podem ser realizadas manualmente, entretanto, esse trabalho pode ser facilitado com a utilização do Buildroot.

O Buildroot é um conjunto de ferramentas (desenvolvidas para serem executadas num sistema com Linux) que automatiza o processo de compilação do Linux visando sua utilização em sistemas embarcados. O Buildroot é capaz de gerar um toolchain de cross-compilação, um sistema de arquivos *root*, uma imagem do Kernel e um bootloader. As seções a seguir detalham a configuração e execução dos módulos que constituem o Buildroot.

Vale ressaltar que o processo descrito nas seções a seguir foi realizado num computador rodando o sistema operacional Ubuntu e com todas as dependências instaladas (ver o capítulo 2 do manual do usuário do Buildroot, referência [3]). Eventuais mensagens de erro provavelmente podem ser corrigidas mediante a instalação das dependências adequadas.

5.1 CONFIGURAÇÕES INICIAIS E A CROSS-COMPILAÇÃO

O Buildroot é capaz de gerar imagens para diversas arquiteturas, incluindo: x86_64, MIPS, ARM, PowerPC, dentre outras. Essa flexibilidade deve-se ao vasto número de toolchains (de diversos desenvolvedores) que se pode baixar e integrar ao ambiente de compilação.

Uma toolchain é um conjunto de programas encadeados (que se executam em sequência) cujo objetivo é gerar programas executáveis a partir de código-fonte. Uma toolchain costuma incluir os seguintes programas: compiladores (das mais diversas linguagens), montador, ligador, debugger e bibliotecas em C que contém funções variadas para desenvolvimento de aplicações compatíveis com diversos sistemas operacionais. Cada arquitetura possui diversas toolchains associadas.

O sistema no qual se realiza a compilação é nomeado *host*, ao passo que o sistema para o qual a compilação está sendo feita é o *target*. Quando *host* e *target* possuem arquiteturas diferentes, diz-se que está sendo realizada uma cross-compilação (ou compilação cruzada). Neste projeto, o *host* é um x86_64 (como todos os processadores Intel/AMD para PCs) e o target é o *aarch64* (pois o LEGv8 é baseado no ARMv8).

O Buildroot possui diversas configurações padrão que abrangem diferentes arquiteturas. Para listar todos os padrões, executa-se (num terminal, a partir da pasta raiz do Buildroot):

```
make list-defconfigs
```

O conjunto de configurações de interesse é aquele apropriado para gerar uma imagem compatível com o qemu. O carregamento dessas configurações default é feito executando-se o comando:

```
make qemu_aarch64_virt_defconfig
```

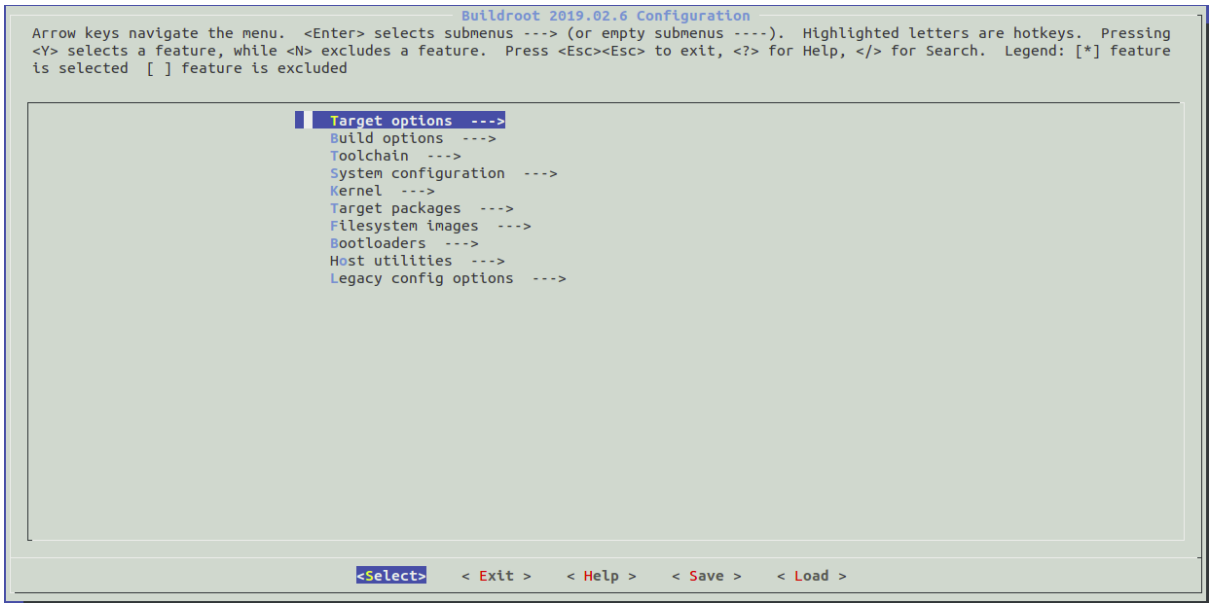
5.2 CONFIGURAÇÕES ADICIONAIS

É possível realizar ajustes finos com uma ferramenta de configuração acessível através do seguinte comando (que abre uma janela como a da Figura 1):

```
make menuconfig
```

Descreve-se brevemente, a seguir, o que está contido em cada um desses submenus e sua relevância para o projeto (aqueles omitidos seguem as configurações *default*):

Figura 1- Tela de configuração do Buildroot após execução do comando make menuconfig



Fonte: elaborado pelos autores com o software Buildroot

A) Target options

Escolha da arquitetura *target* e de particularidades como variantes da arquitetura e especificação da unidade de ponto flutuante. Nenhuma alteração foi realizada em relação as configurações *default*. Nota-se que a variante selecionada é a Cortex-A53, que é uma implementação da arquitetura ARMv8.

