```
**********************************************************
    5224 Mon Jan 26 17:25:38 2015
new/./README.xbuild
README: tell people how to execute the built files
**********************************************************
```

```
 _ _ _
(_) | |_   _ _ __ ___   ___  ___   ___ _ __   __ _ _ __ _ __ ___
| | | | | | | '_ ` _ \ / _ \/ __| / _ \ '_ \ / _` | '__| '_ ` _ \
| | | | |_| | | | | | | (_) \__ \ | (_) | | | | (_| | |  | | | | | |
|_|_|_|\__,_|_| |_| |_|\___/|___/  \___/|_| |_|  \__,_|_|  |_| |_| |_|
```

Welcome brave fool. Don’t Panic! The writer of this README is an even bigger fool than you could be (even if old Ben wonders if those who follow fools are more foolish).

So here’s where you get started building illumos on ARM.

Step 1) You need to get a build environment set up. There’s the easy way and the fun way.

Easy way:

```
cd $HOME
curl -O https://fingolfin.org/illumos/arm/armtc.tar.gz
pfexec tar xvzf armtc.tar.gz -C /
find /opt/armtc
```

Fun way:

Using a normal i386 on i386 build:

```
cd usr/src
dmake setup
cd cmd/sgs
dmake install

mkdir -p /opt/armtc/lib/amd64 /opt/armtc/usr/bin/amd64
cd /opt/armtc/lib/
ln -s amd64 64
cd ../usr/bin
ln -s amd64 64
```

Then from your proto area, install the following:

- o /usr/bin/ld
- o /usr/bin/amd64/ld
- o /lib/libld.so.4
- o /lib/amd64/libld.so.4
- o /lib/liblddbg.so.4
- o /lib/amd64/liblddb.so.4
- o /lib/libelf.so.1
- o /lib/amd64/libelf.so.1

Now that’s all set go grab illumos-arm-extra (git clone gitosis@zelgadis.fingolfin.org:illumos-arm-extra.git) and build that. You’ll need something like:

```
gmake ARCH=arm STRAP=strap LD_ALTEXEC=/opt/armtc/usr/bin/ld install
```

Once that’s done, you’ll need to fix up the rpath there. so from the root of that workspace run:

```
./tools/setrpath proto-arm/usr/ /opt/armtc/usr/lib:/opt/gcc/4.4.4/lib:/lib:/usr/
```

Finally, you can copy all of that into your arm compiler toolchain directory (use pfexec / sudo as appropriate):



```
cp -r proto-arm/usr /opt/armtc/
```

Step 2) Set up illumos.sh

In a fresh workspace, you’re going to want to set up your illumos.sh with the following:

```
# Enable GCC 4 default
export __GNUC="";
export CW_NO_SHADOW=1
export MACH=arm;
export NATIVE_MACH=i386;
export BUILD64="#"

# Re-set all this MACH-based crud
REF_PROTO_LIST=$PARENT_WS/usr/src/proto_list_${MACH}; export REF_PROTO_LIST
ROOT="$CODEMGR_WS/proto/root_${MACH}"; export ROOT
PARENT_ROOT=$PARENT_WS/proto/root_$MACH; export PARENT_ROOT
PKGARCHIVE="${CODEMGR_WS}/packages/${MACH}/nightly"; export PKGARCHIVE
unset GCC_ROOT GNU_ROOT CW_GCC_DIR
export i386_GCC_ROOT=/opt/gcc/4.4.4
export arm_GCC_ROOT=/opt/armtc/usr
export i386_GNU_ROOT=/usr/sfw
export arm_GNU_ROOT=/opt/armtc/usr/gnu

#
# XXX our gcc isn’t called ./usr/bin/gcc fix it up via CW env vars for now.
#
export CW_arm_GCC=/opt/armtc/usr/bin/arm-pc-solaris2.11-gcc-4.6.3

#
# XXX We need to set CPP to our specific cpp, not the generic /usr/ccs/lib/cpp
# as that’s rather, well, x86.
#
export CPP=/opt/armtc/usr/lib/cpp
export AW_CPP=/opt/armtc/usr/lib/cpp
export LD_ALTEXEC=/opt/armtc/usr/bin/ld
```

Step 3) Start your build engines

Once you’ve done that, you’re doing to need to need to use the *new* bldenv to get started building. For the first time you can go ahead and do something like:

```
cd usr/src
ksh93 ./tools/scripts/bldenv.sh ../../illumos.sh
```

This is really just a bit of a bootstrapping weirdness. Once that’s done you can go ahead and continue on.

As a part of this you should see an important two lines:

```
Cross-building enabled
Targeting arm on i386
```

If you don’t, stop. illumos.sh is not configured correctly.

Once you have that you can get going. Start off with a resounding:

```
dmake setup
```

Following this, you can build the kernel as far as we have it for ARM

```
cd uts; dmake install
```

