```
**********************************************************
    3911 Wed Jan 14 19:07:05 2015
new/./README.xbuild
README.xbuild update
**********************************************************
```

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

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
```

Clone this repository again and do a normal i386 on i386 build.

```
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
```

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

```
**********************************************************
   78569 Wed Jan 14 19:07:05 2015
new/usr/src/lib/libdisasm/common/dis_arm.c
libdisasm: fully decode DPI immediate
The immediate value can be rotated.  Instead of dumping the 8-bit immediate
and 4-bit rotate field, we should dump the effective immediate value (both
in decimal and hex) to ease reading the disassembly.  This matches GNU
objdump behavior.
**********************************************************
______unchanged_portion_omitted_
```

```
 666 /*
 667  * There are sixteen data processing instructions (dpi). They come in a few
 668  * different forms which are based on whether immediate values are used and
 669  * whether or not some special purpose shifting is done. We use this one entry
 670  * point to cover all the different types.
 671  *
 672  * From the ARM arch manual:
 673  *
 674  * <opcode1>{<cond>}{S} <Rd>,<shifter>
 675  * <opcode1> := MOV | MVN
 676  * <opcode2>{<cond>} <Rn>,<shifter>
 677  * <opcode2> := CMP, CMN, TST, TEQ
 678  * <opcode3>{<cond>{S} <Rd>,<Rn>, <shifter>
 679  * <opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
 680  *
 681  * 31 - 28|27 26 |25 | 24-21  |20 | 19-16 | 15-12 | 11 - 0
 682  * [ cond | 0  0 | I | opcode | S | Rn    | Rd    | shifter ]
 683  *
 684  * I bit: Determines whether shifter_operand is immediate or register based
 685  * S bit: Determines whether or not the insn updates condition codes
 686  * Rn:    First source operand register
 687  * Rd:    Destination register
 688  * shifter: Specifies the second operand
 689  *
 690  * There are three primary encodings:
 691  *
 692  * 32-bit immediate
 693  * 31 - 28|27 26|25 |24-21 |20|19-16| 15-12|11 - 8    |7 - 0
 694  * [ cond | 0  0| 1 |opcode| S|Rn   | Rd   |rotate_imm|immed_8 ]
 695  *
 696  * Immediate shifts
 697  * 31 - 28|27 26|25 |24-21 |20|19-16|15-12|11 - 7   |6 5  |4|3-0
 698  * [ cond | 0  0| 0 |opcode| S|Rn   |Rd   |shift_imm|shift|0|Rm ]
 699  *
 700  * Register shifts
 701  * 31 - 28|27 26|25 |24-21 |20|19-16|15-12|11 - 8|7|6 5  |4|3-0
 702  * [ cond | 0  0| 0 |opcode| S|Rn   |Rd   |Rs    |0|shift|1|Rm ]
 703  *
 704  * There are four different kinds of shifts that work with both immediate and
 705  * register shifts:
 706  *   o Logical shift left  0b00 (LSL)
 707  *   o Logical shift right 0b01 (LSR)
 708  *   o Arithmetic shift right 0b10 (ASR)
 709  *   o Rotate right 0b11 (ROR)
 710  * There is one special shift which only works with immediate shift format:
 711  *   o If shift_imm = 0 and shift = 0b11, then it is a rotate right with extend
 712  *   (RRX)
 713  *
 714  * Finally there is one special indication for no shift. An immediate shift
 715  * whose shift_imm = shift = 0. This is a shortcut to a direct value from the
 716  * register.
 717  *
 718  * While processing this, we first build up all the information into the
 719  * arm_dpi_inst_t and then from there we go and print out the format based on
 720  * the opcode and shifter. As per the rough grammar above we have to print
```



```
 721  * different sets of instructions in different ways.
 722  */
 723 static int
 724 arm_dis_dpi(uint32_t in, arm_cond_code_t cond, char *buf, size_t buflen)
 725 {
 726         arm_dpi_inst_t dpi_inst;
 727         int ibit, bit4;
 728         size_t len;

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

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

 739         if (ibit) {
 740                 /* 32-bit immediate */
 741                 dpi_inst.dpii_stype = ARM_DPI_SHIFTER_IMM32;
 742                 dpi_inst.dpii_un.dpii_im.dpisi_rot = (in &
 743                     ARM_DPI_IMM_ROT_MASK) >> ARM_DPI_IMM_ROT_SHIFT;
 744                 dpi_inst.dpii_un.dpii_im.dpisi_imm = in & ARM_DPI_IMM_VAL_MASK;
 745         } else if (bit4) {
 746                 /* Register shift */
 747                 dpi_inst.dpii_stype = ARM_DPI_SHIFTER_SREG;
 748                 dpi_inst.dpii_un.dpii_ri.dpisr_val = (in &
 749                     ARM_DPI_REGS_RS_MASK) >> ARM_DPI_REGS_RS_SHIFT;
 750                 dpi_inst.dpii_un.dpii_ri.dpisr_targ = in &
 751                     ARM_DPI_REGS_RM_MASK;
 752                 dpi_inst.dpii_un.dpii_ri.dpisr_code = in &
 753                     ARM_DPI_REGS_SHIFT_MASK >> ARM_DPI_REGS_SHIFT_SHIFT;
 754         } else {
 755                 /* Immediate shift */
 756                 dpi_inst.dpii_stype = ARM_DPI_SHIFTER_SIMM;
 757                 dpi_inst.dpii_un.dpii_si.dpiss_imm = (in &
 758                     ARM_DPI_IMS_SHIMM_MASK) >> ARM_DPI_IMS_SHIMM_SHIFT;
 759                 dpi_inst.dpii_un.dpii_si.dpiss_code = (in &
 760                     ARM_DPI_IMS_SHIFT_MASK) >> ARM_DPI_IMS_SHIFT_SHIFT;
 761                 dpi_inst.dpii_un.dpii_si.dpiss_targ = in & ARM_DPI_IMS_RM_MASK;
 762                 if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_ROR &&
 763                     dpi_inst.dpii_un.dpii_si.dpiss_imm == 0)
 764                         dpi_inst.dpii_un.dpii_si.dpiss_code = DPI_S_RRX;

 766                 if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_LSL &&
 767                     dpi_inst.dpii_un.dpii_si.dpiss_imm == 0)
 768                         dpi_inst.dpii_un.dpii_si.dpiss_code = DPI_S_NONE;
 769         }

 771         /*
 772          * Print everything before the shifter based on the instruction
 773          */
 774         switch (dpi_inst.dpii_op) {
 775         case DPI_OP_MOV:
 776         case DPI_OP_MVN:
 777                 len = snprintf(buf, buflen, "%s%s%s %s",
 778                     arm_dpi_opnames[dpi_inst.dpii_op],
 779                     arm_cond_names[dpi_inst.dpii_cond],
 780                     dpi_inst.dpii_sbit != 0 ? "S" : "",
 781                     arm_reg_names[dpi_inst.dpii_rd]);
 782                 break;
 783         case DPI_OP_CMP:
 784         case DPI_OP_CMN:
 785         case DPI_OP_TST:
 786         case DPI_OP_TEQ:
```

```
787                 len = snprintf(buf, buflen, "%s%s %s",
788                     arm_dpi_opnames[dpi_inst.dpii_op],
789                     arm_cond_names[dpi_inst.dpii_cond],
790                     arm_reg_names[dpi_inst.dpii_rn]);
791                 break;
792         default:
793                 len = snprintf(buf, buflen,
794                     "%s%s%s %s, %s", arm_dpi_opnames[dpi_inst.dpii_op],
795                     arm_cond_names[dpi_inst.dpii_cond],
796                     dpi_inst.dpii_sbit != 0 ? "S" : "",
797                     arm_reg_names[dpi_inst.dpii_rd],
798                     arm_reg_names[dpi_inst.dpii_rn]);
799                 break;
800         }

802         if (len >= buflen)
803                 return (-1);
804         buflen -= len;
805         buf += len;

807         /*
808          * Print the shifter as appropriate
809          */
810         switch (dpi_inst.dpii_stype) {
811         case ARM_DPI_SHIFTER_IMM32: {
812                 uint32_t imm = dpi_inst.dpii_un.dpii_im.dpisi_imm;
813                 int rot = dpi_inst.dpii_un.dpii_im.dpisi_rot * 2;
814                 if (rot != 0)
815                         imm = (imm << (32 - rot)) | (imm >> rot);
816                 if (imm < 10)
817                         len = snprintf(buf, buflen, ", #%d", imm);
818                 else
819                         len = snprintf(buf, buflen, ", #%d  ; %#x", imm, imm);
811         case ARM_DPI_SHIFTER_IMM32:
812                 len = snprintf(buf, buflen, ", #%d, %d",
813                     dpi_inst.dpii_un.dpii_im.dpisi_imm,
814                     dpi_inst.dpii_un.dpii_im.dpisi_rot);
820                 break;
821         }
822 #endif /* ! codereview */
823         case ARM_DPI_SHIFTER_SIMM:
824                 if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_NONE) {
825                         len = snprintf(buf, buflen, ", %s",
826                             arm_reg_names[dpi_inst.dpii_un.dpii_si.dpiss_targ]);
827                         break;
828                 }
829                 if (dpi_inst.dpii_un.dpii_si.dpiss_code == DPI_S_RRX) {
830                         len = snprintf(buf, buflen, ", %s RRX",
831                             arm_reg_names[dpi_inst.dpii_un.dpii_si.dpiss_targ]);
832                         break;
833                 }
834                 len = snprintf(buf, buflen, ", %s, %s #%d",
835                     arm_reg_names[dpi_inst.dpii_un.dpii_si.dpiss_targ],
836                     arm_dpi_shifts[dpi_inst.dpii_un.dpii_si.dpiss_code],
837                     dpi_inst.dpii_un.dpii_si.dpiss_imm);
838                 break;
839         case ARM_DPI_SHIFTER_SREG:
840                 len = snprintf(buf, buflen, ", %s, %s %s",
841                     arm_reg_names[dpi_inst.dpii_un.dpii_ri.dpisr_targ],
842                     arm_dpi_shifts[dpi_inst.dpii_un.dpii_ri.dpisr_code],
843                     arm_reg_names[dpi_inst.dpii_un.dpii_ri.dpisr_val]);
844                 break;
845         }

847         return (len < buflen ? 0 : -1);
848 }
```



```
850 /*
851  * This handles the byte and word size loads and stores. It does not handle the
852  * multi-register loads or the 'extra' ones. The instruction has the generic
853  * form off:
854  *
855  * 31 - 28|27 26 |25|24|23|22|21|20|19-16|15-12|11 - 0
856  * [ cond | 0  0 |I |P |U |B |W |L | Rn | Rd   |mode_specific]
857  *
858  * Here the bits mean the following:
859  *
860  * Rn: The base register used by the addressing mode
861  * Rd: The register to load to or store from
862  * L bit: If L==1 then a load, else store
863  * B bit: If B==1 then work on a byte, else a 32-bit word
864  *
865  * The remaining pieces determine the mode we are operating in:
866  * I bit: If 0 use immediate offsets, otherwise if 1 used register based offsets
867  * P bit: If 0 use post-indexed addressing. If 1, indexing mode is either offset
868  *        addessing or pre-indexed addressing based on the W bit.
869  * U bit: If 1, offset is added to base, if 0 offset is subtracted from base
870  * W bit: This bits interpretation varies based on the P bit. If P is zero then
871  *        W indicates whether a normal memory access is performed or if a read
872  *        from user memory is performed (W = 1).
873  *        If P is 1 then then when W = 0 the base register is not updated and
874  *        when W = 1 the calculated address is written back to the base
875  *        register.
876  *
877  * Based on these combinations there are a total of nine different operating
878  * modes, though not every LDR and STR variant can reach them all.
879  */
880 static int
881 arm_dis_ldstr(uint32_t in, char *buf, size_t buflen)
882 {
883         arm_cond_code_t cc;
884         arm_reg_t rd, rn, rm;
885         int ibit, pbit, ubit, bbit, wbit, lbit;
886         arm_dpi_shift_code_t sc;
887         uint8_t simm;
888         size_t len;

890         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
891         ibit = in & ARM_LS_IBIT_MASK;
892         pbit = in & ARM_LS_PBIT_MASK;
893         ubit = in & ARM_LS_UBIT_MASK;
894         bbit = in & ARM_LS_BBIT_MASK;
895         wbit = in & ARM_LS_WBIT_MASK;
896         lbit = in & ARM_LS_LBIT_MASK;
897         rd = (in & ARM_LS_RD_MASK) >> ARM_LS_RD_SHIFT;
898         rn = (in & ARM_LS_RN_MASK) >> ARM_LS_RN_SHIFT;

900         len = snprintf(buf, buflen, "%s%s%s%s %s, ", lbit != 0 ? "LDR" : "STR",
901             arm_cond_names[cc], bbit != 0 ? "B" : "",
902             (pbit == 0 && wbit != 0) ? "T" : "",
903             arm_reg_names[rd]);
904         if (len >= buflen)
905                 return (-1);

907         /* Figure out the specifics of the encoding for the rest */
908         if (ibit == 0 && pbit != 0) {
909                 /*
910                  * This is the immediate offset mode (A5.2.2). That means that
911                  * we have something of the form [ <Rn>, #+/-<offset_12> ]. All
912                  * of the mode specific bits contribute to offset_12. We also
913                  * handle the pre-indexed version (A5.2.5) which depends on the
914                  * wbit being set.
```

