```
**********************************************************
  120621 Mon Apr 23 16:02:59 2012
new/usr/src/uts/i86pc/os/cpuid.c
*** NO COMMENTS ***
**********************************************************
```

```
/*
 * 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 (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Portions Copyright 2009 Advanced Micro Devices, Inc.
 */
/*
 * Copyright (c) 2011, 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;
uint_t pentiumpro_bug4064495;

uchar_t x86_featureset[BT_SIZEOFMAP(NUM_X86_FEATURES)];

static char *x86_feature_names[NUM_X86_FEATURES] = {
        "lgpg",
```

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


266 /*
267  * These constants determine how many of the elements of the
268  * cpuid we cache in the cpuid_info data structure; the
269  * remaining elements are accessible via the cpuid instruction.
270  */

272 #define NMAX_CPI_STD    6               /* eax = 0 .. 5 */
273 #define NMAX_CPI_EXTD   0x1f            /* eax = 0x80000000 .. 0x8000001e */
272 #define NMAX_CPI_EXTD   0x1c            /* eax = 0x80000000 .. 0x8000001b */

275 /*
276  * Some terminology needs to be explained:
277  *  - Socket: Something that can be plugged into a motherboard.
278  *  - Package: Same as socket
279  *  - Chip: Same as socket. Note that AMD's documentation uses term "chip"
280  *    differently: there, chip is the same as processor node (below)
281  *  - Processor node: Some AMD processors have more than one
282  *    "subprocessor" embedded in a package. These subprocessors (nodes)
283  *    are fully-functional processors themselves with cores, caches,
284  *    memory controllers, PCI configuration spaces. They are connected
285  *    inside the package with Hypertransport links. On single-node
286  *    processors, processor node is equivalent to chip/socket/package.
287  *  - Compute Unit: Some AMD processors pair cores in "compute units" that
288  *    share the FPU and the L1I and L2 caches.
```



```
289  */

291 struct cpuid_info {
292         uint_t cpi_pass;                /* last pass completed */
293         /*
294          * standard function information
295          */
296         uint_t cpi_maxeax;              /* fn 0: %eax */
297         char cpi_vendorstr[13];         /* fn 0: %ebx:%ecx:%edx */
298         uint_t cpi_vendor;              /* enum of cpi_vendorstr */

300         uint_t cpi_family;              /* fn 1: extended family */
301         uint_t cpi_model;               /* fn 1: extended model */
302         uint_t cpi_step;                /* fn 1: stepping */
303         chipid_t cpi_chipid;            /* fn 1: %ebx:  Intel: chip # */
304                                         /*              AMD: package/socket # */
305         uint_t cpi_brandid;             /* fn 1: %ebx: brand ID */
306         int cpi_clogid;                 /* fn 1: %ebx: thread # */
307         uint_t cpi_ncpu_per_chip;       /* fn 1: %ebx: logical cpu count */
308         uint8_t cpi_cacheinfo[16];      /* fn 2: intel-style cache desc */
309         uint_t cpi_ncache;              /* fn 2: number of elements */
310         uint_t cpi_ncpu_shr_last_cache; /* fn 4: %eax: ncpus sharing cache */
311         id_t cpi_last_lvl_cacheid;      /* fn 4: %eax: derived cache id */
312         uint_t cpi_std_4_size;          /* fn 4: number of fn 4 elements */
313         struct cpuid_regs **cpi_std_4;  /* fn 4: %ecx == 0 .. fn4_size */
314         struct cpuid_regs cpi_std[NMAX_CPI_STD];        /* 0 .. 5 */
315         /*
316          * extended function information
317          */
318         uint_t cpi_xmaxeax;             /* fn 0x80000000: %eax */
319         char cpi_brandstr[49];          /* fn 0x8000000[234] */
320         uint8_t cpi_pabits;             /* fn 0x80000006: %eax */
321         uint8_t cpi_vabits;             /* fn 0x80000006: %eax */
322         struct  cpuid_regs cpi_extd[NMAX_CPI_EXTD];     /* 0x800000XX */

324         id_t cpi_coreid;                /* same coreid => strands share core */
325         int cpi_pkgcoreid;              /* core number within single package */
326         uint_t cpi_ncore_per_chip;      /* AMD: fn 0x80000008: %ecx[7-0] */
327                                         /* Intel: fn 4: %eax[31-26] */
328         /*
329          * supported feature information
330          */
331         uint32_t cpi_support[5];
332 #define STD_EDX_FEATURES        0
333 #define AMD_EDX_FEATURES        1
334 #define TM_EDX_FEATURES         2
335 #define STD_ECX_FEATURES        3
336 #define AMD_ECX_FEATURES        4
337         /*
338          * Synthesized information, where known.
339          */
340         uint32_t cpi_chiprev;           /* See X86_CHIPREV_* in x86_archext.h */
341         const char *cpi_chiprevstr;     /* May be NULL if chiprev unknown */
342         uint32_t cpi_socket;            /* Chip package/socket type */

344         struct mwait_info cpi_mwait;    /* fn 5: monitor/mwait info */
345         uint32_t cpi_apicid;
346         uint_t cpi_procnodeid;          /* AMD: nodeID on HT, Intel: chipid */
347         uint_t cpi_procnodes_per_pkg;   /* AMD: # of nodes in the package */
348                                         /* Intel: 1 */
349         uint_t cpi_compunitid;          /* AMD: compute unit ID, Intel: coreid *
350         uint_t cpi_cores_per_compunit;  /* AMD: # of cores in the compute unit *

352         struct xsave_info cpi_xsave;    /* fn D: xsave/xrestor info */
353 };
______unchanged_portion_omitted_
```

```
#endif  /* __xpv */

static void
cpuid_intel_getids(cpu_t *cpu, void *feature)
{
        uint_t i;
        uint_t chipid_shift = 0;
        uint_t coreid_shift = 0;
        struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;

        for (i = 1; i < cpi->cpi_ncpu_per_chip; i <<= 1)
                chipid_shift++;

        cpi->cpi_chipid = cpi->cpi_apicid >> chipid_shift;
        cpi->cpi_clogid = cpi->cpi_apicid & ((1 << chipid_shift) - 1);

        if (is_x86_feature(feature, X86FSET_CMP)) {
                /*
                 * Multi-core (and possibly multi-threaded)
                 * processors.
                 */
                uint_t ncpu_per_core;
                if (cpi->cpi_ncore_per_chip == 1)
                        ncpu_per_core = cpi->cpi_ncpu_per_chip;
                else if (cpi->cpi_ncore_per_chip > 1)
                        ncpu_per_core = cpi->cpi_ncpu_per_chip /
                            cpi->cpi_ncore_per_chip;
                /*
                 * 8bit APIC IDs on dual core Pentiums
                 * look like this:
                 *
                 * +-----------------------+------+------+
                 * | Physical Package ID   |  MC  |  HT  |
                 * +-----------------------+------+------+
                 * <------- chipid -------->
                 * <------- coreid --------------->
                 *                         <--- clogid -->
                 *                         <------>
                 *                         pkgcoreid
                 *
                 * Where the number of bits necessary to
                 * represent MC and HT fields together equals
                 * to the minimum number of bits necessary to
                 * store the value of cpi->cpi_ncpu_per_chip.
                 * Of those bits, the MC part uses the number
                 * of bits necessary to store the value of
                 * cpi->cpi_ncore_per_chip.
                 */
                for (i = 1; i < ncpu_per_core; i <<= 1)
                        coreid_shift++;
                cpi->cpi_coreid = cpi->cpi_apicid >> coreid_shift;
                cpi->cpi_pkgcoreid = cpi->cpi_clogid >> coreid_shift;
        } else if (is_x86_feature(feature, X86FSET_HTT)) {
                /*
                 * Single-core multi-threaded processors.
                 */
                cpi->cpi_coreid = cpi->cpi_chipid;
                cpi->cpi_pkgcoreid = 0;
        }
        cpi->cpi_procnodeid = cpi->cpi_chipid;
        cpi->cpi_compunitid = cpi->cpi_coreid;
}

static void
cpuid_amd_getids(cpu_t *cpu)
```



```
{
        int i, first_half, coreidsz;
        uint32_t nb_caps_reg;
        uint_t node2_1;
        struct cpuid_info *cpi = cpu->cpu_m.mcpu_cpi;
        struct cpuid_regs *cp;

        /*
         * AMD CMP chips currently have a single thread per core.
         *
         * Since no two cpus share a core we must assign a distinct coreid
         * per cpu, and we do this by using the cpu_id.  This scheme does not,
         * however, guarantee that sibling cores of a chip will have sequential
         * coreids starting at a multiple of the number of cores per chip -
         * that is usually the case, but if the ACPI MADT table is presented
         * in a different order then we need to perform a few more gymnastics
         * for the pkgcoreid.
         *
         * All processors in the system have the same number of enabled
         * cores. Cores within a processor are always numbered sequentially
         * from 0 regardless of how many or which are disabled, and there
         * is no way for operating system to discover the real core id when some
         * are disabled.
         *
         * In family 0x15, the cores come in pairs called compute units. They
         * share L1I and L2 caches and the FPU. Enumeration of this features is
         * simplified by the new topology extensions CPUID leaf, indicated by th
         * X86 feature X86FSET_TOPOEXT.
         */

        cpi->cpi_coreid = cpu->cpu_id;
        cpi->cpi_compunitid = cpu->cpu_id;

        if (cpi->cpi_xmaxeax >= 0x80000008) {

                coreidsz = BITX((cpi)->cpi_extd[8].cp_ecx, 15, 12);

                /*
                 * In AMD parlance chip is really a node while Solaris
                 * sees chip as equivalent to socket/package.
                 */
                cpi->cpi_ncore_per_chip =
                    BITX((cpi)->cpi_extd[8].cp_ecx, 7, 0) + 1;
                if (coreidsz == 0) {
                        /* Use legacy method */
                        for (i = 1; i < cpi->cpi_ncore_per_chip; i <<= 1)
                                coreidsz++;
                        if (coreidsz == 0)
                                coreidsz = 1;
                }
        } else {
                /* Assume single-core part */
                cpi->cpi_ncore_per_chip = 1;
                coreidsz = 1;
        }

        cpi->cpi_clogid = cpi->cpi_pkgcoreid =
            cpi->cpi_apicid & ((1<<coreidsz) - 1);
        cpi->cpi_ncpu_per_chip = cpi->cpi_ncore_per_chip;

        /* Get node ID, compute unit ID */
        if (is_x86_feature(x86_featureset, X86FSET_TOPOEXT) &&
            cpi->cpi_xmaxeax >= 0x8000001e) {
                cp = &cpi->cpi_extd[0x1e];
                cp->cp_eax = 0x8000001e;
                (void) __cpuid_insn(cp);
```