B) Toolchain

Este submenu permite escolher qual *toolchain* será usada para gerar a imagem do Linux para a arquitetura selecionada. A *toolchain* escolhida é do tipo externa e o distribuidor é a Linaro.

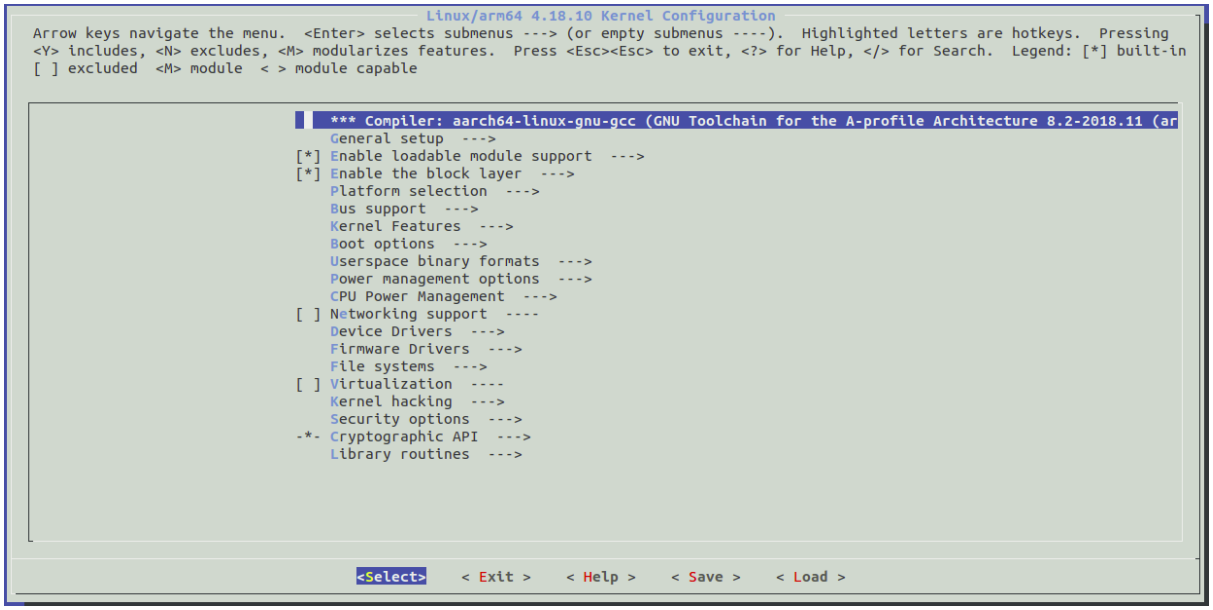
C) System configuration

Possui diversas opções que lidam com o que ocorre após o boot do sistema (e, portanto, configurações referentes ao *root filesystem*) e possibilitam gerenciar um sistema de login, definindo nome de usuário e senha. Optou-se por deixar o nome de usuário como ‘root’.

D) Kernel

Permite selecionar a versão do Kernel que se deseja usar, além de ser possível selecionar o método de compressão. Configurações adicionais no Kernel podem ser realizadas através do comando make linux-menuconfig, que traz um menu como o da Figura 2:

Figura 2 - Configurações adicionais do Kernel do Linux



Fonte: elaborado pelos autores com o software Buildroot

A maioria das opções não foi alterada em relação às configurações *default*, com exceção do *Networking Support* (que foi desabilitado) e com a exclusão de vários *Device Drivers* que dão suporte a dispositivos que não são de interesse para simulação: leitor de cartão SD, placas gráficas, placas de áudio, SATA, dentre outros.

E) Filesystem images

Optou-se por utilizar um *filesystem* do tipo CPIO, sendo que o mesmo é integralmente carregado na memória na inicialização. A utilização deste tipo de *fs* facilita a simulação no QEMU.

5.3 GERANDO AS IMAGENS NECESSÁRIAS

Após a etapa de configuração, é necessário iniciar o processo de montagem através do comando make (sem argumentos adicionais). A simplicidade e flexibilidade

do *Buildroot* tornam-se evidentes nessa etapa. A execução desse script envolve os seguintes passos:

1) O download das dependências necessárias: o código-fonte do Kernel e seus adendos, como extensões e drivers, toolchains externas (como a Linaro, definida na seção 5.2), *rootfs*, *bootloaders*, pacotes extras (nenhum foi selecionado neste caso), dentre outros sources.

