```
**********************************************************
    9810 Fri Apr 25 16:08:00 2014
new/usr/src/uts/i86pc/io/pcplusmp/apic_regops.c
4806 define x2apic feature flag
4807 pcplusmp & apix should use x2apic feature flag
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
 */
#endif /* ! codereview */

#include <sys/cpuvar.h>
#include <sys/psm.h>
#include <sys/archsystm.h>
#include <sys/apic.h>
#include <sys/sunddi.h>
#include <sys/ddi_impldefs.h>
#include <sys/mach_intr.h>
#include <sys/sysmacros.h>
#include <sys/trap.h>
#include <sys/x86_archext.h>
#include <sys/privregs.h>
#include <sys/psm_common.h>

/* Function prototypes of local apic and X2APIC */
static uint64_t local_apic_read(uint32_t reg);
static void local_apic_write(uint32_t reg, uint64_t value);
static int get_local_apic_pri(void);
static void local_apic_write_task_reg(uint64_t value);
static void local_apic_write_int_cmd(uint32_t cpu_id, uint32_t cmd1);
static uint64_t local_x2apic_read(uint32_t msr);
static void local_x2apic_write(uint32_t msr, uint64_t value);
static int get_local_x2apic_pri(void);
static void local_x2apic_write_task_reg(uint64_t value);
static void local_x2apic_write_int_cmd(uint32_t cpu_id, uint32_t cmd1);

/*
 * According to the X2APIC specification:
 *
 *   xAPIC global enable    X2APIC enable         Description
 *   (IA32_APIC_BASE[11])   (IA32_APIC_BASE[10])
```



```
 *   0                      0       APIC is disabled
 *   0                      1       Invalid
 *   1                      0       APIC is enabled in xAPIC mode
 *   1                      1       APIC is enabled in X2APIC mode
 * ------------------------------------------------------------
 */
int     x2apic_enable = 1;
apic_mode_t apic_mode = LOCAL_APIC;     /* Default mode is Local APIC */

/* Uses MMIO (Memory Mapped IO) */
static apic_reg_ops_t local_apic_regs_ops = {
        local_apic_read,
        local_apic_write,
        get_local_apic_pri,
        local_apic_write_task_reg,
        local_apic_write_int_cmd,
        apic_send_EOI,
};

/* X2APIC : Uses RDMSR/WRMSR instructions to access APIC registers */
static apic_reg_ops_t x2apic_regs_ops = {
        local_x2apic_read,
        local_x2apic_write,
        get_local_x2apic_pri,
        local_x2apic_write_task_reg,
        local_x2apic_write_int_cmd,
        apic_send_EOI,
};

int apic_have_32bit_cr8 = 0;

/* The default ops is local APIC (Memory Mapped IO) */
apic_reg_ops_t *apic_reg_ops = &local_apic_regs_ops;

/*
 * APIC register ops related data sturctures and functions.
 */
void apic_send_EOI();
void apic_send_directed_EOI(uint32_t irq);

#define X2APIC_CPUID_BIT        21
#define X2APIC_ENABLE_BIT       10

/*
 * Local APIC Implementation
 */
static uint64_t
local_apic_read(uint32_t reg)
{
        return ((uint32_t)apicadr[reg]);
}
_____unchanged_portion_omitted_

int
apic_detect_x2apic(void)
{
        struct cpuid_regs cp;

        if (x2apic_enable == 0)
                return (0);

        return (is_x86_feature(x86_featureset, X86FSET_X2APIC));
        cp.cp_eax = 1;
        (void) __cpuid_insn(&cp);
```

```
 172         return ((cp.cp_ecx & (0x1 << X2APIC_CPUID_BIT)) ? 1 : 0);
 243 }
```

unchanged_portion_omitted_

```
**********************************************************
  122611 Fri Apr 25 16:08:00 2014
new/usr/src/uts/i86pc/os/cpuid.c
4806 define x2apic feature flag
4807 pcplusmp & apix should use x2apic feature flag
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 by Delphix. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2014 Josef "Jeff" Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Portions Copyright 2009 Advanced Micro Devices, Inc.
 */
/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */
/*
 * Various routines to handle identification
 * and classification of x86 processors.
 */

#include <sys/types.h>
#include <sys/archsystm.h>
#include <sys/x86_archext.h>
#include <sys/kmem.h>
#include <sys/systm.h>
#include <sys/cmn_err.h>
#include <sys/sunddi.h>
#include <sys/sunndi.h>
#include <sys/cpuvar.h>
#include <sys/processor.h>
#include <sys/sysmacros.h>
#include <sys/pg.h>
#include <sys/fp.h>
#include <sys/controlregs.h>
#include <sys/bitmap.h>
#include <sys/auxv_386.h>
#include <sys/memnode.h>
```



```
#include <sys/pci_cfgspace.h>

#ifdef __xpv
#include <sys/hypervisor.h>
#else
#include <sys/ontrap.h>
#endif

/*
 * Pass 0 of cpuid feature analysis happens in locore. It contains special code
 * to recognize Cyrix processors that are not cpuid-compliant, and to deal with
 * them accordingly. For most modern processors, feature detection occurs here
 * in pass 1.
 *
 * Pass 1 of cpuid feature analysis happens just at the beginning of mlsetup()
 * for the boot CPU and does the basic analysis that the early kernel needs.
 * x86_featureset is set based on the return value of cpuid_pass1() of the boot
 * CPU.
 *
 * Pass 1 includes:
 *
 *      o Determining vendor/model/family/stepping and setting x86_type and
 *        x86_vendor accordingly.
 *      o Processing the feature flags returned by the cpuid instruction while
 *        applying any workarounds or tricks for the specific processor.
 *      o Mapping the feature flags into Solaris feature bits (X86_*).
 *      o Processing extended feature flags if supported by the processor,
 *        again while applying specific processor knowledge.
 *      o Determining the CMT characteristics of the system.
 *
 * Pass 1 is done on non-boot CPUs during their initialization and the results
 * are used only as a meager attempt at ensuring that all processors within the
 * system support the same features.
 *
 * Pass 2 of cpuid feature analysis happens just at the beginning
 * of startup().  It just copies in and corrects the remainder
 * of the cpuid data we depend on: standard cpuid functions that we didn't
 * need for pass1 feature analysis, and extended cpuid functions beyond the
 * simple feature processing done in pass1.
 *
 * Pass 3 of cpuid analysis is invoked after basic kernel services; in
 * particular kernel memory allocation has been made available. It creates a
 * readable brand string based on the data collected in the first two passes.
 *
 * Pass 4 of cpuid analysis is invoked after post_startup() when all
 * the support infrastructure for various hardware features has been
 * initialized. It determines which processor features will be reported
 * to userland via the aux vector.
 *
 * All passes are executed on all CPUs, but only the boot CPU determines what
 * features the kernel will use.
 *
 * Much of the worst junk in this file is for the support of processors
 * that didn't really implement the cpuid instruction properly.
 *
 * NOTE: The accessor functions (cpuid_get*) are aware of, and ASSERT upon,
 * the pass numbers.  Accordingly, changes to the pass code may require changes
 * to the accessor code.
 */

uint_t x86_vendor = X86_VENDOR_IntelClone;
uint_t x86_type = X86_TYPE_OTHER;
uint_t x86_clflush_size = 0;

uint_t pentiumpro_bug4046376;
```

```
126 uchar_t x86_featureset[BT_SIZEOFMAP(NUM_X86_FEATURES)];

128 static char *x86_feature_names[NUM_X86_FEATURES] = {
129         "lgpg",
130         "tsc",
131         "msr",
132         "mtrr",
133         "pge",
134         "de",
135         "cmov",
136         "mmx",
137         "mca",
138         "pae",
139         "cv8",
140         "pat",
141         "sep",
142         "sse",
143         "sse2",
144         "htt",
145         "asysc",
146         "nx",
147         "sse3",
148         "cx16",
149         "cmp",
150         "tscp",
151         "mwait",
152         "sse4a",
153         "cpuid",
154         "ssse3",
155         "sse4_1",
156         "sse4_2",
157         "1gpg",
158         "clfsh",
159         "64",
160         "aes",
161         "pclmulqdq",
162         "xsave",
163         "avx",
164         "vmx",
165         "svm",
166         "topoext",
167         "f16c",
168         "rdrand",
169         "x2apic",
 25         "rdrand"
170 };
```

*unchanged_portion_omitted_*

```
936 void
937 cpuid_pass1(cpu_t *cpu, uchar_t *featureset)
938 {
939         uint32_t mask_ecx, mask_edx;
940         struct cpuid_info *cpi;
941         struct cpuid_regs *cp;
942         int xcpuid;
943 #if !defined(__xpv)
944         extern int idle_cpu_prefer_mwait;
945 #endif

947         /*
948          * Space statically allocated for BSP, ensure pointer is set
949          */
950         if (cpu->cpu_id == 0) {
951                 if (cpu->cpu_m.mcpu_cpi == NULL)
952                         cpu->cpu_m.mcpu_cpi = &cpuid_info0;
953         }
```



```
955         add_x86_feature(featureset, X86FSET_CPUID);

957         cpi = cpu->cpu_m.mcpu_cpi;
958         ASSERT(cpi != NULL);
959         cp = &cpi->cpi_std[0];
960         cp->cp_eax = 0;
961         cpi->cpi_maxeax = __cpuid_insn(cp);
962         {
963                 uint32_t *iptr = (uint32_t *)cpi->cpi_vendorstr;
964                 *iptr++ = cp->cp_ebx;
965                 *iptr++ = cp->cp_edx;
966                 *iptr++ = cp->cp_ecx;
967                 *(char *)&cpi->cpi_vendorstr[12] = '\0';
968         }

970         cpi->cpi_vendor = _cpuid_vendorstr_to_vendorcode(cpi->cpi_vendorstr);
971         x86_vendor = cpi->cpi_vendor; /* for compatibility */

973         /*
974          * Limit the range in case of weird hardware
975          */
976         if (cpi->cpi_maxeax > CPI_MAXEAX_MAX)
977                 cpi->cpi_maxeax = CPI_MAXEAX_MAX;
978         if (cpi->cpi_maxeax < 1)
979                 goto pass1_done;

981         cp = &cpi->cpi_std[1];
982         cp->cp_eax = 1;
983         (void) __cpuid_insn(cp);

985         /*
986          * Extract identifying constants for easy access.
987          */
988         cpi->cpi_model = CPI_MODEL(cpi);
989         cpi->cpi_family = CPI_FAMILY(cpi);

991         if (cpi->cpi_family == 0xf)
992                 cpi->cpi_family += CPI_FAMILY_XTD(cpi);

994         /*
995          * Beware: AMD uses "extended model" iff base *FAMILY* == 0xf.
996          * Intel, and presumably everyone else, uses model == 0xf, as
997          * one would expect (max value means possible overflow).  Sigh.
998          */

1000        switch (cpi->cpi_vendor) {
1001        case X86_VENDOR_Intel:
1002                if (IS_EXTENDED_MODEL_INTEL(cpi))
1003                        cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
1004                break;
1005        case X86_VENDOR_AMD:
1006                if (CPI_FAMILY(cpi) == 0xf)
1007                        cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
1008                break;
1009        default:
1010                if (cpi->cpi_model == 0xf)
1011                        cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
1012                break;
1013        }

1015        cpi->cpi_step = CPI_STEP(cpi);
1016        cpi->cpi_brandid = CPI_BRANDID(cpi);

1018        /*
1019         * *default* assumptions:
```

```
1020          * - believe %edx feature word
1021          * - ignore %ecx feature word
1022          * - 32-bit virtual and physical addressing
1023          */
1024         mask_edx = 0xffffffff;
1025         mask_ecx = 0;

1027         cpi->cpi_pabits = cpi->cpi_vabits = 32;

1029         switch (cpi->cpi_vendor) {
1030         case X86_VENDOR_Intel:
1031                 if (cpi->cpi_family == 5)
1032                         x86_type = X86_TYPE_P5;
1033                 else if (IS_LEGACY_P6(cpi)) {
1034                         x86_type = X86_TYPE_P6;
1035                         pentiumpro_bug4046376 = 1;
1036                         /*
1037                          * Clear the SEP bit when it was set erroneously
1038                          */
1039                         if (cpi->cpi_model < 3 && cpi->cpi_step < 3)
1040                                 cp->cp_edx &= ~CPUID_INTC_EDX_SEP;
1041                 } else if (IS_NEW_F6(cpi) || cpi->cpi_family == 0xf) {
1042                         x86_type = X86_TYPE_P4;
1043                         /*
1044                          * We don't currently depend on any of the %ecx
1045                          * features until Prescott, so we'll only check
1046                          * this from P4 onwards.  We might want to revisit
1047                          * that idea later.
1048                          */
1049                         mask_ecx = 0xffffffff;
1050                 } else if (cpi->cpi_family > 0xf)
1051                         mask_ecx = 0xffffffff;
1052                 /*
1053                  * We don't support MONITOR/MWAIT if leaf 5 is not available
1054                  * to obtain the monitor linesize.
1055                  */
1056                 if (cpi->cpi_maxeax < 5)
1057                         mask_ecx &= ~CPUID_INTC_ECX_MON;
1058                 break;
1059         case X86_VENDOR_IntelClone:
1060         default:
1061                 break;
1062         case X86_VENDOR_AMD:
1063 #if defined(OPTERON_ERRATUM_108)
1064                 if (cpi->cpi_family == 0xf && cpi->cpi_model == 0xe) {
1065                         cp->cp_eax = (0xf0f & cp->cp_eax) | 0xc0;
1066                         cpi->cpi_model = 0xc;
1067                 } else
1068 #endif
1069                 if (cpi->cpi_family == 5) {
1070                         /*
1071                          * AMD K5 and K6
1072                          *
1073                          * These CPUs have an incomplete implementation
1074                          * of MCA/MCE which we mask away.
1075                          */
1076                         mask_edx &= ~(CPUID_INTC_EDX_MCE | CPUID_INTC_EDX_MCA);

1078                         /*
1079                          * Model 0 uses the wrong (APIC) bit
1080                          * to indicate PGE.  Fix it here.
1081                          */
1082                         if (cpi->cpi_model == 0) {
1083                                 if (cp->cp_edx & 0x200) {
1084                                         cp->cp_edx &= ~0x200;
1085                                         cp->cp_edx |= CPUID_INTC_EDX_PGE;
```



```
1086                                 }
1087                         }

1089                         /*
1090                          * Early models had problems w/ MMX; disable.
1091                          */
1092                         if (cpi->cpi_model < 6)
1093                                 mask_edx &= ~CPUID_INTC_EDX_MMX;
1094                 }

1096                 /*
1097                  * For newer families, SSE3 and CX16, at least, are valid;
1098                  * enable all
1099                  */
1100                 if (cpi->cpi_family >= 0xf)
1101                         mask_ecx = 0xffffffff;
1102                 /*
1103                  * We don't support MONITOR/MWAIT if leaf 5 is not available
1104                  * to obtain the monitor linesize.
1105                  */
1106                 if (cpi->cpi_maxeax < 5)
1107                         mask_ecx &= ~CPUID_INTC_ECX_MON;

1109 #if !defined(__xpv)
1110                 /*
1111                  * Do not use MONITOR/MWAIT to halt in the idle loop on any AMD
1112                  * processors.  AMD does not intend MWAIT to be used in the cpu
1113                  * idle loop on current and future processors.  10h and future
1114                  * AMD processors use more power in MWAIT than HLT.
1115                  * Pre-family-10h Opterons do not have the MWAIT instruction.
1116                  */
1117                 idle_cpu_prefer_mwait = 0;
1118 #endif

1120                 break;
1121         case X86_VENDOR_TM:
1122                 /*
1123                  * workaround the NT workaround in CMS 4.1
1124                  */
1125                 if (cpi->cpi_family == 5 && cpi->cpi_model == 4 &&
1126                     (cpi->cpi_step == 2 || cpi->cpi_step == 3))
1127                         cp->cp_edx |= CPUID_INTC_EDX_CX8;
1128                 break;
1129         case X86_VENDOR_Centaur:
1130                 /*
1131                  * workaround the NT workarounds again
1132                  */
1133                 if (cpi->cpi_family == 6)
1134                         cp->cp_edx |= CPUID_INTC_EDX_CX8;
1135                 break;
1136         case X86_VENDOR_Cyrix:
1137                 /*
1138                  * We rely heavily on the probing in locore
1139                  * to actually figure out what parts, if any,
1140                  * of the Cyrix cpuid instruction to believe.
1141                  */
1142                 switch (x86_type) {
1143                 case X86_TYPE_CYRIX_486:
1144                         mask_edx = 0;
1145                         break;
1146                 case X86_TYPE_CYRIX_6x86:
1147                         mask_edx = 0;
1148                         break;
1149                 case X86_TYPE_CYRIX_6x86L:
1150                         mask_edx =
1151                             CPUID_INTC_EDX_DE |
```

```
1152                            CPUID_INTC_EDX_CX8;
1153                    break;
1154            case X86_TYPE_CYRIX_6x86MX:
1155                    mask_edx =
1156                        CPUID_INTC_EDX_DE |
1157                        CPUID_INTC_EDX_MSR |
1158                        CPUID_INTC_EDX_CX8 |
1159                        CPUID_INTC_EDX_PGE |
1160                        CPUID_INTC_EDX_CMOV |
1161                        CPUID_INTC_EDX_MMX;
1162                    break;
1163            case X86_TYPE_CYRIX_GXm:
1164                    mask_edx =
1165                        CPUID_INTC_EDX_MSR |
1166                        CPUID_INTC_EDX_CX8 |
1167                        CPUID_INTC_EDX_CMOV |
1168                        CPUID_INTC_EDX_MMX;
1169                    break;
1170            case X86_TYPE_CYRIX_MediaGX:
1171                    break;
1172            case X86_TYPE_CYRIX_MII:
1173            case X86_TYPE_VIA_CYRIX_III:
1174                    mask_edx =
1175                        CPUID_INTC_EDX_DE |
1176                        CPUID_INTC_EDX_TSC |
1177                        CPUID_INTC_EDX_MSR |
1178                        CPUID_INTC_EDX_CX8 |
1179                        CPUID_INTC_EDX_PGE |
1180                        CPUID_INTC_EDX_CMOV |
1181                        CPUID_INTC_EDX_MMX;
1182                    break;
1183            default:
1184                    break;
1185            }
1186            break;
1187    }

1189 #if defined(__xpv)
1190    /*
1191     * Do not support MONITOR/MWAIT under a hypervisor
1192     */
1193    mask_ecx &= ~CPUID_INTC_ECX_MON;
1194    /*
1195     * Do not support XSAVE under a hypervisor for now
1196     */
1197    xsave_force_disable = B_TRUE;

1199 #endif  /* __xpv */

1201    if (xsave_force_disable) {
1202            mask_ecx &= ~CPUID_INTC_ECX_XSAVE;
1203            mask_ecx &= ~CPUID_INTC_ECX_AVX;
1204            mask_ecx &= ~CPUID_INTC_ECX_F16C;
1205    }

1207    /*
1208     * Now we've figured out the masks that determine
1209     * which bits we choose to believe, apply the masks
1210     * to the feature words, then map the kernel's view
1211     * of these feature words into its feature word.
1212     */
1213    cp->cp_edx &= mask_edx;
1214    cp->cp_ecx &= mask_ecx;

1216    /*
1217     * apply any platform restrictions (we don't call this
```



```
1218     * immediately after __cpuid_insn here, because we need the
1219     * workarounds applied above first)
1220     */
1221    platform_cpuid_mangle(cpi->cpi_vendor, 1, cp);

1223    /*
1224     * fold in overrides from the "eeprom" mechanism
1225     */
1226    cp->cp_edx |= cpuid_feature_edx_include;
1227    cp->cp_edx &= ~cpuid_feature_edx_exclude;

1229    cp->cp_ecx |= cpuid_feature_ecx_include;
1230    cp->cp_ecx &= ~cpuid_feature_ecx_exclude;

1232    if (cp->cp_edx & CPUID_INTC_EDX_PSE) {
1233            add_x86_feature(featureset, X86FSET_LARGEPAGE);
1234    }
1235    if (cp->cp_edx & CPUID_INTC_EDX_TSC) {
1236            add_x86_feature(featureset, X86FSET_TSC);
1237    }
1238    if (cp->cp_edx & CPUID_INTC_EDX_MSR) {
1239            add_x86_feature(featureset, X86FSET_MSR);
1240    }
1241    if (cp->cp_edx & CPUID_INTC_EDX_MTRR) {
1242            add_x86_feature(featureset, X86FSET_MTRR);
1243    }
1244    if (cp->cp_edx & CPUID_INTC_EDX_PGE) {
1245            add_x86_feature(featureset, X86FSET_PGE);
1246    }
1247    if (cp->cp_edx & CPUID_INTC_EDX_CMOV) {
1248            add_x86_feature(featureset, X86FSET_CMOV);
1249    }
1250    if (cp->cp_edx & CPUID_INTC_EDX_MMX) {
1251            add_x86_feature(featureset, X86FSET_MMX);
1252    }
1253    if ((cp->cp_edx & CPUID_INTC_EDX_MCE) != 0 &&
1254        (cp->cp_edx & CPUID_INTC_EDX_MCA) != 0) {
1255            add_x86_feature(featureset, X86FSET_MCA);
1256    }
1257    if (cp->cp_edx & CPUID_INTC_EDX_PAE) {
1258            add_x86_feature(featureset, X86FSET_PAE);
1259    }
1260    if (cp->cp_edx & CPUID_INTC_EDX_CX8) {
1261            add_x86_feature(featureset, X86FSET_CX8);
1262    }
1263    if (cp->cp_ecx & CPUID_INTC_ECX_CX16) {
1264            add_x86_feature(featureset, X86FSET_CX16);
1265    }
1266    if (cp->cp_edx & CPUID_INTC_EDX_PAT) {
1267            add_x86_feature(featureset, X86FSET_PAT);
1268    }
1269    if (cp->cp_edx & CPUID_INTC_EDX_SEP) {
1270            add_x86_feature(featureset, X86FSET_SEP);
1271    }
1272    if (cp->cp_edx & CPUID_INTC_EDX_FXSR) {
1273            /*
1274             * In our implementation, fxsave/fxrstor
1275             * are prerequisites before we'll even
1276             * try and do SSE things.
1277             */
1278            if (cp->cp_edx & CPUID_INTC_EDX_SSE) {
1279                    add_x86_feature(featureset, X86FSET_SSE);
1280            }
1281            if (cp->cp_edx & CPUID_INTC_EDX_SSE2) {
1282                    add_x86_feature(featureset, X86FSET_SSE2);
1283            }
```

```
1284                if (cp->cp_ecx & CPUID_INTC_ECX_SSE3) {
1285                        add_x86_feature(featureset, X86FSET_SSE3);
1286                }
1287                if (cp->cp_ecx & CPUID_INTC_ECX_SSSE3) {
1288                        add_x86_feature(featureset, X86FSET_SSSE3);
1289                }
1290                if (cp->cp_ecx & CPUID_INTC_ECX_SSE4_1) {
1291                        add_x86_feature(featureset, X86FSET_SSE4_1);
1292                }
1293                if (cp->cp_ecx & CPUID_INTC_ECX_SSE4_2) {
1294                        add_x86_feature(featureset, X86FSET_SSE4_2);
1295                }
1296                if (cp->cp_ecx & CPUID_INTC_ECX_AES) {
1297                        add_x86_feature(featureset, X86FSET_AES);
1298                }
1299                if (cp->cp_ecx & CPUID_INTC_ECX_PCLMULQDQ) {
1300                        add_x86_feature(featureset, X86FSET_PCLMULQDQ);
1301                }

1303                if (cp->cp_ecx & CPUID_INTC_ECX_XSAVE) {
1304                        add_x86_feature(featureset, X86FSET_XSAVE);

1306                        /* We only test AVX when there is XSAVE */
1307                        if (cp->cp_ecx & CPUID_INTC_ECX_AVX) {
1308                                add_x86_feature(featureset,
1309                                    X86FSET_AVX);

1311                                if (cp->cp_ecx & CPUID_INTC_ECX_F16C)
1312                                        add_x86_feature(featureset,
1313                                            X86FSET_F16C);
1314                        }
1315                }
1316        }
1317        if (cp->cp_ecx & CPUID_INTC_ECX_X2APIC) {
1318                add_x86_feature(featureset, X86FSET_X2APIC);
1319        }
1320 #endif /* ! codereview */
1321        if (cp->cp_edx & CPUID_INTC_EDX_DE) {
1322                add_x86_feature(featureset, X86FSET_DE);
1323        }
1324 #if !defined(__xpv)
1325        if (cp->cp_ecx & CPUID_INTC_ECX_MON) {

1327                /*
1328                 * We require the CLFLUSH instruction for erratum workaround
1329                 * to use MONITOR/MWAIT.