```
807                 cpi->cpi_procnodes_per_pkg = BITX(cp->cp_ecx, 10, 8) + 1;
808                 cpi->cpi_procnodeid = BITX(cp->cp_ecx, 7, 0);
809                 cpi->cpi_cores_per_compunit = BITX(cp->cp_ebx, 15, 8) + 1;
810                 cpi->cpi_compunitid = BITX(cp->cp_ebx, 7, 0);
811         } else if (cpi->cpi_family == 0xf || cpi->cpi_family >= 0x11) {
787         /* Get nodeID */
788         if (cpi->cpi_family == 0xf) {
812                 cpi->cpi_procnodeid = (cpi->cpi_apicid >> coreidsz) & 7;
813                 cpi->cpi_chipid = cpi->cpi_procnodeid;
814         } else if (cpi->cpi_family == 0x10) {
815                 /*
816                  * See if we are a multi-node processor.
817                  * All processors in the system have the same number of nodes
818                  */
819                 nb_caps_reg =  pci_getl_func(0, 24, 3, 0xe8);
820                 if ((cpi->cpi_model < 8) || BITX(nb_caps_reg, 29, 29) == 0) {
821                         /* Single-node */
822                         cpi->cpi_procnodeid = BITX(cpi->cpi_apicid, 5,
823                             coreidsz);
824                         cpi->cpi_chipid = cpi->cpi_procnodeid;
825                 } else {
827                         /*
828                          * Multi-node revision D (2 nodes per package
829                          * are supported)
830                          */
831                         cpi->cpi_procnodes_per_pkg = 2;
833                         first_half = (cpi->cpi_pkgcoreid <=
834                             (cpi->cpi_ncore_per_chip/2 - 1));
836                         if (cpi->cpi_apicid == cpi->cpi_pkgcoreid) {
837                                 /* We are BSP */
838                                 cpi->cpi_procnodeid = (first_half ? 0 : 1);
839                                 cpi->cpi_chipid = cpi->cpi_procnodeid >> 1;
840                         } else {
842                                 /* We are AP */
843                                 /* NodeId[2:1] bits to use for reading F3xe8 */
844                                 node2_1 = BITX(cpi->cpi_apicid, 5, 4) << 1;
846                                 nb_caps_reg =
847                                     pci_getl_func(0, 24 + node2_1, 3, 0xe8);
849                                 /*
850                                  * Check IntNodeNum bit (31:30, but bit 31 is
851                                  * always 0 on dual-node processors)
852                                  */
853                                 if (BITX(nb_caps_reg, 30, 30) == 0)
854                                         cpi->cpi_procnodeid = node2_1 +
855                                             !first_half;
856                                 else
857                                         cpi->cpi_procnodeid = node2_1 +
858                                             first_half;
860                                 cpi->cpi_chipid = cpi->cpi_procnodeid >> 1;
861                         }
862                 }
840         } else if (cpi->cpi_family >= 0x11) {
841                 cpi->cpi_procnodeid = (cpi->cpi_apicid >> coreidsz) & 7;
842                 cpi->cpi_chipid = cpi->cpi_procnodeid;
863         } else {
864                 cpi->cpi_procnodeid = 0;
865                 cpi->cpi_chipid = cpi->cpi_procnodeid;
866         }
```



```
867 }
_____unchanged_portion_omitted_

890 void
891 cpuid_pass1(cpu_t *cpu, uchar_t *featureset)
892 {
893         uint32_t mask_ecx, mask_edx;
894         struct cpuid_info *cpi;
895         struct cpuid_regs *cp;
896         int xcpuid;
897 #if !defined(__xpv)
898         extern int idle_cpu_prefer_mwait;
899 #endif

901         /*
902          * Space statically allocated for BSP, ensure pointer is set
903          */
904         if (cpu->cpu_id == 0) {
905                 if (cpu->cpu_m.mcpu_cpi == NULL)
906                         cpu->cpu_m.mcpu_cpi = &cpuid_info0;
907         }

909         add_x86_feature(featureset, X86FSET_CPUID);

911         cpi = cpu->cpu_m.mcpu_cpi;
912         ASSERT(cpi != NULL);
913         cp = &cpi->cpi_std[0];
914         cp->cp_eax = 0;
915         cpi->cpi_maxeax = __cpuid_insn(cp);
916         {
917                 uint32_t *iptr = (uint32_t *)cpi->cpi_vendorstr;
918                 *iptr++ = cp->cp_ebx;
919                 *iptr++ = cp->cp_edx;
920                 *iptr++ = cp->cp_ecx;
921                 *(char *)&cpi->cpi_vendorstr[12] = '\0';
922         }

924         cpi->cpi_vendor = _cpuid_vendorstr_to_vendorcode(cpi->cpi_vendorstr);
925         x86_vendor = cpi->cpi_vendor; /* for compatibility */

927         /*
928          * Limit the range in case of weird hardware
929          */
930         if (cpi->cpi_maxeax > CPI_MAXEAX_MAX)
931                 cpi->cpi_maxeax = CPI_MAXEAX_MAX;
932         if (cpi->cpi_maxeax < 1)
933                 goto pass1_done;

935         cp = &cpi->cpi_std[1];
936         cp->cp_eax = 1;
937         (void) __cpuid_insn(cp);

939         /*
940          * Extract identifying constants for easy access.
941          */
942         cpi->cpi_model = CPI_MODEL(cpi);
943         cpi->cpi_family = CPI_FAMILY(cpi);

945         if (cpi->cpi_family == 0xf)
946                 cpi->cpi_family += CPI_FAMILY_XTD(cpi);

948         /*
949          * Beware: AMD uses "extended model" iff base *FAMILY* == 0xf.
950          * Intel, and presumably everyone else, uses model == 0xf, as
951          * one would expect (max value means possible overflow).  Sigh.
952          */
```

```
954         switch (cpi->cpi_vendor) {
955         case X86_VENDOR_Intel:
956                 if (IS_EXTENDED_MODEL_INTEL(cpi))
957                         cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
958                 break;
959         case X86_VENDOR_AMD:
960                 if (CPI_FAMILY(cpi) == 0xf)
961                         cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
962                 break;
963         default:
964                 if (cpi->cpi_model == 0xf)
965                         cpi->cpi_model += CPI_MODEL_XTD(cpi) << 4;
966                 break;
967         }

969         cpi->cpi_step = CPI_STEP(cpi);
970         cpi->cpi_brandid = CPI_BRANDID(cpi);

972         /*
973          * *default* assumptions:
974          * - believe %edx feature word
975          * - ignore %ecx feature word
976          * - 32-bit virtual and physical addressing
977          */
978         mask_edx = 0xffffffff;
979         mask_ecx = 0;

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

983         switch (cpi->cpi_vendor) {
984         case X86_VENDOR_Intel:
985                 if (cpi->cpi_family == 5)
986                         x86_type = X86_TYPE_P5;
987                 else if (IS_LEGACY_P6(cpi)) {
988                         x86_type = X86_TYPE_P6;
989                         pentiumpro_bug4046376 = 1;
990                         pentiumpro_bug4064495 = 1;
991                         /*
992                          * Clear the SEP bit when it was set erroneously
993                          */
994                         if (cpi->cpi_model < 3 && cpi->cpi_step < 3)
995                                 cp->cp_edx &= ~CPUID_INTC_EDX_SEP;
996                 } else if (IS_NEW_F6(cpi) || cpi->cpi_family == 0xf) {
997                         x86_type = X86_TYPE_P4;
998                         /*
999                          * We don't currently depend on any of the %ecx
1000                         * features until Prescott, so we'll only check
1001                         * this from P4 onwards.  We might want to revisit
1002                         * that idea later.
1003                         */
1004                        mask_ecx = 0xffffffff;
1005                } else if (cpi->cpi_family > 0xf)
1006                        mask_ecx = 0xffffffff;
1007                /*
1008                 * We don't support MONITOR/MWAIT if leaf 5 is not available
1009                 * to obtain the monitor linesize.
1010                 */
1011                if (cpi->cpi_maxeax < 5)
1012                        mask_ecx &= ~CPUID_INTC_ECX_MON;
1013                break;
1014        case X86_VENDOR_IntelClone:
1015        default:
1016                break;
1017        case X86_VENDOR_AMD:
1018 #if defined(OPTERON_ERRATUM_108)
```



```
1019                if (cpi->cpi_family == 0xf && cpi->cpi_model == 0xe) {
1020                        cp->cp_eax = (0xf0f & cp->cp_eax) | 0xc0;
1021                        cpi->cpi_model = 0xc;
1022                } else
1023 #endif
1024                if (cpi->cpi_family == 5) {
1025                        /*
1026                         * AMD K5 and K6
1027                         *
1028                         * These CPUs have an incomplete implementation
1029                         * of MCA/MCE which we mask away.
1030                         */
1031                        mask_edx &= ~(CPUID_INTC_EDX_MCE | CPUID_INTC_EDX_MCA);

1033                        /*
1034                         * Model 0 uses the wrong (APIC) bit
1035                         * to indicate PGE.  Fix it here.
1036                         */
1037                        if (cpi->cpi_model == 0) {
1038                                if (cp->cp_edx & 0x200) {
1039                                        cp->cp_edx &= ~0x200;
1040                                        cp->cp_edx |= CPUID_INTC_EDX_PGE;
1041                                }
1042                        }

1044                        /*
1045                         * Early models had problems w/ MMX; disable.
1046                         */
1047                        if (cpi->cpi_model < 6)
1048                                mask_edx &= ~CPUID_INTC_EDX_MMX;
1049                }

1051                /*
1052                 * For newer families, SSE3 and CX16, at least, are valid;
1053                 * enable all
1054                 */
1055                if (cpi->cpi_family >= 0xf)
1056                        mask_ecx = 0xffffffff;
1057                /*
1058                 * We don't support MONITOR/MWAIT if leaf 5 is not available
1059                 * to obtain the monitor linesize.
1060                 */
1061                if (cpi->cpi_maxeax < 5)
1062                        mask_ecx &= ~CPUID_INTC_ECX_MON;

1064 #if !defined(__xpv)
1065                /*
1066                 * Do not use MONITOR/MWAIT to halt in the idle loop on any AMD
1067                 * processors.  AMD does not intend MWAIT to be used in the cpu
1068                 * idle loop on current and future processors.  10h and future
1069                 * AMD processors use more power in MWAIT than HLT.
1070                 * Pre-family-10h Opterons do not have the MWAIT instruction.
1071                 */
1072                idle_cpu_prefer_mwait = 0;
1073 #endif

1075                break;
1076        case X86_VENDOR_TM:
1077                /*
1078                 * workaround the NT workaround in CMS 4.1
1079                 */
1080                if (cpi->cpi_family == 5 && cpi->cpi_model == 4 &&
1081                    (cpi->cpi_step == 2 || cpi->cpi_step == 3))
1082                        cp->cp_edx |= CPUID_INTC_EDX_CX8;
1083                break;
1084        case X86_VENDOR_Centaur:
```