2) O *build* propriamente dito. Esse processo envolve selecionar quais os arquivos fonte deverão passar pelo processo de *build* (analisando-se as opções previamente selecionadas) para então invocar a *toolchain* definida pelo usuário; que compila, liga e monta os códigos compilados. Além dos códigos fonte do Kernel e do *bootloader* diversas bibliotecas auxiliares também passam por *build* nesta etapa.

O processo de *build* para as configurações mencionadas resulta na geração de 2 arquivos: um contendo um *filesystem* simples num formato comprimido (*cpio*, como previamente selecionado na seção 6) e outro que é a imagem do Linux propriamente dita, codificada binariamente. O filesystem e a imagem geradas estão presentes no diretório *images* sob o nome de *rootfs.cpio* e *Image*, respectivamente

O *rootfs* pode ser explorado com um gerenciador de arquivos comprimidos. As pastas nele contido são aquelas que se encontram na raiz de um sistema Linux comum: bin, dev, etc, usr, dentre outras. Destaca-se a pasta bin, que possui comandos executáveis a partir do ambiente simulado (ver seção 6). Tais comandos estão disponíveis porque o processo *default* do Buildroot também faz o build de um conjunto de programas conhecido como *Busybox*, que é um agregado de funções simples (criar/visualizar diretórios, compactar/descompactar arquivos, listar arquivos num diretório, dentre outros) e executáveis através de um terminal. O *Busybox* também é baixado, se necessário, após a execução do comando make.

Como a imagem gerada é uma sequência binária (isto é, não possui uma codificação definida), não é possível ler seu conteúdo diretamente, como se faria com um txt, por exemplo. Há duas alternativas para interpretar o conteúdo: gerar um dump hexadecimal (isto é, converter cadeias de bits em caracteres ASCII e armazenar num

txt) ou realizar um *objdump*, que consiste em ler as cadeias de bits e traduzi-las em instruções da arquitetura AArch64.

A primeira das opções envolve utilizar o comando xxd presente por padrão em diversas distribuições do Linux:

```
xxd -c 16 Image image.txt
```

O texto de saída possui uma codificação hexadecimal e sua respectiva representação ASCII de toda sequência de bits que compõe a imagem compilada.

A segunda alternativa dá maior interpretabilidade ao texto gerado. O processo de *disassembly* (ou *objdump*) envolve ler a sequência binária e procurar uma equivalência com as instruções do AArch64, fazendo a codificação das instruções. Tal processo pode ser automatizado através da própria Toolchain utilizada para o *build*, através do comando:

```
./buildroot-2019.02.6/output/host/bin/aarch64-buildroot-linux-uclibc-objdump -m
aarch64 -b binary -D ./images/Image > objdump_kernelImage.txt
```

Nota-se que a *toolchain* não está no repositório, sendo necessária adaptar o comando para apontar para os arquivos baixados pelo Buildroot ou baixar e extrair a *toolchain* diretamente do site da Linaro.

O *objdump* dispõe seu output num formato legível e interpretável:

```
Endereço:  <Instrução codificada> (hexa)  Mnemônico  Operando
```

Nem todas as linhas da imagem podem ser traduzidas para instruções da arquitetura AArch64. Isso ocorre porque nem todas as cadeias de bit são, de fato, código executável. Os primeiros 64 bytes, por exemplo, servem parcialmente como cabeçalho para indicar, por exemplo, o tamanho da imagem. Outros trechos da imagem também servem como área de dados e não cabe interpretá-los através do *objdump*.

6 EXECUÇÃO EM AMBIENTE SIMULADO

Tendo a imagem do Kernel compilada e um *filesystem* adequado, o passo remanescente para poder simular um sistema de Linux simples é preparar o QEMU para execução. Para adequar o ambiente de simulação à arquitetura AArch64, executa-se o seguinte comando a partir do diretório raiz do QEMU (a opção target-list indica qual a arquitetura que se deseja emular):

```
./configure --target-list=aarch64-softmmu
```

Para se iniciar a simulação, é necessário rodar o seguinte comando:

```
./qemu/qemu.git/aarch64-softmmu/qemu-system-aarch64 -cpu cortex-a53 -machine
type=virt -nographic -kernel ./images/Image -initrd ./images/rootfs.cpio
```

Seleciona-se o Cortex-a53 como CPU pois ele é uma implementação da arquitetura ARMv8. Os demais argumentos estão detalhados na seção 4.

Iniciada a simulação, são mostradas no terminal diversas mensagens de descobrimento e inicialização de dispositivos: memória RAM, temporizadores, interfaces seriais, portas de acesso à rede, dentre outros.

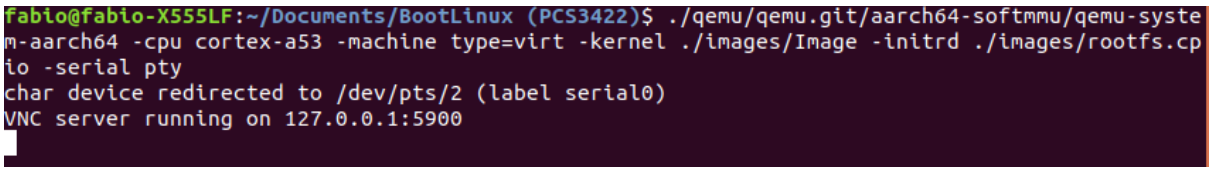
Depois das mensagens iniciais, o controle é transferido ao usuário, para que o mesmo insira suas credenciais. Como definido na seção 4 o usuário é *'root'*. A partir deste momento, o usuário pode navegar através do *filesystem* gerado e executar funções simples como: navegar até um determinado diretório (cd nome_do_diretório) ou subir um nível (cd /.), listar todo o conteúdo de um diretório (ls), criar uma nova pasta (mkdir nome_da_pasta); dentre outras opções.