```
915                  */
916                 len += snprintf(buf + len, buflen - len, "[%s, #%s%d]%s",
917                     arm_reg_names[rn], ubit != 0 ? "" : "-",
918                     in & ARM_LS_IMM_MASK, wbit != 0 ? "!" : "");
919         } else if (ibit != 0 && pbit != 0) {
920                 /*
921                  * This handles A5.2.2, A5.2.3, A5.2.6, and A5.2.7. We can have
922                  * one of two options. If the non-rm bits (11-4) are all zeros
923                  * then we have a special case of a register offset is just
924                  * being added. Otherwise we have a scaled register offset where
925                  * the shift code matters.
926                  */
927                 rm = in & ARM_LS_REG_RM_MASK;
928                 len += snprintf(buf + len, buflen - len, "[%s, %s%s",
929                     arm_reg_names[rn], ubit != 0 ? "" : "-",
930                     arm_reg_names[rm]);
931                 if (len >= buflen)
932                         return (-1);
933                 if ((in & ARM_LS_REG_NRM_MASK) != 0) {
934                         simm = (in & ARM_LS_SCR_SIMM_MASK) >>
935                             ARM_LS_SCR_SIMM_SHIFT;
936                         sc = (in & ARM_LS_SCR_SCODE_MASK) >>
937                             ARM_LS_SCR_SCODE_SHIFT;
938
939                         if (simm == 0 && sc == DPI_S_ROR)
940                                 sc = DPI_S_RRX;
941
942                         len += snprintf(buf + len, buflen - len, "%s",
943                             arm_dpi_shifts[sc]);
944                         if (len >= buflen)
945                                 return (-1);
946                         if (sc != DPI_S_RRX) {
947                                 len += snprintf(buf + len, buflen - len, " #%d",
948                                     simm);
949                                 if (len >= buflen)
950                                         return (-1);
951                         }
952                 }
953                 len += snprintf(buf + len, buflen - len, "]%s",
954                     wbit != 0 ? "!" : "");
955         } else if (ibit == 0 && pbit == 0 && wbit == 0) {
956                 /* A5.2.8 immediate post-indexed */
957                 len += snprintf(buf + len, buflen - len, "[%s], #%s%d",
958                     arm_reg_names[rn], ubit != 0 ? "" : "-",
959                     in & ARM_LS_IMM_MASK);
960         } else if (ibit != 0 && pbit == 0 && wbit == 0) {
961                 /* A5.2.9 and A5.2.10 */
962                 rm = in & ARM_LS_REG_RM_MASK;
963                 len += snprintf(buf + len, buflen - len, "[%s], %s%s",
964                     arm_reg_names[rn], ubit != 0 ? "" : "-",
965                     arm_reg_names[rm]);
966                 if ((in & ARM_LS_REG_NRM_MASK) != 0) {
967                         simm = (in & ARM_LS_SCR_SIMM_MASK) >>
968                             ARM_LS_SCR_SIMM_SHIFT;
969                         sc = (in & ARM_LS_SCR_SCODE_MASK) >>
970                             ARM_LS_SCR_SCODE_SHIFT;
971
972                         if (simm == 0 && sc == DPI_S_ROR)
973                                 sc = DPI_S_RRX;
974
975                         len += snprintf(buf + len, buflen - len, "%s",
976                             arm_dpi_shifts[sc]);
977                         if (len >= buflen)
978                                 return (-1);
979                         if (sc != DPI_S_RRX)
980                                 len += snprintf(buf + len, buflen - len,
```



```
981                                     " #%d", simm);
982                 }
983         }
984
985         return (len < buflen ? 0 : -1);
986 }
987
988 /*
989  * This handles load and store multiple instructions. The general format is as
990  * follows:
991  *
992  * 31 - 28|27 26 25|24|23|22|21|20|19-16|15-0
993  * [ cond | 1  0 0 |P |U |S |W |L | Rn | register set
994  *
995  * The register set has one bit per register. If a bit is set it indicates that
996  * register and if it is not set then it indicates that the register is not
997  * included in this.
998  *
999  * S bit: If the instruction is a LDM and we load the PC, the S == 1 tells us to
1000 * load the CPSR from SPSR after the other regs are loaded. If the instruction
1001 * is a STM or LDM without touching the PC it indicates that if we are
1002 * privileged we should send the banked registers.
1003 *
1004 * L bit: Where this is a load or store. Load is active high.
1005 *
1006 * P bit: If P == 0 then Rn is included in the memory region transfers and its
1007 * location is dependent on the U bit. It is at the top (U == 0) or bottom (U ==
1008 * 1). If P == 1 then it is excluded and lies one word beyond the top (U == 0)
1009 * or bottom based on the U bit.
1010 *
1011 * U bit: If U == 1 then the transfer is made upwards and if U == 0 then the
1012 * transfer is made downwards.
1013 *
1014 * W bit: If set then we incremet the base register after the transfer. It is
1015 * modified by 4 times the number of registers in the list. If the U bit is
1016 * positive then that value is added to Rn otherwise it is subtracted.
1017 *
1018 * The overal layout for this is
1019 * (LDM|STM){<cond>}<addressing mode> Rn{!}, <registers>{^}. Here the ! is based
1020 * on having the W bit set. The ^ bit depends on whether S is set or not.
1021 *
1022 * There are four normal addressing modes: IA, IB, DA, DB. There are also
1023 * corresponding stack addressing modes that exist. However we have no way of
1024 * knowing which are the ones being used, therefore we are going to default to
1025 * the non-stack versions which are listed as the primary.
1026 *
1027 * Finally the last useful bit is how the registers list is specified. It is a
1028 * comma separated list inside of { }. However, a user may separate a contiguous
1029 * range by the use of a -, eg. R0 - R4. However, it is impossible for us to map
1030 * back directly to what the user did. So for now, we punt on second down and
1031 * instead just list each indidvidual register rather than attempt a joining
1032 * routine.
1033 */
1034 static int
1035 arm_dis_ldstr_multi(uint32_t in, char *buf, size_t buflen)
1036 {
1037         int sbit, wbit, lbit, ii, cont;
1038         uint16_t regs, addr_mode;
1039         arm_reg_t rn;
1040         arm_cond_code_t cc;
1041         size_t len;
1042
1043         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1044         sbit = in & ARM_LSM_SBIT_MASK;
1045         wbit = in & ARM_LSM_WBIT_MASK;
1046         lbit = in & ARM_LSM_LBIT_MASK;
```

```
1047         rn = (in & ARM_LSM_RN_MASK) >> ARM_LSM_RN_SHIFT;
1048         regs = in & ARM_LSM_RLIST_MASK;
1049         addr_mode = (in & ARM_LSM_ADDR_MASK) >> ARM_LSM_ADDR_SHIFT;

1051         len = snprintf(buf, buflen, "%s%s%s %s%s, { ",
1052             lbit != 0 ? "LDM" : "STM",
1053             arm_cond_names[cc],
1054             arm_lsm_mode_names[addr_mode],
1055             arm_reg_names[rn],
1056             wbit != 0 ? "!" : "");

1058         cont = 0;
1059         for (ii = 0; ii < 16; ii++) {
1060                 if (!(regs & (1 << ii)))
1061                         continue;

1063                 len += snprintf(buf + len, buflen - len, "%s%s",
1064                     cont > 0 ? ", " : "", arm_reg_names[ii]);
1065                 if (len >= buflen)
1066                         return (-1);
1067                 cont++;
1068         }

1070         len += snprintf(buf + len, buflen - len, " }%s", sbit != 0 ? "^" : "");
1071         return (len >= buflen ? -1 : 0);
1072 }

1074 /*
1075  * Here we need to handle miscillaneous loads and stores. This is used to load
1076  * and store signed and unsigned half words. To load a signed byte. And to load
1077  * and store double words. There is no specific store routines for signed bytes
1078  * and halfwords as they are supposed to use the SRB and STRH. There are two
1079  * primary encodings this time. The general case looks like:
1080  *
1081  * 31 - 28|27 - 25|24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1082  * [ cond |   0   |P |U |I |W |L | Rn | Rd   |amode|1|S|H|1|amode ]
1083  *
1084  * The I, P, U, and W bits specify the addressing mode.
1085  * The L, S, and H bits describe the type and size.
1086  * Rn: The base register used by the addressing mode
1087  * Rd: The register to load to or store from
1088  *
1089  * The other bits specifically mean:
1090  * I bit: If set to one the address specific pieces are immediate. Otherwise
1091  * they aren't.
1092  * P bit: If P is 0 used post-indexed addressing. If P is 1 its behavior is
1093  * based on the value of W.
1094  * U bit: If U is one the offset is added to the base otherwise subtracted
1095  * W bit: When P is one a value of W == 1 says that the resulting memory address
1096  * should be written back to the base register. The base register isn't touched
1097  * when W is zero.
1098  *
1099  * The L, S, and H bits combine in the following table:
1100  *
1101  *  L | S | H | Meaning
1102  *  ------------------
1103  *  0 | 0 | 1 | store halfword
1104  *  0 | 1 | 0 | load doubleword
1105  *  0 | 1 | 1 | store doubleword
1106  *  1 | 0 | 1 | load unsigned half word
1107  *  1 | 1 | 0 | load signed byte
1108  *  1 | 1 | 1 | load signed halfword
1109  *
1110  * The final format of this is:
1111  * LDR|STR{<cond>}H|SH|SB|D <rd>, address_mode
1112  */
```



```
1113 static int
1114 arm_dis_els(uint32_t in, char *buf, size_t buflen)
1115 {
1116         arm_cond_code_t cc;
1117         arm_reg_t rn, rd;
1118         const char *iname, *suffix;
1119         int lbit, sbit, hbit, pbit, ubit, ibit, wbit;
1120         uint8_t imm;
1121         size_t len;

1123         lbit = in & ARM_ELS_LBIT_MASK;
1124         sbit = in & ARM_ELS_SBIT_MASK;
1125         hbit = in & ARM_ELS_SBIT_MASK;

1127         if (lbit || (sbit && hbit == 0))
1128                 iname = "LDR";
1129         else
1130                 iname = "STR";

1132         if (sbit == 0 && hbit)
1133                 suffix = "H";
1134         else if (lbit == 0)
1135                 suffix = "D";
1136         else if (sbit && hbit == 0)
1137                 suffix = "SB";
1138         else if (sbit && hbit)
1139                 suffix = "SH";

1141         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1142         rn = (in & ARM_ELS_RN_MASK) >> ARM_ELS_RN_SHIFT;
1143         rd = (in & ARM_ELS_RD_MASK) >> ARM_ELS_RD_SHIFT;

1145         len = snprintf(buf, buflen, "%s%s%s %s, ", iname, arm_cond_names[cc],
1146             suffix, arm_reg_names[rd]);
1147         if (len >= buflen)
1148                 return (-1);

1150         pbit = in & ARM_ELS_PBIT_MASK;
1151         ubit = in & ARM_ELS_UBIT_MASK;
1152         ibit = in & ARM_ELS_IBIT_MASK;
1153         wbit = in & ARM_ELS_WBIT_MASK;

1155         if (pbit && ibit) {
1156                 /* Handle A5.3.2 and A5.3.4 immediate offset and pre-indexed */
1157                 /* Bits 11-8 form the upper 4 bits of imm */
1158                 imm = (in & ARM_ELS_UP_AM_MASK) >> (ARM_ELS_UP_AM_SHIFT - 4);
1159                 imm |= in & ARM_ELS_LOW_AM_MASK;
1160                 len += snprintf(buf + len, buflen - len, "[%s, #%s%d]%s",
1161                     arm_reg_names[rn],
1162                     ubit != 0 ? "" : "-", imm,
1163                     wbit != 0 ? "!" : "");
1164         } else if (pbit && ibit == 0) {
1165                 /* Handle A5.3.3 and A5.3.5 register offset and pre-indexed */
1166                 len += snprintf(buf + len, buflen - len, "[%s %s%s]%s",
1167                     arm_reg_names[rn],
1168                     ubit != 0 ? "" : "-",
1169                     arm_reg_names[in & ARM_ELS_LOW_AM_MASK],
1170                     wbit != 0 ? "!" : "");
1171         } else if (pbit == 0 && ibit) {
1172                 /* A5.3.6 Immediate post-indexed */
1173                 /* Bits 11-8 form the upper 4 bits of imm */
1174                 imm = (in & ARM_ELS_UP_AM_MASK) >> (ARM_ELS_UP_AM_SHIFT - 4);
1175                 imm |= in & ARM_ELS_LOW_AM_MASK;
1176                 len += snprintf(buf + len, buflen - len, "[%s], #%s%d",
1177                     arm_reg_names[rn], ubit != 0 ? "" : "-", imm);
1178         } else if (pbit == 0 && ibit == 0) {
```

```
1179                /* Handle A 5.3.7 Register post-indexed */
1180                len += snprintf(buf + len, buflen - len, "[%s], %s%s",
1181                    arm_reg_names[rn], ubit != 0 ? "" : "-",
1182                    arm_reg_names[in & ARM_ELS_LOW_AM_MASK]);
1183        }

1185        return (len >= buflen ? -1 : 0);
1186 }

1188 /*
1189  * Handle SWP and SWPB out of the extra loads/stores extensions.
1190  */
1191 static int
1192 arm_dis_swap(uint32_t in, char *buf, size_t buflen)
1193 {
1194        arm_cond_code_t cc;
1195        arm_reg_t rn, rd, rm;

1197        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1198        rn = (in & ARM_ELS_RN_MASK) >> ARM_ELS_RN_SHIFT;
1199        rd = (in & ARM_ELS_RD_MASK) >> ARM_ELS_RD_SHIFT;
1200        rm = in & ARM_ELS_RN_MASK;

1202        if (snprintf(buf, buflen, "SWP%s%s %s, %s, [%s]",
1203            arm_cond_names[cc],
1204            (in & ARM_ELS_SWAP_BYTE_MASK) ? "B" : "",
1205            arm_reg_names[rd], arm_reg_names[rm], arm_reg_names[rn]) >=
1206            buflen)
1207                return (-1);

1209        return (0);
1210 }

1212 /*
1213  * Handle LDREX and STREX out of the extra loads/stores extensions.
1214  */
1215 static int
1216 arm_dis_lsexcl(uint32_t in, char *buf, size_t buflen)
1217 {
1218        arm_cond_code_t cc;
1219        arm_reg_t rn, rd, rm;
1220        int lbit;
1221        size_t len;

1223        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1224        rn = (in & ARM_ELS_RN_MASK) >> ARM_ELS_RN_SHIFT;
1225        rd = (in & ARM_ELS_RD_MASK) >> ARM_ELS_RD_SHIFT;
1226        rm = in & ARM_ELS_RN_MASK;
1227        lbit = in & ARM_ELS_LBIT_MASK;

1229        len = snprintf(buf, buflen, "%s%sEX %s, ",
1230            lbit != 0 ? "LDR" : "STR",
1231            arm_cond_names[cc], arm_reg_names[rd]);
1232        if (len >= buflen)
1233                return (-1);

1235        if (lbit)
1236                len += snprintf(buf + len, buflen - len, "[%s]",
1237                    arm_reg_names[rn]);
1238        else
1239                len += snprintf(buf + len, buflen - len, "%s, [%s]",
1240                    arm_reg_names[rm], arm_reg_names[rn]);
1241        return (len >= buflen ? -1 : 0);
1242 }

1244 /*
```



```
1245  * This is designed to handle the multiplication instruction extension space.
1246  * Note that this doesn't actually cover all of the multiplication instructions
1247  * available in ARM, but all of the ones that are in this space. This includes
1248  * the following instructions:
1249  *
1250  *
1251  * There are three basic encoding formats:
1252  *
1253  * Multipy (acc):
1254  * 31 - 28|27 - 24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1255  * [ cond |   0   |0 |0 | A |S |Rn  | Rd   |Rs   |1|0|0|1|Rm ]
1256  *
1257  * Unsigned multipy acc acc long
1258  * 31 - 28|27 - 24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1259  * [ cond |   0   |0 |1 |0 |0 |RdHi |RdLo  |Rs   |1|0|0|1|Rm ]
1260  *
1261  * Multiply (acc) long:
1262  * 31 - 28|27 - 24|23|22|21|20|19-16|15-12|11-8 |7|6|5|4|3-0
1263  * [ cond |   0   |1 |Un|A |S |RdHi| RdLo |Rs   |1|0|0|1|Rm ]
1264  *
1265  * A bit: Accumulate
1266  * Un bit: Unsigned is active low, signed is active high
1267  * S bit: Indicates whethere the status register should be updated.
1268  *
1269  * MLA(S) and MUL(S) make up the first type of instructions.
1270  * UMAAL makes up the second group.
1271  * (U|S)MULL(S), (U|S)MLAL(S), Make up the third.
1272  */
1273 static int
1274 arm_dis_extmul(uint32_t in, char *buf, size_t buflen)
1275 {
1276        arm_cond_code_t cc;
1277        arm_reg_t rd, rn, rs, rm;
1278        size_t len;