1330                 */
1331                if (cp->cp_edx & CPUID_INTC_EDX_CLFSH) {
1332                        cpi->cpi_mwait.support |= MWAIT_SUPPORT;
1333                        add_x86_feature(featureset, X86FSET_MWAIT);
1334                } else {
1335                        extern int idle_cpu_assert_cflush_monitor;

1337                        /*
1338                         * All processors we are aware of which have
1339                         * MONITOR/MWAIT also have CLFLUSH.
1340                         */
1341                        if (idle_cpu_assert_cflush_monitor) {
1342                                ASSERT((cp->cp_ecx & CPUID_INTC_ECX_MON) &&
1343                                    (cp->cp_edx & CPUID_INTC_EDX_CLFSH));
1344                        }
1345                }
1346        }
1347 #endif  /* __xpv */

1349        if (cp->cp_ecx & CPUID_INTC_ECX_VMX) {
```



```
1350                add_x86_feature(featureset, X86FSET_VMX);
1351        }

1353        if (cp->cp_ecx & CPUID_INTC_ECX_RDRAND)
1354                add_x86_feature(featureset, X86FSET_RDRAND);

1356        /*
1357         * Only need it first time, rest of the cpus would follow suit.
1358         * we only capture this for the bootcpu.
1359         */
1360        if (cp->cp_edx & CPUID_INTC_EDX_CLFSH) {
1361                add_x86_feature(featureset, X86FSET_CLFSH);
1362                x86_clflush_size = (BITX(cp->cp_ebx, 15, 8) * 8);
1363        }
1364        if (is_x86_feature(featureset, X86FSET_PAE))
1365                cpi->cpi_pabits = 36;

1367        /*
1368         * Hyperthreading configuration is slightly tricky on Intel
1369         * and pure clones, and even trickier on AMD.
1370         *
1371         * (AMD chose to set the HTT bit on their CMP processors,
1372         * even though they're not actually hyperthreaded.  Thus it
1373         * takes a bit more work to figure out what's really going
1374         * on ... see the handling of the CMP_LGCY bit below)
1375         */
1376        if (cp->cp_edx & CPUID_INTC_EDX_HTT) {
1377                cpi->cpi_ncpu_per_chip = CPI_CPU_COUNT(cpi);
1378                if (cpi->cpi_ncpu_per_chip > 1)
1379                        add_x86_feature(featureset, X86FSET_HTT);
1380        } else {
1381                cpi->cpi_ncpu_per_chip = 1;
1382        }

1384        /*
1385         * Work on the "extended" feature information, doing
1386         * some basic initialization for cpuid_pass2()
1387         */
1388        xcpuid = 0;
1389        switch (cpi->cpi_vendor) {
1390        case X86_VENDOR_Intel:
1391                if (IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf)
1392                        xcpuid++;
1393                break;
1394        case X86_VENDOR_AMD:
1395                if (cpi->cpi_family > 5 ||
1396                    (cpi->cpi_family == 5 && cpi->cpi_model >= 1))
1397                        xcpuid++;
1398                break;
1399        case X86_VENDOR_Cyrix:
1400                /*
1401                 * Only these Cyrix CPUs are -known- to support
1402                 * extended cpuid operations.
1403                 */
1404                if (x86_type == X86_TYPE_VIA_CYRIX_III ||
1405                    x86_type == X86_TYPE_CYRIX_GXm)
1406                        xcpuid++;
1407                break;
1408        case X86_VENDOR_Centaur:
1409        case X86_VENDOR_TM:
1410        default:
1411                xcpuid++;
1412                break;
1413        }

1415        if (xcpuid) {
```

```
1416                cp = &cpi->cpi_extd[0];
1417                cp->cp_eax = 0x80000000;
1418                cpi->cpi_xmaxeax = __cpuid_insn(cp);
1419        }

1421        if (cpi->cpi_xmaxeax & 0x80000000) {

1423                if (cpi->cpi_xmaxeax > CPI_XMAXEAX_MAX)
1424                        cpi->cpi_xmaxeax = CPI_XMAXEAX_MAX;

1426                switch (cpi->cpi_vendor) {
1427                case X86_VENDOR_Intel:
1428                case X86_VENDOR_AMD:
1429                        if (cpi->cpi_xmaxeax < 0x80000001)
1430                                break;
1431                        cp = &cpi->cpi_extd[1];
1432                        cp->cp_eax = 0x80000001;
1433                        (void) __cpuid_insn(cp);

1435                        if (cpi->cpi_vendor == X86_VENDOR_AMD &&
1436                            cpi->cpi_family == 5 &&
1437                            cpi->cpi_model == 6 &&
1438                            cpi->cpi_step == 6) {
1439                                /*
1440                                 * K6 model 6 uses bit 10 to indicate SYSC
1441                                 * Later models use bit 11. Fix it here.
1442                                 */
1443                                if (cp->cp_edx & 0x400) {
1444                                        cp->cp_edx &= ~0x400;
1445                                        cp->cp_edx |= CPUID_AMD_EDX_SYSC;
1446                                }
1447                        }

1449                        platform_cpuid_mangle(cpi->cpi_vendor, 0x80000001, cp);

1451                        /*
1452                         * Compute the additions to the kernel's feature word.
1453                         */
1454                        if (cp->cp_edx & CPUID_AMD_EDX_NX) {
1455                                add_x86_feature(featureset, X86FSET_NX);
1456                        }

1458                        /*
1459                         * Regardless whether or not we boot 64-bit,
1460                         * we should have a way to identify whether
1461                         * the CPU is capable of running 64-bit.
1462                         */
1463                        if (cp->cp_edx & CPUID_AMD_EDX_LM) {
1464                                add_x86_feature(featureset, X86FSET_64);
1465                        }

1467 #if defined(__amd64)
1468                        /* 1 GB large page - enable only for 64 bit kernel */
1469                        if (cp->cp_edx & CPUID_AMD_EDX_1GPG) {
1470                                add_x86_feature(featureset, X86FSET_1GPG);
1471                        }
1472 #endif

1474                        if ((cpi->cpi_vendor == X86_VENDOR_AMD) &&
1475                            (cpi->cpi_std[1].cp_edx & CPUID_INTC_EDX_FXSR) &&
1476                            (cp->cp_ecx & CPUID_AMD_ECX_SSE4A)) {
1477                                add_x86_feature(featureset, X86FSET_SSE4A);
1478                        }

1480                        /*
1481                         * If both the HTT and CMP_LGCY bits are set,
```



```
1482                         * then we're not actually HyperThreaded.  Read
1483                         * "AMD CPUID Specification" for more details.
1484                         */
1485                        if (cpi->cpi_vendor == X86_VENDOR_AMD &&
1486                            is_x86_feature(featureset, X86FSET_HTT) &&
1487                            (cp->cp_ecx & CPUID_AMD_ECX_CMP_LGCY)) {
1488                                remove_x86_feature(featureset, X86FSET_HTT);
1489                                add_x86_feature(featureset, X86FSET_CMP);
1490                        }
1491 #if defined(__amd64)
1492                        /*
1493                         * It's really tricky to support syscall/sysret in
1494                         * the i386 kernel; we rely on sysenter/sysexit
1495                         * instead.  In the amd64 kernel, things are -way-
1496                         * better.
1497                         */
1498                        if (cp->cp_edx & CPUID_AMD_EDX_SYSC) {
1499                                add_x86_feature(featureset, X86FSET_ASYSC);
1500                        }

1502                        /*
1503                         * While we're thinking about system calls, note
1504                         * that AMD processors don't support sysenter
1505                         * in long mode at all, so don't try to program them.
1506                         */
1507                        if (x86_vendor == X86_VENDOR_AMD) {
1508                                remove_x86_feature(featureset, X86FSET_SEP);
1509                        }
1510 #endif
1511                        if (cp->cp_edx & CPUID_AMD_EDX_TSCP) {
1512                                add_x86_feature(featureset, X86FSET_TSCP);
1513                        }

1515                        if (cp->cp_ecx & CPUID_AMD_ECX_SVM) {
1516                                add_x86_feature(featureset, X86FSET_SVM);
1517                        }

1519                        if (cp->cp_ecx & CPUID_AMD_ECX_TOPOEXT) {
1520                                add_x86_feature(featureset, X86FSET_TOPOEXT);
1521                        }
1522                        break;
1523                default:
1524                        break;
1525                }

1527                /*
1528                 * Get CPUID data about processor cores and hyperthreads.
1529                 */
1530                switch (cpi->cpi_vendor) {
1531                case X86_VENDOR_Intel:
1532                        if (cpi->cpi_maxeax >= 4) {
1533                                cp = &cpi->cpi_std[4];
1534                                cp->cp_eax = 4;
1535                                cp->cp_ecx = 0;
1536                                (void) __cpuid_insn(cp);
1537                                platform_cpuid_mangle(cpi->cpi_vendor, 4, cp);
1538                        }
1539                        /*FALLTHROUGH*/
1540                case X86_VENDOR_AMD:
1541                        if (cpi->cpi_xmaxeax < 0x80000008)
1542                                break;
1543                        cp = &cpi->cpi_extd[8];
1544                        cp->cp_eax = 0x80000008;
1545                        (void) __cpuid_insn(cp);
1546                        platform_cpuid_mangle(cpi->cpi_vendor, 0x80000008, cp);
```

```
1548                        /*
1549                         * Virtual and physical address limits from
1550                         * cpuid override previously guessed values.
1551                         */
1552                        cpi->cpi_pabits = BITX(cp->cp_eax, 7, 0);
1553                        cpi->cpi_vabits = BITX(cp->cp_eax, 15, 8);
1554                        break;
1555                default:
1556                        break;
1557                }

1559                /*
1560                 * Derive the number of cores per chip
1561                 */
1562                switch (cpi->cpi_vendor) {
1563                case X86_VENDOR_Intel:
1564                        if (cpi->cpi_maxeax < 4) {
1565                                cpi->cpi_ncore_per_chip = 1;
1566                                break;
1567                        } else {
1568                                cpi->cpi_ncore_per_chip =
1569                                    BITX((cpi)->cpi_std[4].cp_eax, 31, 26) + 1;
1570                        }
1571                        break;
1572                case X86_VENDOR_AMD:
1573                        if (cpi->cpi_xmaxeax < 0x80000008) {
1574                                cpi->cpi_ncore_per_chip = 1;
1575                                break;
1576                        } else {
1577                                /*
1578                                 * On family 0xf cpuid fn 2 ECX[7:0] "NC" is
1579                                 * 1 less than the number of physical cores on
1580                                 * the chip.  In family 0x10 this value can
1581                                 * be affected by "downcoring" - it reflects
1582                                 * 1 less than the number of cores actually
1583                                 * enabled on this node.
1584                                 */
1585                                cpi->cpi_ncore_per_chip =
1586                                    BITX((cpi)->cpi_extd[8].cp_ecx, 7, 0) + 1;
1587                        }
1588                        break;
1589                default:
1590                        cpi->cpi_ncore_per_chip = 1;
1591                        break;
1592                }

1594                /*
1595                 * Get CPUID data about TSC Invariance in Deep C-State.
1596                 */
1597                switch (cpi->cpi_vendor) {
1598                case X86_VENDOR_Intel:
1599                        if (cpi->cpi_maxeax >= 7) {
1600                                cp = &cpi->cpi_extd[7];
1601                                cp->cp_eax = 0x80000007;
1602                                cp->cp_ecx = 0;
1603                                (void) __cpuid_insn(cp);
1604                        }
1605                        break;
1606                default:
1607                        break;
1608                }
1609        } else {
1610                cpi->cpi_ncore_per_chip = 1;
1611        }

1613        /*
```



```
1614         * If more than one core, then this processor is CMP.
1615         */
1616        if (cpi->cpi_ncore_per_chip > 1) {
1617                add_x86_feature(featureset, X86FSET_CMP);
1618        }

1620        /*
1621         * If the number of cores is the same as the number
1622         * of CPUs, then we cannot have HyperThreading.
1623         */
1624        if (cpi->cpi_ncpu_per_chip == cpi->cpi_ncore_per_chip) {
1625                remove_x86_feature(featureset, X86FSET_HTT);
1626        }

1628        cpi->cpi_apicid = CPI_APIC_ID(cpi);
1629        cpi->cpi_procnodes_per_pkg = 1;
1630        cpi->cpi_cores_per_compunit = 1;
1631        if (is_x86_feature(featureset, X86FSET_HTT) == B_FALSE &&
1632            is_x86_feature(featureset, X86FSET_CMP) == B_FALSE) {
1633                /*
1634                 * Single-core single-threaded processors.
1635                 */
1636                cpi->cpi_chipid = -1;
1637                cpi->cpi_clogid = 0;
1638                cpi->cpi_coreid = cpu->cpu_id;
1639                cpi->cpi_pkgcoreid = 0;
1640                if (cpi->cpi_vendor == X86_VENDOR_AMD)
1641                        cpi->cpi_procnodeid = BITX(cpi->cpi_apicid, 3, 0);
1642                else
1643                        cpi->cpi_procnodeid = cpi->cpi_chipid;
1644        } else if (cpi->cpi_ncpu_per_chip > 1) {
1645                if (cpi->cpi_vendor == X86_VENDOR_Intel)
1646                        cpuid_intel_getids(cpu, featureset);
1647                else if (cpi->cpi_vendor == X86_VENDOR_AMD)
1648                        cpuid_amd_getids(cpu);
1649                else {
1650                        /*
1651                         * All other processors are currently
1652                         * assumed to have single cores.
1653                         */
1654                        cpi->cpi_coreid = cpi->cpi_chipid;
1655                        cpi->cpi_pkgcoreid = 0;
1656                        cpi->cpi_procnodeid = cpi->cpi_chipid;
1657                        cpi->cpi_compunitid = cpi->cpi_chipid;
1658                }
1659        }

1661        /*
1662         * Synthesize chip "revision" and socket type
1663         */
1664        cpi->cpi_chiprev = _cpuid_chiprev(cpi->cpi_vendor, cpi->cpi_family,
1665            cpi->cpi_model, cpi->cpi_step);
1666        cpi->cpi_chiprevstr = _cpuid_chiprevstr(cpi->cpi_vendor,
1667            cpi->cpi_family, cpi->cpi_model, cpi->cpi_step);
1668        cpi->cpi_socket = _cpuid_skt(cpi->cpi_vendor, cpi->cpi_family,
1669            cpi->cpi_model, cpi->cpi_step);

1671 pass1_done:
1672        cpi->cpi_pass = 1;
1673 }

1675 /*
1676  * Make copies of the cpuid table entries we depend on, in
1677  * part for ease of parsing now, in part so that we have only
1678  * one place to correct any of it, in part for ease of
1679  * later export to userland, and in part so we can look at
```

```
1680  * this stuff in a crash dump.
1681  */

1683 /*ARGSUSED*/
1684 void
1685 cpuid_pass2(cpu_t *cpu)
1686 {
1687         uint_t n, nmax;
1688         int i;
1689         struct cpuid_regs *cp;
1690         uint8_t *dp;
1691         uint32_t *iptr;
1692         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

1694         ASSERT(cpi->cpi_pass == 1);

1696         if (cpi->cpi_maxeax < 1)
1697                 goto pass2_done;

1699         if ((nmax = cpi->cpi_maxeax + 1) > NMAX_CPI_STD)
1700                 nmax = NMAX_CPI_STD;
1701         /*
1702          * (We already handled n == 0 and n == 1 in pass 1)
1703          */
1704         for (n = 2, cp = &cpi->cpi_std[2]; n < nmax; n++, cp++) {
1705                 cp->cp_eax = n;

1707                 /*
1708                  * CPUID function 4 expects %ecx to be initialized
1709                  * with an index which indicates which cache to return
1710                  * information about. The OS is expected to call function 4
1711                  * with %ecx set to 0, 1, 2, ... until it returns with
1712                  * EAX[4:0] set to 0, which indicates there are no more
1713                  * caches.
1714                  *
1715                  * Here, populate cpi_std[4] with the information returned by
1716                  * function 4 when %ecx == 0, and do the rest in cpuid_pass3()
1717                  * when dynamic memory allocation becomes available.
1718                  *
1719                  * Note: we need to explicitly initialize %ecx here, since
1720                  * function 4 may have been previously invoked.
1721                  */
1722                 if (n == 4)
1723                         cp->cp_ecx = 0;

1725                 (void) __cpuid_insn(cp);
1726                 platform_cpuid_mangle(cpi->cpi_vendor, n, cp);
1727                 switch (n) {
1728                 case 2:
1729                         /*
1730                          * "the lower 8 bits of the %eax register
1731                          * contain a value that identifies the number
1732                          * of times the cpuid [instruction] has to be
1733                          * executed to obtain a complete image of the
1734                          * processor's caching systems."
1735                          *
1736                          * How *do* they make this stuff up?
1737                          */
1738                         cpi->cpi_ncache = sizeof (*cp) *
1739                             BITX(cp->cp_eax, 7, 0);
1740                         if (cpi->cpi_ncache == 0)
1741                                 break;
1742                         cpi->cpi_ncache--;      /* skip count byte */

1744                         /*
1745                          * Well, for now, rather than attempt to implement
```



```
1746                          * this slightly dubious algorithm, we just look
1747                          * at the first 15 ..
1748                          */
1749                         if (cpi->cpi_ncache > (sizeof (*cp) - 1))
1750                                 cpi->cpi_ncache = sizeof (*cp) - 1;

1752                         dp = cpi->cpi_cacheinfo;
1753                         if (BITX(cp->cp_eax, 31, 31) == 0) {
1754                                 uint8_t *p = (void *)&cp->cp_eax;
1755                                 for (i = 1; i < 4; i++)
1756                                         if (p[i] != 0)
1757                                                 *dp++ = p[i];
1758                         }
1759                         if (BITX(cp->cp_ebx, 31, 31) == 0) {
1760                                 uint8_t *p = (void *)&cp->cp_ebx;
1761                                 for (i = 0; i < 4; i++)
1762                                         if (p[i] != 0)
1763                                                 *dp++ = p[i];
1764                         }
1765                         if (BITX(cp->cp_ecx, 31, 31) == 0) {
1766                                 uint8_t *p = (void *)&cp->cp_ecx;
1767                                 for (i = 0; i < 4; i++)
1768                                         if (p[i] != 0)
1769                                                 *dp++ = p[i];
1770                         }
1771                         if (BITX(cp->cp_edx, 31, 31) == 0) {
1772                                 uint8_t *p = (void *)&cp->cp_edx;
1773                                 for (i = 0; i < 4; i++)
1774                                         if (p[i] != 0)
1775                                                 *dp++ = p[i];
1776                         }
1777                         break;

1779                 case 3: /* Processor serial number, if PSN supported */
1780                         break;

1782                 case 4: /* Deterministic cache parameters */
1783                         break;

1785                 case 5: /* Monitor/Mwait parameters */
1786                         {
1787                                 size_t mwait_size;

1789                                 /*
1790                                  * check cpi_mwait.support which was set in cpuid_pass1
1791                                  */
1792                                 if (!(cpi->cpi_mwait.support & MWAIT_SUPPORT))
1793                                         break;

1795                                 /*
1796                                  * Protect ourself from insane mwait line size.
1797                                  * Workaround for incomplete hardware emulator(s).
1798                                  */
1799                                 mwait_size = (size_t)MWAIT_SIZE_MAX(cpi);
1800                                 if (mwait_size < sizeof (uint32_t) ||
1801                                     !ISP2(mwait_size)) {
1802 #if DEBUG
1803                                         cmn_err(CE_NOTE, "Cannot handle cpu %d mwait "
1804                                             "size %ld", cpu->cpu_id, (long)mwait_size);
1805 #endif
1806                                         break;
1807                                 }

1809                                 cpi->cpi_mwait.mon_min = (size_t)MWAIT_SIZE_MIN(cpi);
1810                                 cpi->cpi_mwait.mon_max = mwait_size;
1811                                 if (MWAIT_EXTENSION(cpi)) {
```

```
1812                                cpi->cpi_mwait.support |= MWAIT_EXTENSIONS;
1813                                if (MWAIT_INT_ENABLE(cpi))
1814                                        cpi->cpi_mwait.support |=
1815                                            MWAIT_ECX_INT_ENABLE;
1816                        }
1817                        break;
1818                }
1819                default:
1820                        break;
1821                }
1822        }

1824        if (cpi->cpi_maxeax >= 0xB && cpi->cpi_vendor == X86_VENDOR_Intel) {
1825                struct cpuid_regs regs;

1827                cp = &regs;
1828                cp->cp_eax = 0xB;
1829                cp->cp_edx = cp->cp_ebx = cp->cp_ecx = 0;

1831                (void) __cpuid_insn(cp);

1833                /*
1834                 * Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero, which
1835                 * indicates that the extended topology enumeration leaf is
1836                 * available.
1837                 */
1838                if (cp->cp_ebx) {
1839                        uint32_t x2apic_id;
1840                        uint_t coreid_shift = 0;
1841                        uint_t ncpu_per_core = 1;
1842                        uint_t chipid_shift = 0;
1843                        uint_t ncpu_per_chip = 1;
1844                        uint_t i;
1845                        uint_t level;

1847                        for (i = 0; i < CPI_FNB_ECX_MAX; i++) {
1848                                cp->cp_eax = 0xB;
1849                                cp->cp_ecx = i;

1851                                (void) __cpuid_insn(cp);
1852                                level = CPI_CPU_LEVEL_TYPE(cp);

1854                                if (level == 1) {
1855                                        x2apic_id = cp->cp_edx;
1856                                        coreid_shift = BITX(cp->cp_eax, 4, 0);
1857                                        ncpu_per_core = BITX(cp->cp_ebx, 15, 0);
1858                                } else if (level == 2) {
1859                                        x2apic_id = cp->cp_edx;
1860                                        chipid_shift = BITX(cp->cp_eax, 4, 0);
1861                                        ncpu_per_chip = BITX(cp->cp_ebx, 15, 0);
1862                                }
1863                        }

1865                        cpi->cpi_apicid = x2apic_id;
1866                        cpi->cpi_ncpu_per_chip = ncpu_per_chip;
1867                        cpi->cpi_ncore_per_chip = ncpu_per_chip /
1868                            ncpu_per_core;
1869                        cpi->cpi_chipid = x2apic_id >> chipid_shift;
1870                        cpi->cpi_clogid = x2apic_id & ((1 << chipid_shift) - 1);
1871                        cpi->cpi_coreid = x2apic_id >> coreid_shift;
1872                        cpi->cpi_pkgcoreid = cpi->cpi_clogid >> coreid_shift;
1873                }

1875                /* Make cp NULL so that we don't stumble on others */
1876                cp = NULL;
1877        }
```



```
1879        /*
1880         * XSAVE enumeration
1881         */
1882        if (cpi->cpi_maxeax >= 0xD) {
1883                struct cpuid_regs regs;
1884                boolean_t cpuid_d_valid = B_TRUE;

1886                cp = &regs;
1887                cp->cp_eax = 0xD;
1888                cp->cp_edx = cp->cp_ebx = cp->cp_ecx = 0;

1890                (void) __cpuid_insn(cp);

1892                /*
1893                 * Sanity checks for debug
1894                 */
1895                if ((cp->cp_eax & XFEATURE_LEGACY_FP) == 0 ||
1896                    (cp->cp_eax & XFEATURE_SSE) == 0) {
1897                        cpuid_d_valid = B_FALSE;
1898                }

1900                cpi->cpi_xsave.xsav_hw_features_low = cp->cp_eax;
1901                cpi->cpi_xsave.xsav_hw_features_high = cp->cp_edx;
1902                cpi->cpi_xsave.xsav_max_size = cp->cp_ecx;

1904                /*
1905                 * If the hw supports AVX, get the size and offset in the save
1906                 * area for the ymm state.