Existe também a opção de emular uma comunicação serial com uma porta do computador host (usualmente nomeada TTYS# ou TTYUSB#), que é como seria feita a comunicação serial com a placa real onde se implementaria o projeto ou uma porta serial virtual. Para emular uma porta serial virtual, executa-se o comando:

```
./qemu/qemu.git/aarch64-softmmu/qemu-system-aarch64 -cpu cortex-a53 -machine
type=virt -kernel ./images/Image -initrd ./images/rootfs.cpio -serial pty
```

A execução desse comando gera uma resposta como a da Figura 3:

Figura 3 - Redirecionando a saída do QEMU para uma serial virtual



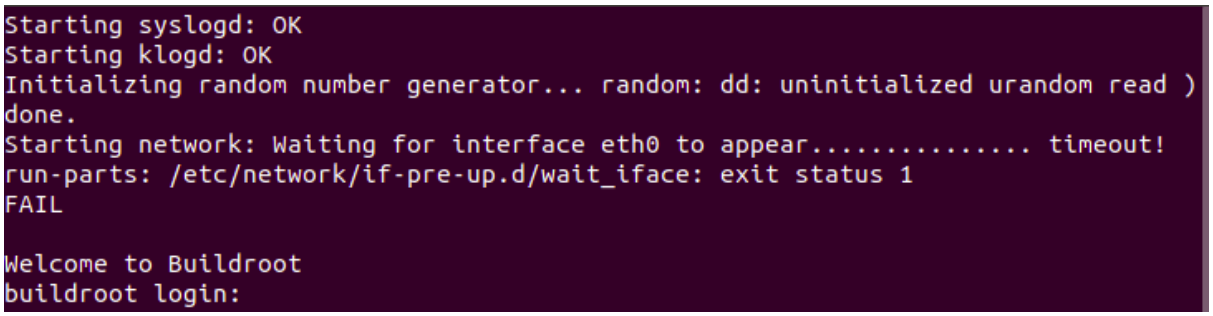
Fonte: elaborado pelos autores com o software QEMU

Para se visualizar o trânsito de mensagens nessa porta serial, pode-se utilizar qualquer terminal compatível, mas a solução adotada foi com o Minicom. Para configurar corretamente o minicom, é preciso executar o seguinte comando:

sudo minicom -s

Seleciona-se a opção ‘Serial Port Setup’ e altera-se as seguintes configurações: Serial Device: /dev/pts/N (onde N pode variar, mas é observável na figura acima) e Bps/Par/Bits: 9600 8N1. Uma vez que a configuração for realizada, escolhe-se a opção ‘Save Setup as dfl’ e seleciona-se ‘Exit’, que deve iniciar a comunicação imediatamente. A saída no terminal deve se assemelhar à Figura 4:

Figura 4 - Saída do terminal virtual durante o boot



Fonte: elaborado pelos autores com o software QEMU

As mensagens iniciais de boot não são visualizáveis porque o Minicom possui um pequeno atraso. É importante iniciar primeiro a simulação e depois rodar o Minicom, pois como a porta é virtual, ela não é instanciada até que seja rodado o QEMU.

7 ADAPTAÇÕES PARA REALIZAR O BOOT NO HARDWARE REAL

O QEMU é extremamente útil no sentido de que permite emular uma máquina ARM sem a necessidade de possuí-la fisicamente. A desvantagem é que sua utilização esconde diversos aspectos de implementação do usuário final, que são de crucial entendimento ao tentar fazer o boot de Linux numa FPGA, onde todos os elementos são descritos do zero.

A opção -machine type=virt utilizada na seção 6 emula recursos que não estão presentes por padrão quando se implementa um sistema embarcado numa FPGA, tais como: interfaces seriais, memórias flash, um bootloader e outros.

As adaptações necessárias podem ser divididas em 3 frentes: fazer o *build* tendo como *target* o LEGv8, compatibilizar o sistema com a interface serial desenvolvida pelo outro grupo dessa disciplina e integrar um bootloader apropriado.

7.1 AS DIFERENÇAS ENTRE O LEGV8 E O ARMV8

O ISA LEGv8 é um subconjunto das instruções ARMv8 idealizado por D. A. Patterson e J. L. Henessy para servir de exemplo didático em sua obra “Computer Organization and Design ARM Edition: The Hardware Software Interface”. Por esse motivo, não existe nenhuma toolchain capaz de gerar código para processadores do tipo LEGv8.

Com a toolchain utilizada para realizar o build, também é possível visualizar a quais as instruções correspondem os binários gerados através do disassembler. Para se mostrar as instruções utilizadas pela toolchain e não implementadas no LEGv8, realizou-se o disassembly da imagem gerada do Kernel.