1280        /*
1281         * RdHi is equal to rd here. RdLo is equal to Rn here.
1282         */
1283        rd = (in & ARM_EMULT_RD_MASK) >> ARM_EMULT_RD_SHIFT;
1284        rn = (in & ARM_EMULT_RN_MASK) >> ARM_EMULT_RN_SHIFT;
1285        rs = (in & ARM_EMULT_RS_MASK) >> ARM_EMULT_RS_SHIFT;
1286        rm = in & ARM_EMULT_RM_MASK;

1288        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

1290        if ((in & ARM_EMULT_MA_MASK) == 0) {
1291                if (in & ARM_EMULT_ABIT_MASK) {
1292                        len = snprintf(buf, buflen, "MLA%s%s %s, %s, %s, %s",
1293                            arm_cond_names[cc],
1294                            (in & ARM_EMULT_SBIT_MASK) ? "S" : "",
1295                            arm_reg_names[rd], arm_reg_names[rm],
1296                            arm_reg_names[rs], arm_reg_names[rs]);
1297                } else {
1298                        len = snprintf(buf, buflen, "MUL%s%s %s, %s, %s",
1299                            arm_cond_names[cc],
1300                            (in & ARM_EMULT_SBIT_MASK) ? "S" : "",
1301                            arm_reg_names[rd], arm_reg_names[rm],
1302                            arm_reg_names[rs]);

1304                }
1305        } else if ((in & ARM_EMULT_UMA_MASK) == ARM_EMULT_UMA_TARG) {
1306                len = snprintf(buf, buflen, "UMAAL%s %s, %s, %s, %s",
1307                    arm_cond_names[cc], arm_reg_names[rn], arm_reg_names[rd],
1308                    arm_reg_names[rm], arm_reg_names[rs]);
1309        } else if ((in & ARM_EMULT_MAL_MASK) == ARM_EMULT_MAL_TARG) {
1310                len = snprintf(buf, buflen, "%s%s%s%s %s, %s, %s, %s",
```

```
1311                    (in & ARM_EMULT_UNBIT_MASK) ? "S" : "U",
1312                    (in & ARM_EMULT_ABIT_MASK) ? "MLAL" : "MULL",
1313                    arm_cond_names[cc],
1314                    (in & ARM_EMULT_SBIT_MASK) ? "S" : "",
1315                    arm_reg_names[rn], arm_reg_names[rd], arm_reg_names[rm],
1316                    arm_reg_names[rs]);
1317            } else {
1318                    /* Not a supported instruction in this space */
1319                    return (-1);
1320            }
1321            return (len >= buflen ? -1 : 0);
1322    }

1324    /*
1325     * Here we handle the three different cases of moving to and from the various
1326     * status registers in both register mode and in immediate mode.
1327     */
1328    static int
1329    arm_dis_status_regs(uint32_t in, char *buf, size_t buflen)
1330    {
1331            arm_cond_code_t cc;
1332            arm_reg_t rd, rm;
1333            uint8_t field;
1334            int imm;
1335            size_t len;

1337            cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

1339            if ((in & ARM_CDSP_MRS_MASK) == ARM_CDSP_MRS_TARG) {
1340                    rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1341                    if (snprintf(buf, buflen, "MRS%s %s, %s", arm_cond_names[cc],
1342                        arm_reg_names[rd],
1343                        (in & ARM_CDSP_STATUS_RBIT) != 0 ? "SPSR" : "CPSR") >=
1344                        buflen)
1345                            return (-1);
1346                    return (0);
1347            }

1349            field = (in & ARM_CDSP_MSR_F_MASK) >> ARM_CDSP_MSR_F_SHIFT;
1350            len = snprintf(buf, buflen, "MSR%s %s_%s, ", arm_cond_names[cc],
1351                (in & ARM_CDSP_STATUS_RBIT) != 0 ? "SPSR" : "CPSR",
1352                arm_cdsp_msr_field_names[field]);
1353            if (len >= buflen)
1354                    return (-1);

1356            if (in & ARM_CDSP_MSR_ISIMM_MASK) {
1357                    imm = in & ARM_CDSP_MSR_IMM_MASK;
1358                    imm <<= (in & ARM_CDSP_MSR_RI_MASK) >> ARM_CDSP_MSR_RI_SHIFT;
1359                    len += snprintf(buf + len, buflen - len, "#%d", imm);
1360            } else {
1361                    rm = in & ARM_CDSP_RM_MASK;
1362                    len += snprintf(buf + len, buflen - len, "%s",
1363                        arm_reg_names[rm]);
1364            }

1366            return (len >= buflen ? -1 : 0);
1367    }

1369    /*
1370     * Here we need to handle the Control And DSP instruction extension space. This
1371     * consists of several different instructions. Unlike other extension spaces
1372     * there isn't as much tha tis similar here as there is stuff that is different.
1373     * Oh well, that's a part of life. Instead we do a little bit of additional
1374     * parsing here.
1375     *
1376     * The first group that we separate out are the instructions that interact with
```



```
1377     * the status registers. Those are handled in their own function.
1378     */
1379    static int
1380    arm_dis_cdsp_ext(uint32_t in, char *buf, size_t buflen)
1381    {
1382            uint16_t imm, op;
1383            arm_cond_code_t cc;
1384            arm_reg_t rd, rm, rn, rs;
1385            size_t len;

1387            if ((in & ARM_CDSP_STATUS_MASK) == ARM_CDSP_STATUS_TARG)
1388                    return (arm_dis_status_regs(in, buf, buflen));

1390            cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

1392            /*
1393             * This gets the Branch/exchange as well as the Branch and link/exchange
1394             * pieces. These generally also transform the instruction set into
1395             * something we can't actually disassemble. Here the lower mask and
1396             * target is the opposite. eg. the target bits are not what we want.
1397             */
1398            if ((in & ARM_CDSP_BEX_UP_MASK) == ARM_CDSP_BEX_UP_TARG &&
1399                (in & ARM_CDSP_BEX_LOW_MASK) != ARM_CDSP_BEX_NLOW_TARG) {
1400                    rm = in & ARM_CDSP_RM_MASK;
1401                    imm = (in & ARM_CDSP_BEX_TYPE_MASK) >> ARM_CDSP_BEX_TYPE_SHIFT;
1402                    if (snprintf(buf, buflen, "B%s%s %s",
1403                        imm == ARM_CDSP_BEX_TYPE_X ? "X" :
1404                        imm == ARM_CDSP_BEX_TYPE_J ? "XJ" : "LX",
1405                        arm_cond_names[cc], arm_reg_names[rm]) >= buflen)
1406                            return (-1);
1407                    return (0);
1408            }

1410            /* Count leading zeros */
1411            if ((in & ARM_CDSP_CLZ_MASK) == ARM_CDSP_CLZ_TARG) {
1412                    rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1413                    rm = in & ARM_CDSP_RM_MASK;
1414                    if (snprintf(buf, buflen, "CLZ%s %s, %s", arm_cond_names[cc],
1415                        arm_reg_names[rd], arm_reg_names[rm]) >= buflen)
1416                            return (-1);
1417                    return (0);
1418            }

1420            if ((in & ARM_CDSP_SAT_MASK) == ARM_CDSP_SAT_TARG) {
1421                    rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1422                    rn = (in & ARM_CDSP_RN_MASK) >> ARM_CDSP_RN_SHIFT;
1423                    rm = in & ARM_CDSP_RM_MASK;
1424                    imm = (in & ARM_CDSP_SAT_OP_MASK) >> ARM_CDSP_SAT_OP_SHIFT;
1425                    if (snprintf(buf, buflen, "Q%s%s %s, %s, %s",
1426                        arm_cdsp_sat_opnames[imm], arm_cond_names[cc],
1427                        arm_reg_names[rd], arm_reg_names[rm],
1428                        arm_reg_names[rn]) >= buflen)
1429                            return (-1);
1430                    return (0);
1431            }

1433            /*
1434             * Breakpoint instructions are a bit different. While they are in the
1435             * conditional instruction namespace, they actually aren't defined to
1436             * take a condition. That's just how it rolls. The breakpoint is a
1437             * 16-bit value. The upper 12 bits are stored together and the lower
1438             * four together.
1439             */
1440            if ((in & ARM_CDSP_BKPT_MASK) == ARM_CDSP_BKPT_TARG) {
1441                    if (cc != ARM_COND_NACC)
1442                            return (-1);
```

```
1443                imm = (in & ARM_CDSP_BKPT_UIMM_MASK) >>
1444                    ARM_CDSP_BKPT_UIMM_SHIFT;
1445                imm <<= 4;
1446                imm |= (in & ARM_CDSP_BKPT_LIMM_MASK);
1447                if (snprintf(buf, buflen, "BKPT %d", imm) >= buflen)
1448                        return (1);
1449                return (0);
1450        }

1452        /*
1453         * Here we need to handle another set of multiplies. Specifically the
1454         * Signed multiplies. This is SMLA<x><y>, SMLAW<y>, SMULW<y>,
1455         * SMLAL<x><y>, SMUL<x><y>. These instructions all follow the form:
1456         *
1457         * 31 - 28|27-25|24|23|22-21|20|19-16|15-12|11 - 8|7|6|5|4|3-0
1458         * [ cond |  0  | 1| 0| op. | 0|Rn   |Rd   |Rs    |1|y|x|0|Rm ]
1459         *
1460         * If x is one a T is used for that part of the name. Otherwise a B is.
1461         * The same holds true for y.
1462         *
1463         * These instructions map to the following opcodes:
1464         * SMLA<x><y>: 00,
1465         * SMLAW<y>: 01 and x is zero,
1466         * SMULW<y>: 01 and x is one ,
1467         * SMLAL<x><y>: 10,
1468         * SMUL<xy><y>: 11
1469         */
1470        if ((in & ARM_CDSP_SMUL_MASK) == ARM_CDSP_SMUL_TARG) {
1471                rd = (in & ARM_CDSP_RD_MASK) >> ARM_CDSP_RD_SHIFT;
1472                rn = (in & ARM_CDSP_RN_MASK) >> ARM_CDSP_RN_SHIFT;
1473                rs = (in & ARM_CDSP_RS_MASK) >> ARM_CDSP_RS_SHIFT;
1474                rm = in & ARM_CDSP_RM_MASK;
1475                op = (in & ARM_CDSP_SMUL_OP_MASK) >> ARM_CDSP_SMUL_OP_SHIFT;

1477                switch (op) {
1478                case 0:
1479                        len = snprintf(buf, buflen, "SMLA%s%s%s %s, %s, %s, %s",
1480                            (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "T" : "B",
1481                            (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" : "B",
1482                            arm_cond_names[cc], arm_reg_names[rd],
1483                            arm_reg_names[rm], arm_reg_names[rs],
1484                            arm_reg_names[rn]);
1485                        break;
1486                case 1:
1487                        if (in & ARM_CDSP_SMUL_X_MASK) {
1488                                len = snprintf(buf, buflen,
1489                                    "SMULW%s%s %s, %s, %s",
1490                                    (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" :
1491                                    "B", arm_cond_names[cc], arm_reg_names[rd],
1492                                    arm_reg_names[rm], arm_reg_names[rs]);
1493                        } else {
1494                                len = snprintf(buf, buflen,
1495                                    "SMLAW%s%s %s, %s, %s %s",
1496                                    (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" :
1497                                    "B", arm_cond_names[cc], arm_reg_names[rd],
1498                                    arm_reg_names[rm], arm_reg_names[rs],
1499                                    arm_reg_names[rn]);
1500                        }
1501                        break;
1502                case 2:
1503                        len = snprintf(buf, buflen,
1504                            "SMLAL%s%s%s %s, %s, %s, %s",
1505                            (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "T" : "B",
1506                            (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" : "B",
1507                            arm_cond_names[cc], arm_reg_names[rd],
1508                            arm_reg_names[rn], arm_reg_names[rm],
```



```
1509                                arm_reg_names[rs]);
1510                        break;
1511                case 3:
1512                        len = snprintf(buf, buflen, "SMUL%s%s%s %s, %s, %s",
1513                            (in & ARM_CDSP_SMUL_X_MASK) != 0 ? "T" : "B",
1514                            (in & ARM_CDSP_SMUL_Y_MASK) != 0 ? "T" : "B",
1515                            arm_cond_names[cc], arm_reg_names[rd],
1516                            arm_reg_names[rm], arm_reg_names[rs]);
1517                        break;
1518                default:
1519                        return (-1);
1520                }
1521                return (len >= buflen ? -1 : 0);
1522        }

1524        /*
1525         * If we got here then this is some other instructin we don't know
1526         * about in the instruction extensino space.
1527         */
1528        return (-1);
1529 }

1531 /*
1532  * Coprocessor double register transfers
1533  *
1534  * MCRR:
1535  * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11-8|7-4|3-0
1536  * [ cond |1 1 0| 0| 0| 1| 0| 0| Rn  |  Rd |cp #|op |CRm
1537  *
1538  * MRRC:
1539  * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11-8|7-4|3-0
1540  * [ cond |1 1 0| 0| 0| 1| 0| 1| Rn  |  Rd |cp #|op |CRm
1541  *
1542  */
1543 static int
1544 arm_dis_coproc_drt(uint32_t in, char *buf, size_t buflen)
1545 {
1546        arm_cond_code_t cc;
1547        arm_reg_t rd, rn, rm;
1548        uint8_t coproc, op;
1549        const char *ccn;
1550        size_t len;

1552        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1553        coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
1554        rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
1555        rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
1556        rm = in & ARM_COPROC_RM_MASK;
1557        op = (in & ARM_COPROC_DRT_OP_MASK) >> ARM_COPROC_DRT_OP_SHIFT;

1559        if (cc == ARM_COND_NACC)
1560                ccn = "2";
1561        else
1562                ccn = arm_cond_names[cc];

1564        len = snprintf(buf, buflen, "%s%s %s, #%d, %s, %s, C%s",
1565            (in & ARM_COPROC_DRT_DIR_MASK) != 0 ? "MRRC" : "MCRR",
1566            ccn, arm_coproc_names[coproc], op, arm_reg_names[rd],
1567            arm_reg_names[rn], arm_reg_names[rm]);
1568        return (len >= buflen ? -1 : 0);
1569 }