```
1085                 /*
1086                  * workaround the NT workarounds again
1087                  */
1088                 if (cpi->cpi_family == 6)
1089                         cp->cp_edx |= CPUID_INTC_EDX_CX8;
1090                 break;
1091         case X86_VENDOR_Cyrix:
1092                 /*
1093                  * We rely heavily on the probing in locore
1094                  * to actually figure out what parts, if any,
1095                  * of the Cyrix cpuid instruction to believe.
1096                  */
1097                 switch (x86_type) {
1098                 case X86_TYPE_CYRIX_486:
1099                         mask_edx = 0;
1100                         break;
1101                 case X86_TYPE_CYRIX_6x86:
1102                         mask_edx = 0;
1103                         break;
1104                 case X86_TYPE_CYRIX_6x86L:
1105                         mask_edx =
1106                             CPUID_INTC_EDX_DE |
1107                             CPUID_INTC_EDX_CX8;
1108                         break;
1109                 case X86_TYPE_CYRIX_6x86MX:
1110                         mask_edx =
1111                             CPUID_INTC_EDX_DE |
1112                             CPUID_INTC_EDX_MSR |
1113                             CPUID_INTC_EDX_CX8 |
1114                             CPUID_INTC_EDX_PGE |
1115                             CPUID_INTC_EDX_CMOV |
1116                             CPUID_INTC_EDX_MMX;
1117                         break;
1118                 case X86_TYPE_CYRIX_GXm:
1119                         mask_edx =
1120                             CPUID_INTC_EDX_MSR |
1121                             CPUID_INTC_EDX_CX8 |
1122                             CPUID_INTC_EDX_CMOV |
1123                             CPUID_INTC_EDX_MMX;
1124                         break;
1125                 case X86_TYPE_CYRIX_MediaGX:
1126                         break;
1127                 case X86_TYPE_CYRIX_MII:
1128                 case X86_TYPE_VIA_CYRIX_III:
1129                         mask_edx =
1130                             CPUID_INTC_EDX_DE |
1131                             CPUID_INTC_EDX_TSC |
1132                             CPUID_INTC_EDX_MSR |
1133                             CPUID_INTC_EDX_CX8 |
1134                             CPUID_INTC_EDX_PGE |
1135                             CPUID_INTC_EDX_CMOV |
1136                             CPUID_INTC_EDX_MMX;
1137                         break;
1138                 default:
1139                         break;
1140                 }
1141                 break;
1142         }

1144 #if defined(__xpv)
1145         /*
1146          * Do not support MONITOR/MWAIT under a hypervisor
1147          */
1148         mask_ecx &= ~CPUID_INTC_ECX_MON;
1149         /*
1150          * Do not support XSAVE under a hypervisor for now
```



```
1151          */
1152         xsave_force_disable = B_TRUE;

1154 #endif  /* __xpv */

1156         if (xsave_force_disable) {
1157                 mask_ecx &= ~CPUID_INTC_ECX_XSAVE;
1158                 mask_ecx &= ~CPUID_INTC_ECX_AVX;
1159         }

1161         /*
1162          * Now we've figured out the masks that determine
1163          * which bits we choose to believe, apply the masks
1164          * to the feature words, then map the kernel's view
1165          * of these feature words into its feature word.
1166          */
1167         cp->cp_edx &= mask_edx;
1168         cp->cp_ecx &= mask_ecx;

1170         /*
1171          * apply any platform restrictions (we don't call this
1172          * immediately after __cpuid_insn here, because we need the
1173          * workarounds applied above first)
1174          */
1175         platform_cpuid_mangle(cpi->cpi_vendor, 1, cp);

1177         /*
1178          * fold in overrides from the "eeprom" mechanism
1179          */
1180         cp->cp_edx |= cpuid_feature_edx_include;
1181         cp->cp_edx &= ~cpuid_feature_edx_exclude;

1183         cp->cp_ecx |= cpuid_feature_ecx_include;
1184         cp->cp_ecx &= ~cpuid_feature_ecx_exclude;

1186         if (cp->cp_edx & CPUID_INTC_EDX_PSE) {
1187                 add_x86_feature(featureset, X86FSET_LARGEPAGE);
1188         }
1189         if (cp->cp_edx & CPUID_INTC_EDX_TSC) {
1190                 add_x86_feature(featureset, X86FSET_TSC);
1191         }
1192         if (cp->cp_edx & CPUID_INTC_EDX_MSR) {
1193                 add_x86_feature(featureset, X86FSET_MSR);
1194         }
1195         if (cp->cp_edx & CPUID_INTC_EDX_MTRR) {
1196                 add_x86_feature(featureset, X86FSET_MTRR);
1197         }
1198         if (cp->cp_edx & CPUID_INTC_EDX_PGE) {
1199                 add_x86_feature(featureset, X86FSET_PGE);
1200         }
1201         if (cp->cp_edx & CPUID_INTC_EDX_CMOV) {
1202                 add_x86_feature(featureset, X86FSET_CMOV);
1203         }
1204         if (cp->cp_edx & CPUID_INTC_EDX_MMX) {
1205                 add_x86_feature(featureset, X86FSET_MMX);
1206         }
1207         if ((cp->cp_edx & CPUID_INTC_EDX_MCE) != 0 &&
1208             (cp->cp_edx & CPUID_INTC_EDX_MCA) != 0) {
1209                 add_x86_feature(featureset, X86FSET_MCA);
1210         }
1211         if (cp->cp_edx & CPUID_INTC_EDX_PAE) {
1212                 add_x86_feature(featureset, X86FSET_PAE);
1213         }
1214         if (cp->cp_edx & CPUID_INTC_EDX_CX8) {
1215                 add_x86_feature(featureset, X86FSET_CX8);
1216         }
```

```
1217         if (cp->cp_ecx & CPUID_INTC_ECX_CX16) {
1218                 add_x86_feature(featureset, X86FSET_CX16);
1219         }
1220         if (cp->cp_edx & CPUID_INTC_EDX_PAT) {
1221                 add_x86_feature(featureset, X86FSET_PAT);
1222         }
1223         if (cp->cp_edx & CPUID_INTC_EDX_SEP) {
1224                 add_x86_feature(featureset, X86FSET_SEP);
1225         }
1226         if (cp->cp_edx & CPUID_INTC_EDX_FXSR) {
1227                 /*
1228                  * In our implementation, fxsave/fxrstor
1229                  * are prerequisites before we'll even
1230                  * try and do SSE things.
1231                  */
1232                 if (cp->cp_edx & CPUID_INTC_EDX_SSE) {
1233                         add_x86_feature(featureset, X86FSET_SSE);
1234                 }
1235                 if (cp->cp_edx & CPUID_INTC_EDX_SSE2) {
1236                         add_x86_feature(featureset, X86FSET_SSE2);
1237                 }
1238                 if (cp->cp_ecx & CPUID_INTC_ECX_SSE3) {
1239                         add_x86_feature(featureset, X86FSET_SSE3);
1240                 }
1241                 if (cp->cp_ecx & CPUID_INTC_ECX_SSSE3) {
1242                         add_x86_feature(featureset, X86FSET_SSSE3);
1243                 }
1244                 if (cp->cp_ecx & CPUID_INTC_ECX_SSE4_1) {
1245                         add_x86_feature(featureset, X86FSET_SSE4_1);
1246                 }
1247                 if (cp->cp_ecx & CPUID_INTC_ECX_SSE4_2) {
1248                         add_x86_feature(featureset, X86FSET_SSE4_2);
1249                 }
1250                 if (cp->cp_ecx & CPUID_INTC_ECX_AES) {
1251                         add_x86_feature(featureset, X86FSET_AES);
1252                 }
1253                 if (cp->cp_ecx & CPUID_INTC_ECX_PCLMULQDQ) {
1254                         add_x86_feature(featureset, X86FSET_PCLMULQDQ);
1255                 }

1257                 if (cp->cp_ecx & CPUID_INTC_ECX_XSAVE) {
1258                         add_x86_feature(featureset, X86FSET_XSAVE);
1259                         /* We only test AVX when there is XSAVE */
1260                         if (cp->cp_ecx & CPUID_INTC_ECX_AVX) {
1261                                 add_x86_feature(featureset,
1262                                     X86FSET_AVX);
1263                         }
1264                 }
1265         }
1266         if (cp->cp_edx & CPUID_INTC_EDX_DE) {
1267                 add_x86_feature(featureset, X86FSET_DE);
1268         }
1269 #if !defined(__xpv)
1270         if (cp->cp_ecx & CPUID_INTC_ECX_MON) {

1272                 /*
1273                  * We require the CLFLUSH instruction for erratum workaround
1274                  * to use MONITOR/MWAIT.
1275                  */
1276                 if (cp->cp_edx & CPUID_INTC_EDX_CLFSH) {
1277                         cpi->cpi_mwait.support |= MWAIT_SUPPORT;
1278                         add_x86_feature(featureset, X86FSET_MWAIT);
1279                 } else {
1280                         extern int idle_cpu_assert_cflush_monitor;

1282                         /*
```



```
1283                          * All processors we are aware of which have
1284                          * MONITOR/MWAIT also have CLFLUSH.
1285                          */
1286                         if (idle_cpu_assert_cflush_monitor) {
1287                                 ASSERT((cp->cp_ecx & CPUID_INTC_ECX_MON) &&
1288                                     (cp->cp_edx & CPUID_INTC_EDX_CLFSH));
1289                         }
1290                 }
1291         }
1292 #endif  /* __xpv */

1294         if (cp->cp_ecx & CPUID_INTC_ECX_VMX) {
1295                 add_x86_feature(featureset, X86FSET_VMX);
1296         }

1298         /*
1299          * Only need it first time, rest of the cpus would follow suit.
1300          * we only capture this for the bootcpu.
1301          */
1302         if (cp->cp_edx & CPUID_INTC_EDX_CLFSH) {
1303                 add_x86_feature(featureset, X86FSET_CLFSH);
1304                 x86_clflush_size = (BITX(cp->cp_ebx, 15, 8) * 8);
1305         }
1306         if (is_x86_feature(featureset, X86FSET_PAE))
1307                 cpi->cpi_pabits = 36;

1309         /*
1310          * Hyperthreading configuration is slightly tricky on Intel
1311          * and pure clones, and even trickier on AMD.
1312          *
1313          * (AMD chose to set the HTT bit on their CMP processors,
1314          * even though they're not actually hyperthreaded.  Thus it
1315          * takes a bit more work to figure out what's really going
1316          * on ... see the handling of the CMP_LGCY bit below)
1317          */
1318         if (cp->cp_edx & CPUID_INTC_EDX_HTT) {
1319                 cpi->cpi_ncpu_per_chip = CPI_CPU_COUNT(cpi);
1320                 if (cpi->cpi_ncpu_per_chip > 1)
1321                         add_x86_feature(featureset, X86FSET_HTT);
1322         } else {
1323                 cpi->cpi_ncpu_per_chip = 1;
1324         }