**You now have a lovely unix and boot_archive pair in bcm2835/unix (Raspberry**

```
Pi) and qvpb/unix (qemu versatilepb).
You now have a lovely unix and boot_archive pair in bcm2835/unix (raspberry
pi) and qvpb/unix (qemu versatilepb).  These should be booted with a kernel
command line mimicing that of the boot_archive (regardless of the path of the
unix you actually provide).  For example: kernel /platform/bcm2835/kernel/unix
-Bconsole=text
```

```
Step 4) Boot

Now that you have the gate built, you can try to boot the kernel.  This is
where things diverge between qemu and the Raspberry Pi.

Booting qemu is very easy:

qemu-system-arm \
        -kernel $PROTO/platform/qvpb/kernel/loader \
        -initrd $PROTO/platform/qvpb/kernel/initrd \
        -machine versatilepb \
        -cpu arm1176 \
        -m 512 \
        -no-reboot \
        -nographic \
        -append 'kernel /platform/qvpb/kernel/unix -Bconsole=uart'

The loader and kernel messages should appear in the same terminal.

Booting on real hardware is a bit more involved.

  a) Create a FAT16 or FAT32 partition on the SD card.  You'll want it to be
     at least 40 MB.

  b) Create a config.txt on the partition:

        gpu_mem=64
        kernel=loader
        initramfs initrd 0x00800000

  c) Create a cmdline.txt on the partition:

        kernel /platform/bcm2835/kernel/unix -Bconsole=uart

  d) Place Raspberry Pi firmware onto the partition.  You can download
     latest firmware from
     https://github.com/raspberrypi/firmware/tree/master/boot. The firmware
     from January 24th, 2015 is known to work.

        0e52c8cdbfd21631746d6fcdc8f2750af39f4287  bootcode.bin
        aba25d795eaddafd5c8ece3de18873b9928eb6f7  fixup_cd.dat
        38e55d60f896738eec30d0ca4f62b68e48e99184  fixup.dat
        4867e6eab84bb4138e812993112b6a05b7930b89  fixup_x.dat
        fa993851acba366d9e37d59a1d9e9de84b19173f  start_cd.elf
        356060e0f44742d8835294a211b812efcac29f66  start.elf
        b7f01f90d995a36c9d765fd1f4d95a5fcdfd7e41  start_x.elf

  e) Copy $PROTO/platform/bcm2835/kernel/{loader,initrd} onto the partition.
#endif /* ! codereview */
```

```
**********************************************************
   80077 Mon Jan 26 17:25:38 2015
new/usr/src/lib/libdisasm/common/dis_arm.c
libdisasm: print push/pop register list in a more human readable form
Instead of just blindly dumping a list of registers, we detect ranges and
print a list of ranges.
libdisasm: print r9 as "fp"
libdisasm: use "push" and "pop" when appropriate
While there is nothing technically wrong with printing the actual
instruction, it is easier on the eyes to just use these common aliases.
libdisasm: remove shouting
There's no need to print everything upper case.
**********************************************************
```

```
   1 /*
   2  * Common Development and Distribution License ("CDDL"), version 1.0.
   3  * You may only use this file in accordance with the terms of version
   4  * 1.0 of the CDDL.
   5  *
   6  * A full copy of the text of the CDDL should have accompanied this
   7  * source.  A copy of the CDDL is also available via the Internet at
   8  * http://www.illumos.org/license/CDDL.
   9  */
  10 /*
  11  * Copyright (c) 2013, Joyent, Inc.  All rights reserved.
  12  * Copyright (c) 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
  13  */

  15 /*
  16  * This provides basic support for disassembling arm instructions. This is
  17  * derived from the arm reference manual (generic), chapter A3 (ARM DDI 0100l).
  18  * All instructions come in as uint32_t's.
  19  */

  21 #include <libdisasm.h>
  22 #include <stdint.h>
  23 #include <stdio.h>
  24 #include <sys/byteorder.h>

  26 #include "libdisasm_impl.h"

  28 extern size_t strlen(const char *);
  29 #endif /* ! codereview */
  30 extern size_t strlcat(char *, const char *, size_t);

  32 /*
  33  * Condition code mask and shift, aka bits 28-31.
  34  */
  35 #define ARM_CC_MASK     0xf0000000
  36 #define ARM_CC_SHIFT    28

  38 /*
  39  * First level of decoding, aka bits 25-27.
  40  */
  41 #define ARM_L1_DEC_MASK 0x0e000000
  42 #define ARM_L1_DEC_SHIFT        25

  44 /*
  45  * Masks and values for the 0b000 l1 group
  46  */
  47 #define ARM_L1_0_B4_MASK        0x00000010
  48 #define ARM_L1_0_B7_MASK        0x00000080
  49 #define ARM_L1_0_OPMASK 0x01800000
  50 #define ARM_L1_0_SPECOP 0x01000000
  51 #define ARM_L1_0_SMASK  0x00100000
  52 #define ARM_L1_0_ELS_MASK       0x00000060
```



```
  54 /*
  55  * Masks and values for the 0b001 l1 group.
  56  */
  57 #define ARM_L1_1_OPMASK 0x01800000
  58 #define ARM_L1_1_SPECOP 0x01000000
  59 #define ARM_L1_1_SMASK  0x00100000
  60 #define ARM_L1_1_UNDEF_MASK     0x00200000

  62 /*
  63  * Masks and values for the 0b011 l1 group
  64  */
  65 #define ARM_L1_3_B4_MASK        0x00000010
  66 #define ARM_L1_3_ARCHUN_MASK    0x01f000f0

  68 /*
  69  * Masks for the 0b111 l1 group
  70  */
  71 #define ARM_L1_7_COPROCMASK     0x00000010
  72 #define ARM_L1_7_SWINTMASK      0x01000000

  74 /*
  75  * Masks for the data processing instructions (dpi)
  76  */
  77 #define ARM_DPI_OPCODE_MASK     0x01e00000
  78 #define ARM_DPI_OPCODE_SHIFT    21
  79 #define ARM_DPI_IBIT_MASK       0x02000000
  80 #define ARM_DPI_SBIT_MASK       0x00100000
  81 #define ARM_DPI_RN_MASK         0x000f0000
  82 #define ARM_DPI_RN_SHIFT        16
  83 #define ARM_DPI_RD_MASK         0x0000f000
  84 #define ARM_DPI_RD_SHIFT        12
  85 #define ARM_DPI_BIT4_MASK       0x00000010

  87 #define ARM_DPI_IMM_ROT_MASK    0x00000f00
  88 #define ARM_DPI_IMM_ROT_SHIFT   8
  89 #define ARM_DPI_IMM_VAL_MASK    0x000000ff

  91 #define ARM_DPI_IMS_SHIMM_MASK  0x00000f80
  92 #define ARM_DPI_IMS_SHIMM_SHIFT 7
  93 #define ARM_DPI_IMS_SHIFT_MASK  0x00000060
  94 #define ARM_DPI_IMS_SHIFT_SHIFT 5
  95 #define ARM_DPI_IMS_RM_MASK     0x0000000f

  97 #define ARM_DPI_REGS_RS_MASK    0x00000f00
  98 #define ARM_DPI_REGS_RS_SHIFT   8
  99 #define ARM_DPI_REGS_SHIFT_MASK 0x00000060
 100 #define ARM_DPI_REGS_SHIFT_SHIFT        5
 101 #define ARM_DPI_REGS_RM_MASK    0x0000000f

 103 /*
 104  * Definitions for the word and byte LDR and STR instructions
 105  */
 106 #define ARM_LS_IBIT_MASK        0x02000000
 107 #define ARM_LS_PBIT_MASK        0x01000000
 108 #define ARM_LS_UBIT_MASK        0x00800000
 109 #define ARM_LS_BBIT_MASK        0x00400000
 110 #define ARM_LS_WBIT_MASK        0x00200000
 111 #define ARM_LS_LBIT_MASK        0x00100000
 112 #define ARM_LS_RN_MASK          0x000f0000
 113 #define ARM_LS_RN_SHIFT         16
 114 #define ARM_LS_RD_MASK          0x0000f000
 115 #define ARM_LS_RD_SHIFT         12

 117 #define ARM_LS_IMM_MASK         0x00000fff

 119 #define ARM_LS_REG_RM_MASK      0x0000000f
```

```
120 #define ARM_LS_REG_NRM_MASK     0x00000ff0

122 #define ARM_LS_SCR_SIMM_MASK    0x00000f80
123 #define ARM_LS_SCR_SIMM_SHIFT   7
124 #define ARM_LS_SCR_SCODE_MASK   0x00000060
125 #define ARM_LS_SCR_SCODE_SHIFT  5
126 #define ARM_LS_SCR_RM_MASK      0x0000000f

128 /*
129  * Masks for the Load and Store multiple instructions.
130  */
131 #define ARM_LSM_PBIT_MASK       0x01000000
132 #define ARM_LSM_UBIT_MASK       0x00800000
133 #define ARM_LSM_SBIT_MASK       0x00400000
134 #define ARM_LSM_WBIT_MASK       0x00200000
135 #define ARM_LSM_LBIT_MASK       0x00100000
136 #define ARM_LSM_RN_MASK         0x000f0000
137 #define ARM_LSM_RN_SHIFT        16
138 #define ARM_LSM_RLIST_MASK      0x0000ffff
139 #define ARM_LSM_ADDR_MASK       0x01800000
140 #define ARM_LSM_ADDR_SHIFT      23

142 /*
143  * Masks for the Extended and Misc. Loads and stores. This is the extension
144  * space from figure A3-5. Most of them are handled by arm_dis_els() with the
145  * exception or swap / swap byte and load/store register exclusive which due to
146  * its nature is handled elsewhere.
147  */
148 #define ARM_ELS_SWAP_MASK       0x01b00000
149 #define ARM_ELS_SWAP_BYTE_MASK  0x00400000
150 #define ARM_ELS_IS_SWAP         0x01000000
151 #define ARM_ELS_EXCL_MASK       0x01800000
152 #define ARM_ELS_PBIT_MASK       0x01000000
153 #define ARM_ELS_UBIT_MASK       0x00800000
154 #define ARM_ELS_IBIT_MASK       0x00400000
155 #define ARM_ELS_WBIT_MASK       0x00200000
156 #define ARM_ELS_LBIT_MASK       0x00100000
157 #define ARM_ELS_SBIT_MASK       0x00000040
158 #define ARM_ELS_HBIT_MASK       0x00000020
159 #define ARM_ELS_RN_MASK         0x000f0000
160 #define ARM_ELS_RN_SHIFT        16
161 #define ARM_ELS_RD_MASK         0x0000f000
162 #define ARM_ELS_RD_SHIFT        12
163 #define ARM_ELS_UP_AM_MASK      0x00000f00
164 #define ARM_ELS_UP_AM_SHIFT     8
165 #define ARM_ELS_LOW_AM_MASK     0x0000000f

167 /*
168  * Multiply instruction extensino space masks and values
169  */
170 #define ARM_EMULT_UNBIT_MASK    0x00400000
171 #define ARM_EMULT_ABIT_MASK     0x00200000
172 #define ARM_EMULT_SBIT_MASK     0x00100000
173 #define ARM_EMULT_RD_MASK       0x000f0000
174 #define ARM_EMULT_RD_SHIFT      16
175 #define ARM_EMULT_RN_MASK       0x0000f000
176 #define ARM_EMULT_RN_SHIFT      12
177 #define ARM_EMULT_RS_MASK       0x00000f00
178 #define ARM_EMULT_RS_SHIFT      8
179 #define ARM_EMULT_RM_MASK       0x0000000f
180 #define ARM_EMULT_MA_MASK       0x0fc00000
181 #define ARM_EMULT_UMA_MASK      0x0ff00000
182 #define ARM_EMULT_UMA_TARG      0x00400000
183 #define ARM_EMULT_MAL_MASK      0x0f800000
184 #define ARM_EMULT_MAL_TARG      0x00800000
```



```
186 /*
187  * Here we have the masks and target values to indicate instructions from the
188  * Control and DSP extension space. There are a bunch of not quite related
189  * instructions, but that's okay. That's how this thing always rolls.
190  *
191  * The ARM_CDSP_STATUS_MASK and TARG do not catch the move immediate to status
192  * register. That's okay because they get handled and separated out in arm_dis.
193  */
194 #define ARM_CDSP_STATUS_MASK    0x0f9000f0
195 #define ARM_CDSP_STATUS_TARG    0x01000000
196 #define ARM_CDSP_BEX_UP_MASK    0x0ff00000      /* Branch/exchg/link instrs */
197 #define ARM_CDSP_BEX_UP_TARG    0x01200000
198 #define ARM_CDSP_BEX_LOW_MASK   0x000000f0
199 #define ARM_CDSP_BEX_NLOW_TARG  0x00000000      /* Here the target is inverse */
200 #define ARM_CDSP_CLZ_MASK       0x0ff000f0      /* Count leading zeros */
201 #define ARM_CDSP_CLZ_TARG       0x01200030
202 #define ARM_CDSP_SAT_MASK       0x0f9000f0      /* Saturating add/subtract */
203 #define ARM_CDSP_SAT_TARG       0x01000050
204 #define ARM_CDSP_BKPT_MASK      0x0ff000f0      /* Software breakpoint */
205 #define ARM_CDSP_BKPT_TARG      0x01200070
206 #define ARM_CDSP_SMUL_MASK      0x0f900090      /* Signed multiplies (type 2) */
207 #define ARM_CDSP_SMUL_TARG      0x01000080

209 #define ARM_CDSP_RN_MASK        0x000f0000
210 #define ARM_CDSP_RN_SHIFT       16
211 #define ARM_CDSP_RD_MASK        0x0000f000
212 #define ARM_CDSP_RD_SHIFT       12
213 #define ARM_CDSP_RS_MASK        0x00000f00
214 #define ARM_CDSP_RS_SHIFT       8
215 #define ARM_CDSP_RM_MASK        0x0000000f

217 #define ARM_CDSP_STATUS_RBIT    0x00400000
218 #define ARM_CDSP_MRS_MASK       0x00300000      /* Ditinguish MRS and MSR */
219 #define ARM_CDSP_MRS_TARG       0x00000000
220 #define ARM_CDSP_MSR_F_MASK     0x000f0000
221 #define ARM_CDSP_MSR_F_SHIFT    16
222 #define ARM_CDSP_MSR_RI_MASK    0x00000f00
223 #define ARM_CDSP_MSR_RI_SHIFT   8
224 #define ARM_CDSP_MSR_IMM_MASK   0x000000ff
225 #define ARM_CDSP_MSR_ISIMM_MASK 0x02000000

227 #define ARM_CDSP_BEX_TYPE_MASK  0x000000f0
228 #define ARM_CDSP_BEX_TYPE_SHIFT 4
229 #define ARM_CDSP_BEX_TYPE_X     1
230 #define ARM_CDSP_BEX_TYPE_J     2
231 #define ARM_CDSP_BEX_TYPE_L     3

233 #define ARM_CDSP_SAT_OP_MASK    0x00600000
234 #define ARM_CDSP_SAT_OP_SHIFT   21

236 #define ARM_CDSP_BKPT_UIMM_MASK 0x000fff00
237 #define ARM_CDSP_BKPT_UIMM_SHIFT        8
238 #define ARM_CDSP_BKPT_LIMM_MASK 0x0000000f

240 #define ARM_CDSP_SMUL_OP_MASK   0x00600000
241 #define ARM_CDSP_SMUL_OP_SHIFT  21
242 #define ARM_CDSP_SMUL_X_MASK    0x00000020
243 #define ARM_CDSP_SMUL_Y_MASK    0x00000040

245 /*
246  * Interrupt
247  */
248 #define ARM_SWI_IMM_MASK        0x00ffffff

250 /*
251  * Branch and Link pieces.
```

```
252  */
253 #define ARM_BRANCH_LBIT_MASK    0x01000000
254 #define ARM_BRANCH_IMM_MASK     0x00ffffff
255 #define ARM_BRANCH_SIGN_MASK    0x00800000
256 #define ARM_BRANCH_POS_SIGN     0x00ffffff
257 #define ARM_BRANCH_NEG_SIGN     0xff000000
258 #define ARM_BRANCH_SHIFT        2

260 /*
261  * Unconditional instructions
262  */
263 #define ARM_UNI_CPS_MASK        0x0ff10010      /* Change processor state */
264 #define ARM_UNI_CPS_TARG        0x01000000
265 #define ARM_UNI_SE_MASK         0x0fff0078      /* Set endianess */
266 #define ARM_UNI_SE_TARG         0x01010000
267 #define ARM_UNI_PLD_MASK        0x0d70f000      /* Cach preload */
268 #define ARM_UNI_PLD_TARG        0x0550f000
269 #define ARM_UNI_SRS_MASK        0x0e5f0f00      /* Save return state */
270 #define ARM_UNI_SRS_TARG        0x084d0500
271 #define ARM_UNI_RFE_MASK        0x0e500f00      /* Return from exception */
272 #define ARM_UNI_RFE_TARG        0x08100a00
273 #define ARM_UNI_BLX_MASK        0x0e000000      /* Branch with Link / Thumb */
274 #define ARM_UNI_BLX_TARG        0x0a000000
275 #define ARM_UNI_CODRT_MASK      0x0fe00000      /* double reg to coproc */
276 #define ARM_UNI_CODRT_TARG      0x0c400000
277 #define ARM_UNI_CORT_MASK       0x0f000010      /* single reg to coproc */
278 #define ARM_UNI_CORT_TARG       0x0e000010
279 #define ARM_UNI_CODP_MASK       0x0f000010      /* coproc data processing */
280 #define ARM_UNI_CODP_TARG       0x0e000000

282 #define ARM_UNI_CPS_IMOD_MASK   0x000c0000
283 #define ARM_UNI_CPS_IMOD_SHIFT  18
284 #define ARM_UNI_CPS_MMOD_MASK   0x00020000
285 #define ARM_UNI_CPS_A_MASK      0x00000100
286 #define ARM_UNI_CPS_I_MASK      0x00000080
287 #define ARM_UNI_CPS_F_MASK      0x00000040
288 #define ARM_UNI_CPS_MODE_MASK   0x0000001f

290 #define ARM_UNI_SE_BE_MASK      0x00000200

292 #define ARM_UNI_SRS_WBIT_MASK   0x00200000
293 #define ARM_UNI_SRS_MODE_MASK   0x0000000f

295 #define ARM_UNI_RFE_WBIT_MASK   0x00200000

297 #define ARM_UNI_BLX_IMM_MASK    0x00ffffff

299 /*
300  * Definitions of the ARM Media instruction extension space.
301  */
302 #define ARM_MEDIA_L1_MASK       0x01800000      /* First level breakdown */
303 #define ARM_MEDIA_L1_SHIFT      23

305 #define ARM_MEDIA_OP1_MASK      0x00700000
306 #define ARM_MEDIA_OP1_SHIFT     20
307 #define ARM_MEDIA_OP2_MASK      0x000000e0
308 #define ARM_MEDIA_OP2_SHIFT     5

310 #define ARM_MEDIA_RN_MASK       0x000f0000
311 #define ARM_MEDIA_RN_SHIFT      16
312 #define ARM_MEDIA_RD_MASK       0x0000f000
313 #define ARM_MEDIA_RD_SHIFT      12
314 #define ARM_MEDIA_RS_MASK       0x00000f00
315 #define ARM_MEDIA_RS_SHIFT      8
316 #define ARM_MEDIA_RM_MASK       0x0000000f
```



```
318 #define ARM_MEDIA_MULT_X_MASK   0x00000020

320 #define ARM_MEDIA_HPACK_MASK    0x00700020      /* Halfword pack */
321 #define ARM_MEDIA_HPACK_TARG    0x00000000
322 #define ARM_MEDIA_WSAT_MASK     0x00200020      /* Word saturate */
323 #define ARM_MEDIA_WSAT_TARG     0x00200000
324 #define ARM_MEDIA_PHSAT_MASK    0x003000e0      /* Parallel halfword saturate */
325 #define ARM_MEDIA_PHSAT_TARG    0x00200020
326 #define ARM_MEDIA_REV_MASK      0x007000e0      /* Byte rev. word */
327 #define ARM_MEDIA_REV_TARG      0x00300020
328 #define ARM_MEDIA_BRPH_MASK     0x007000e0      /* Byte rev. packed halfword */
329 #define ARM_MEDIA_BRPH_TARG     0x003000a0
330 #define ARM_MEDIA_BRSH_MASK     0x007000e0      /* Byte rev. signed halfword */
331 #define ARM_MEDIA_BRSH_TARG     0x007000a0
332 #define ARM_MEDIA_SEL_MASK      0x008000e0      /* Select bytes */
333 #define ARM_MEDIA_SEL_TARG      0x000000a0
334 #define ARM_MEDIA_SZE_MASK      0x000000e0      /* Sign/zero extend */
335 #define ARM_MEDIA_SZE_TARG      0x00000030

337 #define ARM_MEDIA_HPACK_OP_MASK 0x00000040
338 #define ARM_MEDIA_HPACK_SHIFT_MASK      0x00000f80
339 #define ARM_MEDIA_HPACK_SHIFT_IMM       7

341 #define ARM_MEDIA_SAT_U_MASK    0x00400000
342 #define ARM_MEDIA_SAT_IMM_MASK  0x001f0000
343 #define ARM_MEDIA_SAT_IMM_SHIFT 16
344 #define ARM_MEDIA_SAT_SHI_MASK  0x00000f80
345 #define ARM_MEDIA_SAT_SHI_SHIFT 7
346 #define ARM_MEDIA_SAT_STYPE_MASK        0x00000040

348 #define ARM_MEDIA_SZE_S_MASK    0x00400000
349 #define ARM_MEDIA_SZE_OP_MASK   0x00300000
350 #define ARM_MEDIA_SZE_OP_SHIFT  20
351 #define ARM_MEDIA_SZE_ROT_MASK  0x00000c00
352 #define ARM_MEDIA_SZE_ROT_SHIFT 10

354 /*
355  * Definitions for coprocessor instructions
356  */
357 #define ARM_COPROC_RN_MASK      0x000f0000
358 #define ARM_COPROC_RN_SHIFT     16
359 #define ARM_COPROC_RD_MASK      0x0000f000
360 #define ARM_COPROC_RD_SHIFT     12
361 #define ARM_COPROC_RM_MASK      0x0000000f
362 #define ARM_COPROC_NUM_MASK     0x00000f00
363 #define ARM_COPROC_NUM_SHIFT    8

365 #define ARM_COPROC_CDP_OP1_MASK 0x00f00000
366 #define ARM_COPROC_CDP_OP1_SHIFT        20
367 #define ARM_COPROC_CDP_OP2_MASK 0x000000e0
368 #define ARM_COPROC_CDP_OP2_SHIFT        5

370 #define ARM_COPROC_CRT_OP1_MASK 0x00e00000
371 #define ARM_COPROC_CRT_OP1_SHIFT        21
372 #define ARM_COPROC_CRT_OP2_MASK 0x000000e0
373 #define ARM_COPROC_CRT_OP2_SHIFT        5
374 #define ARM_COPROC_CRT_DIR_MASK 0x00100000      /* MCR or MRC */

376 #define ARM_COPROC_DRT_MASK     0x01e00000
377 #define ARM_COPROC_DRT_TARG     0x00400000
378 #define ARM_COPROC_DRT_OP_MASK  0x000000f0
379 #define ARM_COPROC_DRT_OP_SHIFT 4
380 #define ARM_COPROC_DRT_DIR_MASK 0x00100000      /* MCRR or MRRC */

382 #define ARM_COPROC_LS_P_MASK    0x01000000
383 #define ARM_COPROC_LS_U_MASK    0x00800000
```

```
#define ARM_COPROC_LS_N_MASK    0x00400000
#define ARM_COPROC_LS_W_MASK    0x00200000
#define ARM_COPROC_LS_L_MASK    0x00100000
#define ARM_COPROC_LS_IMM_MASK  0x000000ff

/*
 * This is the table of condition codes that instructions might have. Every
 * instruction starts with a four bit code. The last two codes are special.
 * 0b1110 is the always condition. Therefore we leave off its mneomic extension
 * and treat it as the empty string. The condition code 0b1111 takes us to a
 * separate series of encoded instructions and therefore we go elsewhere with
 * them.
 */
static const char *arm_cond_names[] = {
        "eq",           /* Equal */
        "ne",           /* Not Equal */
        "cs/hs",        /* Carry set/unsigned higher or same */
        "cc/lo",        /* Carry clear/unsigned lower */
        "mi",           /* Minus/negative */
        "pl",           /* Plus/positive or zero */
        "vs",           /* Overflow */
        "vc",           /* No overflow */
        "hi",           /* Unsigned higher */
        "ls",           /* Unsigned lower or same */
        "ge",           /* Signed greater than or equal */
        "lt",           /* Signed less than */
        "gt",           /* Signed greater than */
        "le",           /* Signed less than or equal */
        "EQ",           /* Equal */
        "NE",           /* Not Equal */
        "CS/HS",        /* Carry set/unsigned higher or same */
        "CC/LO",        /* Carry clear/unsigned lower */
        "MI",           /* Minus/negative */
        "PL",           /* Plus/positive or zero */
        "VS",           /* Overflow */
        "VC",           /* No overflow */
        "HI",           /* Unsigned higher */
        "LS",           /* Unsigned lower or same */
        "GE",           /* Signed greater than or equal */
        "LT",           /* Signed less than */
        "GT",           /* Signed greater than */
        "LE",           /* Signed less than or equal */
        "",             /* AL - Always (unconditional) */
        NULL            /* Not a condition code */
};
```

*unchanged_portion_omitted_*

```
/*
 * Registers are encoded surprisingly sanely. It's a 4-bit value that indicates
 * which register in question we're working with.
 */
static const char *arm_reg_names[] = {
        "r0",
        "r1",
        "r2",
        "r3",
        "r4",
        "r5",
        "r6",
        "r7",
        "r8",
        "fp",   /* Alt for r9 */
        "r10",
        "r11",
        "ip",   /* Alt for r12 */
        "sp",   /* Alt for r13 */
```



```
        "lr",   /* Alt for r14 */
        "pc"    /* Alt for r15 */
        "R0",
        "R1",
        "R2",
        "R3",
        "R4",
        "R5",
        "R6",
        "R7",
        "R8",
        "R9",
        "R10",
        "R11",
        "IP",   /* Alt for R12 */
        "SP",   /* Alt for R13 */
        "LR",   /* Alt for R14 */
        "PC"    /* Alt for R15 */
};
```

*unchanged_portion_omitted_*

```
/*
 * These are the opcodes for the instructions which are considered data
 * processing instructions.
 */
static const char *arm_dpi_opnames[] = {
        "and",  /* Logical AND */
        "eor",  /* Logical Exclusive OR */
        "sub",  /* Subtract */
        "rsb",  /* Reverse Subtract */
        "add",  /* Add */
        "adc",  /* Add with Carry */
        "sbc",  /* Subtract with Carry */
        "rsc",  /* Reverse Subtract with Carry */
        "tst",  /* Test */
        "teq",  /* Test Equivalence */
        "cmp",  /* Compare */
        "cmn",  /* Compare negated */
        "orr",  /* Logical (inclusive) OR */
        "mov",  /* Move */
        "bic",  /* Bit clear */
        "mvn"   /* Move not */
        "AND",  /* Logical AND */
        "EOR",  /* Logical Exclusive OR */
        "SUB",  /* Subtract */
        "RSB",  /* Reverse Subtract */
        "ADD",  /* Add */
        "ADC",  /* Add with Carry */
        "SBC",  /* Subtract with Carry */
        "RSC",  /* Reverse Subtract with Carry */
        "TST",  /* Test */
        "TEQ",  /* Test Equivalence */
        "CMP",  /* Compare */
        "CMN",  /* Compare negated */
        "ORR",  /* Logical (inclusive) OR */
        "MOV",  /* Move */
        "BIC",  /* Bit clear */
        "MVN"   /* Move not */
};
```

*unchanged_portion_omitted_*

```
const char *arm_dpi_shifts[] = {
        "lsl",  /* Logical shift left */
        "lsr",  /* Logical shift right */
        "asr",  /* Arithmetic shift right */
        "ror",  /* Rotate right */
```

```
546         "rrx"   /* Rotate right with extend. This is a special case of ror */
172         "LSL",  /* Logical shift left */
173         "LSR",  /* Logical shift right */
174         "ASR",  /* Arithmetic shift right */
175         "ROR",  /* Rotate right */
176         "RRX"   /* Rotate right with extend. This is a special case of ROR */
547 };

549 typedef enum arm_dpi_shift_code {
550         DPI_S_LSL,      /* Logical shift left */
551         DPI_S_LSR,      /* Logical shift right */
552         DPI_S_ASR,      /* Arithmetic shift right */
553         DPI_S_ROR,      /* Rotate right */
554         DPI_S_RRX,      /* Rotate right with extend. Special case of ror */
184         DPI_S_RRX,      /* Rotate right with extend. Special case of ROR */
555         DPI_S_NONE      /* No shift code */
556 } arm_dpi_shift_code_t;
```

*unchanged_portion_omitted_*

```
593 /*
594  * This table contains the names of the load store multiple addressing modes.
595  * The P and U bits are supposed to be combined to index into this. You should
596  * do this by doing P << 1 | U.
597  */
598 static const char *arm_lsm_mode_names[] = {
599         "da",
600         "ia",
601         "db",
602         "ib"
229         "DA",
230         "IA",
231         "DB",
232         "IB"
603 };
```

