Apresentação da infraestrutura ARM para MC404

Rafael Auler Gabriel Krisman Bertazi Guilherme Piccoli

9 de abril de 2014

1 Sugestão para configurar seu ambiente de programação ARM

Esta seção apresenta uma guia rápido para a configuração do seu ambiente de trabalho. Você não necessariamente precisa seguir exatamente esses passos, mas eles oferecem um bom ponto de partida. Note que as seções a seguir detalham o funcionamento do simulador e apresentam mais opções de parâmetros, etc - assim, é recomendavel a leitura integral desse documento.

O primeiro passo consiste em configurar uma pasta para o seu projeto em linguagem de montagem ARM. Crie uma pasta para organizar estes arquivos:

```
$ cd ~
$ mkdir mc404
$ cd mc404
```

Agora copie o *script* para configuração da variável de ambiente automaticamente, por conveniência:

```
$ cp /home/specg12-1/mc404/simulador/set_path.sh .
$ . set_path.sh
```

A última linha (comando `. set_path.sh`) orienta o *shell* do Linux para executar os comandos contidos no arquivo `set_path.sh`. Da próxima vez, você só irá precisar executar a última linha para ter acesso ao ambiente ARM. Agora você já pode executar diversas ferramentas para programação ARM - consulte a Tabela 1 para conhecer vários exemplos.

Função	**Comando**	**Exemplo de uso**
Montador	`arm-eabi-as`	`arm-eabi-as -g input.s -o output.o`
Ligador	`arm-eabi-ld`	`arm-eabi-ld -g input.o -o output`
Investiga o conteúdo de um binário	`arm-eabi-readelf`	`arm-eabi-readelf -a input`
Desmontador	`arm-eabi-objdump`	`arm-eabi-objdump -S input`
Imprime os símbolos definidos em um arquivo objeto	`arm-eabi-nm`	`arm-eabi-nm input`
Elimina símbolos e informações de depuração de um objeto	`arm-eabi-strip`	`arm-eabi-strip input`
Empacota vários arquivos objetos .o em uma biblioteca .a	`arm-eabi-ar`	`arm-eabi-ar rcs lib.a input1.o input2.o`
Simulador	`arm-sim`	`arm-sim --rom=bootstrap_file --sd=sd_card_image`
Simulador que imprime passo a passo cada instrução executada	`arm-sim`	`arm-sim --rom=bootstrap_file --sd=sd_card_image --debug=core`
Simulador que aguarda conexão do GDB	`arm-sim`	`arm-sim --rom=bootstrap_file --sd=sd_card_image -g`
GDB para conectar em um simulador	`arm-eabi-gdb`	`arm-eabi-gdb programa`

Tabela 1: Vários programas do pacote GNU binutils para ARM e simuladores ARM.

Agora crie uma pasta para um projeto "Hello world". Este será seu primeiro programa para ARM.

```
$ mkdir helloworld
$ cd helloworld
$ editor-favorito Makefile
```

Organize seu projeto com um Makefile que deve chamar o montador ARM para transformar arquivos .s (*assembly language files*, ou arquivos em linguagem de montagem ARM) em arquivos .o (objeto relocável). Também crie regras para que os arquivos .o possam ser ligados pelo ligador ARM para finalmente criar um executável final, que será o resultado do processo de produção do binário do seu primeiro programa ARM. Veja este exemplo[1]:

```
all: mkimage

mkimage: helloworld
        #Generate SD card image.
        mksd.sh --so /home/specg12-1/mc404/simulador/dummyos.elf --user helloworld

helloworld: helloworld.o
        #Linking helloworld.o
        arm-eabi-ld helloworld.o -o helloworld -Ttext=0x77802000 -Tdata=0x77802050

%.o : %.s
        #Assembling $*.s
        arm-eabi-as -g $< -o $*.o

dump: helloworld
        arm-eabi-objdump -S helloworld

readelf: helloworld
        arm-eabi-readelf -a helloworld

simulate_only: disk.img
        arm-sim --rom=/home/specg12-1/mc404/simulador/dumboot.bin --sd=disk.img

gdbtarget: disk.img
        arm-sim --rom=/home/specg12-1/mc404/simulador/dumboot.bin --sd=disk.img -g

gdbhost: helloworld
        arm-eabi-gdb helloworld

clean:
        rm -f helloworld helloworld.o
```

