
	Instytut Informatyki Politechniki Śląskiej  Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Języki Asemblerowe	LAB	V

## TEMAT:

Zapoznanie się z instrukcjami MMX i SSE w j. asemblera.

## CEL:

Celem ćwiczenia jest poznanie działania rozkazów MMX i SSE procesorów x86 wykorzystywanych do wykonywania obliczeń zmiennoprzecinkowych oraz jeśli to możliwe stworzenie aplikacji 64-bitowej w środowisku Windows.

## ZAŁOŻENIA:

- Zaimplementować procedurę `_cpuid` rozpoznawania typu procesora oraz MMX, SSE (wg przykładu `cpuid.cpp`),
- Zaimplementować procedurę `RadToDegCpp` (w C++),
- Stworzyć dll'kę assemblerową z funkcją `CpuIdAsm` sprawdzającą dostępność rozkazów SSE, oraz funkcją `RadToDegAsm`,
- Skompilować i uruchomić program wywołując obie procedury `RadToDegCpp`, `RadToDegAsm`.

## WYKONANIE:

Przed wywołaniem procedur `RadToDegxxx` należy wykonać sprawdzenie, czy dany procesor posiada zestaw instrukcji MMX, SSE, SSE2 wykonując rozkaz `CPUID` (asm) i `_cpuid` (C++). Osoby chętne mogą stworzyć projekt w wersji 64bit.

**Sprawozdanie powinno zawierać zrzut ekranowy z postawionym breakpointem (asm), szczegółowy opis procedur z komentarzami oraz wnioski.**

```

// cpuid.cpp
// processor: x86, x64
// Use the __cpuid intrinsic to get information about a CPU

#include <stdio.h>
#include <string.h>
#include <intrin.h>

const char* szFeatures[] =
{
    "x87 FPU On Chip",
    "Virtual-8086 Mode Enhancement",
    "Debugging Extensions",
    "Page Size Extensions",
    "Time Stamp Counter",
    "RDMSR and WRMSR Support",
    "Physical Address Extensions",
    "Machine Check Exception",
    "CMPXCHG8B Instruction",
    "APIC On Chip",
    "Unknown1",
    "SYSENTER and SYSEXIT",
    "Memory Type Range Registers",
    "PTE Global Bit",
    "Machine Check Architecture",
    "Conditional Move/Compare Instruction",
    "Page Attribute Table",
    "36-bit Page Size Extension",
    "Processor Serial Number",
    "CFLUSH Extension",
    "Unknown2",
    "Debug Store",
    "Thermal Monitor and Clock Ctrl",
    "MMX Technology",
    "FXSAVE/FXRSTOR",
    "SSE Extensions",
    "SSE2 Extensions",
    "Self Snoop",
    "Multithreading Technology",
    "Thermal Monitor",
    "Unknown4",
    "Pending Break Enable"
};

int main(int argc, char* argv[])
{
    char CPUString[0x20];
    char CPUBrandString[0x40];
    int CPUInfo[4] = {-1};
    int nSteppingID = 0;
    int nModel = 0;
    int nFamily = 0;
    int nProcessorType = 0;
    int nExtendedmodel = 0;
    int nExtendedfamily = 0;
    int nBrandIndex = 0;
    int nCLFLUSHcachelinesize = 0;
    int nLogicalProcessors = 0;
    int nAPICPhysicalID = 0;
    int nFeatureInfo = 0;
    int nCacheLineSize = 0;
    int nL2Associativity = 0;
    int nCacheSizeK = 0;
    int nPhysicalAddress = 0;
    int nVirtualAddress = 0;