*unchanged_portion_omitted_*

```
629 /*
630  * Names for specific saturating add and subtraction instructions from the
631  * extended control and dsp instructino section.
632  */
633 static const char *arm_cdsp_sat_opnames[] = {
634         "add",
635         "sub",
636         "dadd",
637         "dsub"
264         "ADD",
265         "SUB",
266         "DADD",
267         "DSUB"
638 };

640 static const char *arm_padd_p_names[] = {
641         NULL,   /* 000 */
642         "s",    /* 001 */
643         "q",    /* 010 */
644         "sh",   /* 011 */
272         "S",    /* 001 */
273         "Q",    /* 010 */
274         "SH",   /* 011 */
645         NULL,   /* 100 */
646         "u",    /* 101 */
647         "uq",   /* 110 */
648         "uh",   /* 111 */
276         "U",    /* 101 */
277         "UQ",   /* 110 */
278         "UH",   /* 111 */
```



```
649 };

651 static const char *arm_padd_i_names[] = {
652         "add16",        /* 000 */
653         "addsubx",      /* 001 */
654         "subaddx",      /* 010 */
655         "sub16",        /* 011 */
656         "add8",         /* 100 */
282         "ADD16",        /* 000 */
283         "ADDSUBX",      /* 001 */
284         "SUBADDX",      /* 010 */
285         "SUB16",        /* 011 */
286         "ADD8",         /* 100 */
657         NULL,           /* 101 */
658         NULL,           /* 110 */
659         "sub8",         /* 111 */
289         "SUB8",         /* 111 */
660 };

662 static const char *arm_extend_rot_names[] = {
663         "",             /* 0b00, ROR #0 */
664         ", ror #8",     /* 0b01 */
665         ", ror #16",    /* 0b10 */
666         ", ror #24"     /* 0b11 */
294         ", ROR #8",     /* 0b01 */
295         ", ROR #16",    /* 0b10 */
296         ", ROR #24"     /* 0b11 */
667 };

669 /*
670  * There are sixteen data processing instructions (dpi). They come in a few
671  * different forms which are based on whether immediate values are used and
672  * whether or not some special purpose shifting is done. We use this one entry
673  * point to cover all the different types.
674  *
675  * From the ARM arch manual:
676  *
677  * <opcode1>{<cond>}{S} <Rd>,<shifter>
678  * <opcode1> := MOV | MVN
679  * <opcode2>{<cond>} <Rn>,<shifter>
680  * <opcode2> := CMP, CMN, TST, TEQ
681  * <opcode3>{<cond>{S} <Rd>,<Rn>, <shifter>
682  * <opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
683  *
684  * 31 - 28|27 26 |25 | 24-21  |20 | 19-16 | 15-12 | 11 - 0
685  * [ cond | 0  0 | I | opcode | S | Rn    | Rd    | shifter ]
686  *
687  * I bit: Determines whether shifter_operand is immediate or register based
688  * S bit: Determines whether or not the insn updates condition codes
689  * Rn:    First source operand register
690  * Rd:    Destination register
691  * shifter: Specifies the second operand
692  *
693  * There are three primary encodings:
694  *
695  * 32-bit immediate
696  * 31 - 28|27 26|25 |24-21 |20|19-16| 15-12|11 - 8    |7 - 0
697  * [ cond | 0  0| 1 |opcode| S|Rn   | Rd   |rotate_imm|immed_8 ]
698  *
699  * Immediate shifts
700  * 31 - 28|27 26|25 |24-21 |20|19-16|15-12|11 - 7   |6 5  |4|3-0
701  * [ cond | 0  0| 0 |opcode| S|Rn   |Rd   |shift_imm|shift|0|Rm ]
702  *
703  * Register shifts
704  * 31 - 28|27 26|25 |24-21 |20|19-16|15-12|11 - 8|7|6 5  |4|3-0
705  * [ cond | 0  0| 0 |opcode| S|Rn   |Rd   |Rs    |0|shift|1|Rm ]
```

```
 *
 * There are four different kinds of shifts that work with both immediate and
 * register shifts:
 *   o Logical shift left  0b00 (LSL)
 *   o Logical shift right 0b01 (LSR)
 *   o Arithmetic shift right 0b10 (ASR)
 *   o Rotate right 0b11 (ROR)
 * There is one special shift which only works with immediate shift format:
 *   o If shift_imm = 0 and shift = 0b11, then it is a rotate right with extend
 *   (RRX)
 *
 * Finally there is one special indication for no shift. An immediate shift
 * whose shift_imm = shift = 0. This is a shortcut to a direct value from the
 * register.
 *
 * While processing this, we first build up all the information into the
 * arm_dpi_inst_t and then from there we go and print out the format based on
 * the opcode and shifter. As per the rough grammar above we have to print
 * different sets of instructions in different ways.
 */
static int
arm_dis_dpi(uint32_t in, arm_cond_code_t cond, char *buf, size_t buflen)
{
        arm_dpi_inst_t dpi_inst;
        int ibit, bit4;
        size_t len;

        dpi_inst.dpii_op = (in & ARM_DPI_OPCODE_MASK) >> ARM_DPI_OPCODE_SHIFT;
        dpi_inst.dpii_cond = cond;
        dpi_inst.dpii_rn = (in & ARM_DPI_RN_MASK) >> ARM_DPI_RN_SHIFT;
        dpi_inst.dpii_rd = (in & ARM_DPI_RD_MASK) >> ARM_DPI_RD_SHIFT;
        dpi_inst.dpii_sbit = in & ARM_DPI_SBIT_MASK;

        ibit = in & ARM_DPI_IBIT_MASK;
        bit4 = in & ARM_DPI_BIT4_MASK;

        if (ibit) {
                /* 32-bit immediate */
                dpi_inst.dpii_stype = ARM_DPI_SHIFTER_IMM32;
                dpi_inst.dpii_un.dpii_im.dpisi_rot = (in &
                    ARM_DPI_IMM_ROT_MASK) >> ARM_DPI_IMM_ROT_SHIFT;
                dpi_inst.dpii_un.dpii_im.dpisi_imm = in & ARM_DPI_IMM_VAL_MASK;
        } else if (bit4) {
                /* Register shift */
                dpi_inst.dpii_stype = ARM_DPI_SHIFTER_SREG;
                dpi_inst.dpii_un.dpii_ri.dpisr_val = (in &
                    ARM_DPI_REGS_RS_MASK) >> ARM_DPI_REGS_RS_SHIFT;
                dpi_inst.dpii_un.dpii_ri.dpisr_targ = in &
                    ARM_DPI_REGS_RM_MASK;
                dpi_inst.dpii_un.dpii_ri.dpisr_code = in &
                    ARM_DPI_REGS_SHIFT_MASK >> ARM_DPI_REGS_SHIFT_SHIFT;
        } else {
                /* Immediate shift */
                dpi_inst.dpii_stype = ARM_DPI_SHIFTER_SIMM;
                dpi_inst.dpii_un.dpii_si.dpiss_imm = (in &
                    ARM_DPI_IMS_SHIMM_MASK) >> ARM_DPI_IMS_SHIMM_SHIFT;
                dpi_inst.dpii_un.dpii_si.dpiss_code = (in &
                    ARM_DPI_IMS_SHIFT_MASK) >> ARM_DPI_IMS_SHIFT_SHIFT;
                dpi_inst.dpii_un.dpii_si.dpiss_targ = in & ARM_DPI_IMS_RM_MASK;
                if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_ROR &&
                    dpi_inst.dpii_un.dpii_si.dpiss_imm == 0)
                        dpi_inst.dpii_un.dpii_si.dpiss_code = DPI_S_RRX;

                if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_LSL &&
                    dpi_inst.dpii_un.dpii_si.dpiss_imm == 0)
                        dpi_inst.dpii_un.dpii_si.dpiss_code = DPI_S_NONE;
```



```
        }

        /*
         * Print everything before the shifter based on the instruction
         */
        switch (dpi_inst.dpii_op) {
        case DPI_OP_MOV:
        case DPI_OP_MVN:
                len = snprintf(buf, buflen, "%s%s%s %s",
                    arm_dpi_opnames[dpi_inst.dpii_op],
                    arm_cond_names[dpi_inst.dpii_cond],
                    dpi_inst.dpii_sbit != 0 ? "s" : "",
                    dpi_inst.dpii_sbit != 0 ? "S" : "",
                    arm_reg_names[dpi_inst.dpii_rd]);
                break;
        case DPI_OP_CMP:
        case DPI_OP_CMN:
        case DPI_OP_TST:
        case DPI_OP_TEQ:
                len = snprintf(buf, buflen, "%s%s %s",
                    arm_dpi_opnames[dpi_inst.dpii_op],
                    arm_cond_names[dpi_inst.dpii_cond],
                    arm_reg_names[dpi_inst.dpii_rn]);
                break;
        default:
                len = snprintf(buf, buflen,
                    "%s%s%s %s, %s", arm_dpi_opnames[dpi_inst.dpii_op],
                    arm_cond_names[dpi_inst.dpii_cond],
                    dpi_inst.dpii_sbit != 0 ? "s" : "",
                    dpi_inst.dpii_sbit != 0 ? "S" : "",
                    arm_reg_names[dpi_inst.dpii_rd],
                    arm_reg_names[dpi_inst.dpii_rn]);
                break;
        }

        if (len >= buflen)
                return (-1);
        buflen -= len;
        buf += len;

        /*
         * Print the shifter as appropriate
         */
        switch (dpi_inst.dpii_stype) {
        case ARM_DPI_SHIFTER_IMM32: {
                uint32_t rawimm, imm;
                int rawrot, rot;

                rawimm = dpi_inst.dpii_un.dpii_im.dpisi_imm;
                rawrot = dpi_inst.dpii_un.dpii_im.dpisi_rot;

                rot = rawrot * 2;
                imm = (rawimm << (32 - rot)) | (rawimm >> rot);

                len = snprintf(buf, buflen, ", #%u, %d ; 0x%08x", rawimm,
                    rawrot, imm);
                break;
        }
        case ARM_DPI_SHIFTER_SIMM:
                if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_NONE) {
                        len = snprintf(buf, buflen, ", %s",
                            arm_reg_names[dpi_inst.dpii_un.dpii_si.dpiss_targ]);
                        break;
                }
                if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_RRX) {
                        len = snprintf(buf, buflen, ", %s rrx",
```

```
465                         len = snprintf(buf, buflen, ", %s RRX",
836                             arm_reg_names[dpi_inst.dpii_un.dpii_si.dpiss_targ]);
837                         break;
838                 }
839                 len = snprintf(buf, buflen, ", %s, %s #%d",
840                     arm_reg_names[dpi_inst.dpii_un.dpii_si.dpiss_targ],
841                     arm_dpi_shifts[dpi_inst.dpii_un.dpii_si.dpiss_code],
842                     dpi_inst.dpii_un.dpii_si.dpiss_imm);
843                 break;
844         case ARM_DPI_SHIFTER_SREG:
845                 len = snprintf(buf, buflen, ", %s, %s %s",
846                     arm_reg_names[dpi_inst.dpii_un.dpii_ri.dpisr_targ],
847                     arm_dpi_shifts[dpi_inst.dpii_un.dpii_ri.dpisr_code],
848                     arm_reg_names[dpi_inst.dpii_un.dpii_ri.dpisr_val]);
849                 break;
850         }

852         return (len < buflen ? 0 : -1);
853 }

855 /*
856  * This handles the byte and word size loads and stores. It does not handle the
857  * multi-register loads or the 'extra' ones. The instruction has the generic
858  * form off:
859  *
860  * 31 - 28|27 26 |25|24|23|22|21|20|19-16|15-12|11 - 0
861  * [ cond | 0  0 |I |P |U |B |W |L | Rn | Rd   |mode_specific]
862  *
863  * Here the bits mean the following:
864  *
865  * Rn: The base register used by the addressing mode
866  * Rd: The register to load to or store from
867  * L bit: If L==1 then a load, else store
868  * B bit: If B==1 then work on a byte, else a 32-bit word
869  *
870  * The remaining pieces determine the mode we are operating in:
871  * I bit: If 0 use immediate offsets, otherwise if 1 used register based offsets
872  * P bit: If 0 use post-indexed addressing. If 1, indexing mode is either offset
873  *        addessing or pre-indexed addressing based on the W bit.
874  * U bit: If 1, offset is added to base, if 0 offset is subtracted from base
875  * W bit: This bits interpretation varies based on the P bit. If P is zero then
876  *        W indicates whether a normal memory access is performed or if a read
877  *        from user memory is performed (W = 1).
878  *        If P is 1 then then when W = 0 the base register is not updated and
879  *        when W = 1 the calculated address is written back to the base
880  *        register.
881  *
882  * Based on these combinations there are a total of nine different operating
883  * modes, though not every LDR and STR variant can reach them all.
884  */
885 static int
886 arm_dis_ldstr(uint32_t in, char *buf, size_t buflen)
887 {
888         arm_cond_code_t cc;
889         arm_reg_t rd, rn, rm;
890         int ibit, pbit, ubit, bbit, wbit, lbit;
891         arm_dpi_shift_code_t sc;
892         uint8_t simm;
893         size_t len;

895         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
896         ibit = in & ARM_LS_IBIT_MASK;
897         pbit = in & ARM_LS_PBIT_MASK;
898         ubit = in & ARM_LS_UBIT_MASK;
899         bbit = in & ARM_LS_BBIT_MASK;
900         wbit = in & ARM_LS_WBIT_MASK;
```



```
901         lbit = in & ARM_LS_LBIT_MASK;
902         rd = (in & ARM_LS_RD_MASK) >> ARM_LS_RD_SHIFT;
903         rn = (in & ARM_LS_RN_MASK) >> ARM_LS_RN_SHIFT;

905         len = snprintf(buf, buflen, "%s%s%s%s %s, ", lbit != 0 ? "ldr" : "str",
906             arm_cond_names[cc], bbit != 0 ? "b" : "",
907             (pbit == 0 && wbit != 0) ? "t" : "",
535         len = snprintf(buf, buflen, "%s%s%s%s %s, ", lbit != 0 ? "LDR" : "STR",
536             arm_cond_names[cc], bbit != 0 ? "B" : "",
537             (pbit == 0 && wbit != 0) ? "T" : "",
908             arm_reg_names[rd]);
909         if (len >= buflen)
910                 return (-1);

912         /* Figure out the specifics of the encoding for the rest */
913         if (ibit == 0 && pbit != 0) {
914                 /*
915                  * This is the immediate offset mode (A5.2.2). That means that
916                  * we have something of the form [ <Rn>, #+/-<offset_12> ]. All
917                  * of the mode specific bits contribute to offset_12. We also
918                  * handle the pre-indexed version (A5.2.5) which depends on the
919                  * wbit being set.
920                  */
921                 len += snprintf(buf + len, buflen - len, "[%s, #%s%d]%s",
922                     arm_reg_names[rn], ubit != 0 ? "" : "-",
923                     in & ARM_LS_IMM_MASK, wbit != 0 ? "!" : "");
924         } else if (ibit != 0 && pbit != 0) {
925                 /*
926                  * This handles A5.2.2, A5.2.3, A5.2.6, and A5.2.7. We can have
927                  * one of two options. If the non-rm bits (11-4) are all zeros
928                  * then we have a special case of a register offset is just
929                  * being added. Otherwise we have a scaled register offset where
930                  * the shift code matters.
931                  */
932                 rm = in & ARM_LS_REG_RM_MASK;
933                 len += snprintf(buf + len, buflen - len, "[%s, %s%s",
934                     arm_reg_names[rn], ubit != 0 ? "" : "-",
935                     arm_reg_names[rm]);
936                 if (len >= buflen)
937                         return (-1);
938                 if ((in & ARM_LS_REG_NRM_MASK) != 0) {
939                         simm = (in & ARM_LS_SCR_SIMM_MASK) >>
940                             ARM_LS_SCR_SIMM_SHIFT;
941                         sc = (in & ARM_LS_SCR_SCODE_MASK) >>
942                             ARM_LS_SCR_SCODE_SHIFT;

944                         if (simm == 0 && sc == DPI_S_ROR)
945                                 sc = DPI_S_RRX;

947                         len += snprintf(buf + len, buflen - len, "%s",
948                             arm_dpi_shifts[sc]);
949                         if (len >= buflen)
950                                 return (-1);
951                         if (sc != DPI_S_RRX) {
952                                 len += snprintf(buf + len, buflen - len, " #%d",
953                                     simm);
954                                 if (len >= buflen)
955                                         return (-1);
956                         }
957                 }
958                 len += snprintf(buf + len, buflen - len, "]%s",
959                     wbit != 0 ? "!" : "");
960         } else if (ibit == 0 && pbit == 0 && wbit == 0) {
961                 /* A5.2.8 immediate post-indexed */
962                 len += snprintf(buf + len, buflen - len, "[%s], #%s%d",
963                     arm_reg_names[rn], ubit != 0 ? "" : "-",
```

```
964                     in & ARM_LS_IMM_MASK);
965         } else if (ibit != 0 && pbit == 0 && wbit == 0) {
966                 /* A5.2.9 and A5.2.10 */
967                 rm = in & ARM_LS_REG_RM_MASK;
968                 len += snprintf(buf + len, buflen - len, "[%s], %s%s",
969                     arm_reg_names[rn], ubit != 0 ? "" : "-",
970                     arm_reg_names[rm]);
971                 if ((in & ARM_LS_REG_NRM_MASK) != 0) {
972                         simm = (in & ARM_LS_SCR_SIMM_MASK) >>
973                             ARM_LS_SCR_SIMM_SHIFT;
974                         sc = (in & ARM_LS_SCR_SCODE_MASK) >>
975                             ARM_LS_SCR_SCODE_SHIFT;

977                         if (simm == 0 && sc == DPI_S_ROR)