1326         /*
1327          * Work on the "extended" feature information, doing
1328          * some basic initialization for cpuid_pass2()
1329          */
1330         xcpuid = 0;
1331         switch (cpi->cpi_vendor) {
1332         case X86_VENDOR_Intel:
1333                 if (IS_NEW_F6(cpi) || cpi->cpi_family >= 0xf)
1334                         xcpuid++;
1335                 break;
1336         case X86_VENDOR_AMD:
1337                 if (cpi->cpi_family > 5 ||
1338                     (cpi->cpi_family == 5 && cpi->cpi_model >= 1))
1339                         xcpuid++;
1340                 break;
1341         case X86_VENDOR_Cyrix:
1342                 /*
1343                  * Only these Cyrix CPUs are -known- to support
1344                  * extended cpuid operations.
1345                  */
1346                 if (x86_type == X86_TYPE_VIA_CYRIX_III ||
1347                     x86_type == X86_TYPE_CYRIX_GXm)
1348                         xcpuid++;
```

```
1349                 break;
1350         case X86_VENDOR_Centaur:
1351         case X86_VENDOR_TM:
1352         default:
1353                 xcpuid++;
1354                 break;
1355         }

1357         if (xcpuid) {
1358                 cp = &cpi->cpi_extd[0];
1359                 cp->cp_eax = 0x80000000;
1360                 cpi->cpi_xmaxeax = __cpuid_insn(cp);
1361         }

1363         if (cpi->cpi_xmaxeax & 0x80000000) {

1365                 if (cpi->cpi_xmaxeax > CPI_XMAXEAX_MAX)
1366                         cpi->cpi_xmaxeax = CPI_XMAXEAX_MAX;

1368                 switch (cpi->cpi_vendor) {
1369                 case X86_VENDOR_Intel:
1370                 case X86_VENDOR_AMD:
1371                         if (cpi->cpi_xmaxeax < 0x80000001)
1372                                 break;
1373                         cp = &cpi->cpi_extd[1];
1374                         cp->cp_eax = 0x80000001;
1375                         (void) __cpuid_insn(cp);

1377                         if (cpi->cpi_vendor == X86_VENDOR_AMD &&
1378                             cpi->cpi_family == 5 &&
1379                             cpi->cpi_model == 6 &&
1380                             cpi->cpi_step == 6) {
1381                                 /*
1382                                  * K6 model 6 uses bit 10 to indicate SYSC
1383                                  * Later models use bit 11. Fix it here.
1384                                  */
1385                                 if (cp->cp_edx & 0x400) {
1386                                         cp->cp_edx &= ~0x400;
1387                                         cp->cp_edx |= CPUID_AMD_EDX_SYSC;
1388                                 }
1389                         }

1391                         platform_cpuid_mangle(cpi->cpi_vendor, 0x80000001, cp);

1393                         /*
1394                          * Compute the additions to the kernel's feature word.
1395                          */
1396                         if (cp->cp_edx & CPUID_AMD_EDX_NX) {
1397                                 add_x86_feature(featureset, X86FSET_NX);
1398                         }

1400                         /*
1401                          * Regardless whether or not we boot 64-bit,
1402                          * we should have a way to identify whether
1403                          * the CPU is capable of running 64-bit.
1404                          */
1405                         if (cp->cp_edx & CPUID_AMD_EDX_LM) {
1406                                 add_x86_feature(featureset, X86FSET_64);
1407                         }

1409 #if defined(__amd64)
1410                         /* 1 GB large page - enable only for 64 bit kernel */
1411                         if (cp->cp_edx & CPUID_AMD_EDX_1GPG) {
1412                                 add_x86_feature(featureset, X86FSET_1GPG);
1413                         }
1414 #endif
```



```
1416                         if ((cpi->cpi_vendor == X86_VENDOR_AMD) &&
1417                             (cpi->cpi_std[1].cp_edx & CPUID_INTC_EDX_FXSR) &&
1418                             (cp->cp_ecx & CPUID_AMD_ECX_SSE4A)) {
1419                                 add_x86_feature(featureset, X86FSET_SSE4A);
1420                         }

1422                         /*
1423                          * If both the HTT and CMP_LGCY bits are set,
1424                          * then we're not actually HyperThreaded.  Read
1425                          * "AMD CPUID Specification" for more details.
1426                          */
1427                         if (cpi->cpi_vendor == X86_VENDOR_AMD &&
1428                             is_x86_feature(featureset, X86FSET_HTT) &&
1429                             (cp->cp_ecx & CPUID_AMD_ECX_CMP_LGCY)) {
1430                                 remove_x86_feature(featureset, X86FSET_HTT);
1431                                 add_x86_feature(featureset, X86FSET_CMP);
1432                         }
1433 #if defined(__amd64)
1434                         /*
1435                          * It's really tricky to support syscall/sysret in
1436                          * the i386 kernel; we rely on sysenter/sysexit
1437                          * instead.  In the amd64 kernel, things are -way-
1438                          * better.
1439                          */
1440                         if (cp->cp_edx & CPUID_AMD_EDX_SYSC) {
1441                                 add_x86_feature(featureset, X86FSET_ASYSC);
1442                         }

1444                         /*
1445                          * While we're thinking about system calls, note
1446                          * that AMD processors don't support sysenter
1447                          * in long mode at all, so don't try to program them.
1448                          */
1449                         if (x86_vendor == X86_VENDOR_AMD) {
1450                                 remove_x86_feature(featureset, X86FSET_SEP);
1451                         }
1452 #endif
1453                         if (cp->cp_edx & CPUID_AMD_EDX_TSCP) {
1454                                 add_x86_feature(featureset, X86FSET_TSCP);
1455                         }

1457                         if (cp->cp_ecx & CPUID_AMD_ECX_SVM) {
1458                                 add_x86_feature(featureset, X86FSET_SVM);
1459                         }

1461                         if (cp->cp_ecx & CPUID_AMD_ECX_TOPOEXT) {
1462                                 add_x86_feature(featureset, X86FSET_TOPOEXT);
1463                         }
1464                         break;
1465                 default:
1466                         break;
1467                 }

1469                 /*
1470                  * Get CPUID data about processor cores and hyperthreads.
1471                  */
1472                 switch (cpi->cpi_vendor) {
1473                 case X86_VENDOR_Intel:
1474                         if (cpi->cpi_maxeax >= 4) {
1475                                 cp = &cpi->cpi_std[4];
1476                                 cp->cp_eax = 4;
1477                                 cp->cp_ecx = 0;
1478                                 (void) __cpuid_insn(cp);
1479                                 platform_cpuid_mangle(cpi->cpi_vendor, 4, cp);
1480                         }
```

```
1481                    /*FALLTHROUGH*/
1482            case X86_VENDOR_AMD:
1483                    if (cpi->cpi_xmaxeax < 0x80000008)
1484                            break;
1485                    cp = &cpi->cpi_extd[8];
1486                    cp->cp_eax = 0x80000008;
1487                    (void) __cpuid_insn(cp);
1488                    platform_cpuid_mangle(cpi->cpi_vendor, 0x80000008, cp);

1490                    /*
1491                     * Virtual and physical address limits from
1492                     * cpuid override previously guessed values.
1493                     */
1494                    cpi->cpi_pabits = BITX(cp->cp_eax, 7, 0);
1495                    cpi->cpi_vabits = BITX(cp->cp_eax, 15, 8);
1496                    break;
1497            default:
1498                    break;
1499            }

1501            /*
1502             * Derive the number of cores per chip
1503             */
1504            switch (cpi->cpi_vendor) {
1505            case X86_VENDOR_Intel:
1506                    if (cpi->cpi_maxeax < 4) {
1507                            cpi->cpi_ncore_per_chip = 1;
1508                            break;
1509                    } else {
1510                            cpi->cpi_ncore_per_chip =
1511                                BITX((cpi)->cpi_std[4].cp_eax, 31, 26) + 1;
1512                    }
1513                    break;
1514            case X86_VENDOR_AMD:
1515                    if (cpi->cpi_xmaxeax < 0x80000008) {
1516                            cpi->cpi_ncore_per_chip = 1;
1517                            break;
1518                    } else {
1519                            /*
1520                             * On family 0xf cpuid fn 2 ECX[7:0] "NC" is
1521                             * 1 less than the number of physical cores on
1522                             * the chip.  In family 0x10 this value can
1523                             * be affected by "downcoring" - it reflects
1524                             * 1 less than the number of cores actually
1525                             * enabled on this node.
1526                             */
1527                            cpi->cpi_ncore_per_chip =
1528                                BITX((cpi)->cpi_extd[8].cp_ecx, 7, 0) + 1;
1529                    }
1530                    break;
1531            default:
1532                    cpi->cpi_ncore_per_chip = 1;
1533                    break;
1534            }

1536            /*
1537             * Get CPUID data about TSC Invariance in Deep C-State.
1538             */
1539            switch (cpi->cpi_vendor) {
1540            case X86_VENDOR_Intel:
1541                    if (cpi->cpi_maxeax >= 7) {
1542                            cp = &cpi->cpi_extd[7];
1543                            cp->cp_eax = 0x80000007;
1544                            cp->cp_ecx = 0;
1545                            (void) __cpuid_insn(cp);
1546                    }
```



```
1547                            break;
1548                    default:
1549                            break;
1550                    }
1551            } else {
1552                    cpi->cpi_ncore_per_chip = 1;
1553            }

1555            /*
1556             * If more than one core, then this processor is CMP.
1557             */
1558            if (cpi->cpi_ncore_per_chip > 1) {
1559                    add_x86_feature(featureset, X86FSET_CMP);
1560            }

1562            /*
1563             * If the number of cores is the same as the number
1564             * of CPUs, then we cannot have HyperThreading.
1565             */
1566            if (cpi->cpi_ncpu_per_chip == cpi->cpi_ncore_per_chip) {
1567                    remove_x86_feature(featureset, X86FSET_HTT);
1568            }

1570            cpi->cpi_apicid = CPI_APIC_ID(cpi);
1571            cpi->cpi_procnodes_per_pkg = 1;
1572            cpi->cpi_cores_per_compunit = 1;
1573            if (is_x86_feature(featureset, X86FSET_HTT) == B_FALSE &&
1574                is_x86_feature(featureset, X86FSET_CMP) == B_FALSE) {
1575                    /*
1576                     * Single-core single-threaded processors.
1577                     */
1578                    cpi->cpi_chipid = -1;
1579                    cpi->cpi_clogid = 0;
1580                    cpi->cpi_coreid = cpu->cpu_id;
1581                    cpi->cpi_pkgcoreid = 0;
1582                    if (cpi->cpi_vendor == X86_VENDOR_AMD)
1583                            cpi->cpi_procnodeid = BITX(cpi->cpi_apicid, 3, 0);
1584                    else
1585                            cpi->cpi_procnodeid = cpi->cpi_chipid;
1586            } else if (cpi->cpi_ncpu_per_chip > 1) {
1587                    if (cpi->cpi_vendor == X86_VENDOR_Intel)
1588                            cpuid_intel_getids(cpu, featureset);
1589                    else if (cpi->cpi_vendor == X86_VENDOR_AMD)
1590                            cpuid_amd_getids(cpu);
1591                    else {
1592                            /*
1593                             * All other processors are currently
1594                             * assumed to have single cores.
1595                             */
1596                            cpi->cpi_coreid = cpi->cpi_chipid;
1597                            cpi->cpi_pkgcoreid = 0;
1598                            cpi->cpi_procnodeid = cpi->cpi_chipid;
1599                            cpi->cpi_compunitid = cpi->cpi_chipid;
1600                    }
1601            }