1571 /*
1572  * This serves as both the entry point for the normal load and stores as well as
1573  * the double register transfers (MCRR and MRCC). If it is a register transfer
1574  * then we quickly send it off.
```

```
 * LDC:
 * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11 - 8|7 - 0
 * [ cond |1 1 0| P| U| N| W| L| Rn  | CRd | cp # | off ]
 *
 * STC:
 * 31 - 28|27-25|24|23|22|21|20|19-16|15-12|11 - 8|7 - 0
 * [ cond |1 1 0| P| U| N| W| L| Rn  | CRd | cp # | off ]
 *
 * Here the bits mean:
 *
 * P bit: If P is zero, it is post-indexed or unindexed based on W. If P is 1
 * then it is offset-addressing or pre-indexed based on W again.
 *
 * U bit: If U is positive then the offset if added, subtracted otherwise.. Note
 * that if P is zero and W is zero, U must be one.
 *
 * N bit: If set that means that we have a Long size, this bit is set by the L
 * suffix, not to be confused with the L bit.
 *
 * W bit: If W is one then the memory address is written back to the base
 * register. Further W = 0 and P = 0 is unindexed addressing. W = 1, P = 0 is
 * post-indexed. W = 0, P = 1 is offset addressing and W = 1, P = 1 is
 * pre-indexed.
 */
static int
arm_dis_coproc_lsdrt(uint32_t in, char *buf, size_t buflen)
{
        arm_cond_code_t cc;
        arm_reg_t rn, rd;
        uint8_t coproc;
        uint32_t imm;
        int pbit, ubit, nbit, wbit, lbit;
        const char *ccn;
        size_t len;

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

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

        pbit = in & ARM_COPROC_LS_P_MASK;
        ubit = in & ARM_COPROC_LS_U_MASK;
        nbit = in & ARM_COPROC_LS_N_MASK;
        wbit = in & ARM_COPROC_LS_W_MASK;
        lbit = in & ARM_COPROC_LS_L_MASK;

        if (cc == ARM_COND_NACC)
                ccn = "2";
        else
                ccn = arm_cond_names[cc];

        len = snprintf(buf, buflen, "%s%s%s %s, C%s, ",
            lbit != 0 ? "LDC" : "STC", ccn, nbit != 0 ? "L" : "",
            arm_coproc_names[coproc], arm_reg_names[rd]);
        if (len >= buflen)
                return (-1);

        if (pbit != 0) {
                imm *= 4;
                len += snprintf(buf + len, buflen - len, "[%s, #%s%d]%s",
                    arm_reg_names[rn],
                    ubit != 0 ? "" : "-", imm,
```



```
                    wbit != 0 ? "!" : "");
        } else if (wbit != 0) {
                imm *= 4;
                len += snprintf(buf + len, buflen - len, "[%s], #%s%d",
                    arm_reg_names[rn], ubit != 0 ? "" : "-", imm);
        } else {
                len += snprintf(buf + len, buflen - len, "[%s], { %d }",
                    arm_reg_names[rn], imm);
        }
        return (len >= buflen ? -1 : 0);
}

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

        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
        coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
        rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
        rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
        rm = in & ARM_COPROC_RM_MASK;
        op1 = (in & ARM_COPROC_CDP_OP1_MASK) >> ARM_COPROC_CDP_OP1_SHIFT;
        op2 = (in & ARM_COPROC_CDP_OP2_MASK) >> ARM_COPROC_CDP_OP2_SHIFT;

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

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

        return (0);
}

/*
 * Here we handle coprocesser single register transfers.
 *
 * MCR:
 * 31 - 28|27 - 24|23-21|20|19-16|15-12|11 - 8|7 - 5|4|3-0
 * [ cond |1 1 1 0| op_1| 0| CRn |  Rd | cp # | op_2|1|CRm ]
 *
 * MRC:
 * 31 - 28|27 - 24|23-21|20|19-16|15-12|11 - 8|7 - 5|4|3-0
 * [ cond |1 1 1 0| op_1| 1| CRn |  Rd | cp # | op_2|1|CRm ]
 */
static int
arm_dis_coproc_rt(uint32_t in, char *buf, size_t buflen)
{
```

```
1707         arm_cond_code_t cc;
1708         arm_reg_t rn, rd, rm;
1709         uint8_t op1, op2, coproc;
1710         const char *ccn;
1711         size_t len;

1713         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1714         coproc = (in & ARM_COPROC_NUM_MASK) >> ARM_COPROC_NUM_SHIFT;
1715         rn = (in & ARM_COPROC_RN_MASK) >> ARM_COPROC_RN_SHIFT;
1716         rd = (in & ARM_COPROC_RD_MASK) >> ARM_COPROC_RD_SHIFT;
1717         rm = in & ARM_COPROC_RM_MASK;
1718         op1 = (in & ARM_COPROC_CRT_OP1_MASK) >> ARM_COPROC_CRT_OP1_SHIFT;
1719         op2 = (in & ARM_COPROC_CRT_OP2_MASK) >> ARM_COPROC_CRT_OP2_SHIFT;

1721         if (cc == ARM_COND_NACC)
1722                 ccn = "2";
1723         else
1724                 ccn = arm_cond_names[cc];

1726         len = snprintf(buf, buflen, "%s%s %s, #%d, %s, C%s, C%s",
1727             (in & ARM_COPROC_CRT_DIR_MASK) != 0 ? "MRC" : "MCR", ccn,
1728             arm_coproc_names[coproc], op1, arm_reg_names[rd],
1729             arm_reg_names[rn], arm_reg_names[rm]);
1730         if (len >= buflen)
1731                 return (-1);

1733         if (op2 != 0)
1734                 if (snprintf(buf + len, buflen - len, ", #%d", op2) >=
1735                     buflen - len)
1736                         return (-1);
1737         return (0);
1738 }

1740 /*
1741  * Here we handle the set of unconditional instructions.
1742  */
1743 static int
1744 arm_dis_uncond_insn(uint32_t in, char *buf, size_t buflen)
1745 {
1746         int imm, sc;
1747         arm_reg_t rn, rm;
1748         size_t len;

1750         /*
1751          * The CPS instruction is a bit complicated. It has the following big
1752          * pattern which maps to a few different ways to use it:
1753          *
1754          *
1755          * 31-28|27-25|24|23-20|19-18|17 |16|15-9|8|7|6|5|4-0
1756          *    1 |  0  | 1| 0   |imod|mmod| 0|SBZ |A|I|F|0|mode
1757          *
1758          * CPS<effect> <iflags> {, #<mode> }
1759          * CPS #<mode>
1760          *
1761          * effect: determines what to do with the A, I, F interrupt bits in the
1762          * CPSR. effect is encoded in the imod field. It is either enable
1763          * interrupts 0b10 or disable interrupts 0b11. Recall that interrupts
1764          * are active low in the CPSR. If effect is not specified then this is
1765          * strictly a mode change which is required.
1766          *
1767          * A, I, F: If effect is specified then the bits which are high are
1768          * modified by the instruction.
1769          *
1770          * mode: Specifies a mode to change to. mmod will be 1 if mode is set.
1771          *
1772          */
```



```
1773         if ((in & ARM_UNI_CPS_MASK) == ARM_UNI_CPS_TARG) {
1774                 imm = (in & ARM_UNI_CPS_IMOD_MASK) > ARM_UNI_CPS_IMOD_SHIFT;

1776                 /* Ob01 is not a valid value for the imod */
1777                 if (imm == 1)
1778                         return (-1);

1780                 if (imm != 0)
1781                         len = snprintf(buf, buflen, "CPS%s %s%s%s%s",
1782                             imm == 2 ? "IE" : "ID",
1783                             (in & ARM_UNI_CPS_A_MASK) ? "a" : "",
1784                             (in & ARM_UNI_CPS_I_MASK) ? "i" : "",
1785                             (in & ARM_UNI_CPS_F_MASK) ? "f" : "",
1786                             (in & ARM_UNI_CPS_MMOD_MASK) ? " ," : "");
1787                 else
1788                         len = snprintf(buf, buflen, "CPS ");
1789                 if (len >= buflen)
1790                         return (-1);

1792                 if (in & ARM_UNI_CPS_MMOD_MASK)
1793                         if (snprintf(buf + len, buflen - len, "#%d",
1794                             in & ARM_UNI_CPS_MODE_MASK) >= buflen - len)
1795                                 return (-1);
1796                 return (0);
1797         }

1799         if ((in & ARM_UNI_SE_MASK) == ARM_UNI_SE_TARG) {
1800                 if (snprintf(buf, buflen, "SETEND %s",
1801                     (in & ARM_UNI_SE_BE_MASK) ? "BE" : "LE") >= buflen)
1802                         return (-1);
1803                 return (0);
1804         }

1806         /*
1807          * The cache preload is like a load, but it has a much simpler set of
1808          * constraints. The only valid bits that you can transform are the I and
1809          * the U bits. We have to use pre-indexed addressing. This means that we
1810          * only have the U bit and the I bit. See arm_dis_ldstr for a full
1811          * explanation of what's happening here.
1812          */
1813         if ((in & ARM_UNI_PLD_MASK) == ARM_UNI_PLD_TARG) {
1814                 rn = (in & ARM_LS_RN_MASK) >> ARM_LS_RN_SHIFT;
1815                 if ((in & ARM_LS_IBIT_MASK) == 0) {
1816                         if (snprintf(buf, buflen, "PLD [%s, #%s%d",
1817                             arm_reg_names[rn],
1818                             (in & ARM_LS_UBIT_MASK) != 0 ? "" : "-",
1819                             in & ARM_LS_IMM_MASK) >= buflen)
1820                                 return (-1);
1821                         return (0);
1822                 }

1824                 rm = in & ARM_LS_REG_RM_MASK;
1825                 len = snprintf(buf, buflen, "PLD [%s, %s%s", arm_reg_names[rn],
1826                     (in & ARM_LS_UBIT_MASK) != 0 ? "" : "-",
1827                     arm_reg_names[rm]);
1828                 if (len >= buflen)
1829                         return (-1);

1831                 if ((in & ARM_LS_REG_NRM_MASK) != 0) {
1832                         imm = (in & ARM_LS_SCR_SIMM_MASK) >>
1833                             ARM_LS_SCR_SIMM_SHIFT;
1834                         sc = (in & ARM_LS_SCR_SCODE_MASK) >>
1835                             ARM_LS_SCR_SCODE_SHIFT;

1837                         if (imm == 0 && sc == DPI_S_ROR)
1838                                 sc = DPI_S_RRX;
```

```
1840                        len += snprintf(buf + len, buflen - len, "%s",
1841                            arm_dpi_shifts[sc]);
1842                        if (len >= buflen)
1843                                return (-1);
1844                        if (sc != DPI_S_RRX) {
1845                                len += snprintf(buf + len, buflen - len,
1846                                    " #%d", imm);
1847                                if (len >= buflen)
1848                                        return (-1);
1849                        }
1850                }
1851                if (snprintf(buf + len, buflen - len, "]") >= buflen - len)
1852                        return (-1);
1853                return (0);
1854        }

1856        /*
1857         * This is a special case of STM, but it works across chip modes.
1858         */
1859        if ((in & ARM_UNI_SRS_MASK) == ARM_UNI_SRS_TARG) {
1860                imm = (in & ARM_LSM_ADDR_MASK) >> ARM_LSM_ADDR_SHIFT;
1861                if (snprintf(buf, buflen, "SRS%s #%d%s",
1862                    arm_lsm_mode_names[imm],
1863                    in & ARM_UNI_SRS_MODE_MASK,
1864                    (in & ARM_UNI_SRS_WBIT_MASK) != 0 ? "!" : "") >= buflen)
1865                        return (-1);
1866                return (0);
1867        }

1869        /*
1870         * RFE is a return from exception instruction that is similar to the LDM
1871         * and STM, but a bit different.
1872         */
1873        if ((in & ARM_UNI_RFE_MASK) == ARM_UNI_RFE_TARG) {
1874                imm = (in & ARM_LSM_ADDR_MASK) >> ARM_LSM_ADDR_SHIFT;
1875                rn = (in & ARM_LS_RN_MASK) >> ARM_LS_RN_SHIFT;
1876                if (snprintf(buf, buflen, "RFE%s %s%s", arm_lsm_mode_names[imm],
1877                    arm_reg_names[rn],
1878                    (in & ARM_UNI_RFE_WBIT_MASK) != 0 ? "!" : "") >= buflen)
1879                        return (-1);
1880                return (0);
1881        }

1883        if ((in & ARM_UNI_BLX_MASK) == ARM_UNI_BLX_TARG) {
1884                if (snprintf(buf, buflen, "BLX %d",
1885                    in & ARM_UNI_BLX_IMM_MASK) >= buflen)
1886                        return (-1);
1887                return (0);
1888        }

1890        if ((in & ARM_UNI_CODRT_MASK) == ARM_UNI_CODRT_TARG) {
1891                return (arm_dis_coproc_lsdrt(in, buf, buflen));
1892        }

1894        if ((in & ARM_UNI_CORT_MASK) == ARM_UNI_CORT_TARG) {
1895                return (arm_dis_coproc_rt(in, buf, buflen));
1896        }

1898        if ((in & ARM_UNI_CODP_MASK) == ARM_UNI_CORT_TARG) {
1899                return (arm_dis_coproc_dp(in, buf, buflen));
1900        }

1902        /*
1903         * An undefined or illegal instruction
1904         */
```



```
1905        return (-1);
1906 }

1908 /*
1909  * Disassemble B and BL instructions. The instruction is given a 24-bit two's
1910  * complement value as an offset address. This value gets sign extended to 30
1911  * bits and then shifted over two bits. This is then added to the PC + 8. So,
1912  * instead of dispalying an absolute address, we're going to display the delta
1913  * that the instruction has instead.
1914  */
1915 static int
1916 arm_dis_branch(dis_handle_t *dhp, uint32_t in, char *buf, size_t buflen)
1917 {
1918        uint32_t addr;
1919        arm_cond_code_t cc;
1920        size_t len;

1922        cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1923        addr = in & ARM_BRANCH_IMM_MASK;
1924        if (in & ARM_BRANCH_SIGN_MASK)
1925                addr |= ARM_BRANCH_NEG_SIGN;
1926        else
1927                addr &= ARM_BRANCH_POS_SIGN;
1928        addr <<= 2;
1929        if ((len = snprintf(buf, buflen, "B%s%s %d",
1930            (in & ARM_BRANCH_LBIT_MASK) != 0 ? "L" : "",
1931            arm_cond_names[cc], (int)addr)) >= buflen)
1932                return (-1);