978                                 sc = DPI_S_RRX;

980                         len += snprintf(buf + len, buflen - len, "%s",
981                             arm_dpi_shifts[sc]);
982                         if (len >= buflen)
983                                 return (-1);
984                         if (sc != DPI_S_RRX)
985                                 len += snprintf(buf + len, buflen - len,
986                                     " #%d", simm);
987                 }
988         }

990         return (len < buflen ? 0 : -1);
991 }

993 static void
994 print_range(char **bufp, size_t *buflenp, uint16_t regs, uint16_t precede)
995 {
996         char *buf = *bufp;
997         size_t buflen = *buflenp;
998         boolean_t cont = B_FALSE;
999         int minreg = -1;
1000        int i;

1002        *buf = '\0';

1004        if (precede && regs)
1005                strlcat(buf, ", ", buflen);

1007        for (i = 0; i < 16; i++) {
1008                boolean_t present = (regs & (1 << i)) != 0;
1009                boolean_t lastreg = (regs & (2 << i)) == 0;

1011                if (!present)
1012                        continue;

1014                if (minreg == -1) {
1015                        if (cont)
1016                                strlcat(buf, ", ", buflen);

1018                        strlcat(buf, arm_reg_names[i], buflen);

1020                        if (!lastreg)
1021                                minreg = i;
1022                } else {
1023                        if (lastreg) {
1024                                strlcat(buf, "-", buflen);
1025                                strlcat(buf, arm_reg_names[i], buflen);
1026                                minreg = -1;
1027                        }
1028                }
```



```
1030                    cont = B_TRUE;
1031            }

1033            *bufp += strlen(buf);
1034            *buflenp -= strlen(buf);
1035    }

1037    static size_t
1038    print_reg_list(char *buf, size_t buflen, uint16_t regs)
1039    {
1040            char *save = buf;

1042            print_range(&buf, &buflen, regs & 0x01ff, 0);
1043            print_range(&buf, &buflen, regs & 0x0200, regs & 0x01ff); /* fp */
1044            print_range(&buf, &buflen, regs & 0x0c00, regs & 0x03ff);
1045            print_range(&buf, &buflen, regs & 0x1000, regs & 0x0fff); /* ip */
1046            print_range(&buf, &buflen, regs & 0x2000, regs & 0x1fff); /* sp */
1047            print_range(&buf, &buflen, regs & 0x4000, regs & 0x3fff); /* lr */
1048            print_range(&buf, &buflen, regs & 0x8000, regs & 0x7fff); /* pc */

1050            return (strlen(save));
1051    }

1053    #endif /* ! codereview */
1054    /*
1055     * This handles load and store multiple instructions. The general format is as
1056     * follows:
1057     *
1058     * 31 - 28|27 26 25|24|23|22|21|20|19-16|15-0
1059     * [ cond | 1  0 0 |P |U |S |W |L | Rn | register set
1060     *
1061     * The register set has one bit per register. If a bit is set it indicates that
1062     * register and if it is not set then it indicates that the register is not
1063     * included in this.
1064     *
1065     * S bit: If the instruction is a LDM and we load the PC, the S == 1 tells us to
1066     * load the CPSR from SPSR after the other regs are loaded. If the instruction
1067     * is a STM or LDM without touching the PC it indicates that if we are
1068     * privileged we should send the banked registers.
1069     *
1070     * L bit: Where this is a load or store. Load is active high.
1071     *
1072     * P bit: If P == 0 then Rn is included in the memory region transfers and its
1073     * location is dependent on the U bit. It is at the top (U == 0) or bottom (U ==
1074     * 1). If P == 1 then it is excluded and lies one word beyond the top (U == 0)
1075     * or bottom based on the U bit.
1076     *
1077     * U bit: If U == 1 then the transfer is made upwards and if U == 0 then the
1078     * transfer is made downwards.
1079     *
1080     * W bit: If set then we incremet the base register after the transfer. It is
1081     * modified by 4 times the number of registers in the list. If the U bit is
1082     * positive then that value is added to Rn otherwise it is subtracted.
1083     *
1084     * The overal layout for this is
1085     * (LDM|STM){<cond>}<addressing mode> Rn{!}, <registers>{^}. Here the ! is based
1086     * on having the W bit set. The ^ bit depends on whether S is set or not.
1087     *
1088     * There are four normal addressing modes: IA, IB, DA, DB. There are also
1089     * corresponding stack addressing modes that exist. However we have no way of
1090     * knowing which are the ones being used, therefore we are going to default to
1091     * the non-stack versions which are listed as the primary.
1092     *
1093     * Finally the last useful bit is how the registers list is specified. It is a
1094     * comma separated list inside of { }. However, a user may separate a contiguous
1095     * range by the use of a -, eg. R0 - R4. However, it is impossible for us to map
```

```
 * back directly to what the user did. So for now, we punt on second down and
 * instead just list each indidvidual register rather than attempt a joining
 * routine.
 */
static int
arm_dis_ldstr_multi(uint32_t in, char *buf, size_t buflen)
{
        int sbit, wbit, lbit;
        int sbit, wbit, lbit, ii, cont;
        uint16_t regs, addr_mode;
        arm_reg_t rn;
        arm_cond_code_t cc;
        size_t len;

        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
        sbit = in & ARM_LSM_SBIT_MASK;
        wbit = in & ARM_LSM_WBIT_MASK;
        lbit = in & ARM_LSM_LBIT_MASK;
        rn = (in & ARM_LSM_RN_MASK) >> ARM_LSM_RN_SHIFT;
        regs = in & ARM_LSM_RLIST_MASK;
        addr_mode = (in & ARM_LSM_ADDR_MASK) >> ARM_LSM_ADDR_SHIFT;

        if ((lbit == 0 && addr_mode == 2 && rn == ARM_REG_R13 && wbit != 0) ||
            (lbit != 0 && addr_mode == 1 && rn == ARM_REG_R13 && wbit != 0))
                len = snprintf(buf, buflen, "%s%s { ",
                    lbit != 0 ? "pop" : "push",
                    arm_cond_names[cc]);
        else
#endif /* ! codereview */
                len = snprintf(buf, buflen, "%s%s%s %s%s, { ",
                    lbit != 0 ? "ldm" : "stm",
            lbit != 0 ? "LDM" : "STM",
                    arm_cond_names[cc],
                    arm_lsm_mode_names[addr_mode],
                    arm_reg_names[rn],
                    wbit != 0 ? "!" : "");

        len += print_reg_list(buf + len, buflen - len, regs);
        cont = 0;
        for (ii = 0; ii < 16; ii++) {
                if (!(regs & (1 << ii)))
                        continue;

                len += snprintf(buf + len, buflen - len, "%s%s",
                    cont > 0 ? ", " : "", arm_reg_names[ii]);
                if (len >= buflen)
                        return (-1);
                cont++;
        }

        len += snprintf(buf + len, buflen - len, " }%s", sbit != 0 ? "^" : "");
        return (len >= buflen ? -1 : 0);
}

/*
 * Here we need to handle miscellaneous loads and stores. This is used to load
 * Here we need to handle miscillaneous loads and stores. This is used to load
 * and store signed and unsigned half words. To load a signed byte. And to load
 * and store double words. There is no specific store routines for signed bytes
 * and halfwords as they are supposed to use the SRB and STRH. There are two
 * primary encodings this time. The general case looks like:
 *
 * 31 - 28|27 - 25|24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
 * [ cond |   0   |P |U |I |W |L | Rn | Rd   |amode|1|S|H|1|amode ]
 *
 * The I, P, U, and W bits specify the addressing mode.
```



```
 * The L, S, and H bits describe the type and size.
 * Rn: The base register used by the addressing mode
 * Rd: The register to load to or store from
 *
 * The other bits specifically mean:
 * I bit: If set to one the address specific pieces are immediate. Otherwise
 * they aren't.
 * P bit: If P is 0 used post-indexed addressing. If P is 1 its behavior is
 * based on the value of W.
 * U bit: If U is one the offset is added to the base otherwise subtracted
 * W bit: When P is one a value of W == 1 says that the resulting memory address
 * should be written back to the base register. The base register isn't touched
 * when W is zero.
 *
 * The L, S, and H bits combine in the following table:
 *
 *  L | S | H | Meaning
 *  ------------------
 *  0 | 0 | 1 | store halfword
 *  0 | 1 | 0 | load doubleword
 *  0 | 1 | 1 | store doubleword
 *  1 | 0 | 1 | load unsigned half word
 *  1 | 1 | 0 | load signed byte
 *  1 | 1 | 1 | load signed halfword
 *
 * The final format of this is:
 * LDR|STR{<cond>}H|SH|SB|D <rd>, address_mode
 */
static int
arm_dis_els(uint32_t in, char *buf, size_t buflen)
{
        arm_cond_code_t cc;
        arm_reg_t rn, rd;
        const char *iname, *suffix;
        int lbit, sbit, hbit, pbit, ubit, ibit, wbit;
        uint8_t imm;
        size_t len;

        lbit = in & ARM_ELS_LBIT_MASK;
        sbit = in & ARM_ELS_SBIT_MASK;
        hbit = in & ARM_ELS_SBIT_MASK;

        if (lbit || (sbit && hbit == 0))
                iname = "ldr";
                iname = "LDR";
        else
                iname = "str";
                iname = "STR";

        if (sbit == 0 && hbit)
                suffix = "h";
                suffix = "H";
        else if (lbit == 0)
                suffix = "d";
                suffix = "D";
        else if (sbit && hbit == 0)
                suffix = "sb";
                suffix = "SB";
        else if (sbit && hbit)
                suffix = "sh";
                suffix = "SH";

        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
        rn = (in & ARM_ELS_RN_MASK) >> ARM_ELS_RN_SHIFT;
        rd = (in & ARM_ELS_RD_MASK) >> ARM_ELS_RD_SHIFT;
```

```
1208         len = snprintf(buf, buflen, "%s%s%s %s, ", iname, arm_cond_names[cc],
1209             suffix, arm_reg_names[rd]);
1210         if (len >= buflen)
1211                 return (-1);

1213         pbit = in & ARM_ELS_PBIT_MASK;
1214         ubit = in & ARM_ELS_UBIT_MASK;
1215         ibit = in & ARM_ELS_IBIT_MASK;
1216         wbit = in & ARM_ELS_WBIT_MASK;

1218         if (pbit && ibit) {
1219                 /* Handle A5.3.2 and A5.3.4 immediate offset and pre-indexed */
1220                 /* Bits 11-8 form the upper 4 bits of imm */
1221                 imm = (in & ARM_ELS_UP_AM_MASK) >> (ARM_ELS_UP_AM_SHIFT - 4);
1222                 imm |= in & ARM_ELS_LOW_AM_MASK;
1223                 len += snprintf(buf + len, buflen - len, "[%s, #%s%d]%s",
1224                     arm_reg_names[rn],
1225                     ubit != 0 ? "" : "-", imm,
1226                     wbit != 0 ? "!" : "");
1227         } else if (pbit && ibit == 0) {
1228                 /* Handle A5.3.3 and A5.3.5 register offset and pre-indexed */
1229                 len += snprintf(buf + len, buflen - len, "[%s %s%s]%s",
1230                     arm_reg_names[rn],
1231                     ubit != 0 ? "" : "-",
1232                     arm_reg_names[in & ARM_ELS_LOW_AM_MASK],
1233                     wbit != 0 ? "!" : "");
1234         } else if (pbit == 0 && ibit) {
1235                 /* A5.3.6 Immediate post-indexed */
1236                 /* Bits 11-8 form the upper 4 bits of imm */
1237                 imm = (in & ARM_ELS_UP_AM_MASK) >> (ARM_ELS_UP_AM_SHIFT - 4);
1238                 imm |= in & ARM_ELS_LOW_AM_MASK;
1239                 len += snprintf(buf + len, buflen - len, "[%s], #%s%d",
1240                     arm_reg_names[rn], ubit != 0 ? "" : "-", imm);
1241         } else if (pbit == 0 && ibit == 0) {
1242                 /* Handle A 5.3.7 Register post-indexed */
1243                 len += snprintf(buf + len, buflen - len, "[%s], %s%s",
1244                     arm_reg_names[rn], ubit != 0 ? "" : "-",
1245                     arm_reg_names[in & ARM_ELS_LOW_AM_MASK]);
1246         }

1248         return (len >= buflen ? -1 : 0);
1249 }

1251 /*
1252  * Handle SWP and SWPB out of the extra loads/stores extensions.
1253  */
1254 static int
1255 arm_dis_swap(uint32_t in, char *buf, size_t buflen)
1256 {
1257         arm_cond_code_t cc;
1258         arm_reg_t rn, rd, rm;

1260         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1261         rn = (in & ARM_ELS_RN_MASK) >> ARM_ELS_RN_SHIFT;
1262         rd = (in & ARM_ELS_RD_MASK) >> ARM_ELS_RD_SHIFT;
1263         rm = in & ARM_ELS_RN_MASK;

1265         if (snprintf(buf, buflen, "swp%s%s %s, %s, [%s]",
 787         if (snprintf(buf, buflen, "SWP%s%s %s, %s, [%s]",
1266             arm_cond_names[cc],
1267             (in & ARM_ELS_SWAP_BYTE_MASK) ? "b" : "",
 789             (in & ARM_ELS_SWAP_BYTE_MASK) ? "B" : "",
1268             arm_reg_names[rd], arm_reg_names[rm], arm_reg_names[rn]) >=
1269             buflen)
1270                 return (-1);
```



```
1272         return (0);
1273 }

1275 /*
1276  * Handle LDREX and STREX out of the extra loads/stores extensions.
1277  */
1278 static int
1279 arm_dis_lsexcl(uint32_t in, char *buf, size_t buflen)
1280 {
1281         arm_cond_code_t cc;
1282         arm_reg_t rn, rd, rm;
1283         int lbit;
1284         size_t len;

1286         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1287         rn = (in & ARM_ELS_RN_MASK) >> ARM_ELS_RN_SHIFT;
1288         rd = (in & ARM_ELS_RD_MASK) >> ARM_ELS_RD_SHIFT;
1289         rm = in & ARM_ELS_RN_MASK;
1290         lbit = in & ARM_ELS_LBIT_MASK;

1292         len = snprintf(buf, buflen, "%s%sex %s, ",
1293             lbit != 0 ? "ldr" : "str",
 814         len = snprintf(buf, buflen, "%s%sEX %s, ",
 815             lbit != 0 ? "LDR" : "STR",
1294             arm_cond_names[cc], arm_reg_names[rd]);
1295         if (len >= buflen)
1296                 return (-1);

1298         if (lbit)
1299                 len += snprintf(buf + len, buflen - len, "[%s]",
1300                     arm_reg_names[rn]);
1301         else
1302                 len += snprintf(buf + len, buflen - len, "%s, [%s]",
1303                     arm_reg_names[rm], arm_reg_names[rn]);
1304         return (len >= buflen ? -1 : 0);
1305 }

1307 /*
1308  * This is designed to handle the multiplication instruction extension space.
1309  * Note that this doesn't actually cover all of the multiplication instructions
1310  * available in ARM, but all of the ones that are in this space. This includes
1311  * the following instructions:
1312  *
1313  *
1314  * There are three basic encoding formats:
1315  *
1316  * Multipy (acc):
1317  * 31 - 28|27 - 24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1318  * [ cond |   0   |0 |0 | A |S |Rn  | Rd   |Rs   |1|0|0|1|Rm ]
1319  *
1320  * Unsigned multipy acc acc long
1321  * 31 - 28|27 - 24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1322  * [ cond |   0   |0 |1 |0 |0 |RdHi |RdLo  |Rs   |1|0|0|1|Rm ]
1323  *
1324  * Multiply (acc) long:
1325  * 31 - 28|27 - 24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1326  * [ cond |   0   |1 |Un|A |S |RdHi| RdLo |Rs   |1|0|0|1|Rm ]
1327  *
1328  * A bit: Accumulate
1329  * Un bit: Unsigned is active low, signed is active high
1330  * S bit: Indicates whethere the status register should be updated.
1331  *
1332  * MLA(S) and MUL(S) make up the first type of instructions.
1333  * UMAAL makes up the second group.
1334  * (U|S)MULL(S), (U|S)MLAL(S), Make up the third.
1335  */
```

```
1336 static int
1337 arm_dis_extmul(uint32_t in, char *buf, size_t buflen)
1338 {
1339         arm_cond_code_t cc;
1340         arm_reg_t rd, rn, rs, rm;
1341         size_t len;

1343         /*
1344          * RdHi is equal to rd here. RdLo is equal to Rn here.
1345          */
1346         rd = (in & ARM_EMULT_RD_MASK) >> ARM_EMULT_RD_SHIFT;
1347         rn = (in & ARM_EMULT_RN_MASK) >> ARM_EMULT_RN_SHIFT;
1348         rs = (in & ARM_EMULT_RS_MASK) >> ARM_EMULT_RS_SHIFT;
1349         rm = in & ARM_EMULT_RM_MASK;

1351         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

1353         if ((in & ARM_EMULT_MA_MASK) == 0) {
1354                 if (in & ARM_EMULT_ABIT_MASK) {
1355                         len = snprintf(buf, buflen, "mla%s%s %s, %s, %s, %s",
 877                         len = snprintf(buf, buflen, "MLA%s%s %s, %s, %s, %s",