1603            /*
1604             * Synthesize chip "revision" and socket type
1605             */
1606            cpi->cpi_chiprev = _cpuid_chiprev(cpi->cpi_vendor, cpi->cpi_family,
1607                cpi->cpi_model, cpi->cpi_step);
1608            cpi->cpi_chiprevstr = _cpuid_chiprevstr(cpi->cpi_vendor,
1609                cpi->cpi_family, cpi->cpi_model, cpi->cpi_step);
1610            cpi->cpi_socket = _cpuid_skt(cpi->cpi_vendor, cpi->cpi_family,
1611                cpi->cpi_model, cpi->cpi_step);
```

```
1613 pass1_done:
1614         cpi->cpi_pass = 1;
1615 }
```

*unchanged_portion_omitted_*

```
3029 uint_t
3030 cpuid_get_compunitid(cpu_t *cpu)
3031 {
3032         ASSERT(cpuid_checkpass(cpu, 1));
3033         return (cpu->cpu_m.mcpu_cpi->cpi_compunitid);
3034 }

3036 uint_t
3037 cpuid_get_cores_per_compunit(cpu_t *cpu)
3038 {
3039         ASSERT(cpuid_checkpass(cpu, 1));
3040         return (cpu->cpu_m.mcpu_cpi->cpi_cores_per_compunit);
3041 }

3043 /*ARGSUSED*/
3044 int
3045 cpuid_have_cr8access(cpu_t *cpu)
3046 {
3047 #if defined(__amd64)
3048         return (1);
3049 #else
3050         struct cpuid_info *cpi;

3052         ASSERT(cpu != NULL);
3053         cpi = cpu->cpu_m.mcpu_cpi;
3054         if (cpi->cpi_vendor == X86_VENDOR_AMD && cpi->cpi_maxeax >= 1 &&
3055             (CPI_FEATURES_XTD_ECX(cpi) & CPUID_AMD_ECX_CR8D) != 0)
3056                 return (1);
3057         return (0);
3058 #endif
3059 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   48131 Mon Apr 23 16:03:00 2012
new/usr/src/uts/i86pc/os/mp_machdep.c
*** NO COMMENTS ***
**********************************************************
```

```
   2 /*
   3  * CDDL HEADER START
   4  *
   5  * The contents of this file are subject to the terms of the
   6  * Common Development and Distribution License (the "License").
   7  * You may not use this file except in compliance with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */
  22 /*
  23  * Copyright (c) 1992, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */
  25 /*
  26  * Copyright (c) 2009-2010, Intel Corporation.
  27  * All rights reserved.
  28  */

  30 #define PSMI_1_7
  31 #include <sys/smp_impldefs.h>
  32 #include <sys/psm.h>
  33 #include <sys/psm_modctl.h>
  34 #include <sys/pit.h>
  35 #include <sys/cmn_err.h>
  36 #include <sys/strlog.h>
  37 #include <sys/clock.h>
  38 #include <sys/debug.h>
  39 #include <sys/rtc.h>
  40 #include <sys/x86_archext.h>
  41 #include <sys/cpupart.h>
  42 #include <sys/cpuvar.h>
  43 #include <sys/cpu_event.h>
  44 #include <sys/cmt.h>
  45 #include <sys/cpu.h>
  46 #include <sys/disp.h>
  47 #include <sys/archsystm.h>
  48 #include <sys/machsystm.h>
  49 #include <sys/sysmacros.h>
  50 #include <sys/memlist.h>
  51 #include <sys/param.h>
  52 #include <sys/promif.h>
  53 #include <sys/cpu_pm.h>
  54 #if defined(__xpv)
  55 #include <sys/hypervisor.h>
  56 #endif
  57 #include <sys/mach_intr.h>
  58 #include <vm/hat_i86.h>
  59 #include <sys/kdi_machimpl.h>
  60 #include <sys/sdt.h>
  61 #include <sys/hpet.h>
```



```
  62 #include <sys/sunddi.h>
  63 #include <sys/sunndi.h>
  64 #include <sys/cpc_pcbe.h>

  66 #define OFFSETOF(s, m)          (size_t)(&(((s *)0)->m))

  68 /*
  69  *      Local function prototypes
  70  */
  71 static int mp_disable_intr(processorid_t cpun);
  72 static void mp_enable_intr(processorid_t cpun);
  73 static void mach_init();
  74 static void mach_picinit();
  75 static int machhztomhz(uint64_t cpu_freq_hz);
  76 static uint64_t mach_getcpufreq(void);
  77 static void mach_fixcpufreq(void);
  78 static int mach_clkinit(int, int *);
  79 static void mach_smpinit(void);
  80 static int mach_softlvl_to_vect(int ipl);
  81 static void mach_get_platform(int owner);
  82 static void mach_construct_info();
  83 static int mach_translate_irq(dev_info_t *dip, int irqno);
  84 static int mach_intr_ops(dev_info_t *, ddi_intr_handle_impl_t *,
  85     psm_intr_op_t, int *);
  86 static void mach_notify_error(int level, char *errmsg);
  87 static hrtime_t dummy_hrtime(void);
  88 static void dummy_scalehrtime(hrtime_t *);
  89 static uint64_t dummy_unscalehrtime(hrtime_t);
  90 void cpu_idle(void);
  91 static void cpu_wakeup(cpu_t *, int);
  92 #ifndef __xpv
  93 void cpu_idle_mwait(void);
  94 static void cpu_wakeup_mwait(cpu_t *, int);
  95 #endif
  96 static int mach_cpu_create_devinfo(cpu_t *cp, dev_info_t **dipp);

  98 /*
  99  *      External reference functions
 100  */
 101 extern void return_instr();
 102 extern uint64_t freq_tsc(uint32_t *);
 103 #if defined(__i386)
 104 extern uint64_t freq_notsc(uint32_t *);
 105 #endif
 106 extern void pc_gethrestime(timestruc_t *);
 107 extern int cpuid_get_coreid(cpu_t *);
 108 extern int cpuid_get_chipid(cpu_t *);

 110 /*
 111  *      PSM functions initialization
 112  */
 113 void (*psm_shutdownf)(int, int) = (void (*)(int, int))return_instr;
 114 void (*psm_preshutdownf)(int, int) = (void (*)(int, int))return_instr;
 115 void (*psm_notifyf)(int)        = (void (*)(int))return_instr;
 116 void (*psm_set_idle_cpuf)(int)  = (void (*)(int))return_instr;
 117 void (*psm_unset_idle_cpuf)(int) = (void (*)(int))return_instr;
 118 void (*psminitf)()              = mach_init;
 119 void (*picinitf)()              = return_instr;
 120 int (*clkinitf)(int, int *)     = (int (*)(int, int *))return_instr;
 121 int (*ap_mlsetup)()             = (int (*)(void))return_instr;
 122 void (*send_dirintf)()          = return_instr;
 123 void (*setspl)(int)             = (void (*)(int))return_instr;
 124 int (*addspl)(int, int, int, int) = (int (*)(int, int, int, int))return_instr;
 125 int (*delspl)(int, int, int, int) = (int (*)(int, int, int, int))return_instr;
 126 int (*get_pending_spl)(void)    = (int (*)(void))return_instr;
 127 int (*addintr)(void *, int, avfunc, char *, int, caddr_t, caddr_t,
```

```
128     uint64_t *, dev_info_t *) = NULL;
129 void (*remintr)(void *, int, avfunc, int) = NULL;
130 void (*kdisetsoftint)(int, struct av_softinfo *)=
131         (void (*)(int, struct av_softinfo *))return_instr;
132 void (*setsoftint)(int, struct av_softinfo *)=
133         (void (*)(int, struct av_softinfo *))return_instr;
134 int (*slvltovect)(int)          = (int (*)(int))return_instr;
135 int (*setlvl)(int, int *)       = (int (*)(int, int *))return_instr;
136 void (*setlvlx)(int, int)       = (void (*)(int, int))return_instr;
137 int (*psm_disable_intr)(int)    = mp_disable_intr;
138 void (*psm_enable_intr)(int)    = mp_enable_intr;
139 hrtime_t (*gethrtimef)(void)    = dummy_hrtime;
140 hrtime_t (*gethrtimeunscaledf)(void)    = dummy_hrtime;
141 void (*scalehrtimef)(hrtime_t *)        = dummy_scalehrtime;
142 uint64_t (*unscalehrtimef)(hrtime_t)    = dummy_unscalehrtime;
143 int (*psm_translate_irq)(dev_info_t *, int) = mach_translate_irq;
144 void (*gethrestimef)(timestruc_t *) = pc_gethrestime;
145 void (*psm_notify_error)(int, char *) = (void (*)(int, char *))NULL;
146 int (*psm_get_clockirq)(int) = NULL;
147 int (*psm_get_ipivect)(int, int) = NULL;
148 uchar_t (*psm_get_ioapicid)(uchar_t) = NULL;
149 uint32_t (*psm_get_localapicid)(uint32_t) = NULL;
150 uchar_t (*psm_xlate_vector_by_irq)(uchar_t) = NULL;

152 int (*psm_clkinit)(int) = NULL;
153 void (*psm_timer_reprogram)(hrtime_t) = NULL;
154 void (*psm_timer_enable)(void) = NULL;
155 void (*psm_timer_disable)(void) = NULL;
156 void (*psm_post_cyclic_setup)(void *arg) = NULL;
157 int (*psm_intr_ops)(dev_info_t *, ddi_intr_handle_impl_t *, psm_intr_op_t,
158     int *) = mach_intr_ops;
159 int (*psm_state)(psm_state_request_t *) = (int (*)(psm_state_request_t *))
160     return_instr;

162 void (*notify_error)(int, char *) = (void (*)(int, char *))return_instr;
163 void (*hrtime_tick)(void)       = return_instr;

165 int (*psm_cpu_create_devinfo)(cpu_t *, dev_info_t **) = mach_cpu_create_devinfo;
166 int (*psm_cpu_get_devinfo)(cpu_t *, dev_info_t **) = NULL;

168 /* global IRM pool for APIX (PSM) module */
169 ddi_irm_pool_t *apix_irm_pool_p = NULL;

171 /*
172  * True if the generic TSC code is our source of hrtime, rather than whatever
173  * the PSM can provide.
174  */
175 #ifdef __xpv
176 int tsc_gethrtime_enable = 0;
177 #else
178 int tsc_gethrtime_enable = 1;
179 #endif
180 int tsc_gethrtime_initted = 0;

182 /*
183  * True if the hrtime implementation is "hires"; namely, better than microdata.
184  */
185 int gethrtime_hires = 0;

187 /*
188  * Local Static Data
189  */
190 static struct psm_ops mach_ops;
191 static struct psm_ops *mach_set[4] = {&mach_ops, NULL, NULL, NULL};
192 static ushort_t mach_ver[4] = {0, 0, 0, 0};
```