O programa de exemplo que iremos criar é um programa de nível de usuário. Um programa de nível de usuário depende de um sistema operacional para executar, por exemplo, para poder ler dados da entrada e escrever dados na saída. O simulador disponibilizado imita o comportamento da placa i.MX53, que contém um sistema ARM moderno (processador ARM Cortex A8 de 1GHz com vários recursos em hardware, como módulo de criptografia, vídeo, entre outros, dentro do mesmo *chip*, em uma configuração conhecida como SoC ou *System on a Chip*). Entretanto, não podemos explorar todos os recursos do processador ARM no modo usuário, pois algumas instruções só estão disponíveis ao sistema operacional quando o processador está no modo de sistema. Por esse motivo, não utilizaremos um sistema operacional. Mas, para rodar um programa helloworld para imprimir na tela, utilizaremos um sistema operacional simplificado, chamado DummyOS, que comunica-se com o módulo *hardware* UART da placa i.MX53 para escrever dados na tela. Dessa maneira, observe que o Makefile, após ligar o seu programa de usuário, utiliza como entrada também o código do DummyOS em dummyos.elf, para gerar a imagem do cartão SD que o simulador carrega. Ainda, passamos os parâmetros `-Ttext=<address> e -Tdata=<address>` para orientar o *linker* a colocar o nosso código no endereço correto de memória, que é onde o sistema operacional DummyOS espera que ele esteja.

O simulador do ARM imita parcialmente o comportamento de um sistema real. Assim, quando o simulador é ligado, o *program counter* possui o valor 0x0, que é onde está a primeira instrução a ser executada. Na plataforma real, esta região do espaço de endereçamento é ocupada por uma memória do tipo ROM[2], e nela se encontra o código de *bootstrap*, ou seja, o código que faz a primeira

[1] Caso você for copiar o Makefile do exemplo fornecido, lembre-se de substituir os espaços antes dos comandos por tabulações explícitas. Tabulações são exigidas pelo GNU make para reconhecer os comandos de execução de uma regra.

[2] *Read-only Memory*.

inicialização do sistema e carrega o sistema operacional de um dispositivo de armazenamento, no nosso caso, um cartão SD. O parâmetro `--rom=/home/specg12-1/mc404/simulador/dumboot.bin` passado ao simulador faz com que este carregue o código de *bootstrap* da imagem dumboot.bin na posição 0x0 da memória ROM.

O Dumboot é capaz de carregar a imagem de um cartão SD e prepará-la para ser executada no sistema. Para preparar a imagem do cartão SD você deverá usar a ferramenta mksd.sh, informando o sistema operacional (DummyOS) e o código de usuário que será executado.

Agora crie o arquivo helloworld.s com o seguinte conteúdo:

```
@ Este é um programa ARM de exemplo que imprime
@ uma string na saída padrão utilizando a syscall write

.text                   @inicio do trecho de codigo

        .align 4        @alinha instrucoes de 4 em 4 bytes
        .globl main     @torna o simbolo main visivel fora do arquivo

main:
        mov r0, #1      @copia o valor 1 em r0, indicando que a saida da syscall write sera em stdout
        ldr r1, =string @copia em r1 o endereco da string
        mov r2, #6      @copia em r2 o tamanho da string - note que r0,r1 e r2 sao os argumentos da syscall write

        mov r7, #4      @copia o valor 4 para r7, indicando a escolha da syscall write
        svc 0x0         @chama syscalls - no caso, a instrucao acima indica que a syscall write eh a que sera chamada

        mov r7, #1      @move o valor 1 para r7, indicando a escolha da syscall exit
        svc 0x0


.data                   @inicio do trecho de dados

        .align 4
string: .asciz "MC404\n"  @coloca a string na memoria
```

Em seguida, produza o binário do programa e a imagem do cartão SD com:

```
$ make helloworld # Produz o binário
$ make mksd # Produz a imagem do cartão.
$ make readelf # mostra o conteúdo das seções do binário ELF
$ make dump # chama o desmontador para mostrar o conteúdo montado
```

Para conferir que se trata de um binário ARM, execute:

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, version 1, statically linked, not
stripped
```

O comando *file* identifica a natureza de um arquivo no Linux. Neste caso, o comando confirma que o arquivo helloworld é um binário ELF (*Executable and Linkable Format*), que é o formato de arquivos relocáveis e executáveis utilizado no Linux, 32-bit do tipo LSB (*little-endian*). A ligação é estática, indicando que este arquivo não depende de bibliotecas dinâmicas para sua execução, ou seja, é um executável auto-contido.

2 Utilizando os simuladores ARM

Esta seção introduz as ferramentas necessárias para a simulação dos programas produzidos para o processador ARM. De uma maneira geral, os programadores de *softwares* para dispositivos embarcados, como aqueles que utilizam o processador ARM (por exemplo, celulares[3]), sempre contam com um simulador da plataforma para testar seus programas. A plataforma é o conjunto que compreende o processador, os componentes adicionais de *hardware* e o sistema operacional, se disponível.

[3]No caso de celulares, existe ainda o exemplo da plataforma Android para desenvolvimento em Java. O simulador utilizado é um simulador de ARM que roda todo o sistema operacional Linux e a máquina virtual Java para executar os programas criados.

2.1 Simulador básico

O simulador disponibilizado nesta disciplina simula a plataforma i.MX53 parcialmente. Isto é, alguns componentes de *hardware* disponíveis na placa real não estão no simulador, mas eles não serão necessários na disciplina. Como programa de modo usuário, o seu programa ARM poderá fazer uso de chamadas de sistema para obter dados de entrada e escrever dados na saída e interagir com o usuário. Veja a Tabela 2 para exemplos de chamadas de sistema. Para realizar uma chamada de sistema no ARM, utilize a instrução `svc` (ou `swi`, que é um sinônimo) com o operando 0. Antes de chamar esta instrução, entretanto, você deverá ter colocado em `R7` o número da chamada de sistema. Consulta a Tabela 5 para os números das chamadas de sistema para a plataforma Linux-ARM (o DummyOS implementa apenas a *syscalls* `write` e `exit` segundo a especificação do Linux).

syscall	**Parâmetros**	**Descrição**
`read`	`r0` - descritor do arquivo. `r1` - ponteiro para o *buffer* de leitura. `r2` - número de caracteres a serem lidos.	A syscall `read`[4] lê `r2` *bytes* do arquivo cujo descritor está em `r0` e armazena na área de memória apontada por `r1`. O descritor de número 0 indica a entrada padrão aberta pelo *shell*.
`write`	`r0` - descritor do arquivo. `r1` - ponteiro para o *buffer* de escrita. `r2` - número de caracteres a serem escritos.	A *syscall* `write` escreve `r2` *bytes* que estão na área de memória apontada por `r1` no arquivo cujo descritor está em `r0`. O descritor de número 1 indica a saída padrão (famosa *stdout* em C) aberta pelo *shell*.

Tabela 2: Exemplo de chamadas de sistema do Linux.

No exemplo *Hello World*, a chamada de sistema `write` é chamada para escrever a string `"Hello world.\n"` na saída padrão. O simulador conecta a saída padrão do sistema operacional que roda o simulador (o sistema hospedeiro) na saída do dispositivo de *hardware* UART, que realiza comunicação por cabos seriais. Um exemplo muito comum é o uso de cabos RS-232 para comunicar um computador a um sistema embarcado ARM. Dessa maneira, ao rodar o simulador com este programa, o resultado deve ser a impressão da *string* na saída padrão do *shell* que estiver rodando o simulador. Para testar esse resultado, após produzir o executável final de seu programa ARM utilizando o montador e o ligador conforme descrito na seção anterior, e gerar a imagem do cartão SD, você pode executar o simulador da seguinte forma:

```
$ arm-sim --rom=/home/specg12-1/mc404/simulador/dumboot.bin --sd=disk.img -cycles=100
```

A saída produzida deve incluir uma mensagem de inicialização do SystemC, um simulador de sistemas digitais. Após a mensagem *Starting simulation* e até a mensagem *Simulation finished*, a saída apresentada é produzida pelo seu programa ARM (e pelo DummyOS, que está atendendo a *syscall* `write` para escrever na tela). Note que o parâmetro "`-cycles=100`" é utilizado para especificar quantos ciclos serão simulados. Como este é um simulador de plataforma fiel à realidade, você deve atentar ao fato de que um processador nunca pára de executar código a não ser que desligado. Assim, você pode executar o simulador sem especificar um limite máximo de ciclos para simular, mas ele nunca irá terminar a execução e você poderá encerrá-lo com Ctrl+C. Em um sistema operacional real como o Linux, o processador está sempre executando código de algum programa modo usuário ou está em um laço infinito esperando tarefas aparecerem (o processo de sistema *idle*). No caso do DummyOS, assim que você termina o seu aplicativo com a *syscall exit*, ele entra em um laço infinito aguardando o processador ser reiniciado ou desligado.

Short option	Long option	Description
-g	--enable-gdb	Wait for GDB connection.
-p	--gdb-port=[port]	Define port to wait for GDB connection.
-b	--batch-cycles=[cycles]	Run n+1 processor cycles for each plataform cycle.
-c	--cycles=[cycles]	Run for [cycles] plataform cycles.
-D	--debug=[device]	Activate device depuration mode.
-f	--trace-flow	Activate flow debug mode. (experimental)
-r	--rom=[image]	Load image to ROM device.
-s	--sd=[image]	Load image to SD card device.
-y	--load-sys=[file]	Load ELF image as system code. (deprecated)
-?	--help	Give this help list.
	--usage	Give a short usage message.
-v	--version	Print program version.

Tabela 3: Parâmetros do simulador

Para saber exatamente quantas instruções serão simuladas, multiplique o número de ciclos fornecido no parâmetro `-cycles=` por 100. Ou seja, no exemplo anterior, 10000 instruções serão simuladas. Outros parâmetros interessantes para o simulador de plataforma são apresentados na Tabela 3.

2.2 Simulador com GDB

Muitas vezes não é suficiente a simulação que apresenta apenas a saída do seu programa. Para encontrar erros no programa, é importante a utilização de um método que permita a você conhecer o estado do processador após a execução de cada instrução. Para isso, o programa GNU gdb pode ser utilizado. Em sua utilização para programas nativos da arquitetura Intel, basta executar o comando `gdb nome-do-programa` e, no *shell* do gdb, digitar `run`. Entretanto, o gdb utilizado neste curso é uma versão especial para dispositivos ARM para desenvolvimento em regime de *cross compilation*. Este regime de desenvolvimento é caracterizado quando você produz código para um processador diferente daquele no qual você está utilizando o compilador. Por exemplo, produzir programas para a arquitetura Intel IA-32 utilizando ferramentas no seu Linux para Intel IA-32 é um ambiente de desenvolvimento nativo. Produzir programas para a arquitetura ARM utilizando ferramentas que rodam em um Intel IA-32 (como é o caso da disciplina) é um ambiente de *cross compilation*.

Para utilizar o gdb em um ambiente de *cross compiling*, é necessário conectar o gdb ao simulador, pois o gdb, por si só, não é capaz de executar o programa, uma vez que o gdb está rodando sobre Intel. Para isso, iremos utilizar um parâmetro do simulador ARM que aguarda uma conexão do gdb na porta 5000 antes de iniciar a execução:

```
$ arm-sim --rom=/home/specg12-1/mc404/simulador/dumboot.bin --sd=disk.img -cycles=100 -g
```

O simulador irá inicializar e suspender a execução até que um gdb se conecte pela porta 5000. Em outro *shell*, você deverá executar o gdb para ARM:

```
# abra um novo shell
$ cd ~/mc404        # Acesse sua pasta da disciplina
$ . set_path.sh     # Configura paths
$ cd helloworld     # Acesse a pasta do projeto helloworld
$ arm-eabi-gdb helloworld

GNU gdb 6.4
Copyright 2005 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=arm-eabi".
(gdb)
```

Nesse momento, a string `(gdb)` indica que o *shell* gdb está pronto para receber comandos. Iremos então instruí-lo para se conectar ao simulador:

```
(gdb) target remote localhost:5000
```

Após esse comando, repare, no *shell* em que o simulador está aberto, que a simulação já foi iniciada. O gdb, entretanto, possui total controle sobre a execução do programa, e o interrompe antes que a primeira instrução do programa seja executada. A cada interrupção, o gdb imprime a linha do código em que está. Se o programa foi feito em C, a linha do código em C é exibida. No nosso caso, linhas do código em linguagem de montagem são exibidas. Como você está no código do DummyOS, a primeira instrução é um salto para o tratador do evento *reset*, que ocorre toda vez que o processador é reiniciado ou ligado. Pule rapidamente para o código do seu programa usuário através da configuração de um *breakpoint* na sua função `main`:

```
(gdb) b main
Breakpoint 1 at 0x77802000: file helloworld.s, line 7.
(gdb) c
Continuing.

Breakpoint 1, main () at helloworld.s:7
7 mov r0, #1 @copia o valor 1 em r0, indicando que a saida da syscall write...
Current language:  auto; currently asm

(gdb) si   # si é sinônimo de step instruction, ou próxima instrução
8 ldr r1, =string @copia em r1 o endereço da string
(gdb)      # pressionar enter repete o último comando.
9 mov r2, #6 @copia em r2 o tamanho da string
(gdb)
```

Note que se você iniciou o gdb sem indicar o programa helloworld como parâmetro, o gdb não irá ter carregado os símbolos de depuração e não irá exibir em qual linha do código fonte você está. Ainda, se você não montou o programa helloworld com a opção "-g", o montador também não irá ter incluído no binário final os símbolos de depuração necessários para o gdb. A Tabela 4 contém exemplos de comandos que o *shell* interativo do gdb aceita. O gdb é um *software* livre muito utilizado e também possui um extenso manual para consulta.

Em nosso exemplo, para analisar de perto a execução passo a passo do programa `helloworld`, e até modificar o código em tempo de execução, procede-se com as seguintes etapas:

```
(gdb) target remote localhost:5000
Remote debugging using localhost:5000
0x00000000 in ?? ()
(gdb) b main
Breakpoint 1 at 0x77802000: file helloworld.s, line 7.
(gdb) c
Continuing.