A Figura 5 mostra um trecho do arquivo mencionado no parágrafo anterior, destacando duas instruções não pertencentes ao LEGv8:

Figura 5 - Instruções não implementadas no LEGv8

```
980044:       b0001240        adrp    x0, 0xbc9000
980048:       90001281        adrp    x1, 0xbd0000
98004c:       cb000021        sub     x1, x1, x0
980050:       97da746c        bl      0x1d200
980054:       b0001240        adrp    x0, 0xbc9000
980058:       90001281        adrp    x1, 0xbd0000
98005c:       cb000021        sub     x1, x1, x0
980060:       a8817c1f        stp     xzr, xzr, [x0], #16
980064:       a8817c1f        stp     xzr, xzr, [x0], #16
980068:       a8817c1f        stp     xzr, xzr, [x0], #16
98006c:       a8817c1f        stp     xzr, xzr, [x0], #16
980070:       f1010021        subs    x1, x1, #0x40
980074:       54ffff61        b.ne    0x980060  // b.any
980078:       d280e227        mov     x7, #0x711                      // #1809
98007c:       b0001240        adrp    x0, 0xbc9000
980080:       b0ffddc3        adrp    x3, 0x539000
```

Fonte: elaborado pelos autores através da execução do disassembler

Seria necessário, então, substituir as instruções não implementadas por outras equivalentes, quando possível; ou removê-las. Esse trabalho fica a cargo de outro grupo, responsável por criar uma toolchain compatível com o LEGv8. Uma vez munidos dessa toolchain customizada, seria necessário fazer o build do Kernel novamente.

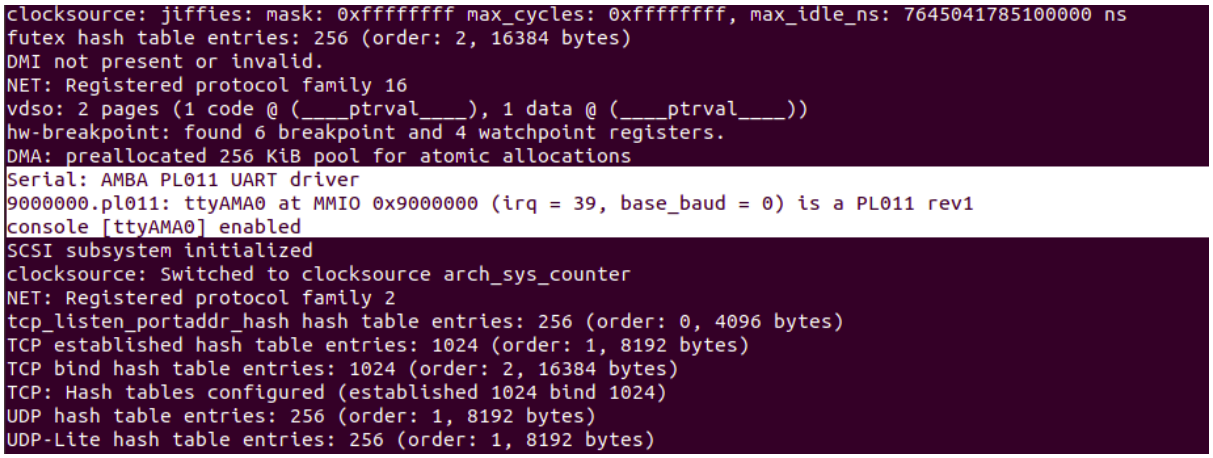
7.2 A INTERFACE SERIAL

Sem que haja uma interface serial, é impossível se comunicar com o sistema para executar comandos através de um terminal, como foi feito na seção 6. O QEMU tem sua própria maneira de virtualizar uma comunicação serial, mas a implementação real da comunicação requer um maior trabalho.

Para que a troca de informações seja possível, seriam necessários uma interface serial e um barramento compatíveis. Além disso, seus respectivos drivers devem estar devidamente compilados (com a toolchain mencionado no tópico 7.1) e instalados na máquina.

A questão é: qual interface serial escolher, como implementá-la e configurá-la? Analisando-se as mensagens de boot (figura 6), observa-se que a comunicação serial emulada é feita através do periférico nomeado “AMBA PL011 UART”.