1934        /* Per the ARM manuals, we have to account for the extra 8 bytes here */
1935        if (dhp->dh_lookup(dhp->dh_data, dhp->dh_addr + (int)addr + 8, NULL, 0,
1936            NULL, NULL) == 0) {
1937                len += snprintf(buf + len, buflen - len, "\t<");
1938                if (len >= buflen)
1939                        return (-1);
1940                dhp->dh_lookup(dhp->dh_data, dhp->dh_addr + (int)addr + 8,
1941                    buf + len, buflen - len, NULL, NULL);
1942                strlcat(buf, ">", buflen);
1943        }

1945        return (0);
1946 }

1948 /*
1949  * There are six instructions that are covered here: ADD16, ADDSUBX, SUBADDX,
1950  * SUB16, ADD8, and SUB8. They can hae the following variations: S, Q, SH, U,
1951  * UQ, and UH. It has two differnt sets of bits to determine the opcode: 22-20
1952  * and then 7-5.
1953  *
1954  * These instructions have the general form of:
1955  *
1956  * 31 - 28|27-25|24|23|22-20|19-16|15-12|11 - 8|7-5|4|3-0
1957  * [ cond |0 1 1| 0| 0| opP |Rn    |Rd    |SBO    |opI|1|Rm ]
1958  *
1959  * Here we use opP to refer to the prefix of the instruction, eg. S, Q, etc.
1960  * Where as opI refers to which instruction it is, eg. ADD16, ADD8, etc. We use
1961  * string tables for both of these in arm_padd_p_names and arm_padd_i_names. If
1962  * there is an empty entry that means that the instruction in question doesn't
1963  * exist.
1964  */
1965 static int
1966 arm_dis_padd(uint32_t in, char *buf, size_t buflen)
1967 {
1968        arm_reg_t rn, rd, rm;
1969        arm_cond_code_t cc;
1970        uint8_t opp, opi;
```

```
1971         const char *pstr, *istr;

1973         opp = (in & ARM_MEDIA_OP1_MASK) >> ARM_MEDIA_OP1_SHIFT;
1974         opi = (in & ARM_MEDIA_OP2_MASK) >> ARM_MEDIA_OP2_SHIFT;

1976         pstr = arm_padd_p_names[opp];
1977         istr = arm_padd_i_names[opi];

1979         if (pstr == NULL || istr == NULL)
1980                 return (-1);

1982         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
1983         rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
1984         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
1985         rm = in & ARM_MEDIA_RM_MASK;

1987         if (snprintf(buf, buflen, "%s%%s %s, %s, %s", pstr, istr,
1988             arm_cond_names[cc], arm_reg_names[rd], arm_reg_names[rn],
1989             arm_reg_names[rm]) >= buflen)
1990                 return (-1);
1991         return (0);
1992 }

1994 /*
1995  * Disassemble the extend instructions from ARMv6. There are six instructions:
1996  *
1997  * XTAB16, XTAB, XTAH, XTB16, XTB, XTFH. These can exist with one of the
1998  * following prefixes: S, U. The opcode exists in bits 22-20. We have the
1999  * following rules from there:
2000  *
2001  * If bit 22 is one then we are using the U prefix, otherwise the S prefix. Then
2002  * we have the following opcode maps in the lower two bits:
2003  * XTAB16       00 iff Rn != 0xf
2004  * XTAB         10 iff Rn != 0xf
2005  * XTAH         11 iff Rn != 0xf
2006  * XTB16        00 iff Rn = 0xf
2007  * XTB          10 iff Rn = 0xf
2008  * XTH          11 iff Rn = 0xf
2009  */
2010 static int
2011 arm_dis_extend(uint32_t in, char *buf, size_t buflen)
2012 {
2013         uint8_t op, rot;
2014         int sbit;
2015         arm_cond_code_t cc;
2016         arm_reg_t rn, rm, rd;
2017         const char *opn;
2018         size_t len;


2021         rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2022         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2023         rm = in & ARM_MEDIA_RM_MASK;
2024         op = (in & ARM_MEDIA_SZE_OP_MASK) >> ARM_MEDIA_SZE_OP_SHIFT;
2025         rot = (in & ARM_MEDIA_SZE_ROT_MASK) >> ARM_MEDIA_SZE_ROT_SHIFT;
2026         sbit = in & ARM_MEDIA_SZE_S_MASK;
2027         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;

2029         switch (op) {
2030         case 0x0:
2031                 opn = rn == ARM_REG_R15 ? "XTAB16" : "XTB16";
2032                 break;
2033         case 0x2:
2034                 opn = rn == ARM_REG_R15 ? "XTAB" : "XTB";
2035                 break;
2036         case 0x3:
```



```
2037                 opn = rn == ARM_REG_R15 ? "XTAH" : "XTH";
2038                 break;
2039         default:
2040                 return (-1);
2041                 break;
2042         }

2044         if (rn == ARM_REG_R15) {
2045                 len = snprintf(buf, buflen, "%s%s%s %s, %s",
2046                     sbit != 0 ? "U" : "S",
2047                     opn, arm_cond_names[cc], arm_reg_names[rd],
2048                     arm_reg_names[rn]);
2049         } else {
2050                 len = snprintf(buf, buflen, "%s%s%s %s, %s, %s",
2051                     sbit != 0 ? "U" : "S",
2052                     opn, arm_cond_names[cc], arm_reg_names[rd],
2053                     arm_reg_names[rn], arm_reg_names[rm]);
2054         }

2056         if (len >= buflen)
2057                 return (-1);

2059         if (snprintf(buf + len, buflen - len, "%s",
2060             arm_extend_rot_names[rot]) >= buflen - len)
2061                 return (-1);
2062         return (0);
2063 }

2065 /*
2066  * The media instructions and extensions can be divided into different groups of
2067  * instructions. We first use bits 23 and 24 to figure out where to send it. We
2068  * call this group of bits the l1 mask.
2069  */
2070 static int
2071 arm_dis_media(uint32_t in, char *buf, size_t buflen)
2072 {
2073         uint8_t l1, op1, op2;
2074         arm_cond_code_t cc;
2075         arm_reg_t rd, rn, rs, rm;
2076         int xbit;
2077         size_t len;

2079         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
2080         l1 = (in & ARM_MEDIA_L1_MASK) >> ARM_MEDIA_L1_SHIFT;
2081         switch (l1) {
2082         case 0x0:
2083                 return (arm_dis_padd(in, buf, buflen));
2084                 break;
2085         case 0x1:
2086                 if ((in & ARM_MEDIA_HPACK_MASK) == ARM_MEDIA_HPACK_TARG) {
2087                         rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2088                         rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2089                         rm = in & ARM_MEDIA_RM_MASK;
2090                         op1 = (in & ARM_MEDIA_HPACK_SHIFT_MASK) >>
2091                             ARM_MEDIA_HPACK_SHIFT_IMM;
2092                         len = snprintf(buf, buflen, "%s%s %s, %s, %s",
2093                             (in & ARM_MEDIA_HPACK_OP_MASK) != 0 ?
2094                             "PKHTB" : "PKHBT", arm_cond_names[cc],
2095                             arm_reg_names[rd], arm_reg_names[rn],
2096                             arm_reg_names[rd]);
2097                         if (len >= buflen)
2098                                 return (-1);

2100                         if (op1 != 0) {
2101                                 if (in & ARM_MEDIA_HPACK_OP_MASK)
2102                                         len += snprintf(buf + len, buflen - len,
```

```
2103                                        ", ASR %d", op1);
2104                        else
2105                                len += snprintf(buf + len, buflen - len,
2106                                    ", LSL %d", op1);
2107                }
2108                return (len >= buflen ? -1 : 0);
2109        }

2111        if ((in & ARM_MEDIA_WSAT_MASK) == ARM_MEDIA_WSAT_TARG) {
2112                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2113                rm = in & ARM_MEDIA_RM_MASK;
2114                op1 = (in & ARM_MEDIA_SAT_IMM_MASK) >>
2115                    ARM_MEDIA_SAT_IMM_SHIFT;
2116                op2 = (in & ARM_MEDIA_SAT_SHI_MASK) >>
2117                    ARM_MEDIA_SAT_SHI_SHIFT;
2118                len = snprintf(buf, buflen, "%s%s %s, #%d, %s",
2119                    (in & ARM_MEDIA_SAT_U_MASK) != 0 ? "USAT" : "SSAT",
2120                    arm_cond_names[cc], arm_reg_names[rd], op1,
2121                    arm_reg_names[rm]);

2123                if (len >= buflen)
2124                        return (-1);

2126                /*
2127                 * The shift is optional in the assembler and encoded as
2128                 * LSL 0. However if we get ASR 0, that means ASR #32.
2129                 * An ARM_MEDIA_SAT_STYPE_MASK of 0 is LSL, 1 is ASR.
2130                 */
2131                if (op2 != 0 || (in & ARM_MEDIA_SAT_STYPE_MASK) == 1) {
2132                        if (op2 == 0)
2133                                op2 = 32;
2134                        if (snprintf(buf + len, buflen - len,
2135                            ", %s #%d",
2136                            (in & ARM_MEDIA_SAT_STYPE_MASK) != 0 ?
2137                            "ASR" : "LSL", op2) >= buflen - len)
2138                                return (-1);
2139                }
2140                return (0);
2141        }

2143        if ((in & ARM_MEDIA_PHSAT_MASK) == ARM_MEDIA_PHSAT_TARG) {
2144                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2145                rm = in & ARM_MEDIA_RM_MASK;
2146                op1 = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2147                if (snprintf(buf, buflen, "%s%s %s, #%d, %s",
2148                    (in & ARM_MEDIA_SAT_U_MASK) != 0 ?
2149                    "USAT16" : "SSAT16",
2150                    arm_cond_names[cc], arm_reg_names[rd], op1,
2151                    arm_reg_names[rm]) >= buflen)
2152                        return (-1);
2153                return (0);
2154        }

2156        if ((in & ARM_MEDIA_REV_MASK) == ARM_MEDIA_REV_TARG) {
2157                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2158                rm = in & ARM_MEDIA_RM_MASK;
2159                if (snprintf(buf, buflen, "REV%s %s, %s",
2160                    arm_cond_names[cc], arm_reg_names[rd],
2161                    arm_reg_names[rd]) >= buflen)
2162                        return (-1);
2163                return (0);
2164        }

2166        if ((in & ARM_MEDIA_BRPH_MASK) == ARM_MEDIA_BRPH_TARG) {
2167                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2168                rm = in & ARM_MEDIA_RM_MASK;
```



```
2169                if (snprintf(buf, buflen, "REV16%s %s, %s",
2170                    arm_cond_names[cc], arm_reg_names[rd],
2171                    arm_reg_names[rd]) >= buflen)
2172                        return (-1);
2173                return (0);
2174        }

2176        if ((in & ARM_MEDIA_BRSH_MASK) == ARM_MEDIA_BRSH_TARG) {
2177                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2178                rm = in & ARM_MEDIA_RM_MASK;
2179                if (snprintf(buf, buflen, "REVSH%s %s, %s",
2180                    arm_cond_names[cc], arm_reg_names[rd],
2181                    arm_reg_names[rd]) >= buflen)
2182                        return (-1);
2183                return (0);
2184        }

2186        if ((in & ARM_MEDIA_SEL_MASK) == ARM_MEDIA_SEL_TARG) {
2187                rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2188                rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2189                rm = in & ARM_MEDIA_RM_MASK;
2190                if (snprintf(buf, buflen, "SEL%s %s, %s, %s",
2191                    arm_cond_names[cc], arm_reg_names[rd],
2192                    arm_reg_names[rn], arm_reg_names[rm]) >= buflen)
2193                        return (-1);
2194                return (0);
2195        }

2197        if ((in & ARM_MEDIA_SZE_MASK) == ARM_MEDIA_SZE_TARG)
2198                return (arm_dis_extend(in, buf, buflen));
2199        /* Unknown instruction */
2200        return (-1);
2201        break;
2202    case 0x2:
2203        /*
2204         * This consists of the following multiply instructions:
2205         * SMLAD, SMLSD, SMLALD, SMUAD, and SMUSD.
2206         *
2207         * SMLAD and SMUAD encoding are the same, switch on Rn == R15
2208         * 22-20 are 000 7-6 are 00
2209         * SMLSD and SMUSD encoding are the same, switch on Rn == R15
2210         * 22-20 are 000 7-6 are 01
2211         * SMLALD: 22-20 are 100 7-6 are 00
2212         */
2213        rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2214        rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2215        rs = (in & ARM_MEDIA_RS_MASK) >> ARM_MEDIA_RS_SHIFT;
2216        rm = in & ARM_MEDIA_RM_MASK;
2217        op1 = (in & ARM_MEDIA_OP1_MASK) >> ARM_MEDIA_OP1_SHIFT;
2218        op2 = (in & ARM_MEDIA_OP2_MASK) >> ARM_MEDIA_OP2_SHIFT;
2219        xbit = in & ARM_MEDIA_MULT_X_MASK;

2221        if (op1 == 0x0) {
2222                if (op2 != 0x0 && op2 != 0x1)
2223                        return (-1);
2224                if (rn == ARM_REG_R15) {
2225                        len = snprintf(buf, buflen, "%s%s%s %s, %s, %s",
2226                            op2 != 0 ? "SMUSD" : "SMUAD",
2227                            xbit != 0 ? "X" : "X",
2228                            arm_cond_names[cc], arm_reg_names[rd],
2229                            arm_reg_names[rm], arm_reg_names[rs]);
2230                } else {
2231                        len = snprintf(buf, buflen,
2232                            "%s%s%s %s, %s, %s, %s",
2233                            op2 != 0 ? "SMLSD" : "SMLAD",
2234                            xbit != 0 ? "X" : "",
```