```
194 /*
195  * virtualization support for psm
196  */
197 void *psm_vt_ops = NULL;
198 /*
199  * If non-zero, idle cpus will become "halted" when there's
200  * no work to do.
201  */
202 int     idle_cpu_use_hlt = 1;

204 #ifndef __xpv
205 /*
206  * If non-zero, idle cpus will use mwait if available to halt instead of hlt.
207  */
208 int     idle_cpu_prefer_mwait = 1;
209 /*
210  * Set to 0 to avoid MONITOR+CLFLUSH assertion.
211  */
212 int     idle_cpu_assert_cflush_monitor = 1;

214 /*
215  * If non-zero, idle cpus will not use power saving Deep C-States idle loop.
216  */
217 int     idle_cpu_no_deep_c = 0;
218 /*
219  * Non-power saving idle loop and wakeup pointers.
220  * Allows user to toggle Deep Idle power saving feature on/off.
221  */
222 void    (*non_deep_idle_cpu)() = cpu_idle;
223 void    (*non_deep_idle_disp_enq_thread)(cpu_t *, int);

225 /*
226  * Object for the kernel to access the HPET.
227  */
228 hpet_t hpet;

230 #endif  /* ifndef __xpv */

232 uint_t cp_haltset_fanout = 0;

234 /*ARGSUSED*/
235 int
236 pg_plat_hw_shared(cpu_t *cp, pghw_type_t hw)
237 {
238         switch (hw) {
239         case PGHW_IPIPE:
240                 if (is_x86_feature(x86_featureset, X86FSET_HTT)) {
241                         /*
242                          * Hyper-threading is SMT
243                          */
244                         return (1);
245                 } else {
246                         return (0);
247                 }
248         case PGHW_FPU:
249                 if (cpuid_get_cores_per_compunit(cp) > 1)
250                         return (1);
251                 else
252                         return (0);
253         case PGHW_PROCNODE:
254                 if (cpuid_get_procnodes_per_pkg(cp) > 1)
255                         return (1);
256                 else
257                         return (0);
258         case PGHW_CHIP:
259                 if (is_x86_feature(x86_featureset, X86FSET_CMP) ||
```

```
260                     is_x86_feature(x86_featureset, X86FSET_HTT))
261                         return (1);
262                 else
263                         return (0);
264         case PGHW_CACHE:
265                 if (cpuid_get_ncpu_sharing_last_cache(cp) > 1)
266                         return (1);
267                 else
268                         return (0);
269         case PGHW_POW_ACTIVE:
270                 if (cpupm_domain_id(cp, CPUPM_DTYPE_ACTIVE) != (id_t)-1)
271                         return (1);
272                 else
273                         return (0);
274         case PGHW_POW_IDLE:
275                 if (cpupm_domain_id(cp, CPUPM_DTYPE_IDLE) != (id_t)-1)
276                         return (1);
277                 else
278                         return (0);
279         default:
280                 return (0);
281         }
282 }
```

*unchanged_portion_omitted_*

```
302 /*
303  * Return a physical instance identifier for known hardware sharing
304  * relationships
305  */
306 id_t
307 pg_plat_hw_instance_id(cpu_t *cpu, pghw_type_t hw)
308 {
309         switch (hw) {
310         case PGHW_IPIPE:
311                 return (cpuid_get_coreid(cpu));
312         case PGHW_CACHE:
313                 return (cpuid_get_last_lvl_cacheid(cpu));
314         case PGHW_FPU:
315                 return (cpuid_get_compunitid(cpu));
316         case PGHW_PROCNODE:
317                 return (cpuid_get_procnodeid(cpu));
318         case PGHW_CHIP:
319                 return (cpuid_get_chipid(cpu));
320         case PGHW_POW_ACTIVE:
321                 return (cpupm_domain_id(cpu, CPUPM_DTYPE_ACTIVE));
322         case PGHW_POW_IDLE:
323                 return (cpupm_domain_id(cpu, CPUPM_DTYPE_IDLE));
324         default:
325                 return (-1);
326         }
327 }

329 /*
330  * Express preference for optimizing for sharing relationship
331  * hw1 vs hw2
332  */
333 pghw_type_t
334 pg_plat_hw_rank(pghw_type_t hw1, pghw_type_t hw2)
335 {
336         int i, rank1, rank2;

338         static pghw_type_t hw_hier[] = {
339                 PGHW_IPIPE,
340                 PGHW_CACHE,
341                 PGHW_FPU,
342                 PGHW_PROCNODE,
```



```
343                 PGHW_CHIP,
344                 PGHW_POW_IDLE,
345                 PGHW_POW_ACTIVE,
346                 PGHW_NUM_COMPONENTS
347         };

349         for (i = 0; hw_hier[i] != PGHW_NUM_COMPONENTS; i++) {
350                 if (hw_hier[i] == hw1)
351                         rank1 = i;
352                 if (hw_hier[i] == hw2)
353                         rank2 = i;
354         }

356         if (rank1 > rank2)
357                 return (hw1);
358         else
359                 return (hw2);
360 }

362 /*
363  * Override the default CMT dispatcher policy for the specified
364  * hardware sharing relationship
365  */
366 pg_cmt_policy_t
367 pg_plat_cmt_policy(pghw_type_t hw)
368 {
369         /*
370          * For shared caches, also load balance across them to
371          * maximize aggregate cache capacity
372          *
373          * On AMD family 0x15 CPUs, cores come in pairs called
374          * compute units, sharing the FPU and the L1I and L2
375          * caches. Use balancing and cache affinity.
376          */
377         switch (hw) {
378         case PGHW_FPU:
379         case PGHW_CACHE:
380                 return (CMT_BALANCE|CMT_AFFINITY);
381         default:
382                 return (CMT_NO_POLICY);
383         }
384 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   26594 Mon Apr 23 16:03:01 2012
new/usr/src/uts/intel/sys/x86_archext.h
*** NO COMMENTS ***
**********************************************************
```

```
/*
 * 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 (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 */

#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_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 */
#define CPUID_INTC_ECX_AVX      0x10000000      /* AVX supported */

#define FMT_CPUID_INTC_ECX                                              \
        "\20"                                                           \
        "\35avx\34osxsav\33xsave"                                       \
        "\32aes"                                                        \
        "\30popcnt\27movbe\25sse4.2\24sse4.1\23dca"                     \
```

```
128         "\20\17etprd\16cx16\13cid\12ssse3\11tm2"                \
129         "\10est\7smx\6vmx\5dscpl\4mon\2pclmulqdq\1sse3"

131 /*
132  * cpuid instruction feature flags in %edx (extended function 0x80000001)
133  */

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

169 #define FMT_CPUID_AMD_EDX                                       \
170         "\20"                                                   \
171         "\40a3d\37a3d+\36lm\34tscp\32ffxsr\31fxsr"              \
172         "\30mmx\27mmxext\25nx\22pse\21pat"                      \
173         "\20cmov\17mca\16pge\15mtrr\14syscall\12apic\11cx8"     \
174         "\10mce\7pae\6msr\5tsc\4pse\3de\2vme\1fpu"

176 #define CPUID_AMD_ECX_AHF64     0x00000001      /* LAHF and SAHF in long mode */
177 #define CPUID_AMD_ECX_CMP_LGCY  0x00000002      /* AMD: multicore chip */
178 #define CPUID_AMD_ECX_SVM       0x00000004      /* AMD: secure VM */
179 #define CPUID_AMD_ECX_EAS       0x00000008      /* extended apic space */
180 #define CPUID_AMD_ECX_CR8D      0x00000010      /* AMD: 32-bit mov %cr8 */
181 #define CPUID_AMD_ECX_LZCNT     0x00000020      /* AMD: LZCNT insn */
182 #define CPUID_AMD_ECX_SSE4A     0x00000040      /* AMD: SSE4A insns */
183 #define CPUID_AMD_ECX_MAS       0x00000080      /* AMD: MisAlignSse mnode */
184 #define CPUID_AMD_ECX_3DNP      0x00000100      /* AMD: 3DNowPrefectch */
185 #define CPUID_AMD_ECX_OSVW      0x00000200      /* AMD: OSVW */
186 #define CPUID_AMD_ECX_IBS       0x00000400      /* AMD: IBS */
187 #define CPUID_AMD_ECX_SSE5      0x00000800      /* AMD: SSE5 */
188 #define CPUID_AMD_ECX_SKINIT    0x00001000      /* AMD: SKINIT */
189 #define CPUID_AMD_ECX_WDT       0x00002000      /* AMD: WDT */
190 #define CPUID_AMD_ECX_TOPOEXT   0x00400000      /* AMD: Topology Extensions */

192 #define FMT_CPUID_AMD_ECX                                       \
193         "\20"                                                   \
```



```
194         "\22topoext"                                            \
195         "\14wdt\13skinit\12sse5\11ibs\10osvw\93dnp\8mas"        \
196         "\7sse4a\6lzcnt\5cr8d\3svm\2lcmplgcy\1ahf64"

198 /*
199  * Intel now seems to have claimed part of the "extended" function
200  * space that we previously for non-Intel implementors to use.
201  * More excitingly still, they've claimed bit 20 to mean LAHF/SAHF
202  * is available in long mode i.e. what AMD indicate using bit 0.
203  * On the other hand, everything else is labelled as reserved.
204  */
205 #define CPUID_INTC_ECX_AHF64    0x00100000      /* LAHF and SAHF in long mode */


208 #define P5_MCHADDR      0x0
209 #define P5_CESR         0x11
210 #define P5_CTR0         0x12
211 #define P5_CTR1         0x13

213 #define K5_MCHADDR      0x0
214 #define K5_MCHTYPE      0x01
215 #define K5_TSC          0x10
216 #define K5_TR12         0x12

218 #define REG_PAT         0x277

220 #define REG_MC0_CTL             0x400
221 #define REG_MC5_MISC            0x417
222 #define REG_PERFCTR0            0xc1
223 #define REG_PERFCTR1            0xc2

225 #define REG_PERFEVNT0           0x186
226 #define REG_PERFEVNT1           0x187

228 #define REG_TSC                 0x10    /* timestamp counter */
229 #define REG_APIC_BASE_MSR       0x1b
230 #define REG_X2APIC_BASE_MSR     0x800   /* The MSR address offset of x2APIC */

232 #if !defined(__xpv)
233 /*
234  * AMD C1E
235  */
236 #define MSR_AMD_INT_PENDING_CMP_HALT    0xC0010055
237 #define AMD_ACTONCMPHALT_SHIFT  27
238 #define AMD_ACTONCMPHALT_MASK   3
239 #endif

241 #define MSR_DEBUGCTL            0x1d9

243 #define DEBUGCTL_LBR            0x01
244 #define DEBUGCTL_BTF            0x02

246 /* Intel P6, AMD */
247 #define MSR_LBR_FROM            0x1db
248 #define MSR_LBR_TO              0x1dc
249 #define MSR_LEX_FROM            0x1dd
250 #define MSR_LEX_TO              0x1de