Figura 6 - Serial usada no ambiente simulado



Fonte: elaborado pelos autores com o software QEMU

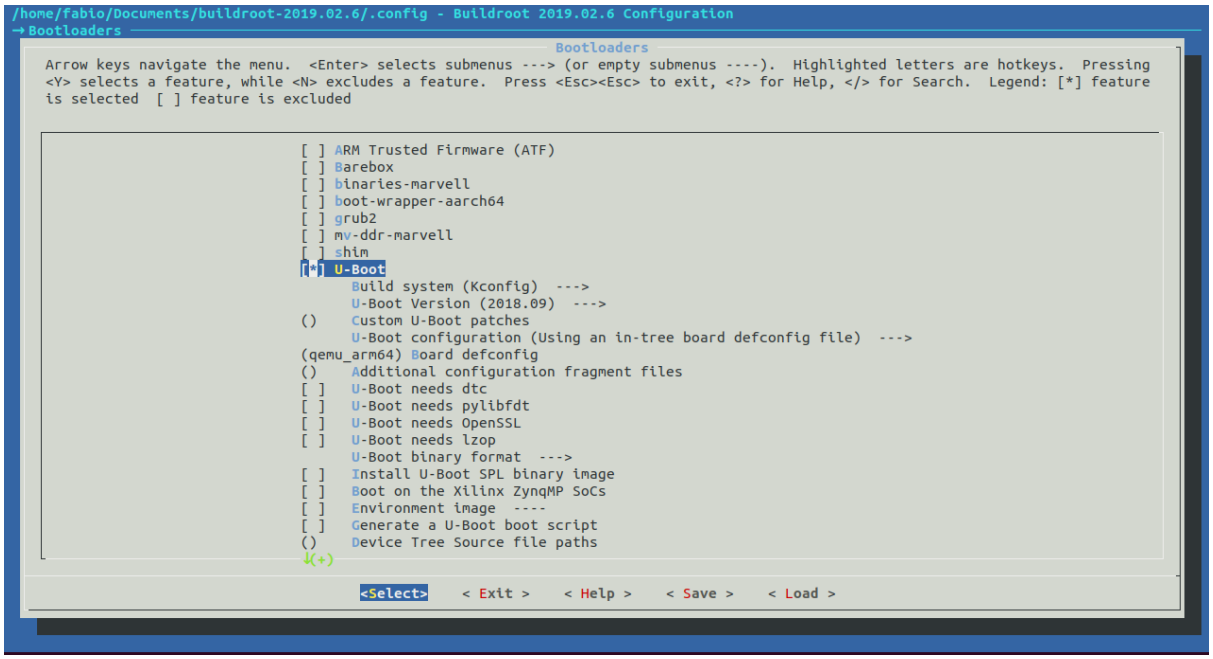
Um dos grupos foi responsável por desenvolver uma interface serial compatível com AMBA, uma arquitetura de barramento de padrão aberto criada pela ARM, que segue justamente as especificações da serial emulada. Ou seja, munidos da implementação da interface serial, bastaria mapeá-la ao endereço 0x90000000 como é feito no ambiente simulado e assume-se que a comunicação serial funcionaria corretamente.

7.3 BOOTLOADER

Para que a inicialização no QEMU funcione corretamente, é necessário passar como parâmetros apenas as imagens do Kernel e do root filesystem e deixar que o sistema inicialize normalmente. O que ocorre é que o QEMU insere no endereço 0x0 da memória emulada uma imagem do u-boot e copia também imagens do Kernel e do RootFS na RAM. O u-boot é um bootloader bastante popular para aplicações embarcadas e possui algumas peculiaridades que serão explicadas posteriormente nesta seção.

É necessário imitar o comportamento descrito no parágrafo anterior no sistema real. O primeiro passo é fazer o build do u-boot. Essa opção está disponível no próprio Buildroot. Para gerar a imagem do u-Boot, basta selecionar a opção Bootloader > U-Boot e preencher a opção Board defconfig com o valor ‘qemu_arm64’, como mostra a Figura 7:

Figura 7 - Configurando o Buildroot para gerar uma imagem do u-Boot



Fonte: elaborado pelos autores com o software Buildroot

A imagem gerada após a execução do comando make é a u-boot.bin e foi colocada no diretório images. O u-boot especifica que na memória devem residir 3 binários para a inicialização correta: a imagem do Kernel, o root file system e a device tree.

A device tree deve ser a mais simples possível, instanciando apenas a CPU, memórias RAM e Flash, timers e a interface serial. A descrição textual da Device Tree utilizada pode ser lida no arquivo /texts/customDeviceTree.txt.

Para que o u-boot reconheça automaticamente a presença dos 3 elementos, é possível inserir um cabeçalho na frente dos 3 binários para identificar seu tipo. A ferramenta utilizada para inserir esses cabeçalhos é nomeada mkimage. Partindo do diretório images, executa-se os seguintes comandos:

mkimage -A arm64 -C none -O linux -T kernel -d Image -a 0x00010000 -e 0x00010000 zImage.uimg

mkimage -A arm64 -C none -O linux -T ramdisk -d rootfs.cpio -a 0x00800000 -e 0x00800000 rootfs.uimg

mkimage -A arm64 -C none -O linux -T flat_dt -d customDeviceTree.dtb -a 0 -e 0 deviceTree.bin

A Tabela 1 mostra o significado de cada argumento utilizado:

Tabela 1 - Utilização do mkimage e significado dos seus argumentos

Argumento	Descrição
-A	Arquitetura target (ARM64, também nomeada AArch64)
-C	O método de compressão utilizado, que é nehum
-O	O sistema operacional que pretende-se inicializar com o u-Boot
-T	Tipo da imagem onde o cabeçalho está sendo inserido: kernel para o Kernel, ramdisk para o sistema de arquivos e flat_dt para a device tree
-d	Nome do source, isto é, a imagem do onde se pretende usar o conteúdo
-a	Endereço de load
-e	Entry point
Último elemento	Nome do arquivo que se pretende gerar, com o cabeçalho inserido no início

Fonte: Utilização do mkimage (referência 28)