1907                 */
1908                if (cpi->cpi_xsave.xsav_hw_features_low & XFEATURE_AVX) {
1909                        cp->cp_eax = 0xD;
1910                        cp->cp_ecx = 2;
1911                        cp->cp_edx = cp->cp_ebx = 0;

1913                        (void) __cpuid_insn(cp);

1915                        if (cp->cp_ebx != CPUID_LEAFD_2_YMM_OFFSET ||
1916                            cp->cp_eax != CPUID_LEAFD_2_YMM_SIZE) {
1917                                cpuid_d_valid = B_FALSE;
1918                        }

1920                        cpi->cpi_xsave.ymm_size = cp->cp_eax;
1921                        cpi->cpi_xsave.ymm_offset = cp->cp_ebx;
1922                }

1924                if (is_x86_feature(x86_featureset, X86FSET_XSAVE)) {
1925                        xsave_state_size = 0;
1926                } else if (cpuid_d_valid) {
1927                        xsave_state_size = cpi->cpi_xsave.xsav_max_size;
1928                } else {
1929                        /* Broken CPUID 0xD, probably in HVM */
1930                        cmn_err(CE_WARN, "cpu%d: CPUID.0xD returns invalid "
1931                            "value: hw_low = %d, hw_high = %d, xsave_size = %d"
1932                            ", ymm_size = %d, ymm_offset = %d\n",
1933                            cpu->cpu_id, cpi->cpi_xsave.xsav_hw_features_low,
1934                            cpi->cpi_xsave.xsav_hw_features_high,
1935                            (int)cpi->cpi_xsave.xsav_max_size,
1936                            (int)cpi->cpi_xsave.ymm_size,
1937                            (int)cpi->cpi_xsave.ymm_offset);

1939                        if (xsave_state_size != 0) {
1940                                /*
1941                                 * This must be a non-boot CPU. We cannot
1942                                 * continue, because boot cpu has already
1943                                 * enabled XSAVE.
```

```
1944                                 */
1945                                ASSERT(cpu->cpu_id != 0);
1946                                cmn_err(CE_PANIC, "cpu%d: we have already "
1947                                    "enabled XSAVE on boot cpu, cannot "
1948                                    "continue.", cpu->cpu_id);
1949                        } else {
1950                                /*
1951                                 * Must be from boot CPU, OK to disable XSAVE.
1952                                 */
1953                                ASSERT(cpu->cpu_id == 0);
1954                                remove_x86_feature(x86_featureset,
1955                                    X86FSET_XSAVE);
1956                                remove_x86_feature(x86_featureset, X86FSET_AVX);
1957                                CPI_FEATURES_ECX(cpi) &= ~CPUID_INTC_ECX_XSAVE;
1958                                CPI_FEATURES_ECX(cpi) &= ~CPUID_INTC_ECX_AVX;
1959                                CPI_FEATURES_ECX(cpi) &= ~CPUID_INTC_ECX_F16C;
1960                                xsave_force_disable = B_TRUE;
1961                        }
1962                }
1963        }


1966        if ((cpi->cpi_xmaxeax & 0x80000000) == 0)
1967                goto pass2_done;

1969        if ((nmax = cpi->cpi_xmaxeax - 0x80000000 + 1) > NMAX_CPI_EXTD)
1970                nmax = NMAX_CPI_EXTD;
1971        /*
1972         * Copy the extended properties, fixing them as we go.
1973         * (We already handled n == 0 and n == 1 in pass 1)
1974         */
1975        iptr = (void *)cpi->cpi_brandstr;
1976        for (n = 2, cp = &cpi->cpi_extd[2]; n < nmax; cp++, n++) {
1977                cp->cp_eax = 0x80000000 + n;
1978                (void) __cpuid_insn(cp);
1979                platform_cpuid_mangle(cpi->cpi_vendor, 0x80000000 + n, cp);
1980                switch (n) {
1981                case 2:
1982                case 3:
1983                case 4:
1984                        /*
1985                         * Extract the brand string
1986                         */
1987                        *iptr++ = cp->cp_eax;
1988                        *iptr++ = cp->cp_ebx;
1989                        *iptr++ = cp->cp_ecx;
1990                        *iptr++ = cp->cp_edx;
1991                        break;
1992                case 5:
1993                        switch (cpi->cpi_vendor) {
1994                        case X86_VENDOR_AMD:
1995                                /*
1996                                 * The Athlon and Duron were the first
1997                                 * parts to report the sizes of the
1998                                 * TLB for large pages. Before then,
1999                                 * we don't trust the data.
2000                                 */
2001                                if (cpi->cpi_family < 6 ||
2002                                    (cpi->cpi_family == 6 &&
2003                                    cpi->cpi_model < 1))
2004                                        cp->cp_eax = 0;
2005                                break;
2006                        default:
2007                                break;
2008                        }
2009                        break;
```



```
2010                case 6:
2011                        switch (cpi->cpi_vendor) {
2012                        case X86_VENDOR_AMD:
2013                                /*
2014                                 * The Athlon and Duron were the first
2015                                 * AMD parts with L2 TLB's.
2016                                 * Before then, don't trust the data.
2017                                 */
2018                                if (cpi->cpi_family < 6 ||
2019                                    cpi->cpi_family == 6 &&
2020                                    cpi->cpi_model < 1)
2021                                        cp->cp_eax = cp->cp_ebx = 0;
2022                                /*
2023                                 * AMD Duron rev A0 reports L2
2024                                 * cache size incorrectly as 1K
2025                                 * when it is really 64K
2026                                 */
2027                                if (cpi->cpi_family == 6 &&
2028                                    cpi->cpi_model == 3 &&
2029                                    cpi->cpi_step == 0) {
2030                                        cp->cp_ecx &= 0xffff;
2031                                        cp->cp_ecx |= 0x400000;
2032                                }
2033                                break;
2034                        case X86_VENDOR_Cyrix:  /* VIA C3 */
2035                                /*
2036                                 * VIA C3 processors are a bit messed
2037                                 * up w.r.t. encoding cache sizes in %ecx
2038                                 */
2039                                if (cpi->cpi_family != 6)
2040                                        break;
2041                                /*
2042                                 * model 7 and 8 were incorrectly encoded
2043                                 *
2044                                 * xxx is model 8 really broken?
2045                                 */
2046                                if (cpi->cpi_model == 7 ||
2047                                    cpi->cpi_model == 8)
2048                                        cp->cp_ecx =
2049                                            BITX(cp->cp_ecx, 31, 24) << 16 |
2050                                            BITX(cp->cp_ecx, 23, 16) << 12 |
2051                                            BITX(cp->cp_ecx, 15, 8) << 8 |
2052                                            BITX(cp->cp_ecx, 7, 0);
2053                                /*
2054                                 * model 9 stepping 1 has wrong associativity
2055                                 */
2056                                if (cpi->cpi_model == 9 && cpi->cpi_step == 1)
2057                                        cp->cp_ecx |= 8 << 12;
2058                                break;
2059                        case X86_VENDOR_Intel:
2060                                /*
2061                                 * Extended L2 Cache features function.
2062                                 * First appeared on Prescott.
2063                                 */
2064                        default:
2065                                break;
2066                        }
2067                        break;
2068                default:
2069                        break;
2070                }
2071        }

2073 pass2_done:
2074        cpi->cpi_pass = 2;
2075 }
```

```
2077 static const char *
2078 intel_cpubrand(const struct cpuid_info *cpi)
2079 {
2080         int i;

2082         if (!is_x86_feature(x86_featureset, X86FSET_CPUID) ||
2083             cpi->cpi_maxeax < 1 || cpi->cpi_family < 5)
2084                 return ("i486");

2086         switch (cpi->cpi_family) {
2087         case 5:
2088                 return ("Intel Pentium(r)");
2089         case 6:
2090                 switch (cpi->cpi_model) {
2091                         uint_t celeron, xeon;
2092                         const struct cpuid_regs *cp;
2093                 case 0:
2094                 case 1:
2095                 case 2:
2096                         return ("Intel Pentium(r) Pro");
2097                 case 3:
2098                 case 4:
2099                         return ("Intel Pentium(r) II");
2100                 case 6:
2101                         return ("Intel Celeron(r)");
2102                 case 5:
2103                 case 7:
2104                         celeron = xeon = 0;
2105                         cp = &cpi->cpi_std[2];  /* cache info */

2107                         for (i = 1; i < 4; i++) {
2108                                 uint_t tmp;

2110                                 tmp = (cp->cp_eax >> (8 * i)) & 0xff;
2111                                 if (tmp == 0x40)
2112                                         celeron++;
2113                                 if (tmp >= 0x44 && tmp <= 0x45)
2114                                         xeon++;
2115                         }

2117                         for (i = 0; i < 2; i++) {
2118                                 uint_t tmp;

2120                                 tmp = (cp->cp_ebx >> (8 * i)) & 0xff;
2121                                 if (tmp == 0x40)
2122                                         celeron++;
2123                                 else if (tmp >= 0x44 && tmp <= 0x45)
2124                                         xeon++;
2125                         }

2127                         for (i = 0; i < 4; i++) {
2128                                 uint_t tmp;

2130                                 tmp = (cp->cp_ecx >> (8 * i)) & 0xff;
2131                                 if (tmp == 0x40)
2132                                         celeron++;
2133                                 else if (tmp >= 0x44 && tmp <= 0x45)
2134                                         xeon++;
2135                         }

2137                         for (i = 0; i < 4; i++) {
2138                                 uint_t tmp;

2140                                 tmp = (cp->cp_edx >> (8 * i)) & 0xff;
2141                                 if (tmp == 0x40)
```



```
2142                                         celeron++;
2143                                 else if (tmp >= 0x44 && tmp <= 0x45)
2144                                         xeon++;
2145                         }

2147                         if (celeron)
2148                                 return ("Intel Celeron(r)");
2149                         if (xeon)
2150                                 return (cpi->cpi_model == 5 ?
2151                                     "Intel Pentium(r) II Xeon(tm)" :
2152                                     "Intel Pentium(r) III Xeon(tm)");
2153                         return (cpi->cpi_model == 5 ?
2154                             "Intel Pentium(r) II or Pentium(r) II Xeon(tm)" :
2155                             "Intel Pentium(r) III or Pentium(r) III Xeon(tm)");
2156                 default:
2157                         break;
2158                 }
2159         default:
2160                 break;
2161         }

2163         /* BrandID is present if the field is nonzero */
2164         if (cpi->cpi_brandid != 0) {
2165                 static const struct {
2166                         uint_t bt_bid;
2167                         const char *bt_str;
2168                 } brand_tbl[] = {
2169                         { 0x1,  "Intel(r) Celeron(r)" },
2170                         { 0x2,  "Intel(r) Pentium(r) III" },
2171                         { 0x3,  "Intel(r) Pentium(r) III Xeon(tm)" },
2172                         { 0x4,  "Intel(r) Pentium(r) III" },
2173                         { 0x6,  "Mobile Intel(r) Pentium(r) III" },
2174                         { 0x7,  "Mobile Intel(r) Celeron(r)" },
2175                         { 0x8,  "Intel(r) Pentium(r) 4" },
2176                         { 0x9,  "Intel(r) Pentium(r) 4" },
2177                         { 0xa,  "Intel(r) Celeron(r)" },
2178                         { 0xb,  "Intel(r) Xeon(tm)" },
2179                         { 0xc,  "Intel(r) Xeon(tm) MP" },
2180                         { 0xe,  "Mobile Intel(r) Pentium(r) 4" },
2181                         { 0xf,  "Mobile Intel(r) Celeron(r)" },
2182                         { 0x11, "Mobile Genuine Intel(r)" },
2183                         { 0x12, "Intel(r) Celeron(r) M" },
2184                         { 0x13, "Mobile Intel(r) Celeron(r)" },
2185                         { 0x14, "Intel(r) Celeron(r)" },
2186                         { 0x15, "Mobile Genuine Intel(r)" },
2187                         { 0x16, "Intel(r) Pentium(r) M" },
2188                         { 0x17, "Mobile Intel(r) Celeron(r)" }
2189                 };
2190                 uint_t btblmax = sizeof (brand_tbl) / sizeof (brand_tbl[0]);
2191                 uint_t sgn;

2193                 sgn = (cpi->cpi_family << 8) |
2194                     (cpi->cpi_model << 4) | cpi->cpi_step;

2196                 for (i = 0; i < btblmax; i++)
2197                         if (brand_tbl[i].bt_bid == cpi->cpi_brandid)
2198                                 break;
2199                 if (i < btblmax) {
2200                         if (sgn == 0x6b1 && cpi->cpi_brandid == 3)
2201                                 return ("Intel(r) Celeron(r)");
2202                         if (sgn < 0xf13 && cpi->cpi_brandid == 0xb)
2203                                 return ("Intel(r) Xeon(tm) MP");
2204                         if (sgn < 0xf13 && cpi->cpi_brandid == 0xe)
2205                                 return ("Intel(r) Xeon(tm)");
2206                         return (brand_tbl[i].bt_str);
2207                 }
```

```
2208         }

2210         return (NULL);
2211 }

2213 static const char *
2214 amd_cpubrand(const struct cpuid_info *cpi)
2215 {
2216         if (!is_x86_feature(x86_featureset, X86FSET_CPUID) ||
2217             cpi->cpi_maxeax < 1 || cpi->cpi_family < 5)
2218                 return ("i486 compatible");

2220         switch (cpi->cpi_family) {
2221         case 5:
2222                 switch (cpi->cpi_model) {
2223                 case 0:
2224                 case 1:
2225                 case 2:
2226                 case 3:
2227                 case 4:
2228                 case 5:
2229                         return ("AMD-K5(r)");
2230                 case 6:
2231                 case 7:
2232                         return ("AMD-K6(r)");
2233                 case 8:
2234                         return ("AMD-K6(r)-2");
2235                 case 9:
2236                         return ("AMD-K6(r)-III");
2237                 default:
2238                         return ("AMD (family 5)");
2239                 }
2240         case 6:
2241                 switch (cpi->cpi_model) {
2242                 case 1:
2243                         return ("AMD-K7(tm)");
2244                 case 0:
2245                 case 2:
2246                 case 4:
2247                         return ("AMD Athlon(tm)");
2248                 case 3:
2249                 case 7:
2250                         return ("AMD Duron(tm)");
2251                 case 6:
2252                 case 8:
2253                 case 10:
2254                         /*
2255                          * Use the L2 cache size to distinguish
2256                          */
2257                         return ((cpi->cpi_extd[6].cp_ecx >> 16) >= 256 ?
2258                             "AMD Athlon(tm)" : "AMD Duron(tm)");
2259                 default:
2260                         return ("AMD (family 6)");
2261                 }
2262         default:
2263                 break;
2264         }

2266         if (cpi->cpi_family == 0xf && cpi->cpi_model == 5 &&
2267             cpi->cpi_brandid != 0) {
2268                 switch (BITX(cpi->cpi_brandid, 7, 5)) {
2269                 case 3:
2270                         return ("AMD Opteron(tm) UP 1xx");
2271                 case 4:
2272                         return ("AMD Opteron(tm) DP 2xx");
2273                 case 5:
```



```
2274                         return ("AMD Opteron(tm) MP 8xx");
2275                 default:
2276                         return ("AMD Opteron(tm)");
2277                 }
2278         }

2280         return (NULL);
2281 }

2283 static const char *
2284 cyrix_cpubrand(struct cpuid_info *cpi, uint_t type)
2285 {
2286         if (!is_x86_feature(x86_featureset, X86FSET_CPUID) ||
2287             cpi->cpi_maxeax < 1 || cpi->cpi_family < 5 ||
2288             type == X86_TYPE_CYRIX_486)
2289                 return ("i486 compatible");

2291         switch (type) {
2292         case X86_TYPE_CYRIX_6x86:
2293                 return ("Cyrix 6x86");
2294         case X86_TYPE_CYRIX_6x86L:
2295                 return ("Cyrix 6x86L");
2296         case X86_TYPE_CYRIX_6x86MX:
2297                 return ("Cyrix 6x86MX");
2298         case X86_TYPE_CYRIX_GXm:
2299                 return ("Cyrix GXm");
2300         case X86_TYPE_CYRIX_MediaGX:
2301                 return ("Cyrix MediaGX");
2302         case X86_TYPE_CYRIX_MII:
2303                 return ("Cyrix M2");
2304         case X86_TYPE_VIA_CYRIX_III:
2305                 return ("VIA Cyrix M3");
2306         default:
2307                 /*
2308                  * Have another wild guess ..
2309                  */
2310                 if (cpi->cpi_family == 4 && cpi->cpi_model == 9)
2311                         return ("Cyrix 5x86");
2312                 else if (cpi->cpi_family == 5) {
2313                         switch (cpi->cpi_model) {
2314                         case 2:
2315                                 return ("Cyrix 6x86");  /* Cyrix M1 */
2316                         case 4:
2317                                 return ("Cyrix MediaGX");
2318                         default:
2319                                 break;
2320                         }
2321                 } else if (cpi->cpi_family == 6) {
2322                         switch (cpi->cpi_model) {
2323                         case 0:
2324                                 return ("Cyrix 6x86MX"); /* Cyrix M2? */
2325                         case 5:
2326                         case 6:
2327                         case 7:
2328                         case 8:
2329                         case 9:
2330                                 return ("VIA C3");
2331                         default:
2332                                 break;
2333                         }
2334                 }
2335                 break;
2336         }
2337         return (NULL);
2338 }
```

```
2340 /*
2341  * This only gets called in the case that the CPU extended
2342  * feature brand string (0x80000002, 0x80000003, 0x80000004)
2343  * aren't available, or contain null bytes for some reason.
2344  */
2345 static void
2346 fabricate_brandstr(struct cpuid_info *cpi)
2347 {
2348         const char *brand = NULL;

2350         switch (cpi->cpi_vendor) {
2351         case X86_VENDOR_Intel:
2352                 brand = intel_cpubrand(cpi);
2353                 break;
2354         case X86_VENDOR_AMD:
2355                 brand = amd_cpubrand(cpi);
2356                 break;
2357         case X86_VENDOR_Cyrix:
2358                 brand = cyrix_cpubrand(cpi, x86_type);
2359                 break;
2360         case X86_VENDOR_NexGen:
2361                 if (cpi->cpi_family == 5 && cpi->cpi_model == 0)
2362                         brand = "NexGen Nx586";
2363                 break;
2364         case X86_VENDOR_Centaur:
2365                 if (cpi->cpi_family == 5)
2366                         switch (cpi->cpi_model) {
2367                         case 4:
2368                                 brand = "Centaur C6";
2369                                 break;
2370                         case 8:
2371                                 brand = "Centaur C2";
2372                                 break;
2373                         case 9:
2374                                 brand = "Centaur C3";
2375                                 break;
2376                         default:
2377                                 break;
2378                         }
2379                 break;
2380         case X86_VENDOR_Rise:
2381                 if (cpi->cpi_family == 5 &&
2382                     (cpi->cpi_model == 0 || cpi->cpi_model == 2))
2383                         brand = "Rise mP6";
2384                 break;
2385         case X86_VENDOR_SiS:
2386                 if (cpi->cpi_family == 5 && cpi->cpi_model == 0)
2387                         brand = "SiS 55x";
2388                 break;
2389         case X86_VENDOR_TM:
2390                 if (cpi->cpi_family == 5 && cpi->cpi_model == 4)
2391                         brand = "Transmeta Crusoe TM3x00 or TM5x00";
2392                 break;
2393         case X86_VENDOR_NSC:
2394         case X86_VENDOR_UMC:
2395         default:
2396                 break;
2397         }
2398         if (brand) {
2399                 (void) strcpy((char *)cpi->cpi_brandstr, brand);
2400                 return;
2401         }

2403         /*
2404          * If all else fails ...
2405          */
```



```
2406         (void) snprintf(cpi->cpi_brandstr, sizeof (cpi->cpi_brandstr),
2407             "%s %d.%d.%d", cpi->cpi_vendorstr, cpi->cpi_family,
2408             cpi->cpi_model, cpi->cpi_step);
2409 }

2411 /*
2412  * This routine is called just after kernel memory allocation
2413  * becomes available on cpu0, and as part of mp_startup() on
2414  * the other cpus.
2415  *
2416  * Fixup the brand string, and collect any information from cpuid
2417  * that requires dynamically allocated storage to represent.
2418  */
2419 /*ARGSUSED*/
2420 void
2421 cpuid_pass3(cpu_t *cpu)
2422 {
2423         int     i, max, shft, level, size;
2424         struct cpuid_regs regs;
2425         struct cpuid_regs *cp;
2426         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

2428         ASSERT(cpi->cpi_pass == 2);

2430         /*
2431          * Function 4: Deterministic cache parameters
2432          *
2433          * Take this opportunity to detect the number of threads
2434          * sharing the last level cache, and construct a corresponding
2435          * cache id. The respective cpuid_info members are initialized
2436          * to the default case of "no last level cache sharing".
2437          */
2438         cpi->cpi_ncpu_shr_last_cache = 1;
2439         cpi->cpi_last_lvl_cacheid = cpu->cpu_id;

2441         if (cpi->cpi_maxeax >= 4 && cpi->cpi_vendor == X86_VENDOR_Intel) {

2443                 /*
2444                  * Find the # of elements (size) returned by fn 4, and along
2445                  * the way detect last level cache sharing details.
2446                  */
2447                 bzero(&regs, sizeof (regs));
2448                 cp = &regs;
2449                 for (i = 0, max = 0; i < CPI_FN4_ECX_MAX; i++) {
2450                         cp->cp_eax = 4;
2451                         cp->cp_ecx = i;

2453                         (void) __cpuid_insn(cp);

2455                         if (CPI_CACHE_TYPE(cp) == 0)
2456                                 break;
2457                         level = CPI_CACHE_LVL(cp);
2458                         if (level > max) {
2459                                 max = level;
2460                                 cpi->cpi_ncpu_shr_last_cache =
2461                                     CPI_NTHR_SHR_CACHE(cp) + 1;
2462                         }
2463                 }
2464                 cpi->cpi_std_4_size = size = i;

2466                 /*
2467                  * Allocate the cpi_std_4 array. The first element
2468                  * references the regs for fn 4, %ecx == 0, which
2469                  * cpuid_pass2() stashed in cpi->cpi_std[4].
2470                  */
2471                 if (size > 0) {
```

```
2472                        cpi->cpi_std_4 =
2473                            kmem_alloc(size * sizeof (cp), KM_SLEEP);
2474                        cpi->cpi_std_4[0] = &cpi->cpi_std[4];

2476                        /*
2477                         * Allocate storage to hold the additional regs
2478                         * for function 4, %ecx == 1 .. cpi_std_4_size.
2479                         *
2480                         * The regs for fn 4, %ecx == 0 has already
2481                         * been allocated as indicated above.
2482                         */
2483                        for (i = 1; i < size; i++) {
2484                                cp = cpi->cpi_std_4[i] =
2485                                    kmem_zalloc(sizeof (regs), KM_SLEEP);
2486                                cp->cp_eax = 4;
2487                                cp->cp_ecx = i;

2489                                (void) __cpuid_insn(cp);
2490                        }
2491                }
2492                /*
2493                 * Determine the number of bits needed to represent
2494                 * the number of CPUs sharing the last level cache.
2495                 *
2496                 * Shift off that number of bits from the APIC id to
2497                 * derive the cache id.
2498                 */
2499                shft = 0;
2500                for (i = 1; i < cpi->cpi_ncpu_shr_last_cache; i <<= 1)
2501                        shft++;
2502                cpi->cpi_last_lvl_cacheid = cpi->cpi_apicid >> shft;
2503        }

2505        /*
2506         * Now fixup the brand string
2507         */
2508        if ((cpi->cpi_xmaxeax & 0x80000000) == 0) {
2509                fabricate_brandstr(cpi);
2510        } else {

2512                /*
2513                 * If we successfully extracted a brand string from the cpuid
2514                 * instruction, clean it up by removing leading spaces and
2515                 * similar junk.