```
2235                                    arm_cond_names[cc], arm_reg_names[rd],
2236                                    arm_reg_names[rm], arm_reg_names[rs],
2237                                    arm_reg_names[rn]);
2239                    }
2240            } else if (op1 == 0x8) {
2241                    if (op2 != 0x0)
2242                            return (-1);
2243                    len = snprintf(buf, buflen, "SMLALD%s%s %s, %s, %s, %s",
2244                        xbit != 0 ? "X" : "",
2245                        arm_cond_names[cc], arm_reg_names[rn],
2246                        arm_reg_names[rd], arm_reg_names[rm],
2247                        arm_reg_names[rs]);
2248            } else
2249                    return (-1);
2251            return (len >= buflen ? -1 : 0);
2252            break;
2253    case 0x3:
2254            /*
2255             * Here we handle USAD8 and USADA8. The main difference is the
2256             * presence of RN. USAD8 is defined as having a value of rn that
2257             * is not r15. If it is r15, then instead it is USADA8.
2258             */
2259            if ((in & ARM_MEDIA_OP1_MASK) != 0)
2260                    return (-1);
2261            if ((in & ARM_MEDIA_OP2_MASK) != 0)
2262                    return (-1);
2264            cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
2265            rn = (in & ARM_MEDIA_RN_MASK) >> ARM_MEDIA_RN_SHIFT;
2266            rd = (in & ARM_MEDIA_RD_MASK) >> ARM_MEDIA_RD_SHIFT;
2267            rs = (in & ARM_MEDIA_RS_MASK) >> ARM_MEDIA_RS_SHIFT;
2268            rm = in & ARM_MEDIA_RM_MASK;
2270            if (rn != ARM_REG_R15)
2271                    len = snprintf(buf, buflen, "USADA8%s %s, %s, %s, %s",
2272                        arm_cond_names[cc], arm_reg_names[rd],
2273                        arm_reg_names[rm], arm_reg_names[rs],
2274                        arm_reg_names[rn]);
2275            else
2276                    len = snprintf(buf, buflen, "USAD8%s %s, %s, %s",
2277                        arm_cond_names[cc], arm_reg_names[rd],
2278                        arm_reg_names[rm], arm_reg_names[rs]);
2279            return (len >= buflen ? -1 : 0);
2280            break;
2281    default:
2282            return (-1);
2283    }
2284 }
2286 /*
2287  * Each instruction in the ARM instruction set is a uint32_t and in our case is
2288  * LE. The upper four bits determine the condition code. If the conditoin code
2289  * is undefined then we know to immediately jump there. Otherwise we go use the
2290  * next three bits to determine where we should go next and how to further
2291  * process the instruction in question. The ARM instruction manual doesn't
2292  * define this field so we're going to call it the L1_DEC or level 1 decoding
2293  * from which it will have to be further subdivided into the specific
2294  * instruction groupings that we care about.
2295  */
2296 static int
2297 arm_dis(dis_handle_t *dhp, uint32_t in, char *buf, size_t buflen)
2298 {
2299         uint8_t l1;
2300         arm_cond_code_t cc;
```



```
2302         cc = (in & ARM_CC_MASK) >> ARM_CC_SHIFT;
2304         if (cc == ARM_COND_NACC)
2305                 return (arm_dis_uncond_insn(in, buf, buflen));
2307         l1 = (in & ARM_L1_DEC_MASK) >> ARM_L1_DEC_SHIFT;
2309         switch (l1) {
2310         case 0x0:
2311                 /*
2312                  * The l0 group is a bit complicated. We have several different
2313                  * groups of instructions to consider. The first question is
2314                  * whether bit 4 is zero or not. If it is, then we have a data
2315                  * processing immediate shift unless the opcode and + S bits
2316                  * (24-20) is of the form 0b10xx0.
2317                  *
2318                  * When bit 4 is 1, we have to then also look at bit 7. If bit
2319                  * 7 is one then we know that this is the class of multiplies /
2320                  * extra load/stores. If bit 7 is zero then we have the same
2321                  * opcode games as we did above.
2322                  */
2323                 if (in & ARM_L1_0_B4_MASK) {
2324                         if (in & ARM_L1_0_B7_MASK) {
2325                                 /*
2326                                  * Both the multiplication extensions and the
2327                                  * load and store extensions live in this
2328                                  * region. The load and store extensions can be
2329                                  * identified by having at least one of bits 5
2330                                  * and 6 set. The exceptions to this are the
2331                                  * SWP and SWPB instructions and the exclusive
2332                                  * load and store instructions which, unlike the
2333                                  * multiplication instructions. These have
2334                                  * specific values for the bits in the range of
2335                                  * 20-24.
2336                                  */
2337                                 if ((in & ARM_L1_0_ELS_MASK) != 0)
2338                                         /* Extra loads/stores */
2339                                         return (arm_dis_els(in, buf, buflen));
2340                                 if ((in & ARM_ELS_SWAP_MASK) == ARM_ELS_IS_SWAP)
2341                                         return (arm_dis_swap(in, buf, buflen));
2342                                 if ((in & ARM_ELS_EXCL_MASK) ==
2343                                     ARM_ELS_EXCL_MASK)
2344                                         return (arm_dis_lsexcl(in, buf,
2345                                             buflen));
2346                                 /* Multiplication instruction extension A3-3. */
2347                                 return (arm_dis_extmul(in, buf, buflen));
2348                         }
2349                         if ((in & ARM_L1_0_OPMASK) == ARM_L1_0_SPECOP &&
2350                             !(in & ARM_L1_0_SMASK)) {
2351                                 /* Misc. Instructions A3-4 */
2352                                 return (arm_dis_cdsp_ext(in, buf, buflen));
2353                         } else {
2354                                 /* data processing register shift */
2355                                 return (arm_dis_dpi(in, cc, buf, buflen));
2356                         }
2357                 } else {
2358                         if ((in & ARM_L1_0_OPMASK) == ARM_L1_0_SPECOP &&
2359                             !(in & ARM_L1_0_SMASK))
2360                                 /* Misc. Instructions A3-4 */
2361                                 return (arm_dis_cdsp_ext(in, buf, buflen));
2362                         else {
2363                                 /* Data processing immediate shift */
2364                                 return (arm_dis_dpi(in, cc, buf, buflen));
2365                         }
2366                 }
```

```
2367                break;
2368        case 0x1:
2369                /*
2370                 * In l1 group 0b001 there are a few ways to tell things apart.
2371                 * We are directed to first look at bits 20-24. Data processing
2372                 * immediate has a 4 bit opcode 24-21 followed by an S bit. We
2373                 * know it is not a data processing immediate if we have
2374                 * something of the form 0b10xx0.
2375                 */
2376                if ((in & ARM_L1_1_OPMASK) == ARM_L1_1_SPECOP &&
2377                    !(in & ARM_L1_1_SMASK)) {
2378                        if (in & ARM_L1_1_UNDEF_MASK) {
2379                                /* Undefined instructions */
2380                                return (-1);
2381                        } else {
2382                                /* Move immediate to status register */
2383                                return (arm_dis_status_regs(in, buf, buflen));
2384                        }
2385                } else {
2386                        /* Data processing immedaite */
2387                        return (arm_dis_dpi(in, cc, buf, buflen));
2388                }
2389                break;
2390        case 0x2:
2391                /* Load/store Immediate offset */
2392                return (arm_dis_ldstr(in, buf, buflen));
2393                break;
2394        case 0x3:
2395                /*
2396                 * Like other sets we use the 4th bit to make an intial
2397                 * determination. If it is zero then this is a load/store
2398                 * register offset class instruction. Following that we have a
2399                 * specical mask of 0x01f000f0 to determine whether this is an
2400                 * architecturally undefined instruction type or not.
2401                 *
2402                 * The architecturally undefined are parts of the current name
2403                 * space that just aren't used, but could be used at some point
2404                 * in the future. For now though, it's an invalid op code.
2405                 */
2406                if (in & ARM_L1_3_B4_MASK) {
2407                        if ((in & ARM_L1_3_ARCHUN_MASK) ==
2408                            ARM_L1_3_ARCHUN_MASK) {
2409                                /* Architecturally undefined */
2410                                return (-1);
2411                        } else {
2412                                /* Media instructions */
2413                                return (arm_dis_media(in, buf, buflen));
2414                        }
2415                } else {
2416                        /* Load/store register offset */
2417                        return (arm_dis_ldstr(in, buf, buflen));
2418                }
2419                break;
2420        case 0x4:
2421                /* Load/store multiple */
2422                return (arm_dis_ldstr_multi(in, buf, buflen));
2423                break;
2424        case 0x5:
2425                /* Branch and Branch with link */
2426                return (arm_dis_branch(dhp, in, buf, buflen));
2427                break;
2428        case 0x6:
2429                /* coprocessor load/store && double register transfers */
2430                return (arm_dis_coproc_lsdrt(in, buf, buflen));
2431                break;
2432        case 0x7:
```



```
2433                /*
2434                 * In l1 group 0b111 you can determine the three groups using
2435                 * the following logic. If the next bit after the l1 group (bit
2436                 * 24) is one than you know that it is a software interrupt.
2437                 * Otherwise it is one of the coprocessor instructions.
2438                 * Furthermore you can tell apart the data processing from the
2439                 * register transfers based on bit 4. If it is zero then it is
2440                 * a data processing instruction, otherwise it is a register
2441                 * transfer.
2442                 */
2443                if (in & ARM_L1_7_SWINTMASK) {
2444                        /*
2445                         * The software interrupt is pretty straightforward. The
2446                         * lower 24 bits are the interrupt number. It's also
2447                         * valid for it to run with a condition code.
2448                         */
2449                        if (snprintf(buf, buflen, "SWI%s %d",
2450                            arm_cond_names[cc],
2451                            in & ARM_SWI_IMM_MASK) >= buflen)
2452                                return (-1);
2453                        return (0);
2454                } else if (in & ARM_L1_7_COPROCMASK) {
2455                        /* coprocessor register transfers */
2456                        return (arm_dis_coproc_rt(in, buf, buflen));
2457                } else {
2458                        /* coprocessor data processing */
2459                        return (arm_dis_coproc_dp(in, buf, buflen));
2460                }
2461                break;
2462        }

2464        return (-1);
2465 }

2467 static int
2468 dis_arm_supports_flags(int flags)
2469 {
2470        int archflags = flags & DIS_ARCH_MASK;

2472        return (archflags == DIS_ARM);
2473 }

2475 /*ARGSUSED*/
2476 static int
2477 dis_arm_handle_attach(dis_handle_t *dhp)
2478 {
2479        return (0);
2480 }

2482 /*ARGSUSED*/
2483 static void
2484 dis_arm_handle_detach(dis_handle_t *dhp)
2485 {
2486 }

2488 static int
2489 dis_arm_disassemble(dis_handle_t *dhp, uint64_t addr, char *buf, size_t buflen)
2490 {
2491        uint32_t in;

2493        buf[0] = '\0';
2494        dhp->dh_addr = addr;
2495        if (dhp->dh_read(dhp->dh_data, addr, &in, sizeof (in)) !=
2496            sizeof (in))
2497                return (-1);
```

```
2499         /* Translate in case we're on sparc? */
2500         in = LE_32(in);

2502         return (arm_dis(dhp, in, buf, buflen));
2503 }

2505 /*
2506  * This is simple in a non Thumb world. If and when we do enter a world where we
2507  * support thumb instructions, then this becomes far less than simple.
2508  */
2509 /*ARGSUSED*/
2510 static uint64_t
2511 dis_arm_previnstr(dis_handle_t *dhp, uint64_t pc, int n)
2512 {
2513         if (n <= 0)
2514                 return (pc);

2516         return (pc - n*4);
2517 }

2519 /*
2520  * If and when we support thumb, then this value should probably become two.
2521  * However, it varies based on whether or not a given instruction is in thumb
2522  * mode.
2523  */
2524 /*ARGSUSED*/
2525 static int
2526 dis_arm_min_instrlen(dis_handle_t *dhp)
2527 {
2528         return (4);
2529 }

2531 /*
2532  * Regardless of thumb, this value does not change.
2533  */
2534 /*ARGSUSED*/
2535 static int
2536 dis_arm_max_instrlen(dis_handle_t *dhp)
2537 {
2538         return (4);
2539 }

2541 /* ARGSUSED */
2542 static int
2543 dis_arm_instrlen(dis_handle_t *dhp, uint64_t pc)
2544 {
2545         return (4);
2546 }

2548 dis_arch_t dis_arch_arm = {
2549         dis_arm_supports_flags,
2550         dis_arm_handle_attach,
2551         dis_arm_handle_detach,
2552         dis_arm_disassemble,
2553         dis_arm_previnstr,
2554         dis_arm_min_instrlen,
2555         dis_arm_max_instrlen,
2556         dis_arm_instrlen
2557 };
```

```
**********************************************************
    3506 Wed Jan 14 19:07:05 2015
new/usr/src/uts/armv6/ml/glocore.s
_locore_start needs to set up an 8-byte aligned stack
Sadly, during removal of special early-boot stacks
(027eb01d0ff8f66be55208b58ecd7cb2b7b27714) the stack passed into mlsetup
lost its 8-byte aligned-ness.
**********************************************************
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2013 (c) Joyent, Inc. All rights reserved.
 */

#include <sys/asm_linkage.h>
#include <sys/machparam.h>
#include <sys/cpu_asm.h>

#include "assym.h"

#if defined(__lint)

#endif

/*
 * Each of the different machines has its own locore.s to take care of getting
 * us into fakebop for the first time. After that, they all return here to a
 * generic locore to take us into mlsetup and then to main forever more.
 */

        /*
         * External globals
         */
        .globl  _locore_start
        .globl  mlsetup
        .globl  sysp
        .globl  bootops
        .globl  bootopsp
        .globl  t0

        .data
        .comm   t0stack, DEFAULTSTKSZ, 32
        .comm   t0, 4094, 32

#if defined(__lint)

/* ARGSUSED */
void
_locore_start(struct boot_syscalls *sysp, struct bootops *bop)
{}

#else   /* __lint */

        /*
         * We got here from _kobj_init() via exitto().  We have a few different
         * tasks that we need to take care of before we hop into mlsetup and
         * then main. We're never going back so we shouldn't feel compelled to
```



```
         * preserve any registers.
         *
         *  o Enable unaligned access
         *  o Enable our I/D-caches
         *  o Save the boot syscalls and bootops for later
         *  o Set up our stack to be the real stack of t0stack.
         *  o Save t0 as curthread
         *  o Set up a struct REGS for mlsetup
         *  o Make sure that we're 8 byte aligned for the call
         */

        ENTRY(_locore_start)


        /*
         * We've been running in t0stack anyway, up to this point, but
         * _locore_start represents what is in effect a fresh start in the
         * real kernel -- We'll never return back through here.
         *
         * So reclaim those few bytes
         */
        ldr     sp, =t0stack
        ldr     r4, =(DEFAULTSTKSZ - REGSIZE)
        add     sp, r4
        bic     sp, sp, #0xff

        /*
         * Save flags and arguments for potential debugging
         */
        str     r0, [sp, #REGOFF_R0]
        str     r1, [sp, #REGOFF_R1]
        str     r2, [sp, #REGOFF_R2]
        str     r3, [sp, #REGOFF_R3]
        mrs     r4, CPSR
        str     r4, [sp, #REGOFF_CPSR]

        /*
         * Save back the bootops and boot_syscalls.
         */
        ldr     r2, =sysp
        str     r0, [r2]
        ldr     r2, =bootops
        str     r1, [r2]
        ldr     r2, =bootopsp
        ldr     r2, [r2]
        str     r1, [r2]

        /*
         * Set up our curthread pointer
         */
        ldr     r0, =t0
        mcr     p15, 0, r0, c13, c0, 4

        /*
         * Go ahead now and enable unaligned access, the L1 I/D caches.
         *
         * Bit 2 is for the D cache
         * Bit 12 is for the I cache
         * Bit 22 is for unaligned access
         */
        mrc     p15, 0, r0, c1, c0, 0
        orr     r0, #0x02
        orr     r0, #0x1000
        orr     r0, #0x400000
        mcr     p15, 0, r0, c1, c0, 0
```