```

```

int nRet = 0;

int nCores = 0;
int nCacheType = 0;
int nCacheLevel = 0;
int nMaxThread = 0;
int nSysLineSize = 0;
int nPhysicalLinePartitions = 0;
int nWaysAssociativity = 0;
int nNumberSets = 0;

unsigned    nIds, nExIds, i;

bool        bSSE3Instructions = false;
bool        bMONITOR_MWAIT = false;
bool        bCPLQualifiedDebugStore = false;
bool        bVirtualMachineExtensions = false;
bool        bEnhancedIntelSpeedStepTechnology = false;
bool        bThermalMonitor2 = false;
bool        bSupplementalSSE3 = false;
bool        bL1ContextID = false;
bool        bCMPXCHG16B = false;
bool        bxTPRUpdateControl = false;
bool        bPerfDebugCapabilityMSR = false;
bool        bSSE41Extensions = false;
bool        bSSE42Extensions = false;
bool        bPOPCNT = false;

bool        bMultithreading = false;

bool        bLAHF_SAHFAvailable = false;
bool        bCmpLegacy = false;
bool        bSVM = false;
bool        bExtApicSpace = false;
bool        bAltMovCr8 = false;
bool        bLZCNT = false;
bool        bSSE4A = false;
bool        bMisalignedSSE = false;
bool        bPREFETCH = false;
bool        bSKINITandDEV = false;
bool        bSYSCALL_SYSRETAavailable = false;
bool        bExecuteDisableBitAvailable = false;
bool        bMMXExtensions = false;
bool        bFFXSr = false;
bool        b1GBSupport = false;
bool        bRDTSCP = false;
bool        b64Available = false;
bool        b3DNowExt = false;
bool        b3DNow = false;
bool        bNestedPaging = false;
bool        bLBRVisualization = false;
bool        bFP128 = false;
bool        bMOVOptimization = false;

bool        bSelfInit = false;
bool        bFullyAssociative = false;

// __cpuid with an InfoType argument of 0 returns the number of
// valid Ids in CPUInfo[0] and the CPU identification string in
// the other three array elements. The CPU identification string is
// not in linear order. The code below arranges the information
// in a human readable form.
__cpuid(CPUInfo, 0);
nIds = CPUInfo[0];
memset(CPUString, 0, sizeof(CPUString));

```

```

*((int*)CPUString) = CPUInfo[1];
*((int*)(CPUString+4)) = CPUInfo[3];
*((int*)(CPUString+8)) = CPUInfo[2];

// Get the information associated with each valid Id
for (i=0; i<=nIds; ++i)
{
    __cpuid(CPUInfo, i);
    printf_s("\nFor InfoType %d\n", i);
    printf_s("CPUInfo[0] = 0x%x\n", CPUInfo[0]);
    printf_s("CPUInfo[1] = 0x%x\n", CPUInfo[1]);
    printf_s("CPUInfo[2] = 0x%x\n", CPUInfo[2]);
    printf_s("CPUInfo[3] = 0x%x\n", CPUInfo[3]);

    // Interpret CPU feature information.
    if (i == 1)
    {
        nSteppingID = CPUInfo[0] & 0xf;
        nModel = (CPUInfo[0] >> 4) & 0xf;
        nFamily = (CPUInfo[0] >> 8) & 0xf;
        nProcessorType = (CPUInfo[0] >> 12) & 0x3;
        nExtendedmodel = (CPUInfo[0] >> 16) & 0xf;
        nExtendedfamily = (CPUInfo[0] >> 20) & 0xff;
        nBrandIndex = CPUInfo[1] & 0xff;
        nCLFLUSHcachelinesize = ((CPUInfo[1] >> 8) & 0xff) * 8;
        nLogicalProcessors = ((CPUInfo[1] >> 16) & 0xff);
        nAPICPhysicalID = (CPUInfo[1] >> 24) & 0xff;
        bSSE3Instructions = (CPUInfo[2] & 0x1) || false;
        bMONITOR_MWAIT = (CPUInfo[2] & 0x8) || false;
        bCPLQualifiedDebugStore = (CPUInfo[2] & 0x10) || false;
        bVirtualMachineExtensions = (CPUInfo[2] & 0x20) || false;
        bEnhancedIntelSpeedStepTechnology = (CPUInfo[2] & 0x80) || false;
        bThermalMonitor2 = (CPUInfo[2] & 0x100) || false;
        bSupplementalSSE3 = (CPUInfo[2] & 0x200) || false;
        bL1ContextID = (CPUInfo[2] & 0x300) || false;
        bCMPXCHG16B = (CPUInfo[2] & 0x2000) || false;
        bXTPRUpdateControl = (CPUInfo[2] & 0x4000) || false;
        bPerfDebugCapabilityMSR = (CPUInfo[2] & 0x8000) || false;
        bSSE41Extensions = (CPUInfo[2] & 0x80000) || false;
        bSSE42Extensions = (CPUInfo[2] & 0x100000) || false;
        bPOPCNT = (CPUInfo[2] & 0x800000) || false;
        nFeatureInfo = CPUInfo[3];
        bMultithreading = (nFeatureInfo & (1 << 28)) || false;
    }
}

// Calling __cpuid with 0x80000000 as the InfoType argument
// gets the number of valid extended IDs.
__cpuid(CPUInfo, 0x80000000);
nExIds = CPUInfo[0];
memset(CPUBrandString, 0, sizeof(CPUBrandString));

// Get the information associated with each extended ID.
for (i=0x80000000; i<=nExIds; ++i)
{
    __cpuid(CPUInfo, i);
    printf_s("\nFor InfoType %x\n", i);
    printf_s("CPUInfo[0] = 0x%x\n", CPUInfo[0]);
    printf_s("CPUInfo[1] = 0x%x\n", CPUInfo[1]);
    printf_s("CPUInfo[2] = 0x%x\n", CPUInfo[2]);
    printf_s("CPUInfo[3] = 0x%x\n", CPUInfo[3]);

    if (i == 0x80000001)
    {
        bLAHF_SAHFAvailable = (CPUInfo[2] & 0x1) || false;
    }
}