2516                 */
2517                if (cpi->cpi_brandstr[0]) {
2518                        size_t maxlen = sizeof (cpi->cpi_brandstr);
2519                        char *src, *dst;

2521                        dst = src = (char *)cpi->cpi_brandstr;
2522                        src[maxlen - 1] = '\0';
2523                        /*
2524                         * strip leading spaces
2525                         */
2526                        while (*src == ' ')
2527                                src++;
2528                        /*
2529                         * Remove any 'Genuine' or "Authentic" prefixes
2530                         */
2531                        if (strncmp(src, "Genuine ", 8) == 0)
2532                                src += 8;
2533                        if (strncmp(src, "Authentic ", 10) == 0)
2534                                src += 10;

2536                        /*
2537                         * Now do an in-place copy.
```



```
2538                         * Map (R) to (r) and (TM) to (tm).
2539                         * The era of teletypes is long gone, and there's
2540                         * -really- no need to shout.
2541                         */
2542                        while (*src != '\0') {
2543                                if (src[0] == '(') {
2544                                        if (strncmp(src + 1, "R)", 2) == 0) {
2545                                                (void) strncpy(dst, "(r)", 3);
2546                                                src += 3;
2547                                                dst += 3;
2548                                                continue;
2549                                        }
2550                                        if (strncmp(src + 1, "TM)", 3) == 0) {
2551                                                (void) strncpy(dst, "(tm)", 4);
2552                                                src += 4;
2553                                                dst += 4;
2554                                                continue;
2555                                        }
2556                                }
2557                                *dst++ = *src++;
2558                        }
2559                        *dst = '\0';

2561                        /*
2562                         * Finally, remove any trailing spaces
2563                         */
2564                        while (--dst > cpi->cpi_brandstr)
2565                                if (*dst == ' ')
2566                                        *dst = '\0';
2567                                else
2568                                        break;
2569                } else
2570                        fabricate_brandstr(cpi);
2571        }
2572        cpi->cpi_pass = 3;
2573 }

2575 /*
2576  * This routine is called out of bind_hwcap() much later in the life
2577  * of the kernel (post_startup()).  The job of this routine is to resolve
2578  * the hardware feature support and kernel support for those features into
2579  * what we're actually going to tell applications via the aux vector.
2580  */
2581 void
2582 cpuid_pass4(cpu_t *cpu, uint_t *hwcap_out)
2583 {
2584        struct cpuid_info *cpi;
2585        uint_t hwcap_flags = 0, hwcap_flags_2 = 0;

2587        if (cpu == NULL)
2588                cpu = CPU;
2589        cpi = cpu->cpu_m.mcpu_cpi;

2591        ASSERT(cpi->cpi_pass == 3);

2593        if (cpi->cpi_maxeax >= 1) {
2594                uint32_t *edx = &cpi->cpi_support[STD_EDX_FEATURES];
2595                uint32_t *ecx = &cpi->cpi_support[STD_ECX_FEATURES];

2597                *edx = CPI_FEATURES_EDX(cpi);
2598                *ecx = CPI_FEATURES_ECX(cpi);

2600                /*
2601                 * [these require explicit kernel support]
2602                 */
2603                if (!is_x86_feature(x86_featureset, X86FSET_SEP))
```

```
2604                        *edx &= ~CPUID_INTC_EDX_SEP;

2606                if (!is_x86_feature(x86_featureset, X86FSET_SSE))
2607                        *edx &= ~(CPUID_INTC_EDX_FXSR|CPUID_INTC_EDX_SSE);
2608                if (!is_x86_feature(x86_featureset, X86FSET_SSE2))
2609                        *edx &= ~CPUID_INTC_EDX_SSE2;

2611                if (!is_x86_feature(x86_featureset, X86FSET_HTT))
2612                        *edx &= ~CPUID_INTC_EDX_HTT;

2614                if (!is_x86_feature(x86_featureset, X86FSET_SSE3))
2615                        *ecx &= ~CPUID_INTC_ECX_SSE3;

2617                if (!is_x86_feature(x86_featureset, X86FSET_SSSE3))
2618                        *ecx &= ~CPUID_INTC_ECX_SSSE3;
2619                if (!is_x86_feature(x86_featureset, X86FSET_SSE4_1))
2620                        *ecx &= ~CPUID_INTC_ECX_SSE4_1;
2621                if (!is_x86_feature(x86_featureset, X86FSET_SSE4_2))
2622                        *ecx &= ~CPUID_INTC_ECX_SSE4_2;
2623                if (!is_x86_feature(x86_featureset, X86FSET_AES))
2624                        *ecx &= ~CPUID_INTC_ECX_AES;
2625                if (!is_x86_feature(x86_featureset, X86FSET_PCLMULQDQ))
2626                        *ecx &= ~CPUID_INTC_ECX_PCLMULQDQ;
2627                if (!is_x86_feature(x86_featureset, X86FSET_XSAVE))
2628                        *ecx &= ~(CPUID_INTC_ECX_XSAVE |
2629                            CPUID_INTC_ECX_OSXSAVE);
2630                if (!is_x86_feature(x86_featureset, X86FSET_AVX))
2631                        *ecx &= ~CPUID_INTC_ECX_AVX;
2632                if (!is_x86_feature(x86_featureset, X86FSET_F16C))
2633                        *ecx &= ~CPUID_INTC_ECX_F16C;

2635                /*
2636                 * [no explicit support required beyond x87 fp context]
2637                 */
2638                if (!fpu_exists)
2639                        *edx &= ~(CPUID_INTC_EDX_FPU | CPUID_INTC_EDX_MMX);

2641                /*
2642                 * Now map the supported feature vector to things that we
2643                 * think userland will care about.
2644                 */
2645                if (*edx & CPUID_INTC_EDX_SEP)
2646                        hwcap_flags |= AV_386_SEP;
2647                if (*edx & CPUID_INTC_EDX_SSE)
2648                        hwcap_flags |= AV_386_FXSR | AV_386_SSE;
2649                if (*edx & CPUID_INTC_EDX_SSE2)
2650                        hwcap_flags |= AV_386_SSE2;
2651                if (*ecx & CPUID_INTC_ECX_SSE3)
2652                        hwcap_flags |= AV_386_SSE3;
2653                if (*ecx & CPUID_INTC_ECX_SSSE3)
2654                        hwcap_flags |= AV_386_SSSE3;
2655                if (*ecx & CPUID_INTC_ECX_SSE4_1)
2656                        hwcap_flags |= AV_386_SSE4_1;
2657                if (*ecx & CPUID_INTC_ECX_SSE4_2)
2658                        hwcap_flags |= AV_386_SSE4_2;
2659                if (*ecx & CPUID_INTC_ECX_MOVBE)
2660                        hwcap_flags |= AV_386_MOVBE;
2661                if (*ecx & CPUID_INTC_ECX_AES)
2662                        hwcap_flags |= AV_386_AES;
2663                if (*ecx & CPUID_INTC_ECX_PCLMULQDQ)
2664                        hwcap_flags |= AV_386_PCLMULQDQ;
2665                if ((*ecx & CPUID_INTC_ECX_XSAVE) &&
2666                    (*ecx & CPUID_INTC_ECX_OSXSAVE)) {
2667                        hwcap_flags |= AV_386_XSAVE;

2669                        if (*ecx & CPUID_INTC_ECX_AVX) {
```



```
2670                                hwcap_flags |= AV_386_AVX;
2671                                if (*ecx & CPUID_INTC_ECX_F16C)
2672                                        hwcap_flags_2 |= AV_386_2_F16C;
2673                        }
2674                }
2675                if (*ecx & CPUID_INTC_ECX_VMX)
2676                        hwcap_flags |= AV_386_VMX;
2677                if (*ecx & CPUID_INTC_ECX_POPCNT)
2678                        hwcap_flags |= AV_386_POPCNT;
2679                if (*edx & CPUID_INTC_EDX_FPU)
2680                        hwcap_flags |= AV_386_FPU;
2681                if (*edx & CPUID_INTC_EDX_MMX)
2682                        hwcap_flags |= AV_386_MMX;

2684                if (*edx & CPUID_INTC_EDX_TSC)
2685                        hwcap_flags |= AV_386_TSC;
2686                if (*edx & CPUID_INTC_EDX_CX8)
2687                        hwcap_flags |= AV_386_CX8;
2688                if (*edx & CPUID_INTC_EDX_CMOV)
2689                        hwcap_flags |= AV_386_CMOV;
2690                if (*ecx & CPUID_INTC_ECX_CX16)
2691                        hwcap_flags |= AV_386_CX16;

2693                if (*ecx & CPUID_INTC_ECX_RDRAND)
2694                        hwcap_flags_2 |= AV_386_2_RDRAND;
2695        }

2697        if (cpi->cpi_xmaxeax < 0x80000001)
2698                goto pass4_done;

2700        switch (cpi->cpi_vendor) {
2701                struct cpuid_regs cp;
2702                uint32_t *edx, *ecx;

2704        case X86_VENDOR_Intel:
2705                /*
2706                 * Seems like Intel duplicated what we necessary
2707                 * here to make the initial crop of 64-bit OS's work.
2708                 * Hopefully, those are the only "extended" bits
2709                 * they'll add.
2710                 */
2711                /*FALLTHROUGH*/

2713        case X86_VENDOR_AMD:
2714                edx = &cpi->cpi_support[AMD_EDX_FEATURES];
2715                ecx = &cpi->cpi_support[AMD_ECX_FEATURES];

2717                *edx = CPI_FEATURES_XTD_EDX(cpi);
2718                *ecx = CPI_FEATURES_XTD_ECX(cpi);

2720                /*
2721                 * [these features require explicit kernel support]
2722                 */
2723                switch (cpi->cpi_vendor) {
2724                case X86_VENDOR_Intel:
2725                        if (!is_x86_feature(x86_featureset, X86FSET_TSCP))
2726                                *edx &= ~CPUID_AMD_EDX_TSCP;
2727                        break;

2729                case X86_VENDOR_AMD:
2730                        if (!is_x86_feature(x86_featureset, X86FSET_TSCP))
2731                                *edx &= ~CPUID_AMD_EDX_TSCP;
2732                        if (!is_x86_feature(x86_featureset, X86FSET_SSE4A))
2733                                *ecx &= ~CPUID_AMD_ECX_SSE4A;
2734                        break;
```

```
2736                 default:
2737                         break;
2738                 }

2740                 /*
2741                  * [no explicit support required beyond
2742                  * x87 fp context and exception handlers]
2743                  */
2744                 if (!fpu_exists)
2745                         *edx &= ~(CPUID_AMD_EDX_MMXamd |
2746                             CPUID_AMD_EDX_3DNow | CPUID_AMD_EDX_3DNowx);

2748                 if (!is_x86_feature(x86_featureset, X86FSET_NX))
2749                         *edx &= ~CPUID_AMD_EDX_NX;
2750 #if !defined(__amd64)
2751                 *edx &= ~CPUID_AMD_EDX_LM;
2752 #endif
2753                 /*
2754                  * Now map the supported feature vector to
2755                  * things that we think userland will care about.
2756                  */
2757 #if defined(__amd64)
2758                 if (*edx & CPUID_AMD_EDX_SYSC)
2759                         hwcap_flags |= AV_386_AMD_SYSC;
2760 #endif
2761                 if (*edx & CPUID_AMD_EDX_MMXamd)
2762                         hwcap_flags |= AV_386_AMD_MMX;
2763                 if (*edx & CPUID_AMD_EDX_3DNow)
2764                         hwcap_flags |= AV_386_AMD_3DNow;
2765                 if (*edx & CPUID_AMD_EDX_3DNowx)
2766                         hwcap_flags |= AV_386_AMD_3DNowx;
2767                 if (*ecx & CPUID_AMD_ECX_SVM)
2768                         hwcap_flags |= AV_386_AMD_SVM;

2770                 switch (cpi->cpi_vendor) {
2771                 case X86_VENDOR_AMD:
2772                         if (*edx & CPUID_AMD_EDX_TSCP)
2773                                 hwcap_flags |= AV_386_TSCP;
2774                         if (*ecx & CPUID_AMD_ECX_AHF64)
2775                                 hwcap_flags |= AV_386_AHF;
2776                         if (*ecx & CPUID_AMD_ECX_SSE4A)
2777                                 hwcap_flags |= AV_386_AMD_SSE4A;
2778                         if (*ecx & CPUID_AMD_ECX_LZCNT)
2779                                 hwcap_flags |= AV_386_AMD_LZCNT;
2780                         break;

2782                 case X86_VENDOR_Intel:
2783                         if (*edx & CPUID_AMD_EDX_TSCP)
2784                                 hwcap_flags |= AV_386_TSCP;
2785                         /*
2786                          * Aarrgh.
2787                          * Intel uses a different bit in the same word.
2788                          */
2789                         if (*ecx & CPUID_INTC_ECX_AHF64)
2790                                 hwcap_flags |= AV_386_AHF;
2791                         break;

2793                 default:
2794                         break;
2795                 }
2796                 break;

2798         case X86_VENDOR_TM:
2799                 cp.cp_eax = 0x80860001;
2800                 (void) __cpuid_insn(&cp);
2801                 cpi->cpi_support[TM_EDX_FEATURES] = cp.cp_edx;
```



```
2802                 break;

2804         default:
2805                 break;
2806         }

2808 pass4_done:
2809         cpi->cpi_pass = 4;
2810         if (hwcap_out != NULL) {
2811                 hwcap_out[0] = hwcap_flags;
2812                 hwcap_out[1] = hwcap_flags_2;
2813         }
2814 }


2817 /*
2818  * Simulate the cpuid instruction using the data we previously
2819  * captured about this CPU.  We try our best to return the truth
2820  * about the hardware, independently of kernel support.
2821  */
2822 uint32_t
2823 cpuid_insn(cpu_t *cpu, struct cpuid_regs *cp)
2824 {
2825         struct cpuid_info *cpi;
2826         struct cpuid_regs *xcp;

2828         if (cpu == NULL)
2829                 cpu = CPU;
2830         cpi = cpu->cpu_m.mcpu_cpi;

2832         ASSERT(cpuid_checkpass(cpu, 3));

2834         /*
2835          * CPUID data is cached in two separate places: cpi_std for standard
2836          * CPUID functions, and cpi_extd for extended CPUID functions.
2837          */
2838         if (cp->cp_eax <= cpi->cpi_maxeax && cp->cp_eax < NMAX_CPI_STD)
2839                 xcp = &cpi->cpi_std[cp->cp_eax];
2840         else if (cp->cp_eax >= 0x80000000 && cp->cp_eax <= cpi->cpi_xmaxeax &&
2841             cp->cp_eax < 0x80000000 + NMAX_CPI_EXTD)
2842                 xcp = &cpi->cpi_extd[cp->cp_eax - 0x80000000];
2843         else
2844                 /*
2845                  * The caller is asking for data from an input parameter which
2846                  * the kernel has not cached.  In this case we go fetch from
2847                  * the hardware and return the data directly to the user.
2848                  */
2849                 return (__cpuid_insn(cp));

2851         cp->cp_eax = xcp->cp_eax;
2852         cp->cp_ebx = xcp->cp_ebx;
2853         cp->cp_ecx = xcp->cp_ecx;
2854         cp->cp_edx = xcp->cp_edx;
2855         return (cp->cp_eax);
2856 }

2858 int
2859 cpuid_checkpass(cpu_t *cpu, int pass)
2860 {
2861         return (cpu != NULL && cpu->cpu_m.mcpu_cpi != NULL &&
2862             cpu->cpu_m.mcpu_cpi->cpi_pass >= pass);
2863 }

2865 int
2866 cpuid_getbrandstr(cpu_t *cpu, char *s, size_t n)
2867 {
```

```
2868         ASSERT(cpuid_checkpass(cpu, 3));

2870         return (snprintf(s, n, "%s", cpu->cpu_m.mcpu_cpi->cpi_brandstr));
2871 }

2873 int
2874 cpuid_is_cmt(cpu_t *cpu)
2875 {
2876         if (cpu == NULL)
2877                 cpu = CPU;

2879         ASSERT(cpuid_checkpass(cpu, 1));

2881         return (cpu->cpu_m.mcpu_cpi->cpi_chipid >= 0);
2882 }

2884 /*
2885  * AMD and Intel both implement the 64-bit variant of the syscall
2886  * instruction (syscallq), so if there's -any- support for syscall,
2887  * cpuid currently says "yes, we support this".
2888  *
2889  * However, Intel decided to -not- implement the 32-bit variant of the
2890  * syscall instruction, so we provide a predicate to allow our caller
2891  * to test that subtlety here.
2892  *
2893  * XXPV Currently, 32-bit syscall instructions don't work via the hypervisor,
2894  *      even in the case where the hardware would in fact support it.
2895  */
2896 /*ARGSUSED*/
2897 int
2898 cpuid_syscall32_insn(cpu_t *cpu)
2899 {
2900         ASSERT(cpuid_checkpass((cpu == NULL ? CPU : cpu), 1));

2902 #if !defined(__xpv)
2903         if (cpu == NULL)
2904                 cpu = CPU;

2906         /*CSTYLED*/
2907         {
2908                 struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

2910                 if (cpi->cpi_vendor == X86_VENDOR_AMD &&
2911                     cpi->cpi_xmaxeax >= 0x80000001 &&
2912                     (CPI_FEATURES_XTD_EDX(cpi) & CPUID_AMD_EDX_SYSC))
2913                         return (1);
2914         }
2915 #endif
2916         return (0);
2917 }

2919 int
2920 cpuid_getidstr(cpu_t *cpu, char *s, size_t n)
2921 {
2922         struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

2924         static const char fmt[] =
2925             "x86 (%s %X family %d model %d step %d clock %d MHz)";
2926         static const char fmt_ht[] =
2927             "x86 (chipid 0x%x %s %X family %d model %d step %d clock %d MHz)";

2929         ASSERT(cpuid_checkpass(cpu, 1));

2931         if (cpuid_is_cmt(cpu))
2932                 return (snprintf(s, n, fmt_ht, cpi->cpi_chipid,
2933                     cpi->cpi_vendorstr, cpi->cpi_std[1].cp_eax,
```



```
2934                     cpi->cpi_family, cpi->cpi_model,
2935                     cpi->cpi_step, cpu->cpu_type_info.pi_clock));
2936         return (snprintf(s, n, fmt,
2937             cpi->cpi_vendorstr, cpi->cpi_std[1].cp_eax,
2938             cpi->cpi_family, cpi->cpi_model,
2939             cpi->cpi_step, cpu->cpu_type_info.pi_clock));
2940 }

2942 const char *
2943 cpuid_getvendorstr(cpu_t *cpu)
2944 {
2945         ASSERT(cpuid_checkpass(cpu, 1));
2946         return ((const char *)cpu->cpu_m.mcpu_cpi->cpi_vendorstr);
2947 }

2949 uint_t
2950 cpuid_getvendor(cpu_t *cpu)
2951 {
2952         ASSERT(cpuid_checkpass(cpu, 1));
2953         return (cpu->cpu_m.mcpu_cpi->cpi_vendor);
2954 }

2956 uint_t
2957 cpuid_getfamily(cpu_t *cpu)
2958 {
2959         ASSERT(cpuid_checkpass(cpu, 1));
2960         return (cpu->cpu_m.mcpu_cpi->cpi_family);
2961 }

2963 uint_t
2964 cpuid_getmodel(cpu_t *cpu)
2965 {
2966         ASSERT(cpuid_checkpass(cpu, 1));
2967         return (cpu->cpu_m.mcpu_cpi->cpi_model);
2968 }

2970 uint_t
2971 cpuid_get_ncpu_per_chip(cpu_t *cpu)
2972 {
2973         ASSERT(cpuid_checkpass(cpu, 1));
2974         return (cpu->cpu_m.mcpu_cpi->cpi_ncpu_per_chip);
2975 }

2977 uint_t
2978 cpuid_get_ncore_per_chip(cpu_t *cpu)
2979 {
2980         ASSERT(cpuid_checkpass(cpu, 1));
2981         return (cpu->cpu_m.mcpu_cpi->cpi_ncore_per_chip);
2982 }

2984 uint_t
2985 cpuid_get_ncpu_sharing_last_cache(cpu_t *cpu)
2986 {
2987         ASSERT(cpuid_checkpass(cpu, 2));
2988         return (cpu->cpu_m.mcpu_cpi->cpi_ncpu_shr_last_cache);
2989 }

2991 id_t
2992 cpuid_get_last_lvl_cacheid(cpu_t *cpu)
2993 {
2994         ASSERT(cpuid_checkpass(cpu, 2));
2995         return (cpu->cpu_m.mcpu_cpi->cpi_last_lvl_cacheid);
2996 }

2998 uint_t
2999 cpuid_getstep(cpu_t *cpu)
```

```
3000 {
3001         ASSERT(cpuid_checkpass(cpu, 1));
3002         return (cpu->cpu_m.mcpu_cpi->cpi_step);
3003 }

3005 uint_t
3006 cpuid_getsig(struct cpu *cpu)
3007 {
3008         ASSERT(cpuid_checkpass(cpu, 1));
3009         return (cpu->cpu_m.mcpu_cpi->cpi_std[1].cp_eax);
3010 }

3012 uint32_t
3013 cpuid_getchiprev(struct cpu *cpu)
3014 {
3015         ASSERT(cpuid_checkpass(cpu, 1));
3016         return (cpu->cpu_m.mcpu_cpi->cpi_chiprev);
3017 }

3019 const char *
3020 cpuid_getchiprevstr(struct cpu *cpu)
3021 {
3022         ASSERT(cpuid_checkpass(cpu, 1));
3023         return (cpu->cpu_m.mcpu_cpi->cpi_chiprevstr);
3024 }

3026 uint32_t
3027 cpuid_getsockettype(struct cpu *cpu)
3028 {
3029         ASSERT(cpuid_checkpass(cpu, 1));
3030         return (cpu->cpu_m.mcpu_cpi->cpi_socket);
3031 }

3033 const char *
3034 cpuid_getsocketstr(cpu_t *cpu)
3035 {
3036         static const char *socketstr = NULL;
3037         struct cpuid_info *cpi;

3039         ASSERT(cpuid_checkpass(cpu, 1));
3040         cpi = cpu->cpu_m.mcpu_cpi;

3042         /* Assume that socket types are the same across the system */
3043         if (socketstr == NULL)
3044                 socketstr = _cpuid_sktstr(cpi->cpi_vendor, cpi->cpi_family,
3045                     cpi->cpi_model, cpi->cpi_step);


3048         return (socketstr);
3049 }

3051 int
3052 cpuid_get_chipid(cpu_t *cpu)
3053 {
3054         ASSERT(cpuid_checkpass(cpu, 1));

3056         if (cpuid_is_cmt(cpu))
3057                 return (cpu->cpu_m.mcpu_cpi->cpi_chipid);
3058         return (cpu->cpu_id);
3059 }

3061 id_t
3062 cpuid_get_coreid(cpu_t *cpu)
3063 {
3064         ASSERT(cpuid_checkpass(cpu, 1));
3065         return (cpu->cpu_m.mcpu_cpi->cpi_coreid);
```



```
3066 }

3068 int
3069 cpuid_get_pkgcoreid(cpu_t *cpu)
3070 {
3071         ASSERT(cpuid_checkpass(cpu, 1));
3072         return (cpu->cpu_m.mcpu_cpi->cpi_pkgcoreid);
3073 }

3075 int
3076 cpuid_get_clogid(cpu_t *cpu)
3077 {
3078         ASSERT(cpuid_checkpass(cpu, 1));
3079         return (cpu->cpu_m.mcpu_cpi->cpi_clogid);
3080 }

3082 int
3083 cpuid_get_cacheid(cpu_t *cpu)
3084 {
3085         ASSERT(cpuid_checkpass(cpu, 1));
3086         return (cpu->cpu_m.mcpu_cpi->cpi_last_lvl_cacheid);
3087 }

3089 uint_t
3090 cpuid_get_procnodeid(cpu_t *cpu)
3091 {
3092         ASSERT(cpuid_checkpass(cpu, 1));
3093         return (cpu->cpu_m.mcpu_cpi->cpi_procnodeid);
3094 }

3096 uint_t
3097 cpuid_get_procnodes_per_pkg(cpu_t *cpu)
3098 {
3099         ASSERT(cpuid_checkpass(cpu, 1));
3100         return (cpu->cpu_m.mcpu_cpi->cpi_procnodes_per_pkg);
3101 }

3103 uint_t
3104 cpuid_get_compunitid(cpu_t *cpu)
3105 {
3106         ASSERT(cpuid_checkpass(cpu, 1));
3107         return (cpu->cpu_m.mcpu_cpi->cpi_compunitid);
3108 }

3110 uint_t
3111 cpuid_get_cores_per_compunit(cpu_t *cpu)
3112 {
3113         ASSERT(cpuid_checkpass(cpu, 1));
3114         return (cpu->cpu_m.mcpu_cpi->cpi_cores_per_compunit);
3115 }

3117 /*ARGSUSED*/
3118 int
3119 cpuid_have_cr8access(cpu_t *cpu)
3120 {
3121 #if defined(__amd64)
3122         return (1);
3123 #else
3124         struct cpuid_info *cpi;

3126         ASSERT(cpu != NULL);
3127         cpi = cpu->cpu_m.mcpu_cpi;
3128         if (cpi->cpi_vendor == X86_VENDOR_AMD && cpi->cpi_maxeax >= 1 &&
3129             (CPI_FEATURES_XTD_ECX(cpi) & CPUID_AMD_ECX_CR8D) != 0)
3130                 return (1);
3131         return (0);
```

```
#endif
}

uint32_t
cpuid_get_apicid(cpu_t *cpu)
{
        ASSERT(cpuid_checkpass(cpu, 1));
        if (cpu->cpu_m.mcpu_cpi->cpi_maxeax < 1) {
                return (UINT32_MAX);
        } else {
                return (cpu->cpu_m.mcpu_cpi->cpi_apicid);
        }
}

void
cpuid_get_addrsize(cpu_t *cpu, uint_t *pabits, uint_t *vabits)
{
        struct cpuid_info *cpi;

        if (cpu == NULL)
                cpu = CPU;
        cpi = cpu->cpu_m.mcpu_cpi;

        ASSERT(cpuid_checkpass(cpu, 1));

        if (pabits)
                *pabits = cpi->cpi_pabits;
        if (vabits)
                *vabits = cpi->cpi_vabits;
}

/*
 * Returns the number of data TLB entries for a corresponding
 * pagesize.  If it can't be computed, or isn't known, the
 * routine returns zero.  If you ask about an architecturally
 * impossible pagesize, the routine will panic (so that the
 * hat implementor knows that things are inconsistent.)