Breakpoint 1, main () at helloworld.s:7
7 mov r0, #1 @copia o valor 1 em r0, indicando que a saida da syscall write...
```

```
Current language:  auto; currently asm
(gdb) si
8 ldr r1, =string @copia em r1 o endereço da string
(gdb) si
9 mov r2, #6 @copia em r2 o tamanho da string
(gdb) info register   #aqui exibimos as informações dos registradores
r0             0x1 1
r1             0x77802050 2004885584
r2             0x1 1
r3             0x53fbc000 1409007616
r4             0x778005e0 2004878816
r5             0x0 0
r6             0x0 0
r7             0x0 0
r8             0x0 0
r9             0x0 0
r10            0x0 0
r11            0x0 0
r12            0x0 0
sp             0x77706000 2003853312
lr             0x7780065c 2004878940
pc             0x77802008 2004885512
fps            0x0 0
cpsr           0x0 0
(gdb) print *(char*) 0x77802050

#exibimos o valor que está no endereço 0x77802050, que é o endereço da string!

$1 = 77 'M'   #letra M
(gdb) set *(char*) 0x77802050='Z'   #trocamos para letra Z
(gdb) si
11 mov r7, #4 @copia o valor 4 para r7, indicando a escolha da syscall write
(gdb) si
12 svc 0x0 @chama syscalls - no caso, a instrucao acima...
(gdb) si
0x778005e8 in ?? ()
(gdb) b helloworld.s:13

#nesse ponto, para evitar percorrer todo o codigo da syscall write do DummyOS,
# atribuimos um novo breakpoint

Breakpoint 2 at 0x77802014: file helloworld.s, line 13.
(gdb) c
Continuing.

Breakpoint 2, main () at helloworld.s:14
14 mov r7, #1 @move o valor 1 para r7, indicando a escolha da syscall exit
(gdb)   #nesse momento, veja a saida modificada do seu programa, no outro shell
```

A saída do programa no simulador, com a memória modificada com a intervenção do gdb, pode ser vista no outro *shell*! Pode ser que o DummyOS não tenha terminado de escrever a frase na tela. Nesse caso, execute mais algumas instruções com `continue` até que as interrupções de escrita sejam completamente atendidas e o texto possa ser visualizado na tela.

Experimente também iniciar o simulador com o parâmetro `--debug=core` para solicitar que, assim

que executar uma instrução, imprima detalhes sobre ela para que possamos visualizar com o maior detalhamento a execução do programa. Repare que a instrução que aparece no console do gdb é a próxima instrução a ser executada, aguardando que você dê o comando `si`. Assim que você executa o comando para pular para próxima instrução, é possível visualizar na tela do simulador a execução da instrução atual em seus detalhes (registradores alterados, endereços de memória acessados etc.). Um bom modo de se trabalhar com o gdb e o simulador é abrir duas janelas de *shell* e visualizar as duas ao mesmo tempo, uma rodando o simulador e a outra o gdb.

Comando	Descrição
si	Executa a próxima instrução. Se o seu programa foi feito em C, use *next*, pois uma linha de código C pode equivaler a várias instruções do processador.
info register	Imprime todos os registradores do processador ARM.
print	Imprime expressões. Pode ser útil para vasculhar o conteúdo da memória do processador ARM. Exemplo: `print *(char*)0x8014` imprime o *byte* que está na posição 0x8014 da memória, interpretado como um caractere.
set	Altera o valor de expressões. Pode ser útil para alterar o conteúdo da memória do processador ARM. Exemplo: `set *(char*)0x8014='T'` altera o valor que está em 0x8014 para 'T'.
break	Configura pontos de parada no programa. Grande parte da arte de depurar programas rapidamente consiste em saber determinar os pontos de parada. Se você não utilizar *breakpoints*, estará fadado a assistir instrução por instrução sendo executada até que encontre o código com erro.
continue	Executa o programa até o próximo ponto de parada. Se não houver pontos de parada, executa até o fim.
quit	Encerra o gdb.

Tabela 4: Exemplo de comandos para o gdb

2.3 Simulador verboso[5]

O simulador verboso já foi comentado e imprime na tela detalhes da execução de cada instrução do programa. Tome cuidado com o fato de que, caso seu programa execute muitas instruções (por exemplo, rodar um laço) e você redirecione a saída do simulador para um arquivo para analisá-lo posteriormente, este arquivo pode facilmente atingir *gigabytes* de tamanho. Portanto, recomenda-se utilizar o simulador verboso apenas para programas que executam poucas instruções. Sua execução é similar ao simulador básico:

```
$ arm-sim --rom=dumboot.bin -sd=sd.img --D core,[tzic,uart,gpt,sd,...]
```

A cada instrução executada, o seguinte cabeçalho é impresso:

[5]Leitura opcional.