252 /* Intel P4 (pre-Prescott, non P4 M) */
253 #define MSR_P4_LBSTK_TOS        0x1da
254 #define MSR_P4_LBSTK_0          0x1db
255 #define MSR_P4_LBSTK_1          0x1dc
256 #define MSR_P4_LBSTK_2          0x1dd
257 #define MSR_P4_LBSTK_3          0x1de

259 /* Intel Pentium M */
```

```
#define MSR_P6M_LBSTK_TOS       0x1c9
#define MSR_P6M_LBSTK_0         0x040
#define MSR_P6M_LBSTK_1         0x041
#define MSR_P6M_LBSTK_2         0x042
#define MSR_P6M_LBSTK_3         0x043
#define MSR_P6M_LBSTK_4         0x044
#define MSR_P6M_LBSTK_5         0x045
#define MSR_P6M_LBSTK_6         0x046
#define MSR_P6M_LBSTK_7         0x047

/* Intel P4 (Prescott) */
#define MSR_PRP4_LBSTK_TOS      0x1da
#define MSR_PRP4_LBSTK_FROM_0   0x680
#define MSR_PRP4_LBSTK_FROM_1   0x681
#define MSR_PRP4_LBSTK_FROM_2   0x682
#define MSR_PRP4_LBSTK_FROM_3   0x683
#define MSR_PRP4_LBSTK_FROM_4   0x684
#define MSR_PRP4_LBSTK_FROM_5   0x685
#define MSR_PRP4_LBSTK_FROM_6   0x686
#define MSR_PRP4_LBSTK_FROM_7   0x687
#define MSR_PRP4_LBSTK_FROM_8   0x688
#define MSR_PRP4_LBSTK_FROM_9   0x689
#define MSR_PRP4_LBSTK_FROM_10  0x68a
#define MSR_PRP4_LBSTK_FROM_11  0x68b
#define MSR_PRP4_LBSTK_FROM_12  0x68c
#define MSR_PRP4_LBSTK_FROM_13  0x68d
#define MSR_PRP4_LBSTK_FROM_14  0x68e
#define MSR_PRP4_LBSTK_FROM_15  0x68f
#define MSR_PRP4_LBSTK_TO_0     0x6c0
#define MSR_PRP4_LBSTK_TO_1     0x6c1
#define MSR_PRP4_LBSTK_TO_2     0x6c2
#define MSR_PRP4_LBSTK_TO_3     0x6c3
#define MSR_PRP4_LBSTK_TO_4     0x6c4
#define MSR_PRP4_LBSTK_TO_5     0x6c5
#define MSR_PRP4_LBSTK_TO_6     0x6c6
#define MSR_PRP4_LBSTK_TO_7     0x6c7
#define MSR_PRP4_LBSTK_TO_8     0x6c8
#define MSR_PRP4_LBSTK_TO_9     0x6c9
#define MSR_PRP4_LBSTK_TO_10    0x6ca
#define MSR_PRP4_LBSTK_TO_11    0x6cb
#define MSR_PRP4_LBSTK_TO_12    0x6cc
#define MSR_PRP4_LBSTK_TO_13    0x6cd
#define MSR_PRP4_LBSTK_TO_14    0x6ce
#define MSR_PRP4_LBSTK_TO_15    0x6cf

#define MCI_CTL_VALUE           0xffffffff

#define MTRR_TYPE_UC            0
#define MTRR_TYPE_WC            1
#define MTRR_TYPE_WT            4
#define MTRR_TYPE_WP            5
#define MTRR_TYPE_WB            6
#define MTRR_TYPE_UC_           7

/*
 * For Solaris we set up the page attritubute table in the following way:
 * PAT0 Write-Back
 * PAT1 Write-Through
 * PAT2 Unchacheable-
 * PAT3 Uncacheable
 * PAT4 Write-Back
 * PAT5 Write-Through
 * PAT6 Write-Combine
 * PAT7 Uncacheable
 * The only difference from h/w default is entry 6.
 */
```



```
#define PAT_DEFAULT_ATTRIBUTE                   \
        ((uint64_t)MTRR_TYPE_WB |               \
        ((uint64_t)MTRR_TYPE_WT << 8) |         \
        ((uint64_t)MTRR_TYPE_UC_ << 16) |       \
        ((uint64_t)MTRR_TYPE_UC << 24) |        \
        ((uint64_t)MTRR_TYPE_WB << 32) |        \
        ((uint64_t)MTRR_TYPE_WT << 40) |        \
        ((uint64_t)MTRR_TYPE_WC << 48) |        \
        ((uint64_t)MTRR_TYPE_UC << 56))

#define X86FSET_LARGEPAGE       0
#define X86FSET_TSC             1
#define X86FSET_MSR             2
#define X86FSET_MTRR            3
#define X86FSET_PGE             4
#define X86FSET_DE              5
#define X86FSET_CMOV            6
#define X86FSET_MMX             7
#define X86FSET_MCA             8
#define X86FSET_PAE             9
#define X86FSET_CX8             10
#define X86FSET_PAT             11
#define X86FSET_SEP             12
#define X86FSET_SSE             13
#define X86FSET_SSE2            14
#define X86FSET_HTT             15
#define X86FSET_ASYSC           16
#define X86FSET_NX              17
#define X86FSET_SSE3            18
#define X86FSET_CX16            19
#define X86FSET_CMP             20
#define X86FSET_TSCP            21
#define X86FSET_MWAIT           22
#define X86FSET_SSE4A           23
#define X86FSET_CPUID           24
#define X86FSET_SSSE3           25
#define X86FSET_SSE4_1          26
#define X86FSET_SSE4_2          27
#define X86FSET_1GPG            28
#define X86FSET_CLFSH           29
#define X86FSET_64              30
#define X86FSET_AES             31
#define X86FSET_PCLMULQDQ       32
#define X86FSET_XSAVE           33
#define X86FSET_AVX             34
#define X86FSET_VMX             35
#define X86FSET_SVM             36
#define X86FSET_TOPOEXT         37

/*
 * 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)
```

```
393 /*
394  * Intel ENERGY_PERF_BIAS MSR indicated by feature bit CPUID.6.ECX[3].
395  */
396 #define CPUID_EPB_SUPPORT       (1 << 3)

398 /*
399  * Intel TSC deadline timer
400  */
401 #define CPUID_DEADLINE_TSC      (1 << 24)

403 /*
404  * x86_type is a legacy concept; this is supplanted
405  * for most purposes by x86_featureset; modern CPUs
406  * should be X86_TYPE_OTHER
407  */
408 #define X86_TYPE_OTHER          0
409 #define X86_TYPE_486            1
410 #define X86_TYPE_P5             2
411 #define X86_TYPE_P6             3
412 #define X86_TYPE_CYRIX_486      4
413 #define X86_TYPE_CYRIX_6x86L    5
414 #define X86_TYPE_CYRIX_6x86     6
415 #define X86_TYPE_CYRIX_GXm      7
416 #define X86_TYPE_CYRIX_6x86MX   8
417 #define X86_TYPE_CYRIX_MediaGX  9
418 #define X86_TYPE_CYRIX_MII      10
419 #define X86_TYPE_VIA_CYRIX_III  11
420 #define X86_TYPE_P4             12

422 /*
423  * x86_vendor allows us to select between
424  * implementation features and helps guide
425  * the interpretation of the cpuid instruction.
426  */
427 #define X86_VENDOR_Intel        0
428 #define X86_VENDORSTR_Intel     "GenuineIntel"

430 #define X86_VENDOR_IntelClone   1

432 #define X86_VENDOR_AMD          2
433 #define X86_VENDORSTR_AMD       "AuthenticAMD"

435 #define X86_VENDOR_Cyrix        3
436 #define X86_VENDORSTR_CYRIX     "CyrixInstead"

438 #define X86_VENDOR_UMC          4
439 #define X86_VENDORSTR_UMC       "UMC UMC UMC "

441 #define X86_VENDOR_NexGen       5
442 #define X86_VENDORSTR_NexGen    "NexGenDriven"

444 #define X86_VENDOR_Centaur      6
445 #define X86_VENDORSTR_Centaur   "CentaurHauls"

447 #define X86_VENDOR_Rise         7
448 #define X86_VENDORSTR_Rise      "RiseRiseRise"

450 #define X86_VENDOR_SiS          8
451 #define X86_VENDORSTR_SiS       "SiS SiS SiS "

453 #define X86_VENDOR_TM           9
454 #define X86_VENDORSTR_TM        "GenuineTMx86"

456 #define X86_VENDOR_NSC          10
457 #define X86_VENDORSTR_NSC       "Geode by NSC"
```



```
459 /*
460  * Vendor string max len + \0
461  */
462 #define X86_VENDOR_STRLEN       13

464 /*
465  * Some vendor/family/model/stepping ranges are commonly grouped under
466  * a single identifying banner by the vendor.  The following encode
467  * that "revision" in a uint32_t with the 8 most significant bits
468  * identifying the vendor with X86_VENDOR_*, the next 8 identifying the
469  * family, and the remaining 16 typically forming a bitmask of revisions
470  * within that family with more significant bits indicating "later" revisions.
471  */

473 #define _X86_CHIPREV_VENDOR_MASK        0xff000000u
474 #define _X86_CHIPREV_VENDOR_SHIFT       24
475 #define _X86_CHIPREV_FAMILY_MASK        0x00ff0000u
476 #define _X86_CHIPREV_FAMILY_SHIFT       16
477 #define _X86_CHIPREV_REV_MASK           0x0000ffffu

479 #define _X86_CHIPREV_VENDOR(x) \
480         (((x) & _X86_CHIPREV_VENDOR_MASK) >> _X86_CHIPREV_VENDOR_SHIFT)
481 #define _X86_CHIPREV_FAMILY(x) \
482         (((x) & _X86_CHIPREV_FAMILY_MASK) >> _X86_CHIPREV_FAMILY_SHIFT)
483 #define _X86_CHIPREV_REV(x) \
484         ((x) & _X86_CHIPREV_REV_MASK)

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

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

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

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

507 /* Revision default */
508 #define X86_CHIPREV_UNKNOWN     0x0

510 /*
511  * Definitions for AMD Family 0xf. Minor revisions C0 and CG are
512  * sufficiently different that we will distinguish them; in all other
513  * case we will identify the major revision.
514  */
515 #define X86_CHIPREV_AMD_F_REV_B _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0001)
516 #define X86_CHIPREV_AMD_F_REV_C0 _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0002)
517 #define X86_CHIPREV_AMD_F_REV_CG _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0004)
518 #define X86_CHIPREV_AMD_F_REV_D _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0008)
519 #define X86_CHIPREV_AMD_F_REV_E _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0010)
520 #define X86_CHIPREV_AMD_F_REV_F _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0020)
521 #define X86_CHIPREV_AMD_F_REV_G _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0xf, 0x0040)

523 /*
```

```
524  * Definitions for AMD Family 0x10.  Rev A was Engineering Samples only.
525  */
526 #define X86_CHIPREV_AMD_10_REV_A \
527         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0001)
528 #define X86_CHIPREV_AMD_10_REV_B \
529         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0002)
530 #define X86_CHIPREV_AMD_10_REV_C \
531         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0004)
532 #define X86_CHIPREV_AMD_10_REV_D \
533         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x10, 0x0008)