Serão gerados, portanto, 3 novos arquivos, nomeados: zImage.uimg, rootfs.uimg e deviceTree.bin. Ainda resta a tarefa de juntar todos os arquivos num só binário. Isso pode ser atingido utilizando-se o comando dd, que junta todos os arquivos num só binário, que foi nomeado flash.bin. Para concatenar todos os arquivos, executa-se os seguintes comandos:

dd if=/dev/zero of=flash.bin bs=1 count=24M

dd if=u-boot of=flash.bin conv=notrunc bs=1 seek=1M

dd if=zImage.uimg of=flash.bin conv=notrunc bs=1 seek=8M

dd if=rootfs.uimg of=flash.bin conv=notrunc bs=1 seek=16M

dd if=rootfs.uimg of=deviceTree.bin conv=notrunc bs=1 seek=24M

Nesse arquivo também estão expressos os endereços de início e término do u-boot, Kernel, sistema de arquivos e device tree.

8 DEPLOY

Por fim, como última etapa temos o Deploy do boot do sistema operacional adaptado, para a sua realização é imprescindível os seguintes pré-requisitos:

1. Placa da linha Cyclone V que inclua entrada para SD Card (como as disponíveis no laboratório digital)
2. Micro SD Card com 4GB ou mais;
3. Um computador host com terminal serial e um espaço para Micro SD Card ou um editor de Micro SD Card.

Como primeira etapa do Deploy, o SD Card necessário para o boot será carregado com um binário no formato img. Para isto, é inserido o SD Card no leitor após determinar o dispositivo ao mesmo por meio do comando:

```
cat /proc/partitions
```

Após este primeiro passo é utilizado o comando *dd* (funcionalidade para criar imagens em discos flexíveis) para escrever a imagem no Card a partir do comando abaixo, tendo como o dispositivo associado */dev/sdx.*

```
sudo dd if=flash.bin of=/dev/sdx bs=1M
```

Por fim, a partir do comando *sync* é enviado as alterações para o SD Card:

```
sudo sync
```

Com o SD Card pronto para ser utilizado, como próximo passo será necessário realizar a configuração da conexão serial na placa, para que o computador host veja a placa como uma porta virtual serial e a distribuição Linux seja construída em drivers para o USB da placa, desse modo nenhuma instalação de drivers se revela necessária. Abaixo é detalhado a configuração apresentada:

A porta virtual serial normalmente é nomeada como /dev/ttyUSB0. A fim de determinar o nome do dispositivo associado com a porta virtual serial do computador host conecte o cabo mini USB da placa no Computador, isto habilita a comunicação entre a placa e o computador e use o comando /dev/ttyUSB novamente para determinar o novo dispositivo serial USB.

Para Linux a aplicação minicom pode ser utilizado para o fim apresentado acima, sendo que a sua configuração é descrita da seguinte forma:

sudo minicom -s

Sendo utilizadas as configurações a seguir:

Serial Device: /dev/ttyUSB0

Bps/Par/Bits: 115200 8N1

Hardware Flow Control: No

Software Flow Control: No

Por fim, Linux deve ser bootado na placa, para isto segue-se o passo a passo abaixo:

1. Inicie o terminal serial;
2. Ligue a placa;
3. Com a memória DDR da placa calibrada, a resposta será como da Figura 4 da seção 6
4. Por fim logue usando ‘root’ como username e o sistema está pronto para ser utilizado

9 CONCLUSÕES

Viabilizar o boot do Linux num sistema embarcado construído do zero, componente por componente, é um processo que requer amplo conhecimento de software e hardware, pois parte dos programas desenvolvidos são específicos para certas arquiteturas ou componentes.

Da parte de software, foi necessário estudar como se dá o processo de boot num sistema embarcado para esquematizar uma solução compatível com as restrições impostas, especialmente a ISA do LEGv8; diferenciando o ambiente simulado (que é de uso comercial e, portanto implementa muito mais *features*) e o real, que é limitado, visto o escopo deste projeto e dos outros realizados pela turma.

Da parte de hardware, foi necessário entender como se estrutura uma device tree, sua importância no processo de boot e quais os componentes a serem declarados para o sistema. Além disso, foi necessário definir uma interface serial compatível com aquela desenvolvida por outros grupos da turma e emular seu funcionamento.

De modo geral, os problemas de implementação de um Linux para o sistema embarcado proposto pela disciplina (baseado num processador LEGv8) foram resolvidos primeiro de maneira individual e depois integrados. A implementação física não foi possível devido às limitações de interdependências entre projetos de diferentes grupos, mas uma solução virtualizada foi apresentada e desenhou-se uma solução de integração com os componentes físicos desenvolvidos pelos outros grupos da disciplina.

10 REFERÊNCIAS BIBLIOGRÁFICAS

[1] ARM. **Arm Architecture Reference Manual.** Disponível em: <https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.135441905.1187878141.1571147300-48504291.1570537975>. Acesso em 11 nov. 2019.

[2] BEYOND LINUX FROM SCRATCH. **About initramfs.** Disponível em: <http://www.linuxfromscratch.org/blfs/view/svn/postlfs/initramfs.html>. Acesso em 11 nov. 2019.

[3] BUILDROOT. **The Buildroot user manual.** Disponível em: <https://www.buildroot.org/downloads/manual/manual.html>. Acesso em 11 nov. 2019.