```
125         /*
126          * mlsetup() takes the struct regs as an argument. main doesn't take
127          * any and should never return. Currently, we have an 8-byte aligned
128          * stack.  We want to push a zero frame pointer to terminate any
129          * stack walking, but that would cause us to end up with only a
130          * 4-byte aligned stack.  So, to keep things nice and correct, we
131          * push a zero value twice - it's similar to a typical function
132          * entry:
133          *      push { r9, lr }
126          * mlsetup() takes the struct regs as an argument. main doesn't take any
127          * and should never return. After the push below, we should have a
128          * 8-byte aligned stack pointer. This is why we subtracted four earlier
129          * on if we were 8-byte aligned.
134          */
135         mov     r9,#0
136         push    { r9 }          /* link register */
137         push    { r9 }          /* frame pointer */
132         push    { r9 }
138         mov     r0, sp
139         bl      mlsetup
140         bl      main
141         /* NOTREACHED */
142         ldr     r0,=__return_from_main
143         ldr     r0,[r0]
144         bl      panic
145         SET_SIZE(_locore_start)
```

*unchanged_portion_omitted_*

```
**********************************************************
   24341 Wed Jan 14 19:07:05 2015
new/usr/src/uts/armv6/os/fakebop.c
fakebop: make a note of multiple ATAG_MEM
fakebop: dump ATAG_ILLUMOS_{STATUS,MAPPING}
fakebop: make the atag dumping code a bit more readable
**********************************************************
______unchanged_portion_omitted_
```

```
207 #define DUMP_ATAG_VAL(name, val)                                        \
208         do {                                                            \
209                 DBG_MSG("\t" name ":");                                 \
210                 bcons_puts("\t");                                       \
211                 DBG_MSG(fakebop_hack_ultostr((val),                     \
212                     &buffer[BUFFERSIZE-1]));                            \
213         } while (0)

215 #endif /* ! codereview */
216 static void
217 fakebop_dump_tags(void *tagstart)
218 {
219         atag_header_t *h = tagstart;
220         atag_core_t *acp;
221         atag_mem_t *amp;
222         atag_cmdline_t *alp;
223         atag_initrd_t *aip;
224         atag_illumos_status_t *aisp;
225         atag_illumos_mapping_t *aimp;
226 #endif /* ! codereview */
227         const char *tname;
228         int i;
229         char *c;

231         DBG_MSG("starting point:");
232         DBG_MSG(fakebop_hack_ultostr((uintptr_t)h, &buffer[BUFFERSIZE-1]));
233         DBG_MSG("first atag size:");
234         DBG_MSG(fakebop_hack_ultostr(h->ah_size, &buffer[BUFFERSIZE-1]));
235         DBG_MSG("first atag tag:");
236         DBG_MSG(fakebop_hack_ultostr(h->ah_tag, &buffer[BUFFERSIZE-1]));
237         while (h != NULL) {
238                 switch (h->ah_tag) {
239                 case ATAG_CORE:
240                         tname = "ATAG_CORE";
241                         break;
242                 case ATAG_MEM:
243                         tname = "ATAG_MEM";
244                         break;
245                 case ATAG_VIDEOTEXT:
246                         tname = "ATAG_VIDEOTEXT";
247                         break;
248                 case ATAG_RAMDISK:
249                         tname = "ATAG_RAMDISK";
250                         break;
251                 case ATAG_INITRD2:
252                         tname = "ATAG_INITRD2";
253                         break;
254                 case ATAG_SERIAL:
255                         tname = "ATAG_SERIAL";
256                         break;
257                 case ATAG_REVISION:
258                         tname = "ATAG_REVISION";
259                         break;
260                 case ATAG_VIDEOLFB:
261                         tname = "ATAG_VIDEOLFB";
262                         break;
263                 case ATAG_CMDLINE:
```



```
264                         tname = "ATAG_CMDLINE";
265                         break;
266                 case ATAG_ILLUMOS_STATUS:
267                         tname = "ATAG_ILLUMOS_STATUS";
268                         break;
269                 case ATAG_ILLUMOS_MAPPING:
270                         tname = "ATAG_ILLUMOS_MAPPING";
271                         break;
272 #endif /* ! codereview */
273                 default:
274                         tname = fakebop_hack_ultostr(h->ah_tag,
275                             &buffer[BUFFERSIZE-1]);
276                         break;
277                 }
278                 DBG_MSG("tag:");
279                 DBG_MSG(tname);
280                 DBG_MSG("size:");
281                 DBG_MSG(fakebop_hack_ultostr(h->ah_size,
282                     &buffer[BUFFERSIZE-1]));
283                 /* Extended information */
284                 switch (h->ah_tag) {
285                 case ATAG_CORE:
286                         if (h->ah_size == 2) {
287                                 DBG_MSG("ATAG_CORE has no extra information");
288                         } else {
289                                 acp = (atag_core_t *)h;
290                                 DUMP_ATAG_VAL("flags", acp->ac_flags);
291                                 DUMP_ATAG_VAL("pagesize", acp->ac_pagesize);
292                                 DUMP_ATAG_VAL("rootdev", acp->ac_rootdev);
207                                 DBG_MSG("\tflags:");
208                                 bcons_puts("\t");
209                                 DBG_MSG(fakebop_hack_ultostr(acp->ac_flags,
210                                     &buffer[BUFFERSIZE-1]));
211                                 DBG_MSG("\tpage:");
212                                 bcons_puts("\t");
213                                 DBG_MSG(fakebop_hack_ultostr(acp->ac_pagesize,
214                                     &buffer[BUFFERSIZE-1]));
215                                 DBG_MSG("\troot:");
216                                 bcons_puts("\t");
217                                 DBG_MSG(fakebop_hack_ultostr(acp->ac_rootdev,
218                                     &buffer[BUFFERSIZE-1]));
293                         }
294                         break;
295                 case ATAG_MEM:
296                         amp = (atag_mem_t *)h;
297                         DUMP_ATAG_VAL("size", amp->am_size);
298                         DUMP_ATAG_VAL("start", amp->am_start);
223                         DBG_MSG("\tsize:");
224                         bcons_puts("\t");
225                         DBG_MSG(fakebop_hack_ultostr(amp->am_size,
226                             &buffer[BUFFERSIZE-1]));
227                         DBG_MSG("\tstart:");
228                         bcons_puts("\t");
229                         DBG_MSG(fakebop_hack_ultostr(amp->am_start,
230                             &buffer[BUFFERSIZE-1]));
299                         break;
300                 case ATAG_INITRD2:
301                         aip = (atag_initrd_t *)h;
302                         DUMP_ATAG_VAL("size", aip->ai_size);
303                         DUMP_ATAG_VAL("start", aip->ai_start);
234                         DBG_MSG("\tsize:");
235                         bcons_puts("\t");
236                         DBG_MSG(fakebop_hack_ultostr(aip->ai_size,
237                             &buffer[BUFFERSIZE-1]));
238                         DBG_MSG("\tstart:");
239                         bcons_puts("\t");
```

```
240                         DBG_MSG(fakebop_hack_ultostr(aip->ai_start,
241                             &buffer[BUFFERSIZE-1]));
304                         break;
305                 case ATAG_CMDLINE:
306                         alp = (atag_cmdline_t *)h;
307                         DBG_MSG("\tcmdline:");
308                         /*
309                          * We have no intelligent thing to wrap our tty at 80
310                          * chars so we just do this a bit more manually for now.
311                          */
312                         i = 0;
313                         c = alp->al_cmdline;
314                         while (*c != '\0') {
315                                 bcons_putchar(*c++);
316                                 if (++i == 72) {
317                                         bcons_puts("\n");
318                                         i = 0;
319                                 }
320                         }
321                         bcons_puts("\n");
322                         break;
323                 case ATAG_ILLUMOS_STATUS:
324                         aisp = (atag_illumos_status_t *)h;
325                         DUMP_ATAG_VAL("version", aisp->ais_version);
326                         DUMP_ATAG_VAL("ptbase", aisp->ais_ptbase);
327                         DUMP_ATAG_VAL("freemem", aisp->ais_freemem);
328                         DUMP_ATAG_VAL("freeused", aisp->ais_freeused);
329                         DUMP_ATAG_VAL("archive", aisp->ais_archive);
330                         DUMP_ATAG_VAL("archivelen", aisp->ais_archivelen);
331                         DUMP_ATAG_VAL("pt_arena", aisp->ais_pt_arena);
332                         DUMP_ATAG_VAL("pt_arena_max", aisp->ais_pt_arena_max);
333                         DUMP_ATAG_VAL("stext", aisp->ais_stext);
334                         DUMP_ATAG_VAL("etext", aisp->ais_etext);
335                         DUMP_ATAG_VAL("sdata", aisp->ais_sdata);
336                         DUMP_ATAG_VAL("edata", aisp->ais_edata);
337                         break;
338                 case ATAG_ILLUMOS_MAPPING:
339                         aimp = (atag_illumos_mapping_t *)h;
340                         DUMP_ATAG_VAL("paddr", aimp->aim_paddr);
341                         DUMP_ATAG_VAL("plen", aimp->aim_plen);
342                         DUMP_ATAG_VAL("vaddr", aimp->aim_vaddr);
343                         DUMP_ATAG_VAL("vlen", aimp->aim_vlen);
344                         DUMP_ATAG_VAL("mapflags", aimp->aim_mapflags);
345                         break;
346 #endif /* ! codereview */
347                 default:
348                         break;
349                 }
350                 h = atag_next(h);
351         }
352 }

354 static void
355 fakebop_getatags(void *tagstart)
356 {
357         atag_mem_t *amp;
358         atag_cmdline_t *alp;
359         atag_header_t *ahp = tagstart;
360         atag_illumos_status_t *aisp;
361         bootinfo_t *bp = &bootinfo;

363         bp->bi_flags = 0;
364         while (ahp != NULL) {
365                 switch (ahp->ah_tag) {
366                 case ATAG_MEM:
367                         /*
```



```
368                          * XXX: we may actually get more than one ATAG_MEM
369                          * if the system has discontiguous physical memory
370                          */
371 #endif /* ! codereview */
372                         amp = (atag_mem_t *)ahp;
373                         bp->bi_memsize = amp->am_size;
374                         bp->bi_memstart = amp->am_start;
375                         break;
376                 case ATAG_CMDLINE:
377                         alp = (atag_cmdline_t *)ahp;
378                         bp->bi_cmdline = alp->al_cmdline;
379                         break;
380                 case ATAG_ILLUMOS_STATUS:
381                         aisp = (atag_illumos_status_t *)ahp;
382                         bp->bi_ramdisk = aisp->ais_archive;
383                         bp->bi_ramsize = aisp->ais_archivelen;
384                         bp->bi_flags |= BI_HAS_RAMDISK;
385                         break;
386                 default:
387                         break;
388                 }
389                 ahp = atag_next(ahp);
390         }
391 }

393 /*
394  * We've been asked to allocate at a specific VA. Allocate the next ragne of
395  * physical addresses and go from there.
396  */
397 static caddr_t
398 fakebop_alloc_hint(caddr_t virt, size_t size, int align)
399 {
400         uintptr_t start = P2ROUNDUP(bop_alloc_pnext, align);
401         if (fakebop_alloc_debug != 0)
402                 bop_printf(NULL, "asked to allocate %d bytes at v/p %p/%p\n",
403                     size, virt, start);
404         if (start + size > bop_alloc_plast)
405                 bop_panic("fakebop_alloc_hint: No more physical address -_-\n");

407         armboot_mmu_map(start, (uintptr_t)virt, size, PF_R | PF_W | PF_X);
408         bop_alloc_pnext = start + size;
409         return (virt);
410 }

412 static caddr_t
413 fakebop_alloc(struct bootops *bops, caddr_t virthint, size_t size, int align)
414 {
415         caddr_t start;

417         if (virthint != NULL)
418                 return (fakebop_alloc_hint(virthint, size, align));
419         if (fakebop_alloc_debug != 0)
420                 bop_printf(bops, "asked to allocate %d bytes\n", size);
421         if (bop_alloc_scratch_next == 0)
422                 bop_panic("fakebop_alloc_init not called");

424         if (align == BO_ALIGN_DONTCARE || align == 0)
425                 align = 4;

427         start = (caddr_t)P2ROUNDUP(bop_alloc_scratch_next, align);
428         if ((uintptr_t)start + size > bop_alloc_scratch_last)
429                 bop_panic("fakebop_alloc: ran out of scratch space!\n");
430         if (fakebop_alloc_debug != 0)
431                 bop_printf(bops, "returning address: %p\n", start);
432         bop_alloc_scratch_next = (uintptr_t)start + size;
```

```
434         return (start);
435 }

437 static void
438 fakebop_free(struct bootops *bops, caddr_t virt, size_t size)
439 {
440         bop_panic("Called into fakebop_free");
441 }

443 static int
444 fakebop_getproplen(struct bootops *bops, const char *pname)
445 {
446         bootprop_t *p;

448         if (fakebop_prop_debug)
449                 bop_printf(NULL, "fakebop_getproplen: asked for %s\n", pname);
450         for (p = bprops; p != NULL; p = p->bp_next) {
451                 if (strcmp(pname, p->bp_name) == 0)
452                         return (p->bp_vlen);
453         }
454         if (fakebop_prop_debug != 0)
455                 bop_printf(NULL, "prop %s not found\n", pname);
456         return (-1);
457 }

459 static int
460 fakebop_getprop(struct bootops *bops, const char *pname, void *value)
461 {
462         bootprop_t *p;

464         if (fakebop_prop_debug)
465                 bop_printf(NULL, "fakebop_getprop: asked for %s\n", pname);
466         for (p = bprops; p != NULL; p = p->bp_next) {
467                 if (strcmp(pname, p->bp_name) == 0)
468                         break;
469         }
470         if (p == NULL) {
471                 if (fakebop_prop_debug)
472                         bop_printf(NULL, "fakebop_getprop: ENOPROP %s\n",
473                             pname);
474                 return (-1);
475         }
476         if (fakebop_prop_debug)
477                 bop_printf(NULL, "fakebop_getprop: copying %d bytes to 0x%x\n",
478                     p->bp_vlen, value);
479         bcopy(p->bp_value, value, p->bp_vlen);
480         return (0);
481 }

483 void
484 bop_printf(bootops_t *bop, const char *fmt, ...)
485 {
486         va_list ap;

488         va_start(ap, fmt);
489         (void) vsnprintf(buffer, BUFFERSIZE, fmt, ap);
490         va_end(ap);
491         bcons_puts(buffer);
492 }

494 static void
495 fakebop_setprop(char *name, int nlen, void *value, int vlen)
496 {
497         size_t size;
498         bootprop_t *bp;
499         caddr_t cur;
```