1356                             arm_cond_names[cc],
1357                             (in & ARM_EMULT_SBIT_MASK) ? "s" : "",
 879                             (in & ARM_EMULT_SBIT_MASK) ? "S" : "",
1358                             arm_reg_names[rd], arm_reg_names[rm],
1359                             arm_reg_names[rs], arm_reg_names[rs]);
1360                 } else {
1361                         len = snprintf(buf, buflen, "mul%s%s %s, %s, %s",
 883                         len = snprintf(buf, buflen, "MUL%s%s %s, %s, %s",
1362                             arm_cond_names[cc],
1363                             (in & ARM_EMULT_SBIT_MASK) ? "s" : "",
 885                             (in & ARM_EMULT_SBIT_MASK) ? "S" : "",
1364                             arm_reg_names[rd], arm_reg_names[rm],
1365                             arm_reg_names[rs]);

1367                 }
1368         } else if ((in & ARM_EMULT_UMA_MASK) == ARM_EMULT_UMA_TARG) {
1369                 len = snprintf(buf, buflen, "umaal%s %s, %s, %s, %s",
 891                 len = snprintf(buf, buflen, "UMAAL%s %s, %s, %s, %s",
1370                     arm_cond_names[cc], arm_reg_names[rn], arm_reg_names[rd],
1371                     arm_reg_names[rm], arm_reg_names[rs]);
1372         } else if ((in & ARM_EMULT_MAL_MASK) == ARM_EMULT_MAL_TARG) {
1373                 len = snprintf(buf, buflen, "%s%s%s%s %s, %s, %s, %s",
1374                     (in & ARM_EMULT_UNBIT_MASK) ? "s" : "u",
1375                     (in & ARM_EMULT_ABIT_MASK) ? "mlal" : "mull",
 896                     (in & ARM_EMULT_UNBIT_MASK) ? "S" : "U",
 897                     (in & ARM_EMULT_ABIT_MASK) ? "MLAL" : "MULL",
1376                     arm_cond_names[cc],
1377                     (in & ARM_EMULT_SBIT_MASK) ? "s" : "",
 899                     (in & ARM_EMULT_SBIT_MASK) ? "S" : "",
1378                     arm_reg_names[rn], arm_reg_names[rd], arm_reg_names[rm],
1379                     arm_reg_names[rs]);
1380         } else {
1381                 /* Not a supported instruction in this space */
1382                 return (-1);
1383         }
1384         return (len >= buflen ? -1 : 0);
1385 }

1387 /*
1388  * Here we handle the three different cases of moving to and from the various
1389  * status registers in both register mode and in immediate mode.
1390  */
1391 static int
1392 arm_dis_status_regs(uint32_t in, char *buf, size_t buflen)
1393 {
```



```
1394         arm_cond_code_t cc;
1395         arm_reg_t rd, rm;
1396         uint8_t field;
1397         int imm;
1398         size_t len;

1400         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

1402         if ((in & ARM_CDSP_MRS_MASK) == ARM_CDSP_MRS_TARG) {
1403                 rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1404                 if (snprintf(buf, buflen, "mrs%s %s, %s", arm_cond_names[cc],
 926                 if (snprintf(buf, buflen, "MRS%s %s, %s", arm_cond_names[cc],
1405                     arm_reg_names[rd],
1406                     (in & ARM_CDSP_STATUS_RBIT) != 0 ? "spsr" : "cpsr") >=
 928                     (in & ARM_CDSP_STATUS_RBIT) != 0 ? "SPSR" : "CPSR") >=
1407                     buflen)
1408                         return (-1);
1409                 return (0);
1410         }

1412         field = (in & ARM_CDSP_MSR_F_MASK) >> ARM_CDSP_MSR_F_SHIFT;
1413         len = snprintf(buf, buflen, "msr%s %s_%s, ", arm_cond_names[cc],
1414             (in & ARM_CDSP_STATUS_RBIT) != 0 ? "spsr" : "cpsr",
 935         len = snprintf(buf, buflen, "MSR%s %s_%s, ", arm_cond_names[cc],
 936             (in & ARM_CDSP_STATUS_RBIT) != 0 ? "SPSR" : "CPSR",
1415             arm_cdsp_msr_field_names[field]);
1416         if (len >= buflen)
1417                 return (-1);

1419         if (in & ARM_CDSP_MSR_ISIMM_MASK) {
1420                 imm = in & ARM_CDSP_MSR_IMM_MASK;
1421                 imm <<= (in & ARM_CDSP_MSR_RI_MASK) >> ARM_CDSP_MSR_RI_SHIFT;
1422                 len += snprintf(buf + len, buflen - len, "#%d", imm);
1423         } else {
1424                 rm = in & ARM_CDSP_RM_MASK;
1425                 len += snprintf(buf + len, buflen - len, "%s",
1426                     arm_reg_names[rm]);
1427         }

1429         return (len >= buflen ? -1 : 0);
1430 }

1432 /*
1433  * Here we need to handle the Control And DSP instruction extension space. This
1434  * consists of several different instructions. Unlike other extension spaces
1435  * there isn't as much tha tis similar here as there is stuff that is different.
1436  * Oh well, that's a part of life. Instead we do a little bit of additional
1437  * parsing here.
1438  *
1439  * The first group that we separate out are the instructions that interact with
1440  * the status registers. Those are handled in their own function.
1441  */
1442 static int
1443 arm_dis_cdsp_ext(uint32_t in, char *buf, size_t buflen)
1444 {
1445         uint16_t imm, op;
1446         arm_cond_code_t cc;
1447         arm_reg_t rd, rm, rn, rs;
1448         size_t len;

1450         if ((in & ARM_CDSP_STATUS_MASK) == ARM_CDSP_STATUS_TARG)
1451                 return (arm_dis_status_regs(in, buf, buflen));

1453         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

1455         /*
```

```
1456          * This gets the Branch/exchange as well as the Branch and link/exchange
1457          * pieces. These generally also transform the instruction set into
1458          * something we can't actually disassemble. Here the lower mask and
1459          * target is the opposite. eg. the target bits are not what we want.
1460          */
1461         if ((in & ARM_CDSP_BEX_UP_MASK) == ARM_CDSP_BEX_UP_TARG &&
1462             (in & ARM_CDSP_BEX_LOW_MASK) != ARM_CDSP_BEX_NLOW_TARG) {
1463                 rm = in & ARM_CDSP_RM_MASK;
1464                 imm = (in & ARM_CDSP_BEX_TYPE_MASK) >> ARM_CDSP_BEX_TYPE_SHIFT;
1465                 if (snprintf(buf, buflen, "b%s%s %s",
1466                     imm == ARM_CDSP_BEX_TYPE_X ? "x" :
1467                     imm == ARM_CDSP_BEX_TYPE_J ? "xj" : "lx",
 987                 if (snprintf(buf, buflen, "B%s%s %s",
 988                     imm == ARM_CDSP_BEX_TYPE_X ? "X" :
 989                     imm == ARM_CDSP_BEX_TYPE_J ? "XJ" : "LX",
1468                     arm_cond_names[cc], arm_reg_names[rm]) >= buflen)
1469                         return (-1);
1470                 return (0);
1471         }

1473         /* Count leading zeros */
1474         if ((in & ARM_CDSP_CLZ_MASK) == ARM_CDSP_CLZ_TARG) {
1475                 rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1476                 rm = in & ARM_CDSP_RM_MASK;
1477                 if (snprintf(buf, buflen, "clz%s %s, %s", arm_cond_names[cc],
 999                 if (snprintf(buf, buflen, "CLZ%s %s, %s", arm_cond_names[cc],
1478                     arm_reg_names[rd], arm_reg_names[rm]) >= buflen)
1479                         return (-1);
1480                 return (0);
1481         }

1483         if ((in & ARM_CDSP_SAT_MASK) == ARM_CDSP_SAT_TARG) {
1484                 rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1485                 rn = (in & ARM_CDSP_RN_MASK) >> ARM_CDSP_RN_SHIFT;
1486                 rm = in & ARM_CDSP_RM_MASK;
1487                 imm = (in & ARM_CDSP_SAT_OP_MASK) >> ARM_CDSP_SAT_OP_SHIFT;
1488                 if (snprintf(buf, buflen, "q%s%s %s, %s, %s",
1010                 if (snprintf(buf, buflen, "Q%s%s %s, %s, %s",
1489                     arm_cdsp_sat_opnames[imm], arm_cond_names[cc],
1490                     arm_reg_names[rd], arm_reg_names[rm],
1491                     arm_reg_names[rn]) >= buflen)
1492                         return (-1);
1493                 return (0);
1494         }

1496         /*
1497          * Breakpoint instructions are a bit different. While they are in the
1498          * conditional instruction namespace, they actually aren't defined to
1499          * take a condition. That's just how it rolls. The breakpoint is a
1500          * 16-bit value. The upper 12 bits are stored together and the lower
1501          * four together.
1502          */
1503         if ((in & ARM_CDSP_BKPT_MASK) == ARM_CDSP_BKPT_TARG) {
1504                 if (cc != ARM_COND_NACC)
1505                         return (-1);
1506                 imm = (in & ARM_CDSP_BKPT_UIMM_MASK) >>
1507                     ARM_CDSP_BKPT_UIMM_SHIFT;
1508                 imm <<= 4;
1509                 imm |= (in & ARM_CDSP_BKPT_LIMM_MASK);
1510                 if (snprintf(buf, buflen, "bkpt %d", imm) >= buflen)
1032                 if (snprintf(buf, buflen, "BKPT %d", imm) >= buflen)
1511                         return (1);
1512                 return (0);
1513         }

1515         /*
```



```
1516          * Here we need to handle another set of multiplies. Specifically the
1517          * Signed multiplies. This is SMLA<x><y>, SMLAW<y>, SMULW<y>,
1518          * SMLAL<x><y>, SMUL<x><y>. These instructions all follow the form:
1519          *
1520          * 31 - 28|27-25|24|23|22-21|20|19-16|15-12|11 - 8|7|6|5|4|3-0
1521          * [ cond |  0  | 1| 0| op. | 0|Rn   |Rd   |Rs    |1|y|x|0|Rm ]
1522          *
1523          * If x is one a T is used for that part of the name. Otherwise a B is.
1524          * The same holds true for y.
1525          *
1526          * These instructions map to the following opcodes:
1527          * SMLA<x><y>: 00,
1528          * SMLAW<y>: 01 and x is zero,
1529          * SMULW<y>: 01 and x is one ,
1530          * SMLAL<x><y>: 10,
1531          * SMUL<xy><y>: 11
1532          */
1533         if ((in & ARM_CDSP_SMUL_MASK) == ARM_CDSP_SMUL_TARG) {
1534                 rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1535                 rn = (in & ARM_CDSP_RN_MASK) >> ARM_CDSP_RN_SHIFT;
1536                 rs = (in & ARM_CDSP_RS_MASK) >> ARM_CDSP_RS_SHIFT;
1537                 rm = in & ARM_CDSP_RM_MASK;
1538                 op = (in & ARM_CDSP_SMUL_OP_MASK) >> ARM_CDSP_SMUL_OP_SHIFT;

1540                 switch (op) {
1541                 case 0:
1542                         len = snprintf(buf, buflen, "smla%s%s%s %s, %s, %s, %s",
1543                             (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "t" : "b",
1544                             (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "t" : "b",
1064                         len = snprintf(buf, buflen, "SMLA%s%s%s %s, %s, %s, %s",
1065                             (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "T" : "B",
1066                             (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" : "B",
1545                             arm_cond_names[cc], arm_reg_names[rd],
1546                             arm_reg_names[rm], arm_reg_names[rs],
1547                             arm_reg_names[rn]);
1548                         break;
1549                 case 1:
1550                         if (in & ARM_CDSP_SMUL_X_MASK) {
1551                                 len = snprintf(buf, buflen,
1552                                     "smulw%s%s %s, %s, %s",
1553                                     (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "t" :
1554                                     "b", arm_cond_names[cc], arm_reg_names[rd],
1074                                     "SMULW%s%s %s, %s, %s",
1075                                     (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" :
1076                                     "B", arm_cond_names[cc], arm_reg_names[rd],
1555                                     arm_reg_names[rm], arm_reg_names[rs]);
1556                         } else {
1557                                 len = snprintf(buf, buflen,
1558                                     "smlaw%s%s %s, %s, %s %s",
1559                                     (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "t" :
1560                                     "b", arm_cond_names[cc], arm_reg_names[rd],
1080                                     "SMLAW%s%s %s, %s, %s %s",
1081                                     (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" :
1082                                     "B", arm_cond_names[cc], arm_reg_names[rd],
1561                                     arm_reg_names[rm], arm_reg_names[rs],
1562                                     arm_reg_names[rn]);
1563                         }
1564                         break;
1565                 case 2:
1566                         len = snprintf(buf, buflen,
1567                             "smlal%s%s%s %s, %s, %s, %s",
1568                             (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "t" : "b",
1569                             (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "t" : "b",
1089                             "SMLAL%s%s%s %s, %s, %s, %s",
1090                             (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "T" : "B",
1091                             (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" : "B",
```

```
1570                            arm_cond_names[cc], arm_reg_names[rd],
1571                            arm_reg_names[rn], arm_reg_names[rm],
1572                            arm_reg_names[rs]);
1573                    break;
1574            case 3:
1575                    len = snprintf(buf, buflen, "smul%s%s%s %s, %s, %s",
1576                        (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "t" : "b",
1577                        (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "t" : "b",
1097                    len = snprintf(buf, buflen, "SMUL%s%s%s %s, %s, %s",
1098                        (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "T" : "B",
1099                        (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" : "B",
1578                        arm_cond_names[cc], arm_reg_names[rd],
1579                        arm_reg_names[rm], arm_reg_names[rs]);
1580                    break;
1581            default:
1582                    return (-1);
1583            }
1584            return (len >= buflen ? -1 : 0);
1585    }

1587    /*
1588     * If we got here then this is some other instructin we don't know
1589     * about in the instruction extensino space.
1590     */
1591    return (-1);
1592 }

1594 /*
1595  * Coprocessor double register transfers
1596  *
1597  * MCRR:
1598  * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11-8|7-4|3-0
1599  * [ cond |1 1 0| 0| 0| 1| 0| 0| Rn  |  Rd |cp #|op |CRm
1600  *
1601  * MRRC:
1602  * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11-8|7-4|3-0
1603  * [ cond |1 1 0| 0| 0| 1| 0| 1| Rn  |  Rd |cp #|op |CRm
1604  *
1605  */
1606 static int
1607 arm_dis_coproc_drt(uint32_t in, char *buf, size_t buflen)
1608 {
1609    arm_cond_code_t cc;
1610    arm_reg_t rd, rn, rm;
1611    uint8_t coproc, op;
1612    const char *ccn;
1613    size_t len;

1615    cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1616    coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
1617    rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
1618    rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
1619    rm = in & ARM_COPROC_RM_MASK;
1620    op = (in & ARM_COPROC_DRT_OP_MASK) >> ARM_COPROC_DRT_OP_SHIFT;

1622    if (cc == ARM_COND_NACC)
1623            ccn = "2";
1624    else
1625            ccn = arm_cond_names[cc];

1627    len = snprintf(buf, buflen, "%s%s %s, #%d, %s, %s, c%s",
1628        (in & ARM_COPROC_DRT_DIR_MASK) != 0 ? "mrrc" : "mcrr",
1149    len = snprintf(buf, buflen, "%s%s %s, #%d, %s, %s, C%s",
1150        (in & ARM_COPROC_DRT_DIR_MASK) != 0 ? "MRRC" : "MCRR",
1629        ccn, arm_coproc_names[coproc], op, arm_reg_names[rd],
1630        arm_reg_names[rn], arm_reg_names[rm]);
```



```
1631    return (len >= buflen ? -1 : 0);
1632 }

1634 /*
1635  * This serves as both the entry point for the normal load and stores as well as
1636  * the double register transfers (MCRR and MRCC). If it is a register transfer
1637  * then we quickly send it off.
1638  * LDC:
1639  * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11 - 8|7 - 0
1640  * [ cond |1 1 0| P| U| N| W| L| Rn  | CRd | cp # | off ]
1641  *
1642  * STC:
1643  * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11 - 8|7 - 0
1644  * [ cond |1 1 0| P| U| N| W| L| Rn  | CRd | cp # | off ]
1645  *
1646  * Here the bits mean:
1647  *