[4] BUSYBOX. **Frequently Asked Questions.** Disponível em: <https://busybox.net/FAQ.html>. Acesso em 11 nov. 2019.

[5] DEACON, W. **Booting AArch64 Linux.** 2012. Disponível em: <https://www.kernel.org/doc/Documentation/arm64/booting.txt>. Acesso em 11 nov. 2019.

[6] ELINUX. **Device Tree What It Is.** Disponível em: <https://elinux.org/Device_Tree_What_It_Is>. Acesso em 11 nov. 2019.

[7] ELINUX. **Toolchains**. Disponível em: <https://elinux.org/Toolchains>. Acesso em 11 nov. 2019.

[8] FREESCALE SEMICONDUCTOR. **Introduction to Device Trees.** 2015. Disponível em: <https://www.nxp.com/docs/en/application-note/AN5125.pdf>. Acesso em 11 nov. 2019.

[9] GITHUB. **U-Boot on QEMU's 'virt' machine on ARM & AArch64.** Disponível em: <https://github.com/ARM-software/u-boot/blob/master/doc/README.qemu-arm>. Acesso em 11 nov. 2019.

[10] HOW TO GEEK. **What is the Linux Kernel and What Does It Do?** Disponível em: <https://www.howtogeek.com/howto/31632/what-is-the-linux-kernel-and-what-does-it-do/>. Acesso em 11 nov. 2019.

[11] JONES, M. **Inside the Linux boot process.** IBM Developer. 2006. Disponível em: <https://developer.ibm.com/articles/l-linuxboot/>. Acesso em 11 nov. 2019.

[12] LINARO. **The Linaro Binary Toolchain Aarch64 Linux.** 2018. Disponível em: <https://releases.linaro.org/components/toolchain/binaries/latest-5/aarch64-linux-gnu/>. Acesso em 11 nov. 2019.

[13] LINUX FOUNDATION. **Device Tree for Dummies.** Disponível em: <http://events17.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies_0.pdf>. Acesso em 11 nov. 2019.

[14] LINUX. **What is Linux.** Disponível em: <https://www.linux.com/what-is-linux/>. Acesso em 11 nov. 2019.

[15] PATTERSON, D. A. E HENNESSY, J. L**. Computer Organization and Design ARM Edition: The Hardware Software Interface.** The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2016

[16] QEMU WIKI. **Main Page.** Disponível em: <https://wiki.qemu.org/Main_Page>. Acesso em 11 nov. 2019.

[17] SABBAGH, M. H. **What Are the Components of a Linux Distribution?** Foss Post. 2006. Disponível em: <https://fosspost.org/education/what-are-the-components-of-a-linux-distribution>. Acesso em 11 nov. 2019.

[18] SINGH, K. **An overview of Linux Boot Process for Embedded Systems.** Embedded Related. 2008. Disponível em: <https://www.embeddedrelated.com/showarticle/59.php>. Acesso em 11 nov. 2019.

[19] SUHANKO, D**. Introdução a sistemas embarcados – Parte 1.** Do bit Ao Byte. Disponível em: <https://www.dobitaobyte.com.br/introducao-sistemas-embarcados-parte-1/>. Acesso em 11 nov. 2019.

[20] TECHDIFFERENCES. **Difference Between Kernel and Operating System.** 2006. Disponível em: <https://techdifferences.com/difference-between-kernel-and-operating-system.html>. Acesso em 11 nov. 2019.

[21] UNIVERSITY OF MICHIGAN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE. **LEGv8 Reference Data.** Disponível em: <http://www.eecs.umich.edu/courses/eecs370/eecs370.f19/resources/materials/ARM-v8-Quick-Reference-Guide.pdf>. Acesso em 11 nov. 2019.

[22] UNIVERSITY OF WASHINGTON COMPUTER SCIENCE & ENGINEERING. **ARMv8 A64 Quick Reference.** Disponível em: <https://courses.cs.washington .edu/courses/cse469/18wi/Materials/arm64.pdf>. Acesso em 11 nov. 2019.

[23] WEIL, S**. QEMU version 4.1.0 User Documentation.** Disponível em: <https://qemu.weilnetz.de/doc/qemu-doc.html>. Acesso em 11 nov. 2019.

[24] WIKIPEDIA. **Linux (núcleo).** Disponível em: <https://pt.wikipedia.org/wiki/Linux_(núcleo)>. Acesso em 11 nov. 2019.

[25] WIKIPEDIA. **Linux.** Disponível em: <https://en.wikipedia.org/wiki/Linux>. Acesso em 11 nov. 2019.

[26] WIKIPEDIA. **Núcleo (sistema operacional).** Disponível em: <https://pt.wikipedia.org/wiki/N%C3%BAcleo_(sistema_operacional)>. Acesso em 11 nov. 2019.

[27] WIKIPEDIA. **QEMU.** Disponível em: <https://en.wikipedia.org/wiki/QEMU>. Acesso em 11 nov. 2019.

[28] IWAMATSU, Nobuhiro. **Mkimage(1): make image for U-boot.** Disponível em: <https://linux.die.net/man/1/mkimage> Acesso em 11 nov. 2019.