```

```

    bCmpLegacy = (CPUInfo[2] & 0x2) || false;
    bSVM = (CPUInfo[2] & 0x4) || false;
    bExtApicSpace = (CPUInfo[2] & 0x8) || false;
    bAltMovCr8 = (CPUInfo[2] & 0x10) || false;
    bLZCNT = (CPUInfo[2] & 0x20) || false;
    bSSE4A = (CPUInfo[2] & 0x40) || false;
    bMisalignedSSE = (CPUInfo[2] & 0x80) || false;
    bPREFETCH = (CPUInfo[2] & 0x100) || false;
    bSKINITandDEV = (CPUInfo[2] & 0x1000) || false;
    bSYSCALL_SYSRETAvalable = (CPUInfo[3] & 0x800) || false;
    bExecutedisableBitAvailable = (CPUInfo[3] & 0x10000) || false;
    bMMXExtensions = (CPUInfo[3] & 0x40000) || false;
    bFFXSRR = (CPUInfo[3] & 0x200000) || false;
    bLGBSupport = (CPUInfo[3] & 0x400000) || false;
    bRDTSCP = (CPUInfo[3] & 0x8000000) || false;
    b64Avalable = (CPUInfo[3] & 0x20000000) || false;
    b3DNowExt = (CPUInfo[3] & 0x40000000) || false;
    b3DNow = (CPUInfo[3] & 0x80000000) || false;
}

// Interpret CPU brand string and cache information.
if (i == 0x80000002)
    memcpy(CPUBrandString, CPUInfo, sizeof(CPUInfo));
else if (i == 0x80000003)
    memcpy(CPUBrandString + 16, CPUInfo, sizeof(CPUInfo));
else if (i == 0x80000004)
    memcpy(CPUBrandString + 32, CPUInfo, sizeof(CPUInfo));
else if (i == 0x80000006)
{
    nCacheLineSize = CPUInfo[2] & 0xff;
    nL2Associativity = (CPUInfo[2] >> 12) & 0xf;
    nCacheSizeK = (CPUInfo[2] >> 16) & 0xffff;
}
else if (i == 0x80000008)
{
    nPhysicalAddress = CPUInfo[0] & 0xff;
    nVirtualAddress = (CPUInfo[0] >> 8) & 0xff;
}
else if (i == 0x8000000A)
{
    bNestedPaging = (CPUInfo[3] & 0x1) || false;
    bLBRVisualization = (CPUInfo[3] & 0x2) || false;
}
else if (i == 0x8000001A)
{
    bFP128 = (CPUInfo[0] & 0x1) || false;
    bMOVOptimization = (CPUInfo[0] & 0x2) || false;
}
}

// Display all the information in user-friendly format.

printf_s("\n\nCPU String: %s\n", CPUString);

if (nIds >= 1)
{
    if (nSteppingID)
        printf_s("Stepping ID = %d\n", nSteppingID);
    if (nModel)
        printf_s("Model = %d\n", nModel);
    if (nFamily)
        printf_s("Family = %d\n", nFamily);
    if (nProcessorType)
        printf_s("Processor Type = %d\n", nProcessorType);
    if (nExtendedmodel)