```
----- PC=0x8000 ----- 0
```

... indicando a posição do *program counter*, que é crucial para que você identifique em que parte do seu programa o simulador está, seguido por um número em notação decimal que indica quantas instruções já foram executadas (0, no caso, indica que esta é a primeira instrução sendo executada, no endereço do ponto de entrada do programa 0x8000).

Cabeçalhos semelhantes serão impressos para outros periféricos simulados. Você pode testá-los passando para o parâmetro -D a lista de periféricos que deseja depurar. A lista de periféricos que podem ser inclusos na opção -D pode ser vista executando o simulador com a opção `--help`. (Cuidado: A saída poderá ficar imensa!)

Caso você não saiba qual instrução do seu programa está no endereço do *program counter* mostrado, lembre-se da ferramenta `objdump` para analisar um binário. Esta ferramenta é um desmontador, capaz de exibir o conteúdo em linguagem de montagem de um binário, relacionando cada instrução a uma posição na memória. Com isso, é possível fazer o mapeamento de endereço de memória para instrução do seu programa:

```
$ arm-eabi-objdump -S helloworld
 helloworld:     file format elf32-littlearm


 Disassembly of section .text:

 00008000 <_start>:
 # write.
         .text
         .align 4
         .globl _start
 _start:
         mov r0, #1
     8000:       e3a00001        mov     r0, #1
```

Assim descobrimos que no endereço 0x8000 estará localizada a instrução `e3a00001`, ou, para humanos, `mov r0, #1`. Esta técnica de desmontagem, contudo, é mais comum quando se tem interesse em analisar um programa cujo código-fonte não está disponível[6] e geralmente não é utilizada por programadores por ser muito trabalhosa.

3 Como funciona o boot do ARM[7]

Sistemas computacionais normalmente executam uma cadeia de programas em sequência para realizar a inicialização da plataforma até conseguir executar um simples programa em nível de usuário, conforme mostrado na Figura 1. Na i.MX53, o primeiro passo do boot acontece a partir do código de *bootstrapping* que fica em um dispositivo de memória ROM, localizado no endereço físico 0x0 do espaço de endereçamento. Quando o processador inicia, o *program counter* aponta para a instrução no endereço 0x0, iniciando assim o processo de *boot*.

No sistema real, o código da ROM decide de qual dispositivo bootar, a partir do valor dos registradores de um periférico chamado *System Reset Controller*. Com o periférico selecionado, o código carrega um programa chamado *bootloader* (U-boot, GRUB, lilo, syslinux,...) de dentro do dispositivo de armazenamento. O *bootloader* é responsável por carregar o *kernel* (por exemplo, o Linux, ou o DummyOS!) do sistema operacional que você vai usar.

[6] A técnica de analisar o código em linguagem de montagem de um programa proprietário é fonte de muita polêmica devido aos direitos autorais. É possível, com isso, inferir o comportamento do programa e alterar, utilizando um editor de binários apropriado, a codificação da instrução e dessa forma modificar o comportamento do programa, por exemplo, para quebrar uma rotina de verificação que conclui se o *software* é legítimo.

[7] Leitura opcional.

O boot do DummyOS é um pouco simplificado pois ele não requer um *bootloader*. Para fazer o boot do DummyOS, usamos na ROM, usamos um programa escrito em linguagem de montagem do ARM chamado Dumboot. Ele faz o *bootstrap* do sistema, carregando o DummyOS do cartão SD e preparando-o para executar. O fluxo de execução do ICBoot pode ser visto na Figura 2.