 */
uint_t
cpuid_get_dtlb_nent(cpu_t *cpu, size_t pagesize)
{
        struct cpuid_info *cpi;
        uint_t dtlb_nent = 0;

        if (cpu == NULL)
                cpu = CPU;
        cpi = cpu->cpu_m.mcpu_cpi;

        ASSERT(cpuid_checkpass(cpu, 1));

        /*
         * Check the L2 TLB info
         */
        if (cpi->cpi_xmaxeax >= 0x80000006) {
                struct cpuid_regs *cp = &cpi->cpi_extd[6];

                switch (pagesize) {

                case 4 * 1024:
                        /*
                         * All zero in the top 16 bits of the register
                         * indicates a unified TLB. Size is in low 16 bits.
                         */
                        if ((cp->cp_ebx & 0xffff0000) == 0)
                                dtlb_nent = cp->cp_ebx & 0x0000ffff;
                        else
```



```
                                dtlb_nent = BITX(cp->cp_ebx, 27, 16);
                        break;

                case 2 * 1024 * 1024:
                        if ((cp->cp_eax & 0xffff0000) == 0)
                                dtlb_nent = cp->cp_eax & 0x0000ffff;
                        else
                                dtlb_nent = BITX(cp->cp_eax, 27, 16);
                        break;

                default:
                        panic("unknown L2 pagesize");
                        /*NOTREACHED*/
                }
        }

        if (dtlb_nent != 0)
                return (dtlb_nent);

        /*
         * No L2 TLB support for this size, try L1.
         */
        if (cpi->cpi_xmaxeax >= 0x80000005) {
                struct cpuid_regs *cp = &cpi->cpi_extd[5];

                switch (pagesize) {
                case 4 * 1024:
                        dtlb_nent = BITX(cp->cp_ebx, 23, 16);
                        break;
                case 2 * 1024 * 1024:
                        dtlb_nent = BITX(cp->cp_eax, 23, 16);
                        break;
                default:
                        panic("unknown L1 d-TLB pagesize");
                        /*NOTREACHED*/
                }
        }

        return (dtlb_nent);
}

/*
 * Return 0 if the erratum is not present or not applicable, positive
 * if it is, and negative if the status of the erratum is unknown.
 *
 * See "Revision Guide for AMD Athlon(tm) 64 and AMD Opteron(tm)
 * Processors" #25759, Rev 3.57, August 2005
 */
int
cpuid_opteron_erratum(cpu_t *cpu, uint_t erratum)
{
        struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;
        uint_t eax;

        /*
         * Bail out if this CPU isn't an AMD CPU, or if it's
         * a legacy (32-bit) AMD CPU.
         */
        if (cpi->cpi_vendor != X86_VENDOR_AMD ||
            cpi->cpi_family == 4 || cpi->cpi_family == 5 ||
            cpi->cpi_family == 6)

                return (0);

        eax = cpi->cpi_std[1].cp_eax;
```

```
#define SH_B0(eax)      (eax == 0xf40 || eax == 0xf50)
#define SH_B3(eax)      (eax == 0xf51)
#define B(eax)          (SH_B0(eax) || SH_B3(eax))

#define SH_C0(eax)      (eax == 0xf48 || eax == 0xf58)

#define SH_CG(eax)      (eax == 0xf4a || eax == 0xf5a || eax == 0xf7a)
#define DH_CG(eax)      (eax == 0xfc0 || eax == 0xfe0 || eax == 0xff0)
#define CH_CG(eax)      (eax == 0xf82 || eax == 0xfb2)
#define CG(eax)         (SH_CG(eax) || DH_CG(eax) || CH_CG(eax))

#define SH_D0(eax)      (eax == 0x10f40 || eax == 0x10f50 || eax == 0x10f70)
#define DH_D0(eax)      (eax == 0x10fc0 || eax == 0x10ff0)
#define CH_D0(eax)      (eax == 0x10f80 || eax == 0x10fb0)
#define D0(eax)         (SH_D0(eax) || DH_D0(eax) || CH_D0(eax))

#define SH_E0(eax)      (eax == 0x20f50 || eax == 0x20f40 || eax == 0x20f70)
#define JH_E1(eax)      (eax == 0x20f10)        /* JH8_E0 had 0x20f30 */
#define DH_E3(eax)      (eax == 0x20fc0 || eax == 0x20ff0)
#define SH_E4(eax)      (eax == 0x20f51 || eax == 0x20f71)
#define BH_E4(eax)      (eax == 0x20fb1)
#define SH_E5(eax)      (eax == 0x20f42)
#define DH_E6(eax)      (eax == 0x20ff2 || eax == 0x20fc2)
#define JH_E6(eax)      (eax == 0x20f12 || eax == 0x20f32)
#define EX(eax)         (SH_E0(eax) || JH_E1(eax) || DH_E3(eax) || \
                            SH_E4(eax) || BH_E4(eax) || SH_E5(eax) || \
                            DH_E6(eax) || JH_E6(eax))

#define DR_AX(eax)      (eax == 0x100f00 || eax == 0x100f01 || eax == 0x100f02)
#define DR_B0(eax)      (eax == 0x100f20)
#define DR_B1(eax)      (eax == 0x100f21)
#define DR_BA(eax)      (eax == 0x100f2a)
#define DR_B2(eax)      (eax == 0x100f22)
#define DR_B3(eax)      (eax == 0x100f23)
#define RB_C0(eax)      (eax == 0x100f40)

        switch (erratum) {
        case 1:
                return (cpi->cpi_family < 0x10);
        case 51:        /* what does the asterisk mean? */
                return (B(eax) || SH_C0(eax) || CG(eax));
        case 52:
                return (B(eax));
        case 57:
                return (cpi->cpi_family <= 0x11);
        case 58:
                return (B(eax));
        case 60:
                return (cpi->cpi_family <= 0x11);
        case 61:
        case 62:
        case 63:
        case 64:
        case 65:
        case 66:
        case 68:
        case 69:
        case 70:
        case 71:
                return (B(eax));
        case 72:
                return (SH_B0(eax));
        case 74:
                return (B(eax));
        case 75:
                return (cpi->cpi_family < 0x10);
```



```
        case 76:
                return (B(eax));
        case 77:
                return (cpi->cpi_family <= 0x11);
        case 78:
                return (B(eax) || SH_C0(eax));
        case 79:
                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax) || EX(eax));
        case 80:
        case 81:
        case 82:
                return (B(eax));
        case 83:
                return (B(eax) || SH_C0(eax) || CG(eax));
        case 85:
                return (cpi->cpi_family < 0x10);
        case 86:
                return (SH_C0(eax) || CG(eax));
        case 88:
#if !defined(__amd64)
                return (0);
#else
                return (B(eax) || SH_C0(eax));
#endif
        case 89:
                return (cpi->cpi_family < 0x10);
        case 90:
                return (B(eax) || SH_C0(eax) || CG(eax));
        case 91:
        case 92:
                return (B(eax) || SH_C0(eax));
        case 93:
                return (SH_C0(eax));
        case 94:
                return (B(eax) || SH_C0(eax) || CG(eax));
        case 95:
#if !defined(__amd64)
                return (0);
#else
                return (B(eax) || SH_C0(eax));
#endif
        case 96:
                return (B(eax) || SH_C0(eax) || CG(eax));
        case 97:
        case 98:
                return (SH_C0(eax) || CG(eax));
        case 99:
                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
        case 100:
                return (B(eax) || SH_C0(eax));
        case 101:
        case 103:
                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
        case 104:
                return (SH_C0(eax) || CG(eax) || D0(eax));
        case 105:
        case 106:
        case 107:
                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
        case 108:
                return (DH_CG(eax));
        case 109:
                return (SH_C0(eax) || CG(eax) || D0(eax));
        case 110:
                return (D0(eax) || EX(eax));
        case 111:
```

```
3396                return (CG(eax));
3397        case 112:
3398                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax) || EX(eax));
3399        case 113:
3400                return (eax == 0x20fc0);
3401        case 114:
3402                return (SH_E0(eax) || JH_E1(eax) || DH_E3(eax));
3403        case 115:
3404                return (SH_E0(eax) || JH_E1(eax));
3405        case 116:
3406                return (SH_E0(eax) || JH_E1(eax) || DH_E3(eax));
3407        case 117:
3408                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax));
3409        case 118:
3410                return (SH_E0(eax) || JH_E1(eax) || SH_E4(eax) || BH_E4(eax) ||
3411                    JH_E6(eax));
3412        case 121:
3413                return (B(eax) || SH_C0(eax) || CG(eax) || D0(eax) || EX(eax));
3414        case 122:
3415                return (cpi->cpi_family < 0x10 || cpi->cpi_family == 0x11);
3416        case 123:
3417                return (JH_E1(eax) || BH_E4(eax) || JH_E6(eax));
3418        case 131:
3419                return (cpi->cpi_family < 0x10);
3420        case 6336786:
3421                /*
3422                 * Test for AdvPowerMgmtInfo.TscPStateInvariant
3423                 * if this is a K8 family or newer processor
3424                 */
3425                if (CPI_FAMILY(cpi) == 0xf) {
3426                        struct cpuid_regs regs;
3427                        regs.cp_eax = 0x80000007;
3428                        (void) __cpuid_insn(&regs);
3429                        return (!(regs.cp_edx & 0x100));
3430                }
3431                return (0);
3432        case 6323525:
3433                return (((((eax >> 12) & 0xff00) + (eax & 0xf00)) |
3434                    (((eax >> 4) & 0xf) | ((eax >> 12) & 0xf0))) < 0xf40);

3436        case 6671130:
3437                /*
3438                 * check for processors (pre-Shanghai) that do not provide
3439                 * optimal management of 1gb ptes in its tlb.
3440                 */
3441                return (cpi->cpi_family == 0x10 && cpi->cpi_model < 4);

3443        case 298:
3444                return (DR_AX(eax) || DR_B0(eax) || DR_B1(eax) || DR_BA(eax) ||
3445                    DR_B2(eax) || RB_C0(eax));

3447        case 721:
3448 #if defined(__amd64)
3449                return (cpi->cpi_family == 0x10 || cpi->cpi_family == 0x12);
3450 #else
3451                return (0);
3452 #endif

3454        default:
3455                return (-1);

3457        }
3458 }

3460 /*
3461  * Determine if specified erratum is present via OSVW (OS Visible Workaround).
```



```
3462  * Return 1 if erratum is present, 0 if not present and -1 if indeterminate.
3463  */
3464 int
3465 osvw_opteron_erratum(cpu_t *cpu, uint_t erratum)
3466 {
3467         struct cpuid_info       *cpi;
3468         uint_t                  osvwid;
3469         static int              osvwfeature = -1;
3470         uint64_t                osvwlength;


3473         cpi = cpu->cpu_m.mcpu_cpi;

3475         /* confirm OSVW supported */
3476         if (osvwfeature == -1) {
3477                 osvwfeature = cpi->cpi_extd[1].cp_ecx & CPUID_AMD_ECX_OSVW;
3478         } else {
3479                 /* assert that osvw feature setting is consistent on all cpus */
3480                 ASSERT(osvwfeature ==
3481                     (cpi->cpi_extd[1].cp_ecx & CPUID_AMD_ECX_OSVW));
3482         }
3483         if (!osvwfeature)
3484                 return (-1);

3486         osvwlength = rdmsr(MSR_AMD_OSVW_ID_LEN) & OSVW_ID_LEN_MASK;

3488         switch (erratum) {
3489         case 298:       /* osvwid is 0 */
3490                 osvwid = 0;
3491                 if (osvwlength <= (uint64_t)osvwid) {
3492                         /* osvwid 0 is unknown */
3493                         return (-1);
3494                 }

3496                 /*
3497                  * Check the OSVW STATUS MSR to determine the state
3498                  * of the erratum where:
3499                  *   0 - fixed by HW
3500                  *   1 - BIOS has applied the workaround when BIOS
3501                  *   workaround is available. (Or for other errata,
3502                  *   OS workaround is required.)
3503                  * For a value of 1, caller will confirm that the
3504                  * erratum 298 workaround has indeed been applied by BIOS.
3505                  *
3506                  * A 1 may be set in cpus that have a HW fix
3507                  * in a mixed cpu system. Regarding erratum 298:
3508                  *   In a multiprocessor platform, the workaround above
3509                  *   should be applied to all processors regardless of
3510                  *   silicon revision when an affected processor is
3511                  *   present.
3512                  */

3514                 return (rdmsr(MSR_AMD_OSVW_STATUS +
3515                     (osvwid / OSVW_ID_CNT_PER_MSR)) &
3516                     (1ULL << (osvwid % OSVW_ID_CNT_PER_MSR)));

3518         default:
3519                 return (-1);
3520         }
3521 }

3523 static const char assoc_str[] = "associativity";
3524 static const char line_str[] = "line-size";
3525 static const char size_str[] = "size";

3527 static void
```

```
3528 add_cache_prop(dev_info_t *devi, const char *label, const char *type,
3529     uint32_t val)
3530 {
3531         char buf[128];
3532
3533         /*
3534          * ndi_prop_update_int() is used because it is desirable for
3535          * DDI_PROP_HW_DEF and DDI_PROP_DONTSLEEP to be set.
3536          */
3537         if (snprintf(buf, sizeof (buf), "%s-%s", label, type) < sizeof (buf))
3538                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, devi, buf, val);
3539 }
3540
3541 /*
3542  * Intel-style cache/tlb description
3543  *
3544  * Standard cpuid level 2 gives a randomly ordered
3545  * selection of tags that index into a table that describes
3546  * cache and tlb properties.
3547  */
3548
3549 static const char l1_icache_str[] = "l1-icache";
3550 static const char l1_dcache_str[] = "l1-dcache";
3551 static const char l2_cache_str[] = "l2-cache";
3552 static const char l3_cache_str[] = "l3-cache";
3553 static const char itlb4k_str[] = "itlb-4K";
3554 static const char dtlb4k_str[] = "dtlb-4K";
3555 static const char itlb2M_str[] = "itlb-2M";
3556 static const char itlb4M_str[] = "itlb-4M";
3557 static const char dtlb4M_str[] = "dtlb-4M";
3558 static const char dtlb24_str[] = "dtlb0-2M-4M";
3559 static const char itlb424_str[] = "itlb-4K-2M-4M";
3560 static const char itlb24_str[] = "itlb-2M-4M";
3561 static const char dtlb44_str[] = "dtlb-4K-4M";
3562 static const char sl1_dcache_str[] = "sectored-l1-dcache";
3563 static const char sl2_cache_str[] = "sectored-l2-cache";
3564 static const char itrace_str[] = "itrace-cache";
3565 static const char sl3_cache_str[] = "sectored-l3-cache";
3566 static const char sh_l2_tlb4k_str[] = "shared-l2-tlb-4k";
3567
3568 static const struct cachetab {
3569         uint8_t         ct_code;
3570         uint8_t         ct_assoc;
3571         uint16_t        ct_line_size;
3572         size_t          ct_size;
3573         const char      *ct_label;
3574 } intel_ctab[] = {
3575         /*
3576          * maintain descending order!
3577          *
3578          * Codes ignored - Reason
3579          * ----------------------
3580          * 40H - intel_cpuid_4_cache_info() disambiguates l2/l3 cache
3581          * f0H/f1H - Currently we do not interpret prefetch size by design
3582          */
3583         { 0xe4, 16, 64, 8*1024*1024, l3_cache_str},
3584         { 0xe3, 16, 64, 4*1024*1024, l3_cache_str},
3585         { 0xe2, 16, 64, 2*1024*1024, l3_cache_str},
3586         { 0xde, 12, 64, 6*1024*1024, l3_cache_str},
3587         { 0xdd, 12, 64, 3*1024*1024, l3_cache_str},
3588         { 0xdc, 12, 64, ((1*1024*1024)+(512*1024)), l3_cache_str},
3589         { 0xd8, 8, 64, 4*1024*1024, l3_cache_str},
3590         { 0xd7, 8, 64, 2*1024*1024, l3_cache_str},
3591         { 0xd6, 8, 64, 1*1024*1024, l3_cache_str},
3592         { 0xd2, 4, 64, 2*1024*1024, l3_cache_str},
3593         { 0xd1, 4, 64, 1*1024*1024, l3_cache_str},
```



```
3594         { 0xd0, 4, 64, 512*1024, l3_cache_str},
3595         { 0xca, 4, 0, 512, sh_l2_tlb4k_str},
3596         { 0xc0, 4, 0, 8, dtlb44_str },
3597         { 0xba, 4, 0, 64, dtlb4k_str },
3598         { 0xb4, 4, 0, 256, dtlb4k_str },
3599         { 0xb3, 4, 0, 128, dtlb4k_str },
3600         { 0xb2, 4, 0, 64, itlb4k_str },
3601         { 0xb0, 4, 0, 128, itlb4k_str },
3602         { 0x87, 8, 64, 1024*1024, l2_cache_str},
3603         { 0x86, 4, 64, 512*1024, l2_cache_str},
3604         { 0x85, 8, 32, 2*1024*1024, l2_cache_str},
3605         { 0x84, 8, 32, 1024*1024, l2_cache_str},
3606         { 0x83, 8, 32, 512*1024, l2_cache_str},
3607         { 0x82, 8, 32, 256*1024, l2_cache_str},
3608         { 0x80, 8, 64, 512*1024, l2_cache_str},
3609         { 0x7f, 2, 64, 512*1024, l2_cache_str},
3610         { 0x7d, 8, 64, 2*1024*1024, sl2_cache_str},
3611         { 0x7c, 8, 64, 1024*1024, sl2_cache_str},
3612         { 0x7b, 8, 64, 512*1024, sl2_cache_str},
3613         { 0x7a, 8, 64, 256*1024, sl2_cache_str},
3614         { 0x79, 8, 64, 128*1024, sl2_cache_str},
3615         { 0x78, 8, 64, 1024*1024, l2_cache_str},
3616         { 0x73, 8, 0, 64*1024, itrace_str},
3617         { 0x72, 8, 0, 32*1024, itrace_str},
3618         { 0x71, 8, 0, 16*1024, itrace_str},
3619         { 0x70, 8, 0, 12*1024, itrace_str},
3620         { 0x68, 4, 64, 32*1024, sl1_dcache_str},
3621         { 0x67, 4, 64, 16*1024, sl1_dcache_str},
3622         { 0x66, 4, 64, 8*1024, sl1_dcache_str},
3623         { 0x60, 8, 64, 16*1024, sl1_dcache_str},
3624         { 0x5d, 0, 0, 256, dtlb44_str},
3625         { 0x5c, 0, 0, 128, dtlb44_str},
3626         { 0x5b, 0, 0, 64, dtlb44_str},
3627         { 0x5a, 4, 0, 32, dtlb24_str},
3628         { 0x59, 0, 0, 16, dtlb4k_str},
3629         { 0x57, 4, 0, 16, dtlb4k_str},
3630         { 0x56, 4, 0, 16, dtlb4M_str},
3631         { 0x55, 0, 0, 7, itlb24_str},
3632         { 0x52, 0, 0, 256, itlb424_str},
3633         { 0x51, 0, 0, 128, itlb424_str},
3634         { 0x50, 0, 0, 64, itlb424_str},
3635         { 0x4f, 0, 0, 32, itlb4k_str},
3636         { 0x4e, 24, 64, 6*1024*1024, l2_cache_str},
3637         { 0x4d, 16, 64, 16*1024*1024, l3_cache_str},
3638         { 0x4c, 12, 64, 12*1024*1024, l3_cache_str},
3639         { 0x4b, 16, 64, 8*1024*1024, l3_cache_str},
3640         { 0x4a, 12, 64, 6*1024*1024, l3_cache_str},
3641         { 0x49, 16, 64, 4*1024*1024, l3_cache_str},
3642         { 0x48, 12, 64, 3*1024*1024, l2_cache_str},
3643         { 0x47, 8, 64, 8*1024*1024, l3_cache_str},
3644         { 0x46, 4, 64, 4*1024*1024, l3_cache_str},
3645         { 0x45, 4, 32, 2*1024*1024, l2_cache_str},
3646         { 0x44, 4, 32, 1024*1024, l2_cache_str},
3647         { 0x43, 4, 32, 512*1024, l2_cache_str},
3648         { 0x42, 4, 32, 256*1024, l2_cache_str},
3649         { 0x41, 4, 32, 128*1024, l2_cache_str},
3650         { 0x3e, 4, 64, 512*1024, sl2_cache_str},
3651         { 0x3d, 6, 64, 384*1024, sl2_cache_str},
3652         { 0x3c, 4, 64, 256*1024, sl2_cache_str},
3653         { 0x3b, 2, 64, 128*1024, sl2_cache_str},
3654         { 0x3a, 6, 64, 192*1024, sl2_cache_str},
3655         { 0x39, 4, 64, 128*1024, sl2_cache_str},
3656         { 0x30, 8, 64, 32*1024, l1_icache_str},
3657         { 0x2c, 8, 64, 32*1024, l1_dcache_str},
3658         { 0x29, 8, 64, 4096*1024, sl3_cache_str},
3659         { 0x25, 8, 64, 2048*1024, sl3_cache_str},
```

```
3660         { 0x23, 8, 64, 1024*1024, sl3_cache_str},
3661         { 0x22, 4, 64, 512*1024, sl3_cache_str},
3662         { 0x0e, 6, 64, 24*1024, l1_dcache_str},
3663         { 0x0d, 4, 32, 16*1024, l1_dcache_str},
3664         { 0x0c, 4, 32, 16*1024, l1_dcache_str},
3665         { 0x0b, 4, 0, 4, itlb4M_str},
3666         { 0x0a, 2, 32, 8*1024, l1_dcache_str},
3667         { 0x08, 4, 32, 16*1024, l1_icache_str},
3668         { 0x06, 4, 32, 8*1024, l1_icache_str},
3669         { 0x05, 4, 0, 32, dtlb4M_str},
3670         { 0x04, 4, 0, 8, dtlb4M_str},
3671         { 0x03, 4, 0, 64, dtlb4k_str},
3672         { 0x02, 4, 0, 2, itlb4M_str},
3673         { 0x01, 4, 0, 32, itlb4k_str},
3674         { 0 }
3675 };

3677 static const struct cachetab cyrix_ctab[] = {
3678         { 0x70, 4, 0, 32, "tlb-4K" },
3679         { 0x80, 4, 16, 16*1024, "l1-cache" },
3680         { 0 }
3681 };

3683 /*
3684  * Search a cache table for a matching entry
3685  */
3686 static const struct cachetab *
3687 find_cacheent(const struct cachetab *ct, uint_t code)
3688 {
3689         if (code != 0) {
3690                 for (; ct->ct_code != 0; ct++)
3691                         if (ct->ct_code <= code)
3692                                 break;
3693                 if (ct->ct_code == code)
3694                         return (ct);
3695         }
3696         return (NULL);
3697 }

3699 /*
3700  * Populate cachetab entry with L2 or L3 cache-information using
3701  * cpuid function 4. This function is called from intel_walk_cacheinfo()
3702  * when descriptor 0x49 is encountered. It returns 0 if no such cache
3703  * information is found.