1648  * P bit: If P is zero, it is post-indexed or unindexed based on W. If P is 1
1649  * then it is offset-addressing or pre-indexed based on W again.
1650  *
1651  * U bit: If U is positive then the offset if added, subtracted otherwise.. Note
1652  * that if P is zero and W is zero, U must be one.
1653  *
1654  * N bit: If set that means that we have a Long size, this bit is set by the L
1655  * suffix, not to be confused with the L bit.
1656  *
1657  * W bit: If W is one then the memory address is written back to the base
1658  * register. Further W = 0 and P = 0 is unindexed addressing. W = 1, P = 0 is
1659  * post-indexed. W = 0, P = 1 is offset addressing and W = 1, P = 1 is
1660  * pre-indexed.
1661  */
1662 static int
1663 arm_dis_coproc_lsdrt(uint32_t in, char *buf, size_t buflen)
1664 {
1665    arm_cond_code_t cc;
1666    arm_reg_t rn, rd;
1667    uint8_t coproc;
1668    uint32_t imm;
1669    int pbit, ubit, nbit, wbit, lbit;
1670    const char *ccn;
1671    size_t len;

1673    if ((in & ARM_COPROC_DRT_MASK) == ARM_COPROC_DRT_TARG)
1674            return (arm_dis_coproc_drt(in, buf, buflen));

1676    cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1677    coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
1678    rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
1679    rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
1680    imm = in & ARM_COPROC_LS_IMM_MASK;

1682    pbit = in & ARM_COPROC_LS_P_MASK;
1683    ubit = in & ARM_COPROC_LS_U_MASK;
1684    nbit = in & ARM_COPROC_LS_N_MASK;
1685    wbit = in & ARM_COPROC_LS_W_MASK;
1686    lbit = in & ARM_COPROC_LS_L_MASK;

1688    if (cc == ARM_COND_NACC)
1689            ccn = "2";
1690    else
1691            ccn = arm_cond_names[cc];

1693    len = snprintf(buf, buflen, "%s%s%s %s, c%s, ",
1694        lbit != 0 ? "ldc" : "stc", ccn, nbit != 0 ? "l" : "",
1215    len = snprintf(buf, buflen, "%s%s%s %s, C%s, ",
1216        lbit != 0 ? "LDC" : "STC", ccn, nbit != 0 ? "L" : "",
```

```
1695             arm_coproc_names[coproc], arm_reg_names[rd]);
1696         if (len >= buflen)
1697                 return (-1);

1699         if (pbit != 0) {
1700                 imm *= 4;
1701                 len += snprintf(buf + len, buflen - len, "[%s, #%s%d]%s",
1702                     arm_reg_names[rn],
1703                     ubit != 0 ? "" : "-", imm,
1704                     wbit != 0 ? "!" : "");
1705         } else if (wbit != 0) {
1706                 imm *= 4;
1707                 len += snprintf(buf + len, buflen - len, "[%s], #%s%d",
1708                     arm_reg_names[rn], ubit != 0 ? "" : "-", imm);
1709         } else {
1710                 len += snprintf(buf + len, buflen - len, "[%s], { %d }",
1711                     arm_reg_names[rn], imm);
1712         }
1713         return (len >= buflen ? -1 : 0);
1714 }

1716 /*
1717  * Here we tell a coprocessor to do data processing
1718  *
1719  * CDP:
1720  * 31 - 28|27 - 24|23-20|19-16|15-12|11 - 8|7 - 5|4|3-0
1721  * [ cond |1 1 1 0| op_1| CRn | CRd | cp # | op_2|0|CRm ]
1722  */
1723 static int
1724 arm_dis_coproc_dp(uint32_t in, char *buf, size_t buflen)
1725 {
1726         arm_cond_code_t cc;
1727         arm_reg_t rn, rd, rm;
1728         uint8_t op1, op2, coproc;
1729         const char *ccn;

1731         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1732         coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
1733         rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
1734         rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
1735         rm = in & ARM_COPROC_RM_MASK;
1736         op1 = (in & ARM_COPROC_CDP_OP1_MASK) >> ARM_COPROC_CDP_OP1_SHIFT;
1737         op2 = (in & ARM_COPROC_CDP_OP2_MASK) >> ARM_COPROC_CDP_OP2_SHIFT;

1739         /*
1740          * This instruction is valid with the undefined condition code. When it
1741          * does that, the instruction is intead CDP2 as opposed to CDP.
1742          */
1743         if (cc == ARM_COND_NACC)
1744                 ccn = "2";
1745         else
1746                 ccn = arm_cond_names[cc];

1748         if (snprintf(buf, buflen, "cdp%s %s, #%d, c%s, c%s, c%s, #%d", ccn,
1270         if (snprintf(buf, buflen, "CDP%s %s, #%d, C%s, C%s, C%s, #%d", ccn,
1749             arm_coproc_names[coproc], op1, arm_reg_names[rd],
1750             arm_reg_names[rn], arm_reg_names[rm], op2) >= buflen)
1751                 return (-1);

1753         return (0);
1754 }

1756 /*
1757  * Here we handle coprocesser single register transfers.
1758  *
1759  * MCR:
```



```
1760  * 31 - 28|27 - 24|23-21|20|19-16|15-12|11 - 8|7 - 5|4|3-0
1761  * [ cond |1 1 1 0| op_1| 0| CRn |  Rd | cp # | op_2|1|CRm ]
1762  *
1763  * MRC:
1764  * 31 - 28|27 - 24|23-21|20|19-16|15-12|11 - 8|7 - 5|4|3-0
1765  * [ cond |1 1 1 0| op_1| 1| CRn |  Rd | cp # | op_2|1|CRm ]
1766  */
1767 static int
1768 arm_dis_coproc_rt(uint32_t in, char *buf, size_t buflen)
1769 {
1770         arm_cond_code_t cc;
1771         arm_reg_t rn, rd, rm;
1772         uint8_t op1, op2, coproc;
1773         const char *ccn;
1774         size_t len;

1776         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1777         coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
1778         rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
1779         rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
1780         rm = in & ARM_COPROC_RM_MASK;
1781         op1 = (in & ARM_COPROC_CRT_OP1_MASK) >> ARM_COPROC_CRT_OP1_SHIFT;
1782         op2 = (in & ARM_COPROC_CRT_OP2_MASK) >> ARM_COPROC_CRT_OP2_SHIFT;

1784         if (cc == ARM_COND_NACC)
1785                 ccn = "2";
1786         else
1787                 ccn = arm_cond_names[cc];

1789         len = snprintf(buf, buflen, "%s%s %s, #%d, %s, c%s, c%s",
1790             (in & ARM_COPROC_CRT_DIR_MASK) != 0 ? "mrc" : "mcr", ccn,
1311         len = snprintf(buf, buflen, "%s%s %s, #%d, %s, C%s, C%s",
1312             (in & ARM_COPROC_CRT_DIR_MASK) != 0 ? "MRC" : "MCR", ccn,
1791             arm_coproc_names[coproc], op1, arm_reg_names[rd],
1792             arm_reg_names[rn], arm_reg_names[rm]);
1793         if (len >= buflen)
1794                 return (-1);

1796         if (op2 != 0)
1797                 if (snprintf(buf + len, buflen - len, ", #%d", op2) >=
1798                     buflen - len)
1799                         return (-1);
1800         return (0);
1801 }

1803 /*
1804  * Here we handle the set of unconditional instructions.
1805  */
1806 static int
1807 arm_dis_uncond_insn(uint32_t in, char *buf, size_t buflen)
1808 {
1809         int imm, sc;
1810         arm_reg_t rn, rm;
1811         size_t len;

1813         /*
1814          * The CPS instruction is a bit complicated. It has the following big
1815          * pattern which maps to a few different ways to use it:
1816          *
1817          *
1818          * 31-28|27-25|24|23-20|19-18|17 |16|15-9|8|7|6|5|4-0
1819          *    1 |  0  | 1| 0   |imod|mmod| 0|SBZ |A|I|F|0|mode
1820          *
1821          * CPS<effect> <iflags> {, #<mode> }
1822          * CPS #<mode>
1823          *
```

```
1824          * effect: determines what to do with the A, I, F interrupt bits in the
1825          * CPSR. effect is encoded in the imod field. It is either enable
1826          * interrupts 0b10 or disable interrupts 0b11. Recall that interrupts
1827          * are active low in the CPSR. If effect is not specified then this is
1828          * strictly a mode change which is required.
1829          *
1830          * A, I, F: If effect is specified then the bits which are high are
1831          * modified by the instruction.
1832          *
1833          * mode: Specifies a mode to change to. mmod will be 1 if mode is set.
1834          *
1835          */
1836         if ((in & ARM_UNI_CPS_MASK) == ARM_UNI_CPS_TARG) {
1837                 imm = (in & ARM_UNI_CPS_IMOD_MASK) > ARM_UNI_CPS_IMOD_SHIFT;

1839                 /* Ob01 is not a valid value for the imod */
1840                 if (imm == 1)
1841                         return (-1);

1843                 if (imm != 0)
1844                         len = snprintf(buf, buflen, "cps%s %s%s%s%s",
1845                             imm == 2 ? "ie" : "id",
1366                         len = snprintf(buf, buflen, "CPS%s %s%s%s%s",
1367                             imm == 2 ? "IE" : "ID",
1846                             (in & ARM_UNI_CPS_A_MASK) ? "a" : "",
1847                             (in & ARM_UNI_CPS_I_MASK) ? "i" : "",
1848                             (in & ARM_UNI_CPS_F_MASK) ? "f" : "",
1849                             (in & ARM_UNI_CPS_MMOD_MASK) ? " ," : "");
1850                 else
1851                         len = snprintf(buf, buflen, "cps ");
1373                         len = snprintf(buf, buflen, "CPS ");
1852                 if (len >= buflen)
1853                         return (-1);

1855                 if (in & ARM_UNI_CPS_MMOD_MASK)
1856                         if (snprintf(buf + len, buflen - len, "#%d",
1857                             in & ARM_UNI_CPS_MODE_MASK) >= buflen - len)
1858                                 return (-1);
1859                 return (0);
1860         }

1862         if ((in & ARM_UNI_SE_MASK) == ARM_UNI_SE_TARG) {
1863                 if (snprintf(buf, buflen, "SETEND %s",
1864                     (in & ARM_UNI_SE_BE_MASK) ? "be" : "le") >= buflen)
1386                     (in & ARM_UNI_SE_BE_MASK) ? "BE" : "LE") >= buflen)
1865                         return (-1);
1866                 return (0);
1867         }

1869         /*
1870          * The cache preload is like a load, but it has a much simpler set of
1871          * constraints. The only valid bits that you can transform are the I and
1872          * the U bits. We have to use pre-indexed addressing. This means that we
1873          * only have the U bit and the I bit. See arm_dis_ldstr for a full
1874          * explanation of what's happening here.
1875          */
1876         if ((in & ARM_UNI_PLD_MASK) == ARM_UNI_PLD_TARG) {
1877                 rn = (in & ARM_LS_RN_MASK) >> ARM_LS_RN_SHIFT;
1878                 if ((in & ARM_LS_IBIT_MASK) == 0) {
1879                         if (snprintf(buf, buflen, "pld [%s, #%s%d",
1401                         if (snprintf(buf, buflen, "PLD [%s, #%s%d",
1880                             arm_reg_names[rn],
1881                             (in & ARM_LS_UBIT_MASK) != 0 ? "" : "-",
1882                             in & ARM_LS_IMM_MASK) >= buflen)
1883                                 return (-1);
1884                         return (0);
```



```
1885                 }

1887                 rm = in & ARM_LS_REG_RM_MASK;
1888                 len = snprintf(buf, buflen, "pld [%s, %s%s", arm_reg_names[rn],
1410                 len = snprintf(buf, buflen, "PLD [%s, %s%s", arm_reg_names[rn],
1889                     (in & ARM_LS_UBIT_MASK) != 0 ? "" : "-",
1890                     arm_reg_names[rm]);
1891                 if (len >= buflen)
1892                         return (-1);

1894                 if ((in & ARM_LS_REG_NRM_MASK) != 0) {
1895                         imm = (in & ARM_LS_SCR_SIMM_MASK) >>
1896                             ARM_LS_SCR_SIMM_SHIFT;
1897                         sc = (in & ARM_LS_SCR_SCODE_MASK) >>
1898                             ARM_LS_SCR_SCODE_SHIFT;

1900                         if (imm == 0 && sc == DPI_S_ROR)
1901                                 sc = DPI_S_RRX;

1903                         len += snprintf(buf + len, buflen - len, "%s",
1904                             arm_dpi_shifts[sc]);
1905                         if (len >= buflen)
1906                                 return (-1);
1907                         if (sc != DPI_S_RRX) {
1908                                 len += snprintf(buf + len, buflen - len,
1909                                     " #%d", imm);
1910                                 if (len >= buflen)
1911                                         return (-1);
1912                         }
1913                 }
1914                 if (snprintf(buf + len, buflen - len, "]") >= buflen - len)
1915                         return (-1);
1916                 return (0);
1917         }

1919         /*
1920          * This is a special case of STM, but it works across chip modes.
1921          */
1922         if ((in & ARM_UNI_SRS_MASK) == ARM_UNI_SRS_TARG) {
1923                 imm = (in & ARM_LSM_ADDR_MASK) >> ARM_LSM_ADDR_SHIFT;
1924                 if (snprintf(buf, buflen, "srs%s #%d%s",
1446                 if (snprintf(buf, buflen, "SRS%s #%d%s",
1925                     arm_lsm_mode_names[imm],
1926                     in & ARM_UNI_SRS_MODE_MASK,
1927                     (in & ARM_UNI_SRS_WBIT_MASK) != 0 ? "!" : "") >= buflen)
1928                         return (-1);
1929                 return (0);
1930         }

1932         /*
1933          * RFE is a return from exception instruction that is similar to the LDM
1934          * and STM, but a bit different.
1935          */
1936         if ((in & ARM_UNI_RFE_MASK) == ARM_UNI_RFE_TARG) {
1937                 imm = (in & ARM_LSM_ADDR_MASK) >> ARM_LSM_ADDR_SHIFT;
1938                 rn = (in & ARM_LS_RN_MASK) >> ARM_LS_RN_SHIFT;
1939                 if (snprintf(buf, buflen, "rfe%s %s%s", arm_lsm_mode_names[imm],
1461                 if (snprintf(buf, buflen, "RFE%s %s%s", arm_lsm_mode_names[imm],
1940                     arm_reg_names[rn],
1941                     (in & ARM_UNI_RFE_WBIT_MASK) != 0 ? "!" : "") >= buflen)
1942                         return (-1);
1943                 return (0);
1944         }

1946         if ((in & ARM_UNI_BLX_MASK) == ARM_UNI_BLX_TARG) {
1947                 if (snprintf(buf, buflen, "blx %d",
```

```
1469                 if (snprintf(buf, buflen, "BLX %d",
1948                     in & ARM_UNI_BLX_IMM_MASK) >= buflen)
1949                         return (-1);
1950                 return (0);
1951         }

1953         if ((in & ARM_UNI_CODRT_MASK) == ARM_UNI_CODRT_TARG) {
1954                 return (arm_dis_coproc_lsdrt(in, buf, buflen));
1955         }

1957         if ((in & ARM_UNI_CORT_MASK) == ARM_UNI_CORT_TARG) {
1958                 return (arm_dis_coproc_rt(in, buf, buflen));
1959         }

1961         if ((in & ARM_UNI_CODP_MASK) == ARM_UNI_CORT_TARG) {
1962                 return (arm_dis_coproc_dp(in, buf, buflen));
1963         }

1965         /*
1966          * An undefined or illegal instruction
1967          */
1968         return (-1);
1969 }

1971 /*
1972  * Disassemble B and BL instructions. The instruction is given a 24-bit two's
1973  * complement value as an offset address. This value gets sign extended to 30
1974  * bits and then shifted over two bits. This is then added to the PC + 8. So,
1975  * instead of dispalying an absolute address, we're going to display the delta
1976  * that the instruction has instead.
1977  */
1978 static int
1979 arm_dis_branch(dis_handle_t *dhp, uint32_t in, char *buf, size_t buflen)
1980 {
1981         uint32_t addr;
1982         arm_cond_code_t cc;
1983         size_t len;

1985         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1986         addr = in & ARM_BRANCH_IMM_MASK;
1987         if (in & ARM_BRANCH_SIGN_MASK)
1988                 addr |= ARM_BRANCH_NEG_SIGN;
1989         else
1990                 addr &= ARM_BRANCH_POS_SIGN;
1991         addr <<= 2;
1992         if ((len = snprintf(buf, buflen, "b%s%s %d",
1993             (in & ARM_BRANCH_LBIT_MASK) != 0 ? "l" : "",
1514         if ((len = snprintf(buf, buflen, "B%s%s %d",
1515             (in & ARM_BRANCH_LBIT_MASK) != 0 ? "L" : "",
1994             arm_cond_names[cc], (int)addr)) >= buflen)
1995                 return (-1);

1997         /* Per the ARM manuals, we have to account for the extra 8 bytes here */
1998         if (dhp->dh_lookup(dhp->dh_data, dhp->dh_addr + (int)addr + 8, NULL, 0,
1999             NULL, NULL) == 0) {
2000                 len += snprintf(buf + len, buflen - len, "\t<");
2001                 if (len >= buflen)
2002                         return (-1);
2003                 dhp->dh_lookup(dhp->dh_data, dhp->dh_addr + (int)addr + 8,
2004                     buf + len, buflen - len, NULL, NULL);
2005                 strlcat(buf, ">", buflen);
2006         }

2008         return (0);
2009 }
```