Quando o DummyOS inicia, ele prepara o ambiente que será usado pelo código de usuário, configurando o sistema de interrupções, inicializando as pilhas, carregando o código de usuário e finalmente entregando o controle da execução para o código usuário, que opera em um modo restrito, sem controle total da máquina.

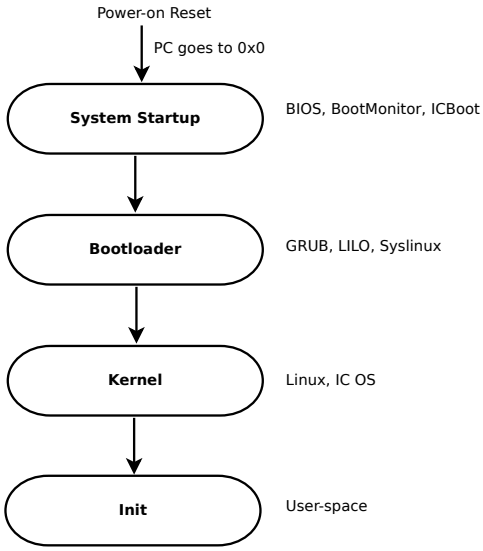

Figura 1: Sequência de boot de um sistema.

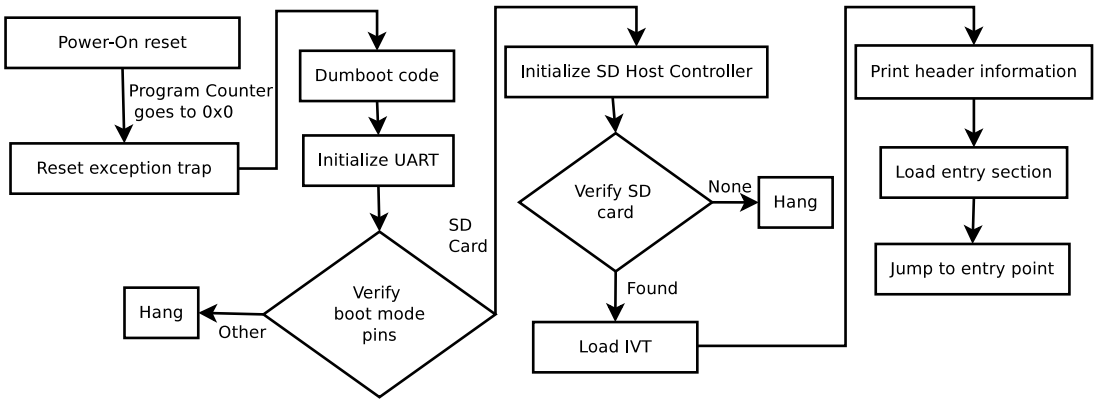

Figura 2: Fluxo de execução do Dumboot.

Nome	Número
exit	1
fork	2
read	3
write	4
open	5
close	6
creat	8
time	13
lseek	19
getpid	20
access	33
kill	37
dup	41
times	43
brk	45
mmap	90
munmap	91
stat	106
lstat	107
fstat	108
uname	122
_llseek	140
readv	145
writev	146
mmap2	192
stat64	195
lstat64	196
fstat64	197
getuid32	199
getgid32	200
geteuid32	201
getegid32	202
fcntl64	221
exit_group	248

Tabela 5: Lista de syscalls da plataforma Linux-ARM. Para consultar os parâmetros, veja a assinatura C com o comando `man 2 nomesyscall` e coloque os parâmetros nos registradores de R0 a R4, da esquerda pra direita. Resultados são retornados em R0.

4 Referências sobre ARM

- Cartão de referência rápida para as instruções ARM: http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf (ignore as instruções que só são suportadas em versões específicas do ARM, como Thumb, Thumb2, v6 e v7)
- Referência da arquitetura ARMv5, utilizada no simulador da disciplina: http://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf (Capítulo A4 descreve instrução por instrução)
- Infocenter ARM: http://infocenter.arm.com/help/index.jsp
- Site com bastante material sobre linguagem de montagem ARM: http://www.coranac.com/tonc/text/asm.htm (ignorar a parte de Thumb).
- Manual do GNU assembler, o montador utilizado: http://sourceware.org/binutils/docs/as/
- Manual do GNU debugger (o gdb, depurador utilizado): http://sourceware.org/gdb/current/onlinedocs/gdb/

Apêndice A: Convenção de Chamadas no ARM

A convenção de chamadas padrão ARM atribui aos 16 registradores da arquitetura as seguintes funções:

- r15: contador de programa;
- r14: registrador de endereço de retorno (a instrução BL, usado em uma chamada de subrotina, armazena o endereço de retorno neste registador);
- r13: ponteiro de pilha;
- r12: registrador para valores intra-procedurais;
- r4 a r11: usados para armazenar variáveis locais;
- r0 a r3: usados para armazenar os valores dos argumentos passados para uma subrotina, e também possuem resultados retornados de uma subrotina.

Se o tipo do valor retornado é muito grande para caber em 128 bits (utilizando os 4 registrados para retorno, de r0 a r3), ou possua um tamanho que não pode ser determinado estaticamente em tempo de compilação, o chamador deve alocar espaço em tempo de execução para o valor retornado e passar o ponteiro para este espaço em r0 para que a função chamada possa trabalhar.

Subrotinas devem preservar o conteúdo de r4 a r11 e o ponteiro de pilha. Tipicamente, salva-se na pilha todos esses registradores ainda no prólogo (início) da função, usa-os como espaço temporário e em seguida, restaura-se os valores da pilha no epílogo (fim) da função. Em particular, subrotinas que chamam outras subrotinas devem salvar o valor de retorno contido no registrador r14 para a pilha antes da chamada. No entanto, tais subrotinas não precisam retornar o valor salvo para r14. Note que basta carregar esse valor diretamente da pilha para r15, o contador de programa, no momento de retornar.

A pilha ARM é decrescente e cheia. Esta convenção de chamadas faz com que uma subrotina ARM típica passe pelas seguintes fases:

- No prólogo, colocar de r4 a r11 na pilha, e colocar o endereço de retorno r14 na pilha também (isto pode ser feito com uma instrução única STM);
- Copiar qualquer argumento passado (r0 a r3) para os registradores temporários locais (r4 a r11);