3704  */
3705 static int
3706 intel_cpuid_4_cache_info(struct cachetab *ct, struct cpuid_info *cpi)
3707 {
3708         uint32_t level, i;
3709         int ret = 0;

3711         for (i = 0; i < cpi->cpi_std_4_size; i++) {
3712                 level = CPI_CACHE_LVL(cpi->cpi_std_4[i]);

3714                 if (level == 2 || level == 3) {
3715                         ct->ct_assoc = CPI_CACHE_WAYS(cpi->cpi_std_4[i]) + 1;
3716                         ct->ct_line_size =
3717                             CPI_CACHE_COH_LN_SZ(cpi->cpi_std_4[i]) + 1;
3718                         ct->ct_size = ct->ct_assoc *
3719                             (CPI_CACHE_PARTS(cpi->cpi_std_4[i]) + 1) *
3720                             ct->ct_line_size *
3721                             (cpi->cpi_std_4[i]->cp_ecx + 1);

3723                         if (level == 2) {
3724                                 ct->ct_label = l2_cache_str;
3725                         } else if (level == 3) {
```



```
3726                                 ct->ct_label = l3_cache_str;
3727                         }
3728                         ret = 1;
3729                 }
3730         }

3732         return (ret);
3733 }

3735 /*
3736  * Walk the cacheinfo descriptor, applying 'func' to every valid element
3737  * The walk is terminated if the walker returns non-zero.
3738  */
3739 static void
3740 intel_walk_cacheinfo(struct cpuid_info *cpi,
3741     void *arg, int (*func)(void *, const struct cachetab *))
3742 {
3743         const struct cachetab *ct;
3744         struct cachetab des_49_ct, des_b1_ct;
3745         uint8_t *dp;
3746         int i;

3748         if ((dp = cpi->cpi_cacheinfo) == NULL)
3749                 return;
3750         for (i = 0; i < cpi->cpi_ncache; i++, dp++) {
3751                 /*
3752                  * For overloaded descriptor 0x49 we use cpuid function 4
3753                  * if supported by the current processor, to create
3754                  * cache information.
3755                  * For overloaded descriptor 0xb1 we use X86_PAE flag
3756                  * to disambiguate the cache information.
3757                  */
3758                 if (*dp == 0x49 && cpi->cpi_maxeax >= 0x4 &&
3759                     intel_cpuid_4_cache_info(&des_49_ct, cpi) == 1) {
3760                         ct = &des_49_ct;
3761                 } else if (*dp == 0xb1) {
3762                         des_b1_ct.ct_code = 0xb1;
3763                         des_b1_ct.ct_assoc = 4;
3764                         des_b1_ct.ct_line_size = 0;
3765                         if (is_x86_feature(x86_featureset, X86FSET_PAE)) {
3766                                 des_b1_ct.ct_size = 8;
3767                                 des_b1_ct.ct_label = itlb2M_str;
3768                         } else {
3769                                 des_b1_ct.ct_size = 4;
3770                                 des_b1_ct.ct_label = itlb4M_str;
3771                         }
3772                         ct = &des_b1_ct;
3773                 } else {
3774                         if ((ct = find_cacheent(intel_ctab, *dp)) == NULL) {
3775                                 continue;
3776                         }
3777                 }

3779                 if (func(arg, ct) != 0) {
3780                         break;
3781                 }
3782         }
3783 }

3785 /*
3786  * (Like the Intel one, except for Cyrix CPUs)
3787  */
3788 static void
3789 cyrix_walk_cacheinfo(struct cpuid_info *cpi,
3790     void *arg, int (*func)(void *, const struct cachetab *))
3791 {
```

```
3792         const struct cachetab *ct;
3793         uint8_t *dp;
3794         int i;

3796         if ((dp = cpi->cpi_cacheinfo) == NULL)
3797                 return;
3798         for (i = 0; i < cpi->cpi_ncache; i++, dp++) {
3799                 /*
3800                  * Search Cyrix-specific descriptor table first ..
3801                  */
3802                 if ((ct = find_cacheent(cyrix_ctab, *dp)) != NULL) {
3803                         if (func(arg, ct) != 0)
3804                                 break;
3805                         continue;
3806                 }
3807                 /*
3808                  * .. else fall back to the Intel one
3809                  */
3810                 if ((ct = find_cacheent(intel_ctab, *dp)) != NULL) {
3811                         if (func(arg, ct) != 0)
3812                                 break;
3813                         continue;
3814                 }
3815         }
3816 }

3818 /*
3819  * A cacheinfo walker that adds associativity, line-size, and size properties
3820  * to the devinfo node it is passed as an argument.
3821  */
3822 static int
3823 add_cacheent_props(void *arg, const struct cachetab *ct)
3824 {
3825         dev_info_t *devi = arg;

3827         add_cache_prop(devi, ct->ct_label, assoc_str, ct->ct_assoc);
3828         if (ct->ct_line_size != 0)
3829                 add_cache_prop(devi, ct->ct_label, line_str,
3830                     ct->ct_line_size);
3831         add_cache_prop(devi, ct->ct_label, size_str, ct->ct_size);
3832         return (0);
3833 }


3836 static const char fully_assoc[] = "fully-associative?";

3838 /*
3839  * AMD style cache/tlb description
3840  *
3841  * Extended functions 5 and 6 directly describe properties of
3842  * tlbs and various cache levels.
3843  */
3844 static void
3845 add_amd_assoc(dev_info_t *devi, const char *label, uint_t assoc)
3846 {
3847         switch (assoc) {
3848         case 0: /* reserved; ignore */
3849                 break;
3850         default:
3851                 add_cache_prop(devi, label, assoc_str, assoc);
3852                 break;
3853         case 0xff:
3854                 add_cache_prop(devi, label, fully_assoc, 1);
3855                 break;
3856         }
3857 }
```



```
3859 static void
3860 add_amd_tlb(dev_info_t *devi, const char *label, uint_t assoc, uint_t size)
3861 {
3862         if (size == 0)
3863                 return;
3864         add_cache_prop(devi, label, size_str, size);
3865         add_amd_assoc(devi, label, assoc);
3866 }

3868 static void
3869 add_amd_cache(dev_info_t *devi, const char *label,
3870     uint_t size, uint_t assoc, uint_t lines_per_tag, uint_t line_size)
3871 {
3872         if (size == 0 || line_size == 0)
3873                 return;
3874         add_amd_assoc(devi, label, assoc);
3875         /*
3876          * Most AMD parts have a sectored cache. Multiple cache lines are
3877          * associated with each tag. A sector consists of all cache lines
3878          * associated with a tag. For example, the AMD K6-III has a sector
3879          * size of 2 cache lines per tag.
3880          */
3881         if (lines_per_tag != 0)
3882                 add_cache_prop(devi, label, "lines-per-tag", lines_per_tag);
3883         add_cache_prop(devi, label, line_str, line_size);
3884         add_cache_prop(devi, label, size_str, size * 1024);
3885 }

3887 static void
3888 add_amd_l2_assoc(dev_info_t *devi, const char *label, uint_t assoc)
3889 {
3890         switch (assoc) {
3891         case 0: /* off */
3892                 break;
3893         case 1:
3894         case 2:
3895         case 4:
3896                 add_cache_prop(devi, label, assoc_str, assoc);
3897                 break;
3898         case 6:
3899                 add_cache_prop(devi, label, assoc_str, 8);
3900                 break;
3901         case 8:
3902                 add_cache_prop(devi, label, assoc_str, 16);
3903                 break;
3904         case 0xf:
3905                 add_cache_prop(devi, label, fully_assoc, 1);
3906                 break;
3907         default: /* reserved; ignore */
3908                 break;
3909         }
3910 }

3912 static void
3913 add_amd_l2_tlb(dev_info_t *devi, const char *label, uint_t assoc, uint_t size)
3914 {
3915         if (size == 0 || assoc == 0)
3916                 return;
3917         add_amd_l2_assoc(devi, label, assoc);
3918         add_cache_prop(devi, label, size_str, size);
3919 }

3921 static void
3922 add_amd_l2_cache(dev_info_t *devi, const char *label,
3923     uint_t size, uint_t assoc, uint_t lines_per_tag, uint_t line_size)
```

```
3924 {
3925         if (size == 0 || assoc == 0 || line_size == 0)
3926                 return;
3927         add_amd_l2_assoc(devi, label, assoc);
3928         if (lines_per_tag != 0)
3929                 add_cache_prop(devi, label, "lines-per-tag", lines_per_tag);
3930         add_cache_prop(devi, label, line_str, line_size);
3931         add_cache_prop(devi, label, size_str, size * 1024);
3932 }

3934 static void
3935 amd_cache_info(struct cpuid_info *cpi, dev_info_t *devi)
3936 {
3937         struct cpuid_regs *cp;

3939         if (cpi->cpi_xmaxeax < 0x80000005)
3940                 return;
3941         cp = &cpi->cpi_extd[5];

3943         /*
3944          * 4M/2M L1 TLB configuration
3945          *
3946          * We report the size for 2M pages because AMD uses two
3947          * TLB entries for one 4M page.
3948          */
3949         add_amd_tlb(devi, "dtlb-2M",
3950             BITX(cp->cp_eax, 31, 24), BITX(cp->cp_eax, 23, 16));
3951         add_amd_tlb(devi, "itlb-2M",
3952             BITX(cp->cp_eax, 15, 8), BITX(cp->cp_eax, 7, 0));

3954         /*
3955          * 4K L1 TLB configuration
3956          */

3958         switch (cpi->cpi_vendor) {
3959                 uint_t nentries;
3960         case X86_VENDOR_TM:
3961                 if (cpi->cpi_family >= 5) {
3962                         /*
3963                          * Crusoe processors have 256 TLB entries, but
3964                          * cpuid data format constrains them to only
3965                          * reporting 255 of them.
3966                          */
3967                         if ((nentries = BITX(cp->cp_ebx, 23, 16)) == 255)
3968                                 nentries = 256;
3969                         /*
3970                          * Crusoe processors also have a unified TLB
3971                          */
3972                         add_amd_tlb(devi, "tlb-4K", BITX(cp->cp_ebx, 31, 24),
3973                             nentries);
3974                         break;
3975                 }
3976                 /*FALLTHROUGH*/
3977         default:
3978                 add_amd_tlb(devi, itlb4k_str,
3979                     BITX(cp->cp_ebx, 31, 24), BITX(cp->cp_ebx, 23, 16));
3980                 add_amd_tlb(devi, dtlb4k_str,
3981                     BITX(cp->cp_ebx, 15, 8), BITX(cp->cp_ebx, 7, 0));
3982                 break;
3983         }

3985         /*
3986          * data L1 cache configuration
3987          */

3989         add_amd_cache(devi, l1_dcache_str,
```



```
3990             BITX(cp->cp_ecx, 31, 24), BITX(cp->cp_ecx, 23, 16),
3991             BITX(cp->cp_ecx, 15, 8), BITX(cp->cp_ecx, 7, 0));

3993         /*
3994          * code L1 cache configuration
3995          */

3997         add_amd_cache(devi, l1_icache_str,
3998             BITX(cp->cp_edx, 31, 24), BITX(cp->cp_edx, 23, 16),
3999             BITX(cp->cp_edx, 15, 8), BITX(cp->cp_edx, 7, 0));

4001         if (cpi->cpi_xmaxeax < 0x80000006)
4002                 return;
4003         cp = &cpi->cpi_extd[6];

4005         /* Check for a unified L2 TLB for large pages */

4007         if (BITX(cp->cp_eax, 31, 16) == 0)
4008                 add_amd_l2_tlb(devi, "l2-tlb-2M",
4009                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4010         else {
4011                 add_amd_l2_tlb(devi, "l2-dtlb-2M",
4012                     BITX(cp->cp_eax, 31, 28), BITX(cp->cp_eax, 27, 16));
4013                 add_amd_l2_tlb(devi, "l2-itlb-2M",
4014                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4015         }

4017         /* Check for a unified L2 TLB for 4K pages */

4019         if (BITX(cp->cp_ebx, 31, 16) == 0) {
4020                 add_amd_l2_tlb(devi, "l2-tlb-4K",
4021                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4022         } else {
4023                 add_amd_l2_tlb(devi, "l2-dtlb-4K",
4024                     BITX(cp->cp_eax, 31, 28), BITX(cp->cp_eax, 27, 16));
4025                 add_amd_l2_tlb(devi, "l2-itlb-4K",
4026                     BITX(cp->cp_eax, 15, 12), BITX(cp->cp_eax, 11, 0));
4027         }

4029         add_amd_l2_cache(devi, l2_cache_str,
4030             BITX(cp->cp_ecx, 31, 16), BITX(cp->cp_ecx, 15, 12),
4031             BITX(cp->cp_ecx, 11, 8), BITX(cp->cp_ecx, 7, 0));
4032 }

4034 /*
4035  * There are two basic ways that the x86 world describes it cache
4036  * and tlb architecture - Intel's way and AMD's way.
4037  *
4038  * Return which flavor of cache architecture we should use
4039  */
4040 static int
4041 x86_which_cacheinfo(struct cpuid_info *cpi)
4042 {
4043         switch (cpi->cpi_vendor) {
4044         case X86_VENDOR_Intel:
4045                 if (cpi->cpi_maxeax >= 2)
4046                         return (X86_VENDOR_Intel);
4047                 break;
4048         case X86_VENDOR_AMD:
4049                 /*
4050                  * The K5 model 1 was the first part from AMD that reported
4051                  * cache sizes via extended cpuid functions.
4052                  */
4053                 if (cpi->cpi_family > 5 ||
4054                     (cpi->cpi_family == 5 && cpi->cpi_model >= 1))
4055                         return (X86_VENDOR_AMD);
```

```
4056                 break;
4057         case X86_VENDOR_TM:
4058                 if (cpi->cpi_family >= 5)
4059                         return (X86_VENDOR_AMD);
4060                 /*FALLTHROUGH*/
4061         default:
4062                 /*
4063                  * If they have extended CPU data for 0x80000005
4064                  * then we assume they have AMD-format cache
4065                  * information.
4066                  *
4067                  * If not, and the vendor happens to be Cyrix,
4068                  * then try our-Cyrix specific handler.
4069                  *
4070                  * If we're not Cyrix, then assume we're using Intel's
4071                  * table-driven format instead.
4072                  */
4073                 if (cpi->cpi_xmaxeax >= 0x80000005)
4074                         return (X86_VENDOR_AMD);
4075                 else if (cpi->cpi_vendor == X86_VENDOR_Cyrix)
4076                         return (X86_VENDOR_Cyrix);
4077                 else if (cpi->cpi_maxeax >= 2)
4078                         return (X86_VENDOR_Intel);
4079                 break;
4080         }
4081         return (-1);
4082 }

4084 void
4085 cpuid_set_cpu_properties(void *dip, processorid_t cpu_id,
4086     struct cpuid_info *cpi)
4087 {
4088         dev_info_t *cpu_devi;
4089         int create;

4091         cpu_devi = (dev_info_t *)dip;

4093         /* device_type */
4094         (void) ndi_prop_update_string(DDI_DEV_T_NONE, cpu_devi,
4095             "device_type", "cpu");

4097         /* reg */
4098         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4099             "reg", cpu_id);

4101         /* cpu-mhz, and clock-frequency */
4102         if (cpu_freq > 0) {
4103                 long long mul;

4105                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4106                     "cpu-mhz", cpu_freq);
4107                 if ((mul = cpu_freq * 1000000LL) <= INT_MAX)
4108                         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4109                             "clock-frequency", (int)mul);
4110         }

4112         if (!is_x86_feature(x86_featureset, X86FSET_CPUID)) {
4113                 return;
4114         }

4116         /* vendor-id */
4117         (void) ndi_prop_update_string(DDI_DEV_T_NONE, cpu_devi,
4118             "vendor-id", cpi->cpi_vendorstr);

4120         if (cpi->cpi_maxeax == 0) {
4121                 return;
```



```
4122         }

4124         /*
4125          * family, model, and step
4126          */
4127         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4128             "family", CPI_FAMILY(cpi));
4129         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4130             "cpu-model", CPI_MODEL(cpi));
4131         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4132             "stepping-id", CPI_STEP(cpi));

4134         /* type */
4135         switch (cpi->cpi_vendor) {
4136         case X86_VENDOR_Intel:
4137                 create = 1;
4138                 break;
4139         default:
4140                 create = 0;
4141                 break;
4142         }
4143         if (create)
4144                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4145                     "type", CPI_TYPE(cpi));

4147         /* ext-family */
4148         switch (cpi->cpi_vendor) {
4149         case X86_VENDOR_Intel:
4150         case X86_VENDOR_AMD:
4151                 create = cpi->cpi_family >= 0xf;
4152                 break;
4153         default:
4154                 create = 0;
4155                 break;
4156         }
4157         if (create)
4158                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4159                     "ext-family", CPI_FAMILY_XTD(cpi));

4161         /* ext-model */
4162         switch (cpi->cpi_vendor) {
4163         case X86_VENDOR_Intel:
4164                 create = IS_EXTENDED_MODEL_INTEL(cpi);
4165                 break;
4166         case X86_VENDOR_AMD:
4167                 create = CPI_FAMILY(cpi) == 0xf;
4168                 break;
4169         default:
4170                 create = 0;
4171                 break;
4172         }
4173         if (create)
4174                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4175                     "ext-model", CPI_MODEL_XTD(cpi));

4177         /* generation */
4178         switch (cpi->cpi_vendor) {
4179         case X86_VENDOR_AMD:
4180                 /*
4181                  * AMD K5 model 1 was the first part to support this
4182                  */
4183                 create = cpi->cpi_xmaxeax >= 0x80000001;
4184                 break;
4185         default:
4186                 create = 0;
4187                 break;
```

```
4188         }
4189         if (create)
4190                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4191                     "generation", BITX((cpi)->cpi_extd[1].cp_eax, 11, 8));
4192
4193         /* brand-id */
4194         switch (cpi->cpi_vendor) {
4195         case X86_VENDOR_Intel:
4196                 /*
4197                  * brand id first appeared on Pentium III Xeon model 8,
4198                  * and Celeron model 8 processors and Opteron
4199                  */
4200                 create = cpi->cpi_family > 6 ||
4201                     (cpi->cpi_family == 6 && cpi->cpi_model >= 8);
4202                 break;
4203         case X86_VENDOR_AMD:
4204                 create = cpi->cpi_family >= 0xf;
4205                 break;
4206         default:
4207                 create = 0;
4208                 break;
4209         }
4210         if (create && cpi->cpi_brandid != 0) {
4211                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4212                     "brand-id", cpi->cpi_brandid);
4213         }
4214
4215         /* chunks, and apic-id */
4216         switch (cpi->cpi_vendor) {
4217                 /*
4218                  * first available on Pentium IV and Opteron (K8)
4219                  */
4220         case X86_VENDOR_Intel:
4221                 create = IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf;
4222                 break;
4223         case X86_VENDOR_AMD:
4224                 create = cpi->cpi_family >= 0xf;
4225                 break;
4226         default:
4227                 create = 0;
4228                 break;
4229         }
4230         if (create) {
4231                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4232                     "chunks", CPI_CHUNKS(cpi));
4233                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4234                     "apic-id", cpi->cpi_apicid);
4235                 if (cpi->cpi_chipid >= 0) {
4236                         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4237                             "chip#", cpi->cpi_chipid);
4238                         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4239                             "clog#", cpi->cpi_clogid);
4240                 }
4241         }
4242
4243         /* cpuid-features */
4244         (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4245             "cpuid-features", CPI_FEATURES_EDX(cpi));
4246
4247
4248         /* cpuid-features-ecx */
4249         switch (cpi->cpi_vendor) {
4250         case X86_VENDOR_Intel:
4251                 create = IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf;
4252                 break;
4253         case X86_VENDOR_AMD:
```



```
4254                 create = cpi->cpi_family >= 0xf;
4255                 break;
4256         default:
4257                 create = 0;
4258                 break;
4259         }
4260         if (create)
4261                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4262                     "cpuid-features-ecx", CPI_FEATURES_ECX(cpi));
4263
4264         /* ext-cpuid-features */
4265         switch (cpi->cpi_vendor) {
4266         case X86_VENDOR_Intel:
4267         case X86_VENDOR_AMD:
4268         case X86_VENDOR_Cyrix:
4269         case X86_VENDOR_TM:
4270         case X86_VENDOR_Centaur:
4271                 create = cpi->cpi_xmaxeax >= 0x80000001;
4272                 break;
4273         default:
4274                 create = 0;
4275                 break;
4276         }
4277         if (create) {
4278                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4279                     "ext-cpuid-features", CPI_FEATURES_XTD_EDX(cpi));
4280                 (void) ndi_prop_update_int(DDI_DEV_T_NONE, cpu_devi,
4281                     "ext-cpuid-features-ecx", CPI_FEATURES_XTD_ECX(cpi));
4282         }
4283
4284         /*
4285          * Brand String first appeared in Intel Pentium IV, AMD K5
4286          * model 1, and Cyrix GXm.  On earlier models we try and
4287          * simulate something similar .. so this string should always
4288          * same -something- about the processor, however lame.
4289          */
4290         (void) ndi_prop_update_string(DDI_DEV_T_NONE, cpu_devi,
4291             "brand-string", cpi->cpi_brandstr);
4292
4293         /*
4294          * Finally, cache and tlb information
4295          */
4296         switch (x86_which_cacheinfo(cpi)) {
4297         case X86_VENDOR_Intel:
4298                 intel_walk_cacheinfo(cpi, cpu_devi, add_cacheent_props);
4299                 break;
4300         case X86_VENDOR_Cyrix:
4301                 cyrix_walk_cacheinfo(cpi, cpu_devi, add_cacheent_props);
4302                 break;
4303         case X86_VENDOR_AMD:
4304                 amd_cache_info(cpi, cpu_devi);
4305                 break;
4306         default:
4307                 break;
4308         }
4309 }
4310
4311 struct l2info {
4312         int *l2i_csz;
4313         int *l2i_lsz;
4314         int *l2i_assoc;
4315         int l2i_ret;
4316 };
4317
4318 /*
4319  * A cacheinfo walker that fetches the size, line-size and associativity
```

```
 * of the L2 cache
 */
static int
intel_l2cinfo(void *arg, const struct cachetab *ct)
{
        struct l2info *l2i = arg;
        int *ip;

        if (ct->ct_label != l2_cache_str &&
            ct->ct_label != sl2_cache_str)
                return (0);     /* not an L2 -- keep walking */

        if ((ip = l2i->l2i_csz) != NULL)
                *ip = ct->ct_size;
        if ((ip = l2i->l2i_lsz) != NULL)
                *ip = ct->ct_line_size;
        if ((ip = l2i->l2i_assoc) != NULL)
                *ip = ct->ct_assoc;
        l2i->l2i_ret = ct->ct_size;
        return (1);             /* was an L2 -- terminate walk */
}

/*
 * AMD L2/L3 Cache and TLB Associativity Field Definition:
 *
 *      Unlike the associativity for the L1 cache and tlb where the 8 bit
 *      value is the associativity, the associativity for the L2 cache and
 *      tlb is encoded in the following table. The 4 bit L2 value serves as
 *      an index into the amd_afd[] array to determine the associativity.