```

```

    printf_s("Extended model = %d\n", nExtendedmodel);
if (nExtendedfamily)
    printf_s("Extended family = %d\n", nExtendedfamily);
if (nBrandIndex)
    printf_s("Brand Index = %d\n", nBrandIndex);
if (nCLFLUSHcachelinesize)
    printf_s("CLFLUSH cache line size = %d\n",
        nCLFLUSHcachelinesize);
if (bMultithreading && (nLogicalProcessors > 0))
    printf_s("Logical Processor Count = %d\n", nLogicalProcessors);
if (nAPICPhysicalID)
    printf_s("APIC Physical ID = %d\n", nAPICPhysicalID);

if (nFeatureInfo || bSSE3Instructions ||
    bMONITOR_MWAIT || bCPLQualifiedDebugStore ||
    bVirtualMachineExtensions || bEnhancedIntelSpeedStepTechnology ||
    bThermalMonitor2 || bSupplementalSSE3 || bL1ContextID ||
    bCMPXCHG16B || bXTPRUpdateControl || bPerfDebugCapabilityMSR ||
    bSSE41Extensions || bSSE42Extensions || bPOPCNT ||
    bLAHF_SAHFAvailable || bCmpLegacy || bSVM ||
    bExtApicSpace || bAltMovCr8 ||
    bLZCNT || bSSE4A || bMisalignedSSE ||
    bPREFETCH || bSKINITandDEV || bSYSCALL_SYSRETAvailable ||
    bExecuteDisableBitAvailable || bMMXExtensions || bFFXSr ||
b1GBSupport ||
    bRDTSCP || b64Available || b3DNowExt || b3DNow || bNestedPaging ||
    bLBRVisualization || bFP128 || bMOVOptimization )
{
    printf_s("\nThe following features are supported:\n");

    if (bSSE3Instructions)
        printf_s("\tSSE3\n");
    if (bMONITOR_MWAIT)
        printf_s("\tMONITOR/MWAIT\n");
    if (bCPLQualifiedDebugStore)
        printf_s("\tCPL Qualified Debug Store\n");
    if (bVirtualMachineExtensions)
        printf_s("\tVirtual Machine Extensions\n");
    if (bEnhancedIntelSpeedStepTechnology)
        printf_s("\tEnhanced Intel SpeedStep Technology\n");
    if (bThermalMonitor2)
        printf_s("\tThermal Monitor 2\n");
    if (bSupplementalSSE3)
        printf_s("\tSupplemental Streaming SIMD Extensions 3\n");
    if (bL1ContextID)
        printf_s("\tL1 Context ID\n");
    if (bCMPXCHG16B)
        printf_s("\tCMPXCHG16B Instruction\n");
    if (bXTPRUpdateControl)
        printf_s("\txTPR Update Control\n");
    if (bPerfDebugCapabilityMSR)
        printf_s("\tPerf\Debug Capability MSR\n");
    if (bSSE41Extensions)
        printf_s("\tSSE4.1 Extensions\n");
    if (bSSE42Extensions)
        printf_s("\tSSE4.2 Extensions\n");
    if (bPOPCNT)
        printf_s("\tPPOPCNT Instruction\n");

    i = 0;
    nIds = 1;
    while (i < (sizeof(szFeatures)/sizeof(const char*)))
    {
        if (nFeatureInfo & nIds)
        {

```

```

        printf_s("\t");
        printf_s(szFeatures[i]);
        printf_s("\n");
    }

    nIds <= 1;
    ++i;
}
if (bLAHF_SAHFAvailable)
    printf_s("\tLAHF/SAHF in 64-bit mode\n");
if (bCmpLegacy)
    printf_s("\tCore multi-processing legacy mode\n");
if (bSVM)
    printf_s("\tSecure Virtual Machine\n");
if (bExtApicSpace)
    printf_s("\tExtended APIC Register Space\n");
if (bAltMovCr8)
    printf_s("\tAltMovCr8\n");
if (bLZCNT)
    printf_s("\tLZCNT instruction\n");
if (bSSE4A)
    printf_s("\tSSE4A (EXTRQ, INSERTQ, MOVNTSD, MOVNTSS)\n");
if (bMisalignedSSE)
    printf_s("\tMisaligned SSE mode\n");
if (bPREFETCH)
    printf_s("\tPREFETCH and PREFETCHW Instructions\n");
if (bSKINITandDEV)
    printf_s("\tSKINIT and DEV support\n");
if (bSYSCALL_SYSRETAvailable)
    printf_s("\tSYSCALL/SYSRET in 64-bit mode\n");
if (bExecuteDisableBitAvailable)
    printf_s("\tExecute Disable Bit\n");
if (bMMXExtensions)
    printf_s("\tExtensions to MMX Instructions\n");
if (bFFXSR)
    printf_s("\tFFXSR\n");
if (b1GBSupport)
    printf_s("\t1GB page support\n");
if (bRDTSCP)
    printf_s("\tRDTSCP instruction\n");
if (b64Available)
    printf_s("\t64 bit Technology\n");
if (b3DNowExt)
    printf_s("\t3Dnow Ext\n");
if (b3DNow)
    printf_s("\t3Dnow! instructions\n");
if (bNestedPaging)
    printf_s("\tNested Paging\n");
if (bLBRVisualization)
    printf_s("\tLBR Visualization\n");
if (bFP128)
    printf_s("\tFP128 optimization\n");
if (bMOVOptimization)
    printf_s("\tMOVU Optimization\n");
}
}

if (nExIds >= 0x80000004)
    printf_s("\nCPU Brand String: %s\n", CPUBrandString);

if (nExIds >= 0x80000006)
{
    printf_s("Cache Line Size = %d\n", nCacheLineSize);
    printf_s("L2 Associativity = %d\n", nL2Associativity);
    printf_s("Cache Size = %dK\n", nCacheSizeK);
}