- Alocar outras variáveis locais para os registradores locais restantes (r4 a r11);

- Fazer cálculos e chamar outras sub-rotinas com BL quando necessário, assumindo que os valores contidos em r0 a r3, r12 e r14 não serão preservados;
- Colocar o resultado em r0;
- No epílogo, resgatar r4 a r11, e r14, o endereço de retorno, da pilha. Entretanto, o endereço de retorno não é resgatado em r14, mas sim diretamente em r15 (isto pode ser feito com uma instrução única LDM e já concretiza o retorno de função, uma vez que o registrador r15, o contador de programa, é sobrescrito).

Apêndice B: Como rodar seus programas ARM em dispositivos com Android[8]

A simulação de um outro processador pode ser muito demorada, mas os benefícios de um simulador geralmente são grandes devido à facilidade de depuração e inspeção de estado que o simulador oferece para o programador. Ainda, pode ser que a plataforma real nem exista ainda. O caso da plataforma ARM-Linux, utilizada nesta disciplina, é especial por envolver um processador muito utilizado comercialmente em sistemas embarcados. Por esse motivo, não é difícil encontrar uma plataforma ARM-Linux real para testar seus programas. O simulador da disciplina simula o chip da plataform Freescale i.MX53, que é uma plataforma física(real) onde você pode executar seus programas e ver o resultado por um cabo RS232 conectado à UART.

Um sistema operacional oferece uma poderosa camada de abstração, permitindo que os mesmos programas rodem sobre diferentes hardwares. O DummyOS atende chamadas POSIX, o padrão do Linux. Isso significa que você pode utilizar o seu programa modo usuário helloworld e testá-lo em um ARM-Linux real, substituindo o DummyOS pelo Linux. Basta mudar o símbolo "`main`"para "`_start`"e remover os parâmetros do linker `-Ttext=0x0` e `dummyos.o` que configuram o DummyOS. Se você possui um celular Android, você já possui um ARM-Linux em funcionamento. O Android foi concebido para hospedar aplicações Java, e não aplicações nativas para o processador ARM escritas em C ou mesmo em linguagem de montagem. Entretanto, podemos ter acesso ao Linux que roda em um celular Android através do programa *android debug bridge*. Nesta seção, vamos detalhar como você pode rodar uma aplicação escrita em linguagem de montagem ARM em um celular Android. Para isso, você deve ter feito as modificações mencionadas no linker, alterado o símbolo `main` para `_start` e gerado o executável helloworld.

Note que o celular pode ter restrições de segurança e impedir que aplicações nativas para ARM (não java) sejam instaladas no celular[9]. Vamos assumir que você possua as ferramentas de desenvolvimento Android para Java mas, ao invés de programas Java, vamos mostrar como um programa nativo Linux-ARM pode ser inserido neste ambiente:

```
- Ligue o simulador ou conecte um celular Android com suporte a debug na porta
- USB. É crucial que o suporte a debug esteja ligado para o adb funcionar.
-
- Importante: Todos os comandos adb aqui são executados com "-e" que conecta
- a um simulador Android rodando na máquina. Mude para -d para redirecionar os
- comandos para um dispositivo USB (seu celular).

$ adb -e push helloworld /data     # instala o helloworld na pasta /data
$ adb -e shell                     # abre um shell ARM-Linux com root

- Você está no shell Linux do seu celular agora

# cd /data
```

[8] Leitura opcional.

[9] Rodar programas escritos em linguagem de montagem ARM em um celular Android é puro hacking e não faz parte do ciclo de desenvolvimento Android, que envolve criar programas seguros que rodam na máquina virtual davikvm Java.

```
# chmod 777 helloworld
# ./helloworld
Hello world.
#
```

Ponto! Você acabou de rodar um programa ARM nativo em seu celular Android.