 *      -1 is undefined. 0 is fully associative.
 */

static int amd_afd[] =
        {-1, 1, 2, -1, 4, -1, 8, -1, 16, -1, 32, 48, 64, 96, 128, 0};

static void
amd_l2cacheinfo(struct cpuid_info *cpi, struct l2info *l2i)
{
        struct cpuid_regs *cp;
        uint_t size, assoc;
        int i;
        int *ip;

        if (cpi->cpi_xmaxeax < 0x80000006)
                return;
        cp = &cpi->cpi_extd[6];

        if ((i = BITX(cp->cp_ecx, 15, 12)) != 0 &&
            (size = BITX(cp->cp_ecx, 31, 16)) != 0) {
                uint_t cachesz = size * 1024;
                assoc = amd_afd[i];

                ASSERT(assoc != -1);

                if ((ip = l2i->l2i_csz) != NULL)
                        *ip = cachesz;
                if ((ip = l2i->l2i_lsz) != NULL)
                        *ip = BITX(cp->cp_ecx, 7, 0);
                if ((ip = l2i->l2i_assoc) != NULL)
                        *ip = assoc;
                l2i->l2i_ret = cachesz;
        }
}

int
getl2cacheinfo(cpu_t *cpu, int *csz, int *lsz, int *assoc)
```



```
{
        struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;
        struct l2info __l2info, *l2i = &__l2info;

        l2i->l2i_csz = csz;
        l2i->l2i_lsz = lsz;
        l2i->l2i_assoc = assoc;
        l2i->l2i_ret = -1;

        switch (x86_which_cacheinfo(cpi)) {
        case X86_VENDOR_Intel:
                intel_walk_cacheinfo(cpi, l2i, intel_l2cinfo);
                break;
        case X86_VENDOR_Cyrix:
                cyrix_walk_cacheinfo(cpi, l2i, intel_l2cinfo);
                break;
        case X86_VENDOR_AMD:
                amd_l2cacheinfo(cpi, l2i);
                break;
        default:
                break;
        }
        return (l2i->l2i_ret);
}

#if !defined(__xpv)

uint32_t *
cpuid_mwait_alloc(cpu_t *cpu)
{
        uint32_t        *ret;
        size_t          mwait_size;

        ASSERT(cpuid_checkpass(CPU, 2));

        mwait_size = CPU->cpu_m.mcpu_cpi->cpi_mwait.mon_max;
        if (mwait_size == 0)
                return (NULL);

        /*
         * kmem_alloc() returns cache line size aligned data for mwait_size
         * allocations.  mwait_size is currently cache line sized.  Neither
         * of these implementation details are guarantied to be true in the
         * future.
         *
         * First try allocating mwait_size as kmem_alloc() currently returns
         * correctly aligned memory.  If kmem_alloc() does not return
         * mwait_size aligned memory, then use mwait_size ROUNDUP.
         *
         * Set cpi_mwait.buf_actual and cpi_mwait.size_actual in case we
         * decide to free this memory.
         */
        ret = kmem_zalloc(mwait_size, KM_SLEEP);
        if (ret == (uint32_t *)P2ROUNDUP((uintptr_t)ret, mwait_size)) {
                cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual = ret;
                cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual = mwait_size;
                *ret = MWAIT_RUNNING;
                return (ret);
        } else {
                kmem_free(ret, mwait_size);
                ret = kmem_zalloc(mwait_size * 2, KM_SLEEP);
                cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual = ret;
                cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual = mwait_size * 2;
                ret = (uint32_t *)P2ROUNDUP((uintptr_t)ret, mwait_size);
                *ret = MWAIT_RUNNING;
                return (ret);
```

```
4452         }
4453 }

4455 void
4456 cpuid_mwait_free(cpu_t *cpu)
4457 {
4458         if (cpu->cpu_m.mcpu_cpi == NULL) {
4459                 return;
4460         }

4462         if (cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual != NULL &&
4463             cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual > 0) {
4464                 kmem_free(cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual,
4465                     cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual);
4466         }

4468         cpu->cpu_m.mcpu_cpi->cpi_mwait.buf_actual = NULL;
4469         cpu->cpu_m.mcpu_cpi->cpi_mwait.size_actual = 0;
4470 }

4472 void
4473 patch_tsc_read(int flag)
4474 {
4475         size_t cnt;

4477         switch (flag) {
4478         case X86_NO_TSC:
4479                 cnt = &_no_rdtsc_end - &_no_rdtsc_start;
4480                 (void) memcpy((void *)tsc_read, (void *)&_no_rdtsc_start, cnt);
4481                 break;
4482         case X86_HAVE_TSCP:
4483                 cnt = &_tscp_end - &_tscp_start;
4484                 (void) memcpy((void *)tsc_read, (void *)&_tscp_start, cnt);
4485                 break;
4486         case X86_TSC_MFENCE:
4487                 cnt = &_tsc_mfence_end - &_tsc_mfence_start;
4488                 (void) memcpy((void *)tsc_read,
4489                     (void *)&_tsc_mfence_start, cnt);
4490                 break;
4491         case X86_TSC_LFENCE:
4492                 cnt = &_tsc_lfence_end - &_tsc_lfence_start;
4493                 (void) memcpy((void *)tsc_read,
4494                     (void *)&_tsc_lfence_start, cnt);
4495                 break;
4496         default:
4497                 break;
4498         }
4499 }

4501 int
4502 cpuid_deep_cstates_supported(void)
4503 {
4504         struct cpuid_info *cpi;
4505         struct cpuid_regs regs;

4507         ASSERT(cpuid_checkpass(CPU, 1));

4509         cpi = CPU->cpu_m.mcpu_cpi;

4511         if (!is_x86_feature(x86_featureset, X86FSET_CPUID))
4512                 return (0);

4514         switch (cpi->cpi_vendor) {
4515         case X86_VENDOR_Intel:
4516                 if (cpi->cpi_xmaxeax < 0x80000007)
4517                         return (0);
```



```
4519                 /*
4520                  * TSC run at a constant rate in all ACPI C-states?
4521                  */
4522                 regs.cp_eax = 0x80000007;
4523                 (void) __cpuid_insn(&regs);
4524                 return (regs.cp_edx & CPUID_TSC_CSTATE_INVARIANCE);

4526         default:
4527                 return (0);
4528         }
4529 }

4531 #endif  /* !__xpv */

4533 void
4534 post_startup_cpu_fixups(void)
4535 {
4536 #ifndef __xpv
4537         /*
4538          * Some AMD processors support C1E state. Entering this state will
4539          * cause the local APIC timer to stop, which we can't deal with at
4540          * this time.
4541          */
4542         if (cpuid_getvendor(CPU) == X86_VENDOR_AMD) {
4543                 on_trap_data_t otd;
4544                 uint64_t reg;

4546                 if (!on_trap(&otd, OT_DATA_ACCESS)) {
4547                         reg = rdmsr(MSR_AMD_INT_PENDING_CMP_HALT);
4548                         /* Disable C1E state if it is enabled by BIOS */
4549                         if ((reg >> AMD_ACTONCMPHALT_SHIFT) &
4550                             AMD_ACTONCMPHALT_MASK) {
4551                                 reg &= ~(AMD_ACTONCMPHALT_MASK <<
4552                                     AMD_ACTONCMPHALT_SHIFT);
4553                                 wrmsr(MSR_AMD_INT_PENDING_CMP_HALT, reg);
4554                         }
4555                 }
4556                 no_trap();
4557         }
4558 #endif  /* !__xpv */
4559 }

4561 /*
4562  * Setup necessary registers to enable XSAVE feature on this processor.
4563  * This function needs to be called early enough, so that no xsave/xrstor
4564  * ops will execute on the processor before the MSRs are properly set up.
4565  *
4566  * Current implementation has the following assumption:
4567  * - cpuid_pass1() is done, so that X86 features are known.
4568  * - fpu_probe() is done, so that fp_save_mech is chosen.
4569  */
4570 void
4571 xsave_setup_msr(cpu_t *cpu)
4572 {
4573         ASSERT(fp_save_mech == FP_XSAVE);
4574         ASSERT(is_x86_feature(x86_featureset, X86FSET_XSAVE));

4576         /* Enable OSXSAVE in CR4. */
4577         setcr4(getcr4() | CR4_OSXSAVE);
4578         /*
4579          * Update SW copy of ECX, so that /dev/cpu/self/cpuid will report
4580          * correct value.
4581          */
4582         cpu->cpu_m.mcpu_cpi->cpi_std[1].cp_ecx |= CPUID_INTC_ECX_OSXSAVE;
4583         setup_xfem();
```

```
}

/*
 * Starting with the Westmere processor the local
 * APIC timer will continue running in all C-states,
 * including the deepest C-states.
 */
int
cpuid_arat_supported(void)
{
        struct cpuid_info *cpi;
        struct cpuid_regs regs;

        ASSERT(cpuid_checkpass(CPU, 1));
        ASSERT(is_x86_feature(x86_featureset, X86FSET_CPUID));

        cpi = CPU->cpu_m.mcpu_cpi;

        switch (cpi->cpi_vendor) {
        case X86_VENDOR_Intel:
                /*
                 * Always-running Local APIC Timer is
                 * indicated by CPUID.6.EAX[2].
                 */
                if (cpi->cpi_maxeax >= 6) {
                        regs.cp_eax = 6;
                        (void) cpuid_insn(NULL, &regs);
                        return (regs.cp_eax & CPUID_CSTATE_ARAT);
                } else {
                        return (0);
                }
        default:
                return (0);
        }
}

/*
 * Check support for Intel ENERGY_PERF_BIAS feature
 */
int
cpuid_iepb_supported(struct cpu *cp)
{
        struct cpuid_info *cpi = cp->cpu_m.mcpu_cpi;
        struct cpuid_regs regs;

        ASSERT(cpuid_checkpass(cp, 1));

        if (!(is_x86_feature(x86_featureset, X86FSET_CPUID)) ||
            !(is_x86_feature(x86_featureset, X86FSET_MSR))) {
                return (0);
        }

        /*
         * Intel ENERGY_PERF_BIAS MSR is indicated by
         * capability bit CPUID.6.ECX.3
         */
        if ((cpi->cpi_vendor != X86_VENDOR_Intel) || (cpi->cpi_maxeax < 6))
                return (0);

        regs.cp_eax = 0x6;
        (void) cpuid_insn(NULL, &regs);
        return (regs.cp_ecx & CPUID_EPB_SUPPORT);
}

/*
 * Check support for TSC deadline timer
```



```
 *
 * TSC deadline timer provides a superior software programming
 * model over local APIC timer that eliminates "time drifts".
 * Instead of specifying a relative time, software specifies an
 * absolute time as the target at which the processor should
 * generate a timer event.
 */
int
cpuid_deadline_tsc_supported(void)
{
        struct cpuid_info *cpi = CPU->cpu_m.mcpu_cpi;
        struct cpuid_regs regs;

        ASSERT(cpuid_checkpass(CPU, 1));
        ASSERT(is_x86_feature(x86_featureset, X86FSET_CPUID));

        switch (cpi->cpi_vendor) {
        case X86_VENDOR_Intel:
                if (cpi->cpi_maxeax >= 1) {
                        regs.cp_eax = 1;
                        (void) cpuid_insn(NULL, &regs);
                        return (regs.cp_ecx & CPUID_DEADLINE_TSC);
                } else {
                        return (0);
                }
        default:
                return (0);
        }
}

#if defined(__amd64) && !defined(__xpv)
/*
 * Patch in versions of bcopy for high performance Intel Nhm processors
 * and later...
 */
void
patch_memops(uint_t vendor)
{
        size_t cnt, i;
        caddr_t to, from;

        if ((vendor == X86_VENDOR_Intel) &&
            is_x86_feature(x86_featureset, X86FSET_SSE4_2)) {
                cnt = &bcopy_patch_end - &bcopy_patch_start;
                to = &bcopy_ck_size;
                from = &bcopy_patch_start;
                for (i = 0; i < cnt; i++) {
                        *to++ = *from++;
                }
        }
}
#endif  /* __amd64 && !__xpv */

/*
 * This function finds the number of bits to represent the number of cores per
 * chip and the number of strands per core for the Intel platforms.
 * It re-uses the x2APIC cpuid code of the cpuid_pass2().
 */
void
cpuid_get_ext_topo(uint_t vendor, uint_t *core_nbits, uint_t *strand_nbits)
{
        struct cpuid_regs regs;
        struct cpuid_regs *cp = &regs;

        if (vendor != X86_VENDOR_Intel) {
                return;
```

```
4716         }

4718         /* if the cpuid level is 0xB, extended topo is available. */
4719         cp->cp_eax = 0;
4720         if (__cpuid_insn(cp) >= 0xB) {

4722                 cp->cp_eax = 0xB;
4723                 cp->cp_edx = cp->cp_ebx = cp->cp_ecx = 0;
4724                 (void) __cpuid_insn(cp);

4726                 /*
4727                  * Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero, which
4728                  * indicates that the extended topology enumeration leaf is
4729                  * available.
4730                  */
4731                 if (cp->cp_ebx) {
4732                         uint_t coreid_shift = 0;
4733                         uint_t chipid_shift = 0;
4734                         uint_t i;
4735                         uint_t level;

4737                         for (i = 0; i < CPI_FNB_ECX_MAX; i++) {
4738                                 cp->cp_eax = 0xB;
4739                                 cp->cp_ecx = i;

4741                                 (void) __cpuid_insn(cp);
4742                                 level = CPI_CPU_LEVEL_TYPE(cp);

4744                                 if (level == 1) {
4745                                         /*
4746                                          * Thread level processor topology
4747                                          * Number of bits shift right APIC ID
4748                                          * to get the coreid.
4749                                          */
4750                                         coreid_shift = BITX(cp->cp_eax, 4, 0);
4751                                 } else if (level == 2) {
4752                                         /*
4753                                          * Core level processor topology
4754                                          * Number of bits shift right APIC ID
4755                                          * to get the chipid.
4756                                          */
4757                                         chipid_shift = BITX(cp->cp_eax, 4, 0);
4758                                 }
4759                         }

4761                         if (coreid_shift > 0 && chipid_shift > coreid_shift) {
4762                                 *strand_nbits = coreid_shift;
4763                                 *core_nbits = chipid_shift - coreid_shift;
4764                         }
4765                 }
4766         }
4767 }
```

```
**********************************************************
   28993 Fri Apr 25 16:08:00 2014
new/usr/src/uts/intel/sys/x86_archext.h
4806 define x2apic feature flag
4807 pcplusmp & apix should use x2apic feature flag
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1995, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 by Delphix. All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright 2012 Jens Elkner <jel+illumos@cs.uni-magdeburg.de>
 * Copyright 2012 Hans Rosenfeld <rosenfeld@grumpf.hope-2000.org>
 */

#ifndef _SYS_X86_ARCHEXT_H
#define _SYS_X86_ARCHEXT_H

#if !defined(_ASM)
#include <sys/regset.h>
#include <sys/processor.h>
#include <vm/seg_enum.h>
#include <vm/page.h>
#endif  /* _ASM */

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * cpuid instruction feature flags in %edx (standard function 1)
 */

#define CPUID_INTC_EDX_FPU      0x00000001      /* x87 fpu present */
#define CPUID_INTC_EDX_VME      0x00000002      /* virtual-8086 extension */
#define CPUID_INTC_EDX_DE       0x00000004      /* debugging extensions */
#define CPUID_INTC_EDX_PSE      0x00000008      /* page size extension */
#define CPUID_INTC_EDX_TSC      0x00000010      /* time stamp counter */
#define CPUID_INTC_EDX_MSR      0x00000020      /* rdmsr and wrmsr */
```



```
#define CPUID_INTC_EDX_PAE      0x00000040      /* physical addr extension */
#define CPUID_INTC_EDX_MCE      0x00000080      /* machine check exception */
#define CPUID_INTC_EDX_CX8      0x00000100      /* cmpxchg8b instruction */
#define CPUID_INTC_EDX_APIC     0x00000200      /* local APIC */
                                                /* 0x400 - reserved */
#define CPUID_INTC_EDX_SEP      0x00000800      /* sysenter and sysexit */
#define CPUID_INTC_EDX_MTRR     0x00001000      /* memory type range reg */
#define CPUID_INTC_EDX_PGE      0x00002000      /* page global enable */
#define CPUID_INTC_EDX_MCA      0x00004000      /* machine check arch */
#define CPUID_INTC_EDX_CMOV     0x00008000      /* conditional move insns */
#define CPUID_INTC_EDX_PAT      0x00010000      /* page attribute table */
#define CPUID_INTC_EDX_PSE36    0x00020000      /* 36-bit pagesize extension */
#define CPUID_INTC_EDX_PSN      0x00040000      /* processor serial number */
#define CPUID_INTC_EDX_CLFSH    0x00080000      /* clflush instruction */
                                                /* 0x100000 - reserved */
#define CPUID_INTC_EDX_DS       0x00200000      /* debug store exists */
#define CPUID_INTC_EDX_ACPI     0x00400000      /* monitoring + clock ctrl */
#define CPUID_INTC_EDX_MMX      0x00800000      /* MMX instructions */
#define CPUID_INTC_EDX_FXSR     0x01000000      /* fxsave and fxrstor */
#define CPUID_INTC_EDX_SSE      0x02000000      /* streaming SIMD extensions */
#define CPUID_INTC_EDX_SSE2     0x04000000      /* SSE extensions */
#define CPUID_INTC_EDX_SS       0x08000000      /* self-snoop */
#define CPUID_INTC_EDX_HTT      0x10000000      /* Hyper Thread Technology */
#define CPUID_INTC_EDX_TM       0x20000000      /* thermal monitoring */
#define CPUID_INTC_EDX_IA64     0x40000000      /* Itanium emulating IA32 */
#define CPUID_INTC_EDX_PBE      0x80000000      /* Pending Break Enable */

#define FMT_CPUID_INTC_EDX                                              \
        "\20"                                                           \
        "\40pbe\37ia64\36tm\35htt\34ss\33sse2\32sse\31fxsr"     \
        "\30mmx\27acpi\26ds\24clfsh\23psn\22pse36\21pat"        \
        "\20cmov\17mca\16pge\15mtrr\14sep\12apic\11cx8"         \
        "\10mce\7pae\6msr\5tsc\4pse\3de\2vme\1fpu"

/*
 * cpuid instruction feature flags in %ecx (standard function 1)
 */

#define CPUID_INTC_ECX_SSE3     0x00000001      /* Yet more SSE extensions */
#define CPUID_INTC_ECX_PCLMULQDQ 0x00000002     /* PCLMULQDQ insn */
                                                /* 0x00000004 - reserved */
#define CPUID_INTC_ECX_MON      0x00000008      /* MONITOR/MWAIT */
#define CPUID_INTC_ECX_DSCPL    0x00000010      /* CPL-qualified debug store */
#define CPUID_INTC_ECX_VMX      0x00000020      /* Hardware VM extensions */
#define CPUID_INTC_ECX_SMX      0x00000040      /* Secure mode extensions */
#define CPUID_INTC_ECX_EST      0x00000080      /* enhanced SpeedStep */
#define CPUID_INTC_ECX_TM2      0x00000100      /* thermal monitoring */
#define CPUID_INTC_ECX_SSSE3    0x00000200      /* Supplemental SSE3 insns */
#define CPUID_INTC_ECX_CID      0x00000400      /* L1 context ID */
                                                /* 0x00000800 - reserved */
                                                /* 0x00001000 - reserved */
#define CPUID_INTC_ECX_CX16     0x00002000      /* cmpxchg16 */
#define CPUID_INTC_ECX_ETPRD    0x00004000      /* extended task pri messages */
                                                /* 0x00008000 - reserved */
                                                /* 0x00010000 - reserved */
                                                /* 0x00020000 - reserved */
#define CPUID_INTC_ECX_DCA      0x00040000      /* direct cache access */
#define CPUID_INTC_ECX_SSE4_1   0x00080000      /* SSE4.1 insns */
#define CPUID_INTC_ECX_SSE4_2   0x00100000      /* SSE4.2 insns */
#define CPUID_INTC_ECX_X2APIC   0x00200000      /* x2APIC */
#endif /* ! codereview */
#define CPUID_INTC_ECX_MOVBE    0x00400000      /* MOVBE insn */
#define CPUID_INTC_ECX_POPCNT   0x00800000      /* POPCNT insn */
#define CPUID_INTC_ECX_AES      0x02000000      /* AES insns */
#define CPUID_INTC_ECX_XSAVE    0x04000000      /* XSAVE/XRESTOR insns */
#define CPUID_INTC_ECX_OSXSAVE  0x08000000      /* OS supports XSAVE insns */
```

```
126 #define CPUID_INTC_ECX_AVX      0x10000000      /* AVX supported */
127 #define CPUID_INTC_ECX_F16C     0x20000000      /* F16C supported */
128 #define CPUID_INTC_ECX_RDRAND   0x40000000      /* RDRAND supported */
129 #define CPUID_INTC_ECX_HV       0x80000000      /* Hypervisor */

131 #define FMT_CPUID_INTC_ECX                                      \
132         "\20"                                                   \
133         "\37rdrand\36f16c\35avx\34osxsav\33xsave"               \
134         "\32aes"                                                \
135         "\30popcnt\27movbe\26x2apic\25sse4.2\24sse4.1\23dca"    \
119         "\30popcnt\27movbe\25sse4.2\24sse4.1\23dca"             \
136         "\20\17etprd\16cx16\13cid\12ssse3\11tm2"                \
137         "\10est\7smx\6vmx\5dscpl\4mon\2pclmulqdq\1sse3"

139 /*
140  * cpuid instruction feature flags in %edx (extended function 0x80000001)
141  */

143 #define CPUID_AMD_EDX_FPU       0x00000001      /* x87 fpu present */
144 #define CPUID_AMD_EDX_VME       0x00000002      /* virtual-8086 extension */
145 #define CPUID_AMD_EDX_DE        0x00000004      /* debugging extensions */
146 #define CPUID_AMD_EDX_PSE       0x00000008      /* page size extensions */
147 #define CPUID_AMD_EDX_TSC       0x00000010      /* time stamp counter */
148 #define CPUID_AMD_EDX_MSR       0x00000020      /* rdmsr and wrmsr */
149 #define CPUID_AMD_EDX_PAE       0x00000040      /* physical addr extension */
150 #define CPUID_AMD_EDX_MCE       0x00000080      /* machine check exception */
151 #define CPUID_AMD_EDX_CX8       0x00000100      /* cmpxchg8b instruction */
152 #define CPUID_AMD_EDX_APIC      0x00000200      /* local APIC */
153                                 /* 0x00000400 - sysc on K6m6 */
154 #define CPUID_AMD_EDX_SYSC      0x00000800      /* AMD: syscall and sysret */
155 #define CPUID_AMD_EDX_MTRR      0x00001000      /* memory type and range reg */
156 #define CPUID_AMD_EDX_PGE       0x00002000      /* page global enable */
157 #define CPUID_AMD_EDX_MCA       0x00004000      /* machine check arch */
158 #define CPUID_AMD_EDX_CMOV      0x00008000      /* conditional move insns */
159 #define CPUID_AMD_EDX_PAT       0x00010000      /* K7: page attribute table */
160 #define CPUID_AMD_EDX_FCMOV     0x00010000      /* FCMOVcc etc. */
161 #define CPUID_AMD_EDX_PSE36     0x00020000      /* 36-bit pagesize extension */
162                                 /* 0x00040000 - reserved */
163                                 /* 0x00080000 - reserved */
164 #define CPUID_AMD_EDX_NX        0x00100000      /* AMD: no-execute page prot */
165                                 /* 0x00200000 - reserved */
166 #define CPUID_AMD_EDX_MMXamd    0x00400000      /* AMD: MMX extensions */
167 #define CPUID_AMD_EDX_MMX       0x00800000      /* MMX instructions */
168 #define CPUID_AMD_EDX_FXSR      0x01000000      /* fxsave and fxrstor */
169 #define CPUID_AMD_EDX_FFXSR     0x02000000      /* fast fxsave/fxrstor */
170 #define CPUID_AMD_EDX_1GPG      0x04000000      /* 1GB page */
171 #define CPUID_AMD_EDX_TSCP      0x08000000      /* rdtscp instruction */
172                                 /* 0x10000000 - reserved */
173 #define CPUID_AMD_EDX_LM        0x20000000      /* AMD: long mode */
174 #define CPUID_AMD_EDX_3DNowx    0x40000000      /* AMD: extensions to 3DNow! */
175 #define CPUID_AMD_EDX_3DNow     0x80000000      /* AMD: 3DNow! instructions */

177 #define FMT_CPUID_AMD_EDX                                       \
178         "\20"                                                   \
179         "\40a3d\37a3d+\36lm\34tscp\32ffxsr\31fxsr"              \
180         "\30mmx\27mmxext\25nx\22pse\21pat"                      \
181         "\20cmov\17mca\16pge\15mtrr\14syscall\12apic\11cx8"     \
182         "\10mce\7pae\6msr\5tsc\4pse\3de\2vme\1fpu"

184 #define CPUID_AMD_ECX_AHF64     0x00000001      /* LAHF and SAHF in long mode */
185 #define CPUID_AMD_ECX_CMP_LGCY  0x00000002      /* AMD: multicore chip */
186 #define CPUID_AMD_ECX_SVM       0x00000004      /* AMD: secure VM */
187 #define CPUID_AMD_ECX_EAS       0x00000008      /* extended apic space */
188 #define CPUID_AMD_ECX_CR8D      0x00000010      /* AMD: 32-bit mov %cr8 */
189 #define CPUID_AMD_ECX_LZCNT     0x00000020      /* AMD: LZCNT insn */
190 #define CPUID_AMD_ECX_SSE4A     0x00000040      /* AMD: SSE4A insns */
```



```
191 #define CPUID_AMD_ECX_MAS       0x00000080      /* AMD: MisAlignSse mnode */
192 #define CPUID_AMD_ECX_3DNP      0x00000100      /* AMD: 3DNowPrefectch */
193 #define CPUID_AMD_ECX_OSVW      0x00000200      /* AMD: OSVW */
194 #define CPUID_AMD_ECX_IBS       0x00000400      /* AMD: IBS */
195 #define CPUID_AMD_ECX_SSE5      0x00000800      /* AMD: SSE5 */
196 #define CPUID_AMD_ECX_SKINIT    0x00001000      /* AMD: SKINIT */
197 #define CPUID_AMD_ECX_WDT       0x00002000      /* AMD: WDT */
198 #define CPUID_AMD_ECX_TOPOEXT   0x00400000      /* AMD: Topology Extensions */

200 #define FMT_CPUID_AMD_ECX                                               \
201         "\20"                                                           \
202         "\22topoext"                                                    \
203         "\14wdt\13skinit\12sse5\11ibs\10osvw\93dnp\8mas"                \
204         "\7sse4a\6lzcnt\5cr8d\3svm\2lcmplgcy\1ahf64"

206 /*
207  * Intel now seems to have claimed part of the "extended" function
208  * space that we previously for non-Intel implementors to use.