```
501         size = sizeof (bootprop_t) + nlen + 1 + vlen;
502         cur = fakebop_alloc(NULL, NULL, size, BO_ALIGN_DONTCARE);
503         bp = (bootprop_t *)cur;
504         if (bprops == NULL) {
505                 bprops = bp;
506                 bp->bp_next = NULL;
507         } else {
508                 bp->bp_next = bprops;
509                 bprops = bp;
510         }
511         cur += sizeof (bootprop_t);
512         bp->bp_name = cur;
513         bcopy(name, cur, nlen);
514         cur += nlen;
515         *cur = '\0';
516         cur++;
517         bp->bp_vlen = vlen;
518         bp->bp_value = cur;
519         if (vlen > 0)
520                 bcopy(value, cur, vlen);

522         if (fakebop_prop_debug)
523                 bop_printf(NULL, "setprop - name: %s, nlen: %d, vlen: %d\n",
524                     bp->bp_name, nlen, bp->bp_vlen);
525 }

527 static void
528 fakebop_setprop_string(char *name, char *value)
529 {
530         if (fakebop_prop_debug)
531                 bop_printf(NULL, "setprop_string: %s->[%s]\n", name, value);
532         fakebop_setprop(name, strlen(name), value, strlen(value) + 1);
533 }

535 static void
536 fakebop_setprop_32(char *name, uint32_t value)
537 {
538         if (fakebop_prop_debug)
539                 bop_printf(NULL, "setprop_32: %s->[%d]\n", name, value);
540         fakebop_setprop(name, strlen(name), (void *)&value, sizeof (value));
541 }

543 static void
544 fakebop_setprop_64(char *name, uint64_t value)
545 {
546         if (fakebop_prop_debug)
547                 bop_printf(NULL, "setprop_64: %s->[%lld]\n", name, value);
548         fakebop_setprop(name, strlen(name), (void *)&value, sizeof (value));
549 }

551 /*
552  * Here we create a bunch of the initial boot properties. This includes what
553  * we've been passed in via the command line. It also includes a few values that
554  * we compute ourselves.
555  */
556 static void
557 fakebop_bootprops_init(void)
558 {
559         int i = 0, proplen = 0, cmdline_len = 0, quote, cont;
560         static int stdout_val = 0;
561         char *c, *prop, *cmdline, *pname;
562         bootinfo_t *bp = &bootinfo;

564         /*
565          * Set the ramdisk properties for kobj_boot_mountroot() can succeed.
```

```
566          */
567         if ((bp->bi_flags & BI_HAS_RAMDISK) != 0) {
568                 fakebop_setprop_64("ramdisk_start",
569                     (uint64_t)(uintptr_t)bp->bi_ramdisk);
570                 fakebop_setprop_64("ramdisk_end",
571                     (uint64_t)(uintptr_t)bp->bi_ramdisk + bp->bi_ramsize);
572         }
573
574         /*
575          * TODO Various arm devices may spit properties at the front just like
576          * i86xpv. We should do something about them at some point.
577          */
578
579         /*
580          * Our boot parameters always wil start with kernel /platform/..., but
581          * the bootloader may in fact stick other properties in front of us. To
582          * deal with that we go ahead and we include them. We'll find the kernel
583          * and our properties set via -B when we finally find something without
584          * an equals sign, mainly kernel.
585          */
586         c = bp->bi_cmdline;
587         prop = strstr(c, "kernel");
588         if (prop == NULL)
589                 bop_panic("failed to find kernel string in boot params!");
590         /* Get us past the first kernel string */
591         prop += 6;
592         while (ISSPACE(prop[0]))
593                 prop++;
594         proplen = 0;
595         while (prop[proplen] != '\0' && !ISSPACE(prop[proplen]))
596                 proplen++;
597         c = prop + proplen + 1;
598         if (proplen > 0) {
599                 prop[proplen] = '\0';
600                 fakebop_setprop_string("boot-file", prop);
601                 /*
602                  * We strip the leading path from whoami so no matter what
603                  * environment we enter into here from it is consistent and
604                  * makes some amount of sense.
605                  */
606                 if (strstr(prop, "/platform") != NULL)
607                         prop = strstr(prop, "/platform");
608                 fakebop_setprop_string("whoami", prop);
609         } else {
610                 bop_panic("no kernel string in boot params!");
611         }
612
613         /*
614          * At this point we have two different sets of properties. Anything that
615          * involves -B is a boot property, otherwise it becomes part of the
616          * kernel command line and must be saved in its own property.
617          */
618         cmdline = fakebop_alloc(NULL, NULL, strlen(c), BO_ALIGN_DONTCARE);
619         cmdline[0] = '\0';
620         while (*c != '\0') {
621
622                 /*
623                  * Just blindly copy it to the commadline if we don't find it.
624                  */
625                 if (c[0] != '-' || c[1] != 'B') {
626                         cmdline[cmdline_len++] = *c;
627                         cmdline[cmdline_len] = '\0';
628                         c++;
629                         continue;
630                 }
```



```
632                 /* Get past "-B" */
633                 c += 2;
634                 while (ISSPACE(*c))
635                         c++;
636
637                 /*
638                  * We have a series of comma separated key-value pairs to sift
639                  * through here. The key and value are separated by an equals
640                  * sign. The value may quoted with either a ' or ". Note, white
641                  * space will also end the value (as it indicates that we have
642                  * moved on from the -B argument.
643                  */
644                 for (;;) {
645                         if (*c == '\0' || ISSPACE(*c))
646                                 break;
647                         prop = strchr(c, '=');
648                         if (prop == NULL)
649                                 break;
650                         pname = c;
651                         *prop = '\0';
652                         prop++;
653                         proplen = 0;
654                         quote = '\0';
655                         for (;;) {
656                                 if (prop[proplen] == '\0')
657                                         break;
658
659                                 if (proplen == 0 && (prop[0] == '\'' ||
660                                     prop[0] == '"')) {
661                                         quote = prop[0];
662                                         proplen++;
663                                         continue;
664                                 }
665
666                                 if (quote != '\0') {
667                                         if (prop[proplen] == quote)
668                                                 quote = '\0';
669                                         proplen++;
670                                         continue;
671                                 }
672
673                                 if (prop[proplen] == ',' ||
674                                     ISSPACE(prop[proplen]))
675                                         break;
676
677                                 /* We just have a normal character */
678                                 proplen++;
679                         }
680
681                         /*
682                          * Save whether we should continue or not and update 'c'
683                          * now as we will most likely clobber the string when we
684                          * are done.
685                          */
686                         cont = (prop[proplen] == ',');
687                         if (prop[proplen] != '\0')
688                                 c = prop + proplen + 1;
689                         else
690                                 c = prop + proplen;
691
692                         if (proplen == 0) {
693                                 fakebop_setprop_string(pname, "true");
694                         } else {
695                                 /*
696                                  * When we copy the prop, do not include the
697                                  * quote.
```

```
698                                  */
699                                 if (prop[0] == prop[proplen - 1] &&
700                                     (prop[0] == '\'' || prop[0] == '"')) {
701                                         prop++;
702                                         proplen -= 2;
703                                 }
704                                 prop[proplen] = '\0';
705                                 fakebop_setprop_string(pname, prop);
706                         }

708                         if (cont == 0)
709                                 break;
710                 }
711         }

713         /*
714          * Yes, we actually set both names here. The latter is set because of
715          * 1275.
716          */
717         fakebop_setprop_string("boot-args", cmdline);
718         fakebop_setprop_string("bootargs", cmdline);

720         /*
721          * Here are some things that we make up, just like our i86pc brethren.
722          */
723         fakebop_setprop_32("stdout", 0);
724         fakebop_setprop_string("mfg-name", "ARMv6");
725         fakebop_setprop_string("impl-arch-name", "ARMv6");
726 }

728 /*
729  * Nominally this should try and look for bootenv.rc, but seriously, let's not.
730  * Instead for now all we're going to to do is look and make sure that console
731  * is set. We *should* do something with it, but we're not.
732  */
733 void
734 boot_prop_finish(void)
735 {
736         int ret;

738         if (fakebop_getproplen(NULL, "console") <= 0)
739                 bop_panic("console not set");
740 }

742 /*ARGSUSED*/
743 int
744 boot_compinfo(int fd, struct compinfo *cbp)
745 {
746         cbp->iscmp = 0;
747         cbp->blksize = MAXBSIZE;
748         return (0);
749 }

751 extern void (*exception_table[])(void);

753 void
754 primordial_handler(void)
755 {
756         bop_panic("TRAP");
757 }

759 void
760 primordial_svc(void *arg)
761 {
762         struct regs *r = (struct regs *)arg;
763         uint32_t *insaddr = (uint32_t *)(r->r_lr - 4);
```



```
765         bop_printf(NULL, "TRAP: svc #%d from %p\n", *insaddr & 0x00ffffff,
766             insaddr);
767 }

769 void
770 primordial_undef(void *arg)
771 {
772         struct regs *r = (struct regs *)arg;
773         uint32_t *insaddr = (uint32_t *)(r->r_lr - 4);

775         bop_printf(NULL, "TRAP: undefined instruction %x at %p\n",
776             *insaddr, insaddr);
777 }

779 void
780 primordial_reset(void *arg)
781 {
782         struct regs *r = (struct regs *)arg;
783         uint32_t *insaddr = (uint32_t *)(r->r_lr - 4);

785         bop_printf(NULL, "TRAP: reset from %p\n",
786             insaddr);
787         bop_panic("cannot recover from reset\n");
788 }

790 void
791 primordial_prefetchabt(void *arg)
792 {
793         struct regs *r = (struct regs *)arg;
794         uint32_t *insaddr = (uint32_t *)(r->r_lr - 4);
795         uint32_t bkptno = ((*insaddr & 0xfff00) >> 4) | (*insaddr & 0xf);

797         bop_printf(NULL, "TRAP: prefetch (or bkpt #%d) at %p\n",
798             bkptno, insaddr);
799 }

801 /*
802  * XXX: This can't be tested without the MMU on, I don't think
803  *
804  * So while I'm sure (surely) we can decode this into a useful "what went
805  * wrong and why", I have no idea how, and couldn't test it if I did.
806  *
807  * I will say that I currently have the awful feeling that it would require an
808  * instruction decoder to decode *insaddr and find the memory refs...
809  */
810 void
811 primordial_dataabt(void *arg)
812 {
813         struct regs *r = (struct regs *)arg;
814         /* XXX: Yes, really +8 see ARM A2.6.6 */
815         uint32_t *insaddr = (uint32_t *)(r->r_lr - 8);

817         bop_printf(NULL, "TRAP: data abort at (insn) %p\n", insaddr);
818         bop_printf(NULL, "r0: %08x\tr1: %08x\n", r->r_r0, r->r_r1);
819         bop_printf(NULL, "r2: %08x\tr3: %08x\n", r->r_r2, r->r_r3);
820         bop_printf(NULL, "r4: %08x\tr5: %08x\n", r->r_r4, r->r_r5);
821         bop_printf(NULL, "r6: %08x\tr7: %08x\n", r->r_r6, r->r_r7);
822         bop_printf(NULL, "r8: %08x\tr9: %08x\n", r->r_r8, r->r_fp);
823         bop_panic("Page Fault! Go fix it\n");
824 }

826 void
827 bad_interrupt(void *arg)
828 {
829         bop_panic("Interrupt with VE == 0 (non-vectored)\n");
```

```
}

/* XXX: This should probably be somewhere else */
void (*trap_table[ARM_EXCPT_NUM])(void *) = {
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};

/*
 * Welcome to the kernel. We need to make a fake version of the boot_ops and the
 * boot_syscalls and then jump our way to _kobj_boot(). Here, we're borrowing
 * the Linux bootloader expectations, mostly because a lot of bootloaders and
 * boards already do this. If it turns out that we want to abstract this in the
 * future, then we should have locore.s do that before we get here.
 */
void
_fakebop_start(void *zeros, uint32_t machid, void *tagstart)
{
        atag_illumos_status_t *aisp;
        bootinfo_t *bip = &bootinfo;
        bootops_t *bops = &bootop;
        extern void _kobj_boot();

        fakebop_getatags(tagstart);
        bcons_init(bip->bi_cmdline);

        /* Now that we have a console, we can usefully handle traps */
        trap_table[ARM_EXCPT_RESET] = primordial_reset;
        trap_table[ARM_EXCPT_UNDINS] = primordial_undef;
        trap_table[ARM_EXCPT_SVC] = primordial_svc;
        trap_table[ARM_EXCPT_PREFETCH] = primordial_prefetchabt;
        trap_table[ARM_EXCPT_DATA] = primordial_dataabt;
        trap_table[ARM_EXCPT_IRQ] = bad_interrupt;
        trap_table[ARM_EXCPT_FIQ] = bad_interrupt;

        bop_printf(NULL, "Testing some exceptions\n");
        __asm__ __volatile__(".word 0xffffffff");
        __asm__ __volatile__("svc #14");
        /* the high and low bit in each field, so we can check the decode */
        __asm__ __volatile__("bkpt #32793");

        /* Clear some lines from the bootloader */
        bop_printf(NULL, "\nWelcome to fakebop -- ARM edition\n");
        if (fakebop_atag_debug != 0)
                fakebop_dump_tags(tagstart);

        aisp = (atag_illumos_status_t *)atag_find(tagstart,
            ATAG_ILLUMOS_STATUS);
        if (aisp == NULL)
                bop_panic("missing ATAG_ILLUMOS_STATUS!\n");

        /*
         * Fill in the bootops vector
         */
        bops->bsys_version = BO_VERSION;
        bops->bsys_alloc = fakebop_alloc;
        bops->bsys_free = fakebop_free;
        bops->bsys_getproplen = fakebop_getproplen;
        bops->bsys_getprop = fakebop_getprop;
        bops->bsys_printf = bop_printf;
```



```
        armboot_mmu_init(tagstart);
        fakebop_alloc_init(tagstart);
        fakebop_bootprops_init();
        bop_printf(NULL, "booting into _kobj\n");

        /*
         * krtld uses _stext, _etext, _sdata, and _edata as part of its segment
         * brks. ARM is a bit different from other versions of unix because we
         * have our exception vector in the binary at the top of our address
         * space. As such, we basically pass in values to krtld that represent
         * what our _etext and _edata would have looked like if not for the
         * exception vector.
         */
        _kobj_boot(&bop_sysp, NULL, bops, aisp->ais_stext, aisp->ais_etext,
            aisp->ais_sdata, aisp->ais_edata, EXCEPTION_ADDRESS);

        bop_panic("Returned from kobj_init\n");
}

void
_fakebop_locore_start(struct boot_syscalls *sysp, struct bootops *bops)
{
        bop_panic("Somehow made it back to fakebop_locore_start...");
}
```

```
**********************************************************
    2818 Wed Jan 14 19:07:05 2015
new/usr/src/uts/armv6/sys/atag.h
atag: remove whitespace error
**********************************************************
______unchanged_portion_omitted_
```