535 /*
536  * Definitions for AMD Family 0x11.
537  */
538 #define X86_CHIPREV_AMD_11 \
539         _X86_CHIPREV_MKREV(X86_VENDOR_AMD, 0x11, 0x0001)


542 /*
543  * Various socket/package types, extended as the need to distinguish
544  * a new type arises.  The top 8 byte identfies the vendor and the
545  * remaining 24 bits describe 24 socket types.
546  */

548 #define _X86_SOCKET_VENDOR_SHIFT        24
549 #define _X86_SOCKET_VENDOR(x)   ((x) >> _X86_SOCKET_VENDOR_SHIFT)
550 #define _X86_SOCKET_TYPE_MASK   0x00ffffff
551 #define _X86_SOCKET_TYPE(x)             ((x) & _X86_SOCKET_TYPE_MASK)

553 #define _X86_SOCKET_MKVAL(vendor, bitval) \
554         ((uint32_t)(vendor) << _X86_SOCKET_VENDOR_SHIFT | (bitval))

556 #define X86_SOCKET_MATCH(s, mask) \
557         (_X86_SOCKET_VENDOR(s) == _X86_SOCKET_VENDOR(mask) && \
558         (_X86_SOCKET_TYPE(s) & _X86_SOCKET_TYPE(mask)) != 0)

560 #define X86_SOCKET_UNKNOWN 0x0
561         /*
562          * AMD socket types
563          */
564 #define X86_SOCKET_754          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000001)
565 #define X86_SOCKET_939          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000002)
566 #define X86_SOCKET_940          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000004)
567 #define X86_SOCKET_S1g1         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000008)
568 #define X86_SOCKET_AM2          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000010)
569 #define X86_SOCKET_F1207        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000020)
570 #define X86_SOCKET_S1g2         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000040)
571 #define X86_SOCKET_S1g3         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000080)
572 #define X86_SOCKET_AM           _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000100)
573 #define X86_SOCKET_AM2R2        _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000200)
574 #define X86_SOCKET_AM3          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000400)
575 #define X86_SOCKET_G34          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x000800)
576 #define X86_SOCKET_ASB2         _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x001000)
577 #define X86_SOCKET_C32          _X86_SOCKET_MKVAL(X86_VENDOR_AMD, 0x002000)

579 /*
580  * xgetbv/xsetbv support
581  */

583 #define XFEATURE_ENABLED_MASK   0x0
584 /*
585  * XFEATURE_ENABLED_MASK values (eax)
586  */
587 #define XFEATURE_LEGACY_FP      0x1
588 #define XFEATURE_SSE            0x2
589 #define XFEATURE_AVX            0x4
```



```
590 #define XFEATURE_MAX            XFEATURE_AVX
591 #define XFEATURE_FP_ALL         (XFEATURE_LEGACY_FP|XFEATURE_SSE|XFEATURE_AVX)

593 #if !defined(_ASM)

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

597 #define NUM_X86_FEATURES        38
594 #define NUM_X86_FEATURES        37
598 extern uchar_t x86_featureset[];

600 extern void free_x86_featureset(void *featureset);
601 extern boolean_t is_x86_feature(void *featureset, uint_t feature);
602 extern void add_x86_feature(void *featureset, uint_t feature);
603 extern void remove_x86_feature(void *featureset, uint_t feature);
604 extern boolean_t compare_x86_featureset(void *setA, void *setB);
605 extern void print_x86_featureset(void *featureset);


608 extern uint_t x86_type;
609 extern uint_t x86_vendor;
610 extern uint_t x86_clflush_size;

612 extern uint_t pentiumpro_bug4046376;
613 extern uint_t pentiumpro_bug4064495;

615 extern uint_t enable486;

617 extern const char CyrixInstead[];

619 #endif

621 #if defined(_KERNEL)

623 /*
624  * This structure is used to pass arguments and get return values back
625  * from the CPUID instruction in __cpuid_insn() routine.
626  */
627 struct cpuid_regs {
628         uint32_t        cp_eax;
629         uint32_t        cp_ebx;
630         uint32_t        cp_ecx;
631         uint32_t        cp_edx;
632 };

634 /*
635  * Utility functions to get/set extended control registers (XCR)
636  * Initial use is to get/set the contents of the XFEATURE_ENABLED_MASK.
637  */
638 extern uint64_t get_xcr(uint_t);
639 extern void set_xcr(uint_t, uint64_t);

641 extern uint64_t rdmsr(uint_t);
642 extern void wrmsr(uint_t, const uint64_t);
643 extern uint64_t xrdmsr(uint_t);
644 extern void xwrmsr(uint_t, const uint64_t);
645 extern int checked_rdmsr(uint_t, uint64_t *);
646 extern int checked_wrmsr(uint_t, uint64_t);

648 extern void invalidate_cache(void);
649 extern ulong_t getcr4(void);
650 extern void setcr4(ulong_t);

652 extern void mtrr_sync(void);

654 extern void cpu_fast_syscall_enable(void *);
```

```
655 extern void cpu_fast_syscall_disable(void *);

657 struct cpu;

659 extern int cpuid_checkpass(struct cpu *, int);
660 extern uint32_t cpuid_insn(struct cpu *, struct cpuid_regs *);
661 extern uint32_t __cpuid_insn(struct cpuid_regs *);
662 extern int cpuid_getbrandstr(struct cpu *, char *, size_t);
663 extern int cpuid_getidstr(struct cpu *, char *, size_t);
664 extern const char *cpuid_getvendorstr(struct cpu *);
665 extern uint_t cpuid_getvendor(struct cpu *);
666 extern uint_t cpuid_getfamily(struct cpu *);
667 extern uint_t cpuid_getmodel(struct cpu *);
668 extern uint_t cpuid_getstep(struct cpu *);
669 extern uint_t cpuid_getsig(struct cpu *);
670 extern uint_t cpuid_get_ncpu_per_chip(struct cpu *);
671 extern uint_t cpuid_get_ncore_per_chip(struct cpu *);
672 extern uint_t cpuid_get_ncpu_sharing_last_cache(struct cpu *);
673 extern id_t cpuid_get_last_lvl_cacheid(struct cpu *);
674 extern int cpuid_get_chipid(struct cpu *);
675 extern id_t cpuid_get_coreid(struct cpu *);
676 extern int cpuid_get_pkgcoreid(struct cpu *);
677 extern int cpuid_get_clogid(struct cpu *);
678 extern int cpuid_get_cacheid(struct cpu *);
679 extern uint32_t cpuid_get_apicid(struct cpu *);
680 extern uint_t cpuid_get_procnodeid(struct cpu *cpu);
681 extern uint_t cpuid_get_procnodes_per_pkg(struct cpu *cpu);
682 extern uint_t cpuid_get_compunitid(struct cpu *cpu);
683 extern uint_t cpuid_get_cores_per_compunit(struct cpu *cpu);
684 extern int cpuid_is_cmt(struct cpu *);
685 extern int cpuid_syscall32_insn(struct cpu *);
686 extern int getl2cacheinfo(struct cpu *, int *, int *, int *);

688 extern uint32_t cpuid_getchiprev(struct cpu *);
689 extern const char *cpuid_getchiprevstr(struct cpu *);
690 extern uint32_t cpuid_getsockettype(struct cpu *);
691 extern const char *cpuid_getsocketstr(struct cpu *);

693 extern int cpuid_have_cr8access(struct cpu *);

695 extern int cpuid_opteron_erratum(struct cpu *, uint_t);

697 struct cpuid_info;

699 extern void setx86isalist(void);
700 extern void cpuid_alloc_space(struct cpu *);
701 extern void cpuid_free_space(struct cpu *);
702 extern void cpuid_pass1(struct cpu *, uchar_t *);
703 extern void cpuid_pass2(struct cpu *);
704 extern void cpuid_pass3(struct cpu *);
705 extern uint_t cpuid_pass4(struct cpu *);
706 extern void cpuid_set_cpu_properties(void *, processorid_t,
707     struct cpuid_info *);

709 extern void cpuid_get_addrsize(struct cpu *, uint_t *, uint_t *);
710 extern uint_t cpuid_get_dtlb_nent(struct cpu *, size_t);

712 #if !defined(__xpv)
713 extern uint32_t *cpuid_mwait_alloc(struct cpu *);
714 extern void cpuid_mwait_free(struct cpu *);
715 extern int cpuid_deep_cstates_supported(void);
716 extern int cpuid_arat_supported(void);
717 extern int cpuid_iepb_supported(struct cpu *);
718 extern int cpuid_deadline_tsc_supported(void);
719 extern int vmware_platform(void);
720 #endif
```



```
722 struct cpu_ucode_info;

724 extern void ucode_alloc_space(struct cpu *);
725 extern void ucode_free_space(struct cpu *);
726 extern void ucode_check(struct cpu *);
727 extern void ucode_cleanup();

729 #if !defined(__xpv)
730 extern  char _tsc_mfence_start;
731 extern  char _tsc_mfence_end;
732 extern  char _tscp_start;
733 extern  char _tscp_end;
734 extern  char _no_rdtsc_start;
735 extern  char _no_rdtsc_end;
736 extern  char _tsc_lfence_start;
737 extern  char _tsc_lfence_end;
738 #endif

740 #if !defined(__xpv)
741 extern  char bcopy_patch_start;
742 extern  char bcopy_patch_end;
743 extern  char bcopy_ck_size;
744 #endif

746 extern void post_startup_cpu_fixups(void);

748 extern uint_t workaround_errata(struct cpu *);

750 #if defined(OPTERON_ERRATUM_93)
751 extern int opteron_erratum_93;
752 #endif

754 #if defined(OPTERON_ERRATUM_91)
755 extern int opteron_erratum_91;
756 #endif

758 #if defined(OPTERON_ERRATUM_100)
759 extern int opteron_erratum_100;
760 #endif

762 #if defined(OPTERON_ERRATUM_121)
763 extern int opteron_erratum_121;
764 #endif

766 #if defined(OPTERON_WORKAROUND_6323525)
767 extern int opteron_workaround_6323525;
768 extern void patch_workaround_6323525(void);
769 #endif

771 #if !defined(__xpv)
772 extern void determine_platform(void);
773 #endif
774 extern int get_hwenv(void);
775 extern int is_controldom(void);

777 extern void xsave_setup_msr(struct cpu *);

779 /*
780  * Defined hardware environments
781  */
782 #define HW_NATIVE       0x00    /* Running on bare metal */
783 #define HW_XEN_PV       0x01    /* Running on Xen Hypervisor paravirutualized */
784 #define HW_XEN_HVM      0x02    /* Running on Xen hypervisor HVM */
785 #define HW_VMWARE       0x03    /* Running on VMware hypervisor */
```

```
 787 #endif  /* _KERNEL */

 789 #endif

 791 #ifdef  __cplusplus
 792 }
```

_____***unchanged_portion_omitted_***