*_____unchanged_portion_omitted_*



```
2057 /*
2058  * Disassemble the extend instructions from ARMv6. There are six instructions:
2059  *
2060  * XTAB16, XTAB, XTAH, XTB16, XTB, XTFH. These can exist with one of the
2061  * following prefixes: S, U. The opcode exists in bits 22-20. We have the
2062  * following rules from there:
2063  *
2064  * If bit 22 is one then we are using the U prefix, otherwise the S prefix. Then
2065  * we have the following opcode maps in the lower two bits:
2066  * XTAB16       00 iff Rn != 0xf
2067  * XTAB         10 iff Rn != 0xf
2068  * XTAH         11 iff Rn != 0xf
2069  * XTB16        00 iff Rn = 0xf
2070  * XTB          10 iff Rn = 0xf
2071  * XTH          11 iff Rn = 0xf
2072  */
2073 static int
2074 arm_dis_extend(uint32_t in, char *buf, size_t buflen)
2075 {
2076         uint8_t op, rot;
2077         int sbit;
2078         arm_cond_code_t cc;
2079         arm_reg_t rn, rm, rd;
2080         const char *opn;
2081         size_t len;


2084         rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2085         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2086         rm = in & ARM_MEDIA_RM_MASK;
2087         op = (in & ARM_MEDIA_SZE_OP_MASK) >> ARM_MEDIA_SZE_OP_SHIFT;
2088         rot = (in & ARM_MEDIA_SZE_ROT_MASK) >> ARM_MEDIA_SZE_ROT_SHIFT;
2089         sbit = in & ARM_MEDIA_SZE_S_MASK;
2090         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

2092         switch (op) {
2093         case 0x0:
2094                 opn = rn == ARM_REG_R15 ? "xtab16" : "xtb16";
1616                 opn = rn == ARM_REG_R15 ? "XTAB16" : "XTB16";
2095                 break;
2096         case 0x2:
2097                 opn = rn == ARM_REG_R15 ? "xtab" : "xtb";
1619                 opn = rn == ARM_REG_R15 ? "XTAB" : "XTB";
2098                 break;
2099         case 0x3:
2100                 opn = rn == ARM_REG_R15 ? "xtah" : "xth";
1622                 opn = rn == ARM_REG_R15 ? "XTAH" : "XTH";
2101                 break;
2102         default:
2103                 return (-1);
2104                 break;
2105         }

2107         if (rn == ARM_REG_R15) {
2108                 len = snprintf(buf, buflen, "%s%s%s %s, %s",
2109                     sbit != 0 ? "u" : "s",
1631                     sbit != 0 ? "U" : "S",
2110                     opn, arm_cond_names[cc], arm_reg_names[rd],
2111                     arm_reg_names[rn]);
2112         } else {
2113                 len = snprintf(buf, buflen, "%s%s%s %s, %s, %s",
2114                     sbit != 0 ? "u" : "s",
1636                     sbit != 0 ? "U" : "S",
2115                     opn, arm_cond_names[cc], arm_reg_names[rd],
2116                     arm_reg_names[rn], arm_reg_names[rm]);
```

```
2117         }

2119         if (len >= buflen)
2120                 return (-1);

2122         if (snprintf(buf + len, buflen - len, "%s",
2123             arm_extend_rot_names[rot]) >= buflen - len)
2124                 return (-1);
2125         return (0);
2126 }

2128 /*
2129  * The media instructions and extensions can be divided into different groups of
2130  * instructions. We first use bits 23 and 24 to figure out where to send it. We
2131  * call this group of bits the l1 mask.
2132  */
2133 static int
2134 arm_dis_media(uint32_t in, char *buf, size_t buflen)
2135 {
2136         uint8_t l1, op1, op2;
2137         arm_cond_code_t cc;
2138         arm_reg_t rd, rn, rs, rm;
2139         int xbit;
2140         size_t len;

2142         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
2143         l1 = (in & ARM_MEDIA_L1_MASK) >> ARM_MEDIA_L1_SHIFT;
2144         switch (l1) {
2145         case 0x0:
2146                 return (arm_dis_padd(in, buf, buflen));
2147                 break;
2148         case 0x1:
2149                 if ((in & ARM_MEDIA_HPACK_MASK) == ARM_MEDIA_HPACK_TARG) {
2150                         rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2151                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2152                         rm = in & ARM_MEDIA_RM_MASK;
2153                         op1 = (in & ARM_MEDIA_HPACK_SHIFT_MASK) >>
2154                             ARM_MEDIA_HPACK_SHIFT_IMM;
2155                         len = snprintf(buf, buflen, "%s%s %s, %s, %s",
2156                             (in & ARM_MEDIA_HPACK_OP_MASK) != 0 ?
2157                             "pkhtb" : "pkhbt", arm_cond_names[cc],
1679                             "PKHTB" : "PKHBT", arm_cond_names[cc],
2158                             arm_reg_names[rd], arm_reg_names[rn],
2159                             arm_reg_names[rd]);
2160                         if (len >= buflen)
2161                                 return (-1);

2163                         if (op1 != 0) {
2164                                 if (in & ARM_MEDIA_HPACK_OP_MASK)
2165                                         len += snprintf(buf + len, buflen - len,
2166                                             ", asr %d", op1);
1688                                             ", ASR %d", op1);
2167                                 else
2168                                         len += snprintf(buf + len, buflen - len,
2169                                             ", lsl %d", op1);
1691                                             ", LSL %d", op1);
2170                         }
2171                         return (len >= buflen ? -1 : 0);
2172                 }

2174                 if ((in & ARM_MEDIA_WSAT_MASK) == ARM_MEDIA_WSAT_TARG) {
2175                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2176                         rm = in & ARM_MEDIA_RM_MASK;
2177                         op1 = (in & ARM_MEDIA_SAT_IMM_MASK) >>
2178                             ARM_MEDIA_SAT_IMM_SHIFT;
2179                         op2 = (in & ARM_MEDIA_SAT_SHI_MASK) >>
```



```
2180                             ARM_MEDIA_SAT_SHI_SHIFT;
2181                         len = snprintf(buf, buflen, "%s%s %s, #%d, %s",
2182                             (in & ARM_MEDIA_SAT_U_MASK) != 0 ? "usat" : "ssat",
1704                             (in & ARM_MEDIA_SAT_U_MASK) != 0 ? "USAT" : "SSAT",
2183                             arm_cond_names[cc], arm_reg_names[rd], op1,
2184                             arm_reg_names[rm]);

2186                         if (len >= buflen)
2187                                 return (-1);

2189                         /*
2190                          * The shift is optional in the assembler and encoded as
2191                          * LSL 0. However if we get ASR 0, that means ASR #32.
2192                          * An ARM_MEDIA_SAT_STYPE_MASK of 0 is LSL, 1 is ASR.
2193                          */
2194                         if (op2 != 0 || (in & ARM_MEDIA_SAT_STYPE_MASK) == 1) {
2195                                 if (op2 == 0)
2196                                         op2 = 32;
2197                                 if (snprintf(buf + len, buflen - len,
2198                                     ", %s #%d",
2199                                     (in & ARM_MEDIA_SAT_STYPE_MASK) != 0 ?
2200                                     "asr" : "lsl", op2) >= buflen - len)
1722                                     "ASR" : "LSL", op2) >= buflen - len)
2201                                         return (-1);
2202                         }
2203                         return (0);
2204                 }

2206                 if ((in & ARM_MEDIA_PHSAT_MASK) == ARM_MEDIA_PHSAT_TARG) {
2207                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2208                         rm = in & ARM_MEDIA_RM_MASK;
2209                         op1 = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2210                         if (snprintf(buf, buflen, "%s%s %s, #%d, %s",
2211                             (in & ARM_MEDIA_SAT_U_MASK) != 0 ?
2212                             "usat16" : "ssat16",
1734                             "USAT16" : "SSAT16",
2213                             arm_cond_names[cc], arm_reg_names[rd], op1,
2214                             arm_reg_names[rm]) >= buflen)
2215                                 return (-1);
2216                         return (0);
2217                 }

2219                 if ((in & ARM_MEDIA_REV_MASK) == ARM_MEDIA_REV_TARG) {
2220                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2221                         rm = in & ARM_MEDIA_RM_MASK;
2222                         if (snprintf(buf, buflen, "rev%s %s, %s",
1744                         if (snprintf(buf, buflen, "REV%s %s, %s",
2223                             arm_cond_names[cc], arm_reg_names[rd],
2224                             arm_reg_names[rd]) >= buflen)
2225                                 return (-1);
2226                         return (0);
2227                 }

2229                 if ((in & ARM_MEDIA_BRPH_MASK) == ARM_MEDIA_BRPH_TARG) {
2230                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2231                         rm = in & ARM_MEDIA_RM_MASK;
2232                         if (snprintf(buf, buflen, "rev16%s %s, %s",
1754                         if (snprintf(buf, buflen, "REV16%s %s, %s",
2233                             arm_cond_names[cc], arm_reg_names[rd],
2234                             arm_reg_names[rd]) >= buflen)
2235                                 return (-1);
2236                         return (0);
2237                 }

2239                 if ((in & ARM_MEDIA_BRSH_MASK) == ARM_MEDIA_BRSH_TARG) {
2240                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
```

```
2241                        rm = in & ARM_MEDIA_RM_MASK;
2242                        if (snprintf(buf, buflen, "revsh%s %s, %s",
1764                        if (snprintf(buf, buflen, "REVSH%s %s, %s",
2243                            arm_cond_names[cc], arm_reg_names[rd],
2244                            arm_reg_names[rd]) >= buflen)
2245                                return (-1);
2246                        return (0);
2247                }

2249                if ((in & ARM_MEDIA_SEL_MASK) == ARM_MEDIA_SEL_TARG) {
2250                        rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2251                        rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2252                        rm = in & ARM_MEDIA_RM_MASK;
2253                        if (snprintf(buf, buflen, "sel%s %s, %s, %s",
1775                        if (snprintf(buf, buflen, "SEL%s %s, %s, %s",
2254                            arm_cond_names[cc], arm_reg_names[rd],
2255                            arm_reg_names[rn], arm_reg_names[rm]) >= buflen)
2256                                return (-1);
2257                        return (0);
2258                }

2260                if ((in & ARM_MEDIA_SZE_MASK) == ARM_MEDIA_SZE_TARG)
2261                        return (arm_dis_extend(in, buf, buflen));
2262                /* Unknown instruction */
2263                return (-1);
2264                break;
2265        case 0x2:
2266                /*
2267                 * This consists of the following multiply instructions:
2268                 * SMLAD, SMLSD, SMLALD, SMUAD, and SMUSD.
2269                 *
2270                 * SMLAD and SMUAD encoding are the same, switch on Rn == R15
2271                 * 22-20 are 000 7-6 are 00
2272                 * SMLSD and SMUSD encoding are the same, switch on Rn == R15
2273                 * 22-20 are 000 7-6 are 01
2274                 * SMLALD: 22-20 are 100 7-6 are 00
2275                 */
2276                rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2277                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2278                rs = (in & ARM_MEDIA_RS_MASK) >> ARM_MEDIA_RS_SHIFT;
2279                rm = in & ARM_MEDIA_RM_MASK;
2280                op1 = (in & ARM_MEDIA_OP1_MASK) >> ARM_MEDIA_OP1_SHIFT;
2281                op2 = (in & ARM_MEDIA_OP2_MASK) >> ARM_MEDIA_OP2_SHIFT;
2282                xbit = in & ARM_MEDIA_MULT_X_MASK;

2284                if (op1 == 0x0) {
2285                        if (op2 != 0x0 && op2 != 0x1)
2286                                return (-1);
2287                        if (rn == ARM_REG_R15) {
2288                                len = snprintf(buf, buflen, "%s%s%s %s, %s, %s",
2289                                    op2 != 0 ? "smusd" : "smuad",
2290                                    xbit != 0 ? "x" : "x",
1811                                    op2 != 0 ? "SMUSD" : "SMUAD",
1812                                    xbit != 0 ? "X" : "X",
2291                                    arm_cond_names[cc], arm_reg_names[rd],
2292                                    arm_reg_names[rm], arm_reg_names[rs]);
2293                        } else {
2294                                len = snprintf(buf, buflen,
2295                                    "%s%s%s %s, %s, %s, %s",
2296                                    op2 != 0 ? "smlsd" : "smlad",
2297                                    xbit != 0 ? "x" : "",
1818                                    op2 != 0 ? "SMLSD" : "SMLAD",
1819                                    xbit != 0 ? "X" : "",
2298                                    arm_cond_names[cc], arm_reg_names[rd],
2299                                    arm_reg_names[rm], arm_reg_names[rs],
2300                                    arm_reg_names[rn]);
```



```
2302                        }
2303                } else if (op1 == 0x8) {
2304                        if (op2 != 0x0)
2305                                return (-1);
2306                        len = snprintf(buf, buflen, "smlald%s%s %s, %s, %s, %s",
2307                            xbit != 0 ? "x" : "",
1828                        len = snprintf(buf, buflen, "SMLALD%s%s %s, %s, %s, %s",
1829                            xbit != 0 ? "X" : "",
2308                            arm_cond_names[cc], arm_reg_names[rn],
2309                            arm_reg_names[rd], arm_reg_names[rm],
2310                            arm_reg_names[rs]);
2311                } else
2312                        return (-1);

2314                return (len >= buflen ? -1 : 0);
2315                break;
2316        case 0x3:
2317                /*
2318                 * Here we handle USAD8 and USADA8. The main difference is the
2319                 * presence of RN. USAD8 is defined as having a value of rn that
2320                 * is not r15. If it is r15, then instead it is USADA8.
2321                 */
2322                if ((in & ARM_MEDIA_OP1_MASK) != 0)
2323                        return (-1);
2324                if ((in & ARM_MEDIA_OP2_MASK) != 0)
2325                        return (-1);

2327                cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
2328                rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2329                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2330                rs = (in & ARM_MEDIA_RS_MASK) >> ARM_MEDIA_RS_SHIFT;
2331                rm = in & ARM_MEDIA_RM_MASK;

2333                if (rn != ARM_REG_R15)
2334                        len = snprintf(buf, buflen, "usada8%s %s, %s, %s, %s",
1856                        len = snprintf(buf, buflen, "USADA8%s %s, %s, %s, %s",
2335                            arm_cond_names[cc], arm_reg_names[rd],
2336                            arm_reg_names[rm], arm_reg_names[rs],
2337                            arm_reg_names[rn]);
2338                else
2339                        len = snprintf(buf, buflen, "usad8%s %s, %s, %s",
1861                        len = snprintf(buf, buflen, "USAD8%s %s, %s, %s",
2340                            arm_cond_names[cc], arm_reg_names[rd],
2341                            arm_reg_names[rm], arm_reg_names[rs]);
2342                return (len >= buflen ? -1 : 0);
2343                break;
2344        default:
2345                return (-1);
2346        }
2347 }

2349 /*
2350  * Each instruction in the ARM instruction set is a uint32_t and in our case is
2351  * LE. The upper four bits determine the condition code. If the conditoin code
2352  * is undefined then we know to immediately jump there. Otherwise we go use the
2353  * next three bits to determine where we should go next and how to further
2354  * process the instruction in question. The ARM instruction manual doesn't
2355  * define this field so we're going to call it the L1_DEC or level 1 decoding
2356  * from which it will have to be further subdivided into the specific
2357  * instruction groupings that we care about.
2358  */
2359 static int
2360 arm_dis(dis_handle_t *dhp, uint32_t in, char *buf, size_t buflen)
2361 {
2362         uint8_t l1;
```

```
2363         arm_cond_code_t cc;

2365         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

2367         if (cc == ARM_COND_NACC)
2368                 return (arm_dis_uncond_insn(in, buf, buflen));

2370         l1 = (in & ARM_L1_DEC_MASK) >> ARM_L1_DEC_SHIFT;

2372         switch (l1) {
2373         case 0x0:
2374                 /*
2375                  * The l0 group is a bit complicated. We have several different
2376                  * groups of instructions to consider. The first question is
2377                  * whether bit 4 is zero or not. If it is, then we have a data
2378                  * processing immediate shift unless the opcode and + S bits
2379                  * (24-20) is of the form 0b10xx0.
2380                  *
2381                  * When bit 4 is 1, we have to then also look at bit 7. If bit
2382                  * 7 is one then we know that this is the class of multiplies /
2383                  * extra load/stores. If bit 7 is zero then we have the same
2384                  * opcode games as we did above.
2385                  */
2386                 if (in & ARM_L1_0_B4_MASK) {
2387                         if (in & ARM_L1_0_B7_MASK) {
2388                                 /*
2389                                  * Both the multiplication extensions and the
2390                                  * load and store extensions live in this
2391                                  * region. The load and store extensions can be
2392                                  * identified by having at least one of bits 5
2393                                  * and 6 set. The exceptions to this are the
2394                                  * SWP and SWPB instructions and the exclusive