209  * More excitingly still, they've claimed bit 20 to mean LAHF/SAHF
210  * is available in long mode i.e. what AMD indicate using bit 0.
211  * On the other hand, everything else is labelled as reserved.
212  */
213 #define CPUID_INTC_ECX_AHF64    0x00100000      /* LAHF and SAHF in long mode */


216 #define P5_MCHADDR      0x0
217 #define P5_CESR         0x11
218 #define P5_CTR0         0x12
219 #define P5_CTR1         0x13

221 #define K5_MCHADDR      0x0
222 #define K5_MCHTYPE      0x01
223 #define K5_TSC          0x10
224 #define K5_TR12         0x12

226 #define REG_PAT         0x277

228 #define REG_MC0_CTL             0x400
229 #define REG_MC5_MISC            0x417
230 #define REG_PERFCTR0            0xc1
231 #define REG_PERFCTR1            0xc2

233 #define REG_PERFEVNT0           0x186
234 #define REG_PERFEVNT1           0x187

236 #define REG_TSC                 0x10    /* timestamp counter */
237 #define REG_APIC_BASE_MSR       0x1b
238 #define REG_X2APIC_BASE_MSR     0x800   /* The MSR address offset of x2APIC */

240 #if !defined(__xpv)
241 /*
242  * AMD C1E
243  */
244 #define MSR_AMD_INT_PENDING_CMP_HALT    0xC0010055
245 #define AMD_ACTONCMPHALT_SHIFT  27
246 #define AMD_ACTONCMPHALT_MASK   3
247 #endif

249 #define MSR_DEBUGCTL            0x1d9

251 #define DEBUGCTL_LBR            0x01
252 #define DEBUGCTL_BTF            0x02

254 /* Intel P6, AMD */
255 #define MSR_LBR_FROM            0x1db
256 #define MSR_LBR_TO              0x1dc
```

```
257 #define MSR_LEX_FROM            0x1dd
258 #define MSR_LEX_TO              0x1de

260 /* Intel P4 (pre-Prescott, non P4 M) */
261 #define MSR_P4_LBSTK_TOS        0x1da
262 #define MSR_P4_LBSTK_0          0x1db
263 #define MSR_P4_LBSTK_1          0x1dc
264 #define MSR_P4_LBSTK_2          0x1dd
265 #define MSR_P4_LBSTK_3          0x1de

267 /* Intel Pentium M */
268 #define MSR_P6M_LBSTK_TOS       0x1c9
269 #define MSR_P6M_LBSTK_0         0x040
270 #define MSR_P6M_LBSTK_1         0x041
271 #define MSR_P6M_LBSTK_2         0x042
272 #define MSR_P6M_LBSTK_3         0x043
273 #define MSR_P6M_LBSTK_4         0x044
274 #define MSR_P6M_LBSTK_5         0x045
275 #define MSR_P6M_LBSTK_6         0x046
276 #define MSR_P6M_LBSTK_7         0x047

278 /* Intel P4 (Prescott) */
279 #define MSR_PRP4_LBSTK_TOS      0x1da
280 #define MSR_PRP4_LBSTK_FROM_0   0x680
281 #define MSR_PRP4_LBSTK_FROM_1   0x681
282 #define MSR_PRP4_LBSTK_FROM_2   0x682
283 #define MSR_PRP4_LBSTK_FROM_3   0x683
284 #define MSR_PRP4_LBSTK_FROM_4   0x684
285 #define MSR_PRP4_LBSTK_FROM_5   0x685
286 #define MSR_PRP4_LBSTK_FROM_6   0x686
287 #define MSR_PRP4_LBSTK_FROM_7   0x687
288 #define MSR_PRP4_LBSTK_FROM_8   0x688
289 #define MSR_PRP4_LBSTK_FROM_9   0x689
290 #define MSR_PRP4_LBSTK_FROM_10  0x68a
291 #define MSR_PRP4_LBSTK_FROM_11  0x68b
292 #define MSR_PRP4_LBSTK_FROM_12  0x68c
293 #define MSR_PRP4_LBSTK_FROM_13  0x68d
294 #define MSR_PRP4_LBSTK_FROM_14  0x68e
295 #define MSR_PRP4_LBSTK_FROM_15  0x68f
296 #define MSR_PRP4_LBSTK_TO_0     0x6c0
297 #define MSR_PRP4_LBSTK_TO_1     0x6c1
298 #define MSR_PRP4_LBSTK_TO_2     0x6c2
299 #define MSR_PRP4_LBSTK_TO_3     0x6c3
300 #define MSR_PRP4_LBSTK_TO_4     0x6c4
301 #define MSR_PRP4_LBSTK_TO_5     0x6c5
302 #define MSR_PRP4_LBSTK_TO_6     0x6c6
303 #define MSR_PRP4_LBSTK_TO_7     0x6c7
304 #define MSR_PRP4_LBSTK_TO_8     0x6c8
305 #define MSR_PRP4_LBSTK_TO_9     0x6c9
306 #define MSR_PRP4_LBSTK_TO_10    0x6ca
307 #define MSR_PRP4_LBSTK_TO_11    0x6cb
308 #define MSR_PRP4_LBSTK_TO_12    0x6cc
309 #define MSR_PRP4_LBSTK_TO_13    0x6cd
310 #define MSR_PRP4_LBSTK_TO_14    0x6ce
311 #define MSR_PRP4_LBSTK_TO_15    0x6cf

313 #define MCI_CTL_VALUE           0xffffffff

315 #define MTRR_TYPE_UC            0
316 #define MTRR_TYPE_WC            1
317 #define MTRR_TYPE_WT            4
318 #define MTRR_TYPE_WP            5
319 #define MTRR_TYPE_WB            6
320 #define MTRR_TYPE_UC_           7

322 /*
```



```
323  * For Solaris we set up the page attritubute table in the following way:
324  * PAT0 Write-Back
325  * PAT1 Write-Through
326  * PAT2 Unchacheable-
327  * PAT3 Uncacheable
328  * PAT4 Write-Back
329  * PAT5 Write-Through
330  * PAT6 Write-Combine
331  * PAT7 Uncacheable
332  * The only difference from h/w default is entry 6.
333  */
334 #define PAT_DEFAULT_ATTRIBUTE                   \
335         ((uint64_t)MTRR_TYPE_WB |               \
336         ((uint64_t)MTRR_TYPE_WT << 8) |         \
337         ((uint64_t)MTRR_TYPE_UC_ << 16) |       \
338         ((uint64_t)MTRR_TYPE_UC << 24) |        \
339         ((uint64_t)MTRR_TYPE_WB << 32) |        \
340         ((uint64_t)MTRR_TYPE_WT << 40) |        \
341         ((uint64_t)MTRR_TYPE_WC << 48) |        \
342         ((uint64_t)MTRR_TYPE_UC << 56))

344 #define X86FSET_LARGEPAGE       0
345 #define X86FSET_TSC             1
346 #define X86FSET_MSR             2
347 #define X86FSET_MTRR            3
348 #define X86FSET_PGE             4
349 #define X86FSET_DE              5
350 #define X86FSET_CMOV            6
351 #define X86FSET_MMX             7
352 #define X86FSET_MCA             8
353 #define X86FSET_PAE             9
354 #define X86FSET_CX8             10
355 #define X86FSET_PAT             11
356 #define X86FSET_SEP             12
357 #define X86FSET_SSE             13
358 #define X86FSET_SSE2            14
359 #define X86FSET_HTT             15
360 #define X86FSET_ASYSC           16
361 #define X86FSET_NX              17
362 #define X86FSET_SSE3            18
363 #define X86FSET_CX16            19
364 #define X86FSET_CMP             20
365 #define X86FSET_TSCP            21
366 #define X86FSET_MWAIT           22
367 #define X86FSET_SSE4A           23
368 #define X86FSET_CPUID           24
369 #define X86FSET_SSSE3           25
370 #define X86FSET_SSE4_1          26
371 #define X86FSET_SSE4_2          27
372 #define X86FSET_1GPG            28
373 #define X86FSET_CLFSH           29
374 #define X86FSET_64              30
375 #define X86FSET_AES             31
376 #define X86FSET_PCLMULQDQ       32
377 #define X86FSET_XSAVE           33
378 #define X86FSET_AVX             34
379 #define X86FSET_VMX             35
380 #define X86FSET_SVM             36
381 #define X86FSET_TOPOEXT         37
382 #define X86FSET_F16C            38
383 #define X86FSET_RDRAND          39
384 #define X86FSET_X2APIC          40
385 #endif /* ! codereview */

387 /*
388  * flags to patch tsc_read routine.
```

```
 */
#define X86_NO_TSC              0x0
#define X86_HAVE_TSCP           0x1
#define X86_TSC_MFENCE          0x2
#define X86_TSC_LFENCE          0x4

/*
 * Intel Deep C-State invariant TSC in leaf 0x80000007.
 */
#define CPUID_TSC_CSTATE_INVARIANCE     (0x100)

/*
 * Intel Deep C-state always-running local APIC timer
 */
#define CPUID_CSTATE_ARAT       (0x4)

/*
 * Intel ENERGY_PERF_BIAS MSR indicated by feature bit CPUID.6.ECX[3].
 */
#define CPUID_EPB_SUPPORT       (1 << 3)

/*
 * Intel TSC deadline timer
 */
#define CPUID_DEADLINE_TSC      (1 << 24)

/*
 * x86_type is a legacy concept; this is supplanted
 * for most purposes by x86_featureset; modern CPUs
 * should be X86_TYPE_OTHER
 */
#define X86_TYPE_OTHER          0
#define X86_TYPE_486            1
#define X86_TYPE_P5             2
#define X86_TYPE_P6             3
#define X86_TYPE_CYRIX_486      4
#define X86_TYPE_CYRIX_6x86L    5
#define X86_TYPE_CYRIX_6x86     6
#define X86_TYPE_CYRIX_GXm      7
#define X86_TYPE_CYRIX_6x86MX   8
#define X86_TYPE_CYRIX_MediaGX  9
#define X86_TYPE_CYRIX_MII      10
#define X86_TYPE_VIA_CYRIX_III  11
#define X86_TYPE_P4             12

/*
 * x86_vendor allows us to select between
 * implementation features and helps guide
 * the interpretation of the cpuid instruction.
 */
#define X86_VENDOR_Intel        0
#define X86_VENDORSTR_Intel     "GenuineIntel"

#define X86_VENDOR_IntelClone   1

#define X86_VENDOR_AMD          2
#define X86_VENDORSTR_AMD       "AuthenticAMD"

#define X86_VENDOR_Cyrix        3
#define X86_VENDORSTR_CYRIX     "CyrixInstead"

#define X86_VENDOR_UMC          4
#define X86_VENDORSTR_UMC       "UMC UMC UMC "

#define X86_VENDOR_NexGen       5
#define X86_VENDORSTR_NexGen    "NexGenDriven"
```



```
#define X86_VENDOR_Centaur      6
#define X86_VENDORSTR_Centaur   "CentaurHauls"

#define X86_VENDOR_Rise         7
#define X86_VENDORSTR_Rise      "RiseRiseRise"

#define X86_VENDOR_SiS          8
#define X86_VENDORSTR_SiS       "SiS SiS SiS "

#define X86_VENDOR_TM           9
#define X86_VENDORSTR_TM        "GenuineTMx86"

#define X86_VENDOR_NSC          10
#define X86_VENDORSTR_NSC       "Geode by NSC"

/*
 * Vendor string max len + \0
 */
#define X86_VENDOR_STRLEN       13

/*
 * Some vendor/family/model/stepping ranges are commonly grouped under
 * a single identifying banner by the vendor.  The following encode
 * that "revision" in a uint32_t with the 8 most significant bits
 * identifying the vendor with X86_VENDOR_*, the next 8 identifying the
 * family, and the remaining 16 typically forming a bitmask of revisions
 * within that family with more significant bits indicating "later" revisions.
 */

#define _X86_CHIPREV_VENDOR_MASK        0xff000000u
#define _X86_CHIPREV_VENDOR_SHIFT       24
#define _X86_CHIPREV_FAMILY_MASK        0x00ff0000u
#define _X86_CHIPREV_FAMILY_SHIFT       16
#define _X86_CHIPREV_REV_MASK           0x0000ffffu

#define _X86_CHIPREV_VENDOR(x) \
        (((x) & _X86_CHIPREV_VENDOR_MASK) >> _X86_CHIPREV_VENDOR_SHIFT)
#define _X86_CHIPREV_FAMILY(x) \
        (((x) & _X86_CHIPREV_FAMILY_MASK) >> _X86_CHIPREV_FAMILY_SHIFT)
#define _X86_CHIPREV_REV(x) \
        ((x) & _X86_CHIPREV_REV_MASK)

/* True if x matches in vendor and family and if x matches the given rev mask */
#define X86_CHIPREV_MATCH(x, mask) \
        (_X86_CHIPREV_VENDOR(x) == _X86_CHIPREV_VENDOR(mask) && \
        _X86_CHIPREV_FAMILY(x) == _X86_CHIPREV_FAMILY(mask) && \
        ((_X86_CHIPREV_REV(x) & _X86_CHIPREV_REV(mask)) != 0))

/* True if x matches in vendor and family, and rev is at least minx */
#define X86_CHIPREV_ATLEAST(x, minx) \
        (_X86_CHIPREV_VENDOR(x) == _X86_CHIPREV_VENDOR(minx) && \
        _X86_CHIPREV_FAMILY(x) == _X86_CHIPREV_FAMILY(minx) && \
        _X86_CHIPREV_REV(x) >= _X86_CHIPREV_REV(minx))

#define _X86_CHIPREV_MKREV(vendor, family, rev) \
        ((uint32_t)(vendor) << _X86_CHIPREV_VENDOR_SHIFT | \
        (family) << _X86_CHIPREV_FAMILY_SHIFT | (rev))

/* True if x matches in vendor, and family is at least minx */
#define X86_CHIPFAM_ATLEAST(x, minx) \
        (_X86_CHIPREV_VENDOR(x) == _X86_CHIPREV_VENDOR(minx) && \
        _X86_CHIPREV_FAMILY(x) >= _X86_CHIPREV_FAMILY(minx))

/* Revision default */
#define X86_CHIPREV_UNKNOWN     0x0
```

```
522 /*
523  * Definitions for AMD Family 0xf. Minor revisions C0 and CG are
524  * sufficiently different that we will distinguish them; in all other
525  * case we will identify the major revision.
526  */
527 #define X86_CHIPREV_AMD_F_REV_B _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0001)
528 #define X86_CHIPREV_AMD_F_REV_C0 _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0002)
529 #define X86_CHIPREV_AMD_F_REV_CG _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0004)
530 #define X86_CHIPREV_AMD_F_REV_D _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0008)
531 #define X86_CHIPREV_AMD_F_REV_E _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0010)
532 #define X86_CHIPREV_AMD_F_REV_F _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0020)
533 #define X86_CHIPREV_AMD_F_REV_G _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0040)

535 /*
536  * Definitions for AMD Family 0x10.  Rev A was Engineering Samples only.
537  */
538 #define X86_CHIPREV_AMD_10_REV_A \
539         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0001)
540 #define X86_CHIPREV_AMD_10_REV_B \
541         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0002)
542 #define X86_CHIPREV_AMD_10_REV_C2 \
543         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0004)
544 #define X86_CHIPREV_AMD_10_REV_C3 \
545         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0008)
546 #define X86_CHIPREV_AMD_10_REV_D0 \
547         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0010)
548 #define X86_CHIPREV_AMD_10_REV_D1 \
549         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0020)
550 #define X86_CHIPREV_AMD_10_REV_E \
551         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0040)

553 /*
554  * Definitions for AMD Family 0x11.
555  */
556 #define X86_CHIPREV_AMD_11_REV_B \
557         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x11, 0x0002)

559 /*
560  * Definitions for AMD Family 0x12.
561  */
562 #define X86_CHIPREV_AMD_12_REV_B \
563         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x12, 0x0002)

565 /*
566  * Definitions for AMD Family 0x14.
567  */
568 #define X86_CHIPREV_AMD_14_REV_B \
569         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x14, 0x0002)
570 #define X86_CHIPREV_AMD_14_REV_C \
571         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x14, 0x0004)

573 /*
574  * Definitions for AMD Family 0x15
575  */
576 #define X86_CHIPREV_AMD_15OR_REV_B2 \
577         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x15, 0x0001)

579 #define X86_CHIPREV_AMD_15TN_REV_A1 \
580         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x15, 0x0002)

582 /*
583  * Various socket/package types, extended as the need to distinguish
584  * a new type arises.  The top 8 byte identfies the vendor and the
585  * remaining 24 bits describe 24 socket types.
586  */
```



```
588 #define _X86_SOCKET_VENDOR_SHIFT        24
589 #define _X86_SOCKET_VENDOR(x)   ((x) >> _X86_SOCKET_VENDOR_SHIFT)
590 #define _X86_SOCKET_TYPE_MASK   0x00ffffff
591 #define _X86_SOCKET_TYPE(x)             ((x) & _X86_SOCKET_TYPE_MASK)

593 #define _X86_SOCKET_MKVAL(vendor, bitval) \
594         ((uint32_t)(vendor) << _X86_SOCKET_VENDOR_SHIFT | (bitval))

596 #define X86_SOCKET_MATCH(s, mask) \
597         (_X86_SOCKET_VENDOR(s) == _X86_SOCKET_VENDOR(mask) && \
598         (_X86_SOCKET_TYPE(s) & _X86_SOCKET_TYPE(mask)) != 0)

600 #define X86_SOCKET_UNKNOWN 0x0
601         /*
602          * AMD socket types
603          */
604 #define X86_SOCKET_754          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000001)
605 #define X86_SOCKET_939          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000002)
606 #define X86_SOCKET_940          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000004)
607 #define X86_SOCKET_S1g1         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000008)
608 #define X86_SOCKET_AM2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000010)
609 #define X86_SOCKET_F1207        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000020)
610 #define X86_SOCKET_S1g2         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000040)
611 #define X86_SOCKET_S1g3         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000080)
612 #define X86_SOCKET_AM           _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000100)
613 #define X86_SOCKET_AM2R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000200)
614 #define X86_SOCKET_AM3          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000400)
615 #define X86_SOCKET_G34          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000800)
616 #define X86_SOCKET_ASB2         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x001000)
617 #define X86_SOCKET_C32          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x002000)
618 #define X86_SOCKET_S1g4         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x004000)
619 #define X86_SOCKET_FT1          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x008000)
620 #define X86_SOCKET_FM1          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x010000)
621 #define X86_SOCKET_FS1          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x020000)
622 #define X86_SOCKET_AM3R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x040000)
623 #define X86_SOCKET_FP2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x080000)
624 #define X86_SOCKET_FS1R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x100000)
625 #define X86_SOCKET_FM2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x200000)

627 /*
628  * xgetbv/xsetbv support
629  */

631 #define XFEATURE_ENABLED_MASK   0x0
632 /*
633  * XFEATURE_ENABLED_MASK values (eax)
634  */
635 #define XFEATURE_LEGACY_FP      0x1
636 #define XFEATURE_SSE            0x2
637 #define XFEATURE_AVX            0x4
638 #define XFEATURE_MAX            XFEATURE_AVX
639 #define XFEATURE_FP_ALL \
640         (XFEATURE_LEGACY_FP|XFEATURE_SSE|XFEATURE_AVX)

642 #if !defined(_ASM)

644 #if defined(_KERNEL) || defined(_KMEMUSER)

646 #define NUM_X86_FEATURES        41
368 #define NUM_X86_FEATURES        40
647 extern uchar_t x86_featureset[];

649 extern void free_x86_featureset(void *featureset);
650 extern boolean_t is_x86_feature(void *featureset, uint_t feature);
651 extern void add_x86_feature(void *featureset, uint_t feature);
```

```
 652 extern void remove_x86_feature(void *featureset, uint_t feature);
 653 extern boolean_t compare_x86_featureset(void *setA, void *setB);
 654 extern void print_x86_featureset(void *featureset);


 657 extern uint_t x86_type;
 658 extern uint_t x86_vendor;
 659 extern uint_t x86_clflush_size;

 661 extern uint_t pentiumpro_bug4046376;

 663 extern const char CyrixInstead[];

 665 #endif

 667 #if defined(_KERNEL)

 669 /*
 670  * This structure is used to pass arguments and get return values back
 671  * from the CPUID instruction in __cpuid_insn() routine.
 672  */
 673 struct cpuid_regs {
 674         uint32_t        cp_eax;
 675         uint32_t        cp_ebx;
 676         uint32_t        cp_ecx;
 677         uint32_t        cp_edx;
 678 };
```

unchanged_portion_omitted_