```

```

}

for (i=0;;i++)
{
    __cpuidex(CPUInfo, 0x4, i);
    if(!(CPUInfo[0] & 0xf0)) break;

    if(i == 0)
    {
        nCores = CPUInfo[0] >> 26;
        printf_s("\n\nNumber of Cores = %d\n", nCores + 1);
    }

    nCacheType = (CPUInfo[0] & 0x1f);
    nCacheLevel = (CPUInfo[0] & 0xe0) >> 5;
    bSelfInit = (CPUInfo[0] & 0x100) >> 8;
    bFullyAssociative = (CPUInfo[0] & 0x200) >> 9;
    nMaxThread = (CPUInfo[0] & 0x3ffc000) >> 14;
    nSysLineSize = (CPUInfo[1] & 0x0fff);
    nPhysicalLinePartitions = (CPUInfo[1] & 0x3ff000) >> 12;
    nWaysAssociativity = (CPUInfo[1]) >> 22;
    nNumberSets = CPUInfo[2];

    printf_s("\n");

    printf_s("ECX Index %d\n", i);
    switch (nCacheType)
    {
        case 0:
            printf_s("    Type: Null\n");
            break;
        case 1:
            printf_s("    Type: Data Cache\n");
            break;
        case 2:
            printf_s("    Type: Instruction Cache\n");
            break;
        case 3:
            printf_s("    Type: Unified Cache\n");
            break;
        default:
            printf_s("    Type: Unknown\n");
    }

    printf_s("    Level = %d\n", nCacheLevel + 1);
    if (bSelfInit)
    {
        printf_s("    Self Initializing\n");
    }
    else
    {
        printf_s("    Not Self Initializing\n");
    }
    if (bFullyAssociative)
    {
        printf_s("    Is Fully Associative\n");
    }
    else
    {
        printf_s("    Is Not Fully Associative\n");
    }
    printf_s("    Max Threads = %d\n",
        nMaxThread+1);
    printf_s("    System Line Size = %d\n",

```



```
        nSysLineSize+1);
printf_s("    Physical Line Partions = %d\n",
        nPhysicalLinePartitions+1);
printf_s("    Ways of Associativity = %d\n",
        nWaysAssociativity+1);
printf_s("    Number of Sets = %d\n",
        nNumberSets+1);
}

return  nRet;
}
```

```

#include <emmintrin.h> // SSE2 instructions library header

double RadToDegNMEACpp (double dRad, BOOL bsSE2)
{
    double dDeg = 0.0;

    double dRem = 0.0,dx;
    int nDeg;
    if (bsSE2) {
        __m128d ma,mb,mDeg,mx,my,mz,mk;

        ma = _mm_set_sd (180.0);
        mb = _mm_set_sd (M_PI);
        mDeg = _mm_div_pd (ma,mb); // convert rad to deg deg =
rad*180/pi

        ma = _mm_set_sd (dRad);
        mb = _mm_mul_pd (ma,mDeg);
        dDeg = mb.m128d_f64[0]; // rad to deg

        nDeg = (int)dDeg; // abs (dMLng)
        dRem = dDeg-nDeg; // rem (dMLng)

        ma = _mm_set_sd (100.0);
        mb = _mm_set_sd (60.0);
        mx = _mm_set_sd (dRem);
        mz = _mm_mul_pd (mx,mb); // dGPS*60
        dx = nDeg*100;
        mk = _mm_set_sd (dx);
        my = _mm_add_pd (mk,mz); // nDeg*100+dGPS*60
        dDeg = my.m128d_f64[0]; // convert to degmmmin.ssdec NMEA format
    }
    else {
        double dCnv = 180.0/M_PI;

        dDeg = dRad*dCnv;

        nDeg = (int)dDeg;
        dRem = dDeg-nDeg;
        dDeg = nDeg*100+dRem*60;
    }

    return dDeg;
} // End of RadToDegNMEACpp

```