2395                                  * load and store instructions which, unlike the
2396                                  * multiplication instructions. These have
2397                                  * specific values for the bits in the range of
2398                                  * 20-24.
2399                                  */
2400                                 if ((in & ARM_L1_0_ELS_MASK) != 0)
2401                                         /* Extra loads/stores */
2402                                         return (arm_dis_els(in, buf, buflen));
2403                                 if ((in & ARM_ELS_SWAP_MASK) == ARM_ELS_IS_SWAP)
2404                                         return (arm_dis_swap(in, buf, buflen));
2405                                 if ((in & ARM_ELS_EXCL_MASK) ==
2406                                     ARM_ELS_EXCL_MASK)
2407                                         return (arm_dis_lsexcl(in, buf,
2408                                             buflen));
2409                                 /* Multiplication instruction extension A3-3. */
2410                                 return (arm_dis_extmul(in, buf, buflen));
2411                         }
2412                         if ((in & ARM_L1_0_OPMASK) == ARM_L1_0_SPECOP &&
2413                             !(in & ARM_L1_0_SMASK)) {
2414                                 /* Misc. Instructions A3-4 */
2415                                 return (arm_dis_cdsp_ext(in, buf, buflen));
2416                         } else {
2417                                 /* data processing register shift */
2418                                 return (arm_dis_dpi(in, cc, buf, buflen));
2419                         }
2420                 } else {
2421                         if ((in & ARM_L1_0_OPMASK) == ARM_L1_0_SPECOP &&
2422                             !(in & ARM_L1_0_SMASK))
2423                                 /* Misc. Instructions A3-4 */
2424                                 return (arm_dis_cdsp_ext(in, buf, buflen));
2425                         else {
2426                                 /* Data processing immediate shift */
2427                                 return (arm_dis_dpi(in, cc, buf, buflen));
2428                         }
```



```
2429                 }
2430                 break;
2431         case 0x1:
2432                 /*
2433                  * In l1 group 0b001 there are a few ways to tell things apart.
2434                  * We are directed to first look at bits 20-24. Data processing
2435                  * immediate has a 4 bit opcode 24-21 followed by an S bit. We
2436                  * know it is not a data processing immediate if we have
2437                  * something of the form 0b10xx0.
2438                  */
2439                 if ((in & ARM_L1_1_OPMASK) == ARM_L1_1_SPECOP &&
2440                     !(in & ARM_L1_1_SMASK)) {
2441                         if (in & ARM_L1_1_UNDEF_MASK) {
2442                                 /* Undefined instructions */
2443                                 return (-1);
2444                         } else {
2445                                 /* Move immediate to status register */
2446                                 return (arm_dis_status_regs(in, buf, buflen));
2447                         }
2448                 } else {
2449                         /* Data processing immedaite */
2450                         return (arm_dis_dpi(in, cc, buf, buflen));
2451                 }
2452                 break;
2453         case 0x2:
2454                 /* Load/store Immediate offset */
2455                 return (arm_dis_ldstr(in, buf, buflen));
2456                 break;
2457         case 0x3:
2458                 /*
2459                  * Like other sets we use the 4th bit to make an intial
2460                  * determination. If it is zero then this is a load/store
2461                  * register offset class instruction. Following that we have a
2462                  * specical mask of 0x01f000f0 to determine whether this is an
2463                  * architecturally undefined instruction type or not.
2464                  *
2465                  * The architecturally undefined are parts of the current name
2466                  * space that just aren't used, but could be used at some point
2467                  * in the future. For now though, it's an invalid op code.
2468                  */
2469                 if (in & ARM_L1_3_B4_MASK) {
2470                         if ((in & ARM_L1_3_ARCHUN_MASK) ==
2471                             ARM_L1_3_ARCHUN_MASK) {
2472                                 /* Architecturally undefined */
2473                                 return (-1);
2474                         } else {
2475                                 /* Media instructions */
2476                                 return (arm_dis_media(in, buf, buflen));
2477                         }
2478                 } else {
2479                         /* Load/store register offset */
2480                         return (arm_dis_ldstr(in, buf, buflen));
2481                 }
2482                 break;
2483         case 0x4:
2484                 /* Load/store multiple */
2485                 return (arm_dis_ldstr_multi(in, buf, buflen));
2486                 break;
2487         case 0x5:
2488                 /* Branch and Branch with link */
2489                 return (arm_dis_branch(dhp, in, buf, buflen));
2490                 break;
2491         case 0x6:
2492                 /* coprocessor load/store && double register transfers */
2493                 return (arm_dis_coproc_lsdrt(in, buf, buflen));
2494                 break;
```

```
2495         case 0x7:
2496                 /*
2497                  * In l1 group 0b111 you can determine the three groups using
2498                  * the following logic. If the next bit after the l1 group (bit
2499                  * 24) is one than you know that it is a software interrupt.
2500                  * Otherwise it is one of the coprocessor instructions.
2501                  * Furthermore you can tell apart the data processing from the
2502                  * register transfers based on bit 4. If it is zero then it is
2503                  * a data processing instruction, otherwise it is a register
2504                  * transfer.
2505                  */
2506                 if (in & ARM_L1_7_SWINTMASK) {
2507                         /*
2508                          * The software interrupt is pretty straightforward. The
2509                          * lower 24 bits are the interrupt number. It's also
2510                          * valid for it to run with a condition code.
2511                          */
2512                         if (snprintf(buf, buflen, "swi%s %d",
2034                         if (snprintf(buf, buflen, "SWI%s %d",
2513                             arm_cond_names[cc],
2514                             in & ARM_SWI_IMM_MASK) >= buflen)
2515                                 return (-1);
2516                         return (0);
2517                 } else if (in & ARM_L1_7_COPROCMASK) {
2518                         /* coprocessor register transfers */
2519                         return (arm_dis_coproc_rt(in, buf, buflen));
2520                 } else {
2521                         /* coprocessor data processing */
2522                         return (arm_dis_coproc_dp(in, buf, buflen));
2523                 }
2524                 break;
2525         }

2527         return (-1);
2528 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    3334 Mon Jan 26 17:25:38 2015
new/usr/src/uts/armv6/bcm2835/ml/locore.s
bcm2835: move strict alignment check disable code into the loader
Since the loader wants to muck with alignment related bits of the SCTLR
anyway, it should set both A and U to the desired values (0 and 1
respectively).
**********************************************************
```

```
   1 /*
   2  * This file and its contents are supplied under the terms of the
   3  * Common Development and Distribution License ("CDDL"), version 1.0.
   4  * You may only use this file in accordance with the terms of version
   5  * 1.0 of the CDDL.
   6  *
   7  * A full copy of the text of the CDDL should have accompanied this
   8  * source.  A copy of the CDDL is also available via the Internet at
   9  * http://www.illumos.org/license/CDDL.
  10  */

  12 /*
  13  * Copyright 2013 (c) Joyent, Inc. All rights reserved.
  14  * Copyright 2015 (c) Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
  15  */

  17 #include <sys/asm_linkage.h>
  18 #include <sys/machparam.h>
  19 #include <sys/cpu_asm.h>

  21 /*
  22  * Every story needs a beginning. This is ours.
  23  */

  25 /*
  26  * We are in a primordial world here. The BMC2835 is going to come along and
  27  * boot us at _start. Normally we would go ahead and use a main() function, but
  28  * for now, we'll do that ourselves. As we've started the world, we also need to
  29  * set up a few things about us, for example our stack pointer. To help us out,
  30  * it's useful to remember the rough memory map. Remember, this is for physcial
  31  * addresses. There is no virtual memory here. These sizes are often manipulated
  32  * by the 'configuration' in the bootloader.
  33  *
  34  * +----------------+ <---- Max physical memory
  35  * |                |
  36  * |                |
  37  * |                |
  38  * +----------------+
  39  * |                |
  40  * |      I/O       |
  41  * |  Peripherals   |
  42  * |                |
  43  * +----------------+ <---- I/O base 0x20000000 (corresponds to 0x7E000000)
  44  * |                |
  45  * |     Main       |
  46  * |    Memory      |
  47  * |                |
  48  * +----------------+ <---- Top of SDRAM
  49  * |                |
  50  * |       VC       |
  51  * |     SDRAM      |
  52  * |                |
  53  * +----------------+ <---- Split determined by bootloader config
  54  * |                |
  55  * |      ARM       |
  56  * |     SDRAM      |
  57  * |                |
  58  * +----------------+ <---- Bottom of physical memory 0x00000000
```



```
  59  *
  60  * With the Raspberry Pi Model B, we have 512 MB of SDRAM. That means we have a
  61  * range of addresses from [0, 0x20000000). If we assume that the minimum amount
  62  * of DRAM is given to the GPU - 32 MB, that means we really have the following
  63  * range: [0, 0x1e000000).
  64  *
  65  * By default, this binary will be loaded into 0x8000. For now, that means we
  66  * will set our initial stack to 0x10000000.
  67  */

  69 /*
  70  * Recall that _start is the traditional entry point for an ELF binary.
  71  */
  72         ENTRY(_start)
  73         ldr sp, =t0stack
  74         ldr r4, =DEFAULTSTKSZ
  75         add sp, r4
  76         bic sp, sp, #0xff

  78         /*
  79          * establish bogus stacks for exceptional CPU states, our exception
  80          * code should never make use of these, and we want loud and violent
  81          * failure should we accidentally try.
  82          */
  83         cps #(CPU_MODE_UND)
  84         mov sp, #-1
  85         cps #(CPU_MODE_ABT)
  86         mov sp, #-1
  87         cps #(CPU_MODE_FIQ)
  88         mov sp, #-1
  89         cps #(CPU_MODE_IRQ)
  90         mov sp, #-1
  91         cps #(CPU_MODE_SVC)

  93         /* Enable highvecs (moves the base of the exception vector) */
  94         mrc     p15, 0, r3, c1, c0, 0
  95         mov     r4, #1
  96         lsl     r4, r4, #13
  97         orr     r3, r3, r4
  98         mcr     p15, 0, r3, c1, c0, 0

 100         /* Disable A (disables strict alignment checks) */
 101         mrc     p15, 0, r3, c1, c0, 0
 102         bic     r3, r3, #2
 103         mcr     p15, 0, r3, c1, c0, 0

 100         /* Enable access to p10 and p11 (privileged mode only) */
 101         mrc     p15, 0, r0, c1, c0, 2
 102         orr     r0, #0x00500000
 103         mcr     p15, 0, r0, c1, c0, 2

 105         bl _fakebop_start
 106         SET_SIZE(_start)
______unchanged_portion_omitted_
```

```
**********************************************************
    2390 Mon Jan 26 17:25:38 2015
new/usr/src/uts/armv6/loader/fakeloader_core.s
bcm2835: move strict alignment check disable code into the loader
Since the loader wants to muck with alignment related bits of the SCTLR
anyway, it should set both A and U to the desired values (0 and 1
respectively).
**********************************************************
______unchanged_portion_omitted_

  40 #if defined(__lint)

  42 /* ARGSUSED */
  43 void
  44 fakeload_unaligned_enable(void)
  45 {}

  47 #else   /* __lint */

  49         /*
  50          * Fix up alignment by turning off A and by turning on U.
  51          */
  52         ENTRY(fakeload_unaligned_enable)
  53         mrc     p15, 0, r0, c1, c0, 0
  54         orr     r0, #0x400000   /* U = 1 */
  55         bic     r0, r0, #2      /* A = 0 */
  54         orr     r0, #0x400000
  56         mcr     p15, 0, r0, c1, c0, 0
  57         bx      lr
  58         SET_SIZE(fakeload_unaligned_enable);
______unchanged_portion_omitted_
```

```
**********************************************************
    4520 Mon Jan 26 17:25:39 2015
new/usr/src/uts/armv6/ml/trap.s
trap: don't mislead about SCTLR.V being not set
It's been quite a while since the exception table got moved via the high
vector bit (SCTLR.V).
**********************************************************
```

```
   1 /*
   2  * This file and its contents are supplied under the terms of the
   3  * Common Development and Distribution License ("CDDL"), version 1.0.
   4  * You may only use this file in accordance with the terms of version
   5  * 1.0 of the CDDL.
   6  *
   7  * A full copy of the text of the CDDL should have accompanied this
   8  * source.  A copy of the CDDL is also available via the Internet at
   9  * http://www.illumos.org/license/CDDL.
  10  */

  12 .file   "trap.s"

  14 #include <sys/asm_linkage.h>
  15 #include <sys/cpu_asm.h>

  17 /*
  18  * Create a section into which to place the exception vector, such that we can
  19  * force it to be mapped where it needs to be.
  19  * force it to be mapped where it needs to be.  Currently at 0x0, since
  20  * we don't set the high vector bit (SCTLR.V)
  20  *
  21  * Each instruction in the vector jumps to its own address + 24, which is the
  22  * matching entry in exception_table.  We do this to insure that we get
  23  * effectively infinite displacement, which we can't achieve in one
  24  * instruction otherwise (and gas complains).
  25  *
  26  * The handler for each exception type saves a trap frame (struct regs, though
  27  * with some bits unusual) and calls the associated handler from trap_table.
  28  *
  29  * XXX: On CPUs with the security extensions, there are actually 2 additional
  30  * exception vectors: secure, and monitor.  We ignore them.
  31  */
  32 .pushsection ".exception_vector", "ax"
  33         ldr     pc, [pc, #24]
  34         ldr     pc, [pc, #24]
  35         ldr     pc, [pc, #24]
  36         ldr     pc, [pc, #24]
  37         ldr     pc, [pc, #24]
  38         ldr     pc, [pc, #24]
  39         ldr     pc, [pc, #24]
  40         ldr     pc, [pc, #24]
  41 .exception_table:
  42         .word handle_reset              /* Reset */
  43         .word handle_undef              /* Undefined insn */
  44         .word handle_svc                /* Supervisor Call */
  45         .word handle_prefetchabt        /* Prefetch abort */
  46         .word handle_dataabt            /* Data abort */
  47         .word 0x00000014                /* Reserved (infinite loops) */
  48         .word handle_irq                /* IRQ */
  49         .word handle_fiq                /* FIQ */
  50 .popsection

  52 /* Clobbers r0 */
  53 #define CALL_HANDLER(scratch, trapno)
  54         ldr     scratch, =(trap_table + (4 * trapno));
  55         ldr     scratch, [scratch];
  56         cmp     scratch, #0;
  57         beq     1f;
```



```
  58         mov     r0, sp;         /* Pass our saved frame to the handler */
  59         blx     scratch;        /* Call the handler */
  60 1:

  62 /*
  63  * XXX: Note that we go to some contortions here to save in 'struct regs' style.
  64  *
  65  * This includes saving our own lr/spsr, for our own return _and_ saving them
  66  * into the 'struct regs', and doing the register save unusually such that we
  67  * get the 'struct regs' in order.
  68  *
  69  * Depending on the exact nature of 'struct regs', perhaps we should be
  70  * bending it to our will, rather than letting it bend us?
  71  *
  72  * XXX: Note also that we're saving the current registers assuming that we
  73  * came from supervisor mode.  We should probably be saving the banked
  74  * registers based on the mode in the spsr.  (or we'll screw up when nested,
  75  * or when userland traps, or...)
  76  */
  77 #define PUSH_TRAP_FRAME(scratch)
  78         srsdb   sp!, #(CPU_MODE_SVC);           /* Save lr and spsr for ourselve
  79         cps     #(CPU_MODE_SVC);
  80         ldr     lr, [sp];                       /* Get lr back for the trap fram
  81         sub     sp, sp, #(4 * 17);              /* Space for all our registers *
  82         stmia   sp, {r0-r14};                   /* XXX: Note we don't save pc */
  83         ldr     scratch, [sp, #(4 * 18)];       /* skip 17 regs and the lr */
  84         str     scratch, [sp, #(4 * 16)];

  86 #define POP_TRAP_FRAME_AND_RET()
  87         ldmia   sp, {r0-r14};
  88         add     sp, sp, #(4 * 19);      /* 17 regs + 2 for lr and spsr */
  89         rfedb   sp;                     /* Return */

  91 /*
  92  * XXX: None of these handlers are even vaguely aware that usermode exists,
  93  * and make no effort to do the traditional things we would probably do on
  94  * returning to user mode.  They will need to be rethought at such a time
  95  */
  96 .globl trap_table

  98 #define DEFINE_HANDLER(name, code)              \
  99         ENTRY(name)                             \
 100                 PUSH_TRAP_FRAME(r0)             \
 101                 CALL_HANDLER(r1, code)          \
 102                 POP_TRAP_FRAME_AND_RET()        \
 103         SET_SIZE(name)
______unchanged_portion_omitted_
```