White Paper
Leonardo Potenza
Software Engineer
Intel Corporation

Segmentation-based Memory Protection Mechanism on Intel® Atom™ Microarchitecture: Coding Optimizations

October 2011

326296

Executive Summary

The Intel® Atom™ processor is explicitly designed to achieve efficient performance and low-power operation, adopting special in-order microarchitecture. The development of highly-optimized software applications requires the use of specific coding techniques.

The purpose of this paper is to emphasize some aspects of the segmented memory protection mechanism implemented in the Intel® Atom™ microarchitecture. A developer can use this information to create a higher-performing solution.

The purpose of this paper is to emphasize some aspects of the segmented memory protection mechanism implemented in the Intel® Atom™ microarchitecture. A developer can use this information to create a higher-performing solution.

The idea is to provide the software developer with guidelines to understanding the different performance characteristics available depending on how the segmented memory is configured and to propose techniques that can be used to improve performance.

Contents

Introduction

The development of a complex multi-tasking software application requires a suitable execution environment. One important feature that almost all the multi-tasking operating systems must provide to ensure *safe* execution of the code is a memory protection mechanism.

The run-time environment must prevent a process from *touching* memory areas that belong to other processes or to the operating system. The IA-32 architecture provides multiple methods for protecting memory, with a view to simplifying OS programming.

Paging - The Flat Memory Model

For various reasons, almost all the modern operating systems prefer to base their memory protection implementation purely on the services provided by the processor's Paging Logic.

The widely adopted memory model is the so called "Flat Memory Model". All applications see the full virtual address space starting at base address 0.

Before executing an application, the operating system creates a set of virtual-to-physical address mapping tables. During the execution, all the generated addresses are submitted to the Paging Logic for a lookup in the application's page tables. In addition to the actual virtual-to-physical address conversion, each page table entry contains a set of other information (for example, the read/write attribute of the page, the privilege level the program in execution must meet) allowing the Paging Logic to grant or deny access to the memory area.

The *Flat Memory Model* implementation has to guarantee that *paging* is the only active memory protection mechanism. The processor's Segmentation Logic must be *disabled*, by programming all the processor segments with base address value 0 and a size equal to the available address space length.

Segmentation

Even if not so common, there are cases where the memory protection mechanism is based on the services provided by the processor's Segmentation Logic.

In the *Segmented Memory Model*, different applications see different address spaces. Moreover, the same application can have different memory areas for different types of segments (Code Segments, Data Segments and Task State Segments). Every segment has its own base address and size. Any attempt to access memory areas outside the segment boundaries is prevented by the Segmentation Logic run-time checks.

The handling of the segmentation mechanism is usually more complicated, both for the application and for the OS kernel code, but the flexibility that can be obtained may be useful in some software solutions.

The use of the segmentation mechanism for memory protection can be found in a number of scenarios such as when dealing with legacy embedded applications or when a conventional OS needs to co-exist with a run-time environment. Some products protect the run-time environment from the standard OS using segmented memory addressing. Sometimes, segmented memory addressing is used instead of paging to help meet some specific real-time requirements.

Segmentation Details on Intel® Atom™ Microarchitecture

The implementation of the segmentation mechanism on Intel® Atom™ microarchitecture is completely backward compatible with the IA-32 instruction set.

Special implementation details should be taken into account when deciding to deploy the code on a platform featuring the Intel® Atom™ processor.

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual* Ref [1] describes how to optimize software to take advantage of the performance characteristics of IA-32 and Intel 64 architecture processors. As reported in the specific chapter dedicated to the Intel® Atom™ microarchitecture, the Address Generation Unit (AGU) assumes that the segment base is zero by default.

The software developer should be aware that, when using segment base values different than zero, the load and store operations (access to and from memory) take longer to complete than a similar operation with a zero stored in the segment base register.

Segment Base Unaligned to Cache Line Boundary

An operating condition that should be avoided is the use of non-zero segment base values that are not aligned to a cache line boundary. The maximum throughput of a MOV operation in this case is reduced to one every nine cycles (instead of one every cycle).

The selection of the segment base values has to be carefully considered at the OS or *run-time framework* implementation level. The software application developer should never need to cope with this particular situation.

Memory Access Through the Code Segment

The Code Segment (CS) register points to the memory area in which the processor instructions are stored. When loading data from memory, the use of the CS can be explicitly specified, overriding the default segment selector:

For example: MOV AL, CS:[BX]

Using a non-zero value for CS, the maximum throughput of a memory read operation is reduced to one every nine cycles (instead of one every cycle), even if the base value is cache aligned.

String Operations Using the ES Base Register

The ES register points to one of the data segments that can be used for efficient and secure access to different types of data structures.

An important recommendation that needs to be considered when writing optimized code that deals with a string operation is to favor instructions that make implicit use of the ES base register for the destination operand. The maximum throughput is one operation every two cycles. The maximum throughput becomes significantly lower (one operation every nine cycles) in all other cases.

However, it is important to note that this performance optimization is useful only for assembly language programmers or compiler developers that need to generate optimized code for the Intel® Atom™ microarchitecture.

Memory Access Through the Data Segment and Stack Segment

Two cases where the software developer's choices can have significant impact on code execution times are when the application needs to access data structures in memory through1) the Stack Segment and 2) the Data Segment.

The Data Segment (DS) register points to the default segment involved when loading data from memory and storing data in memory (unless an explicit segment override has been specified).

The Stack Segment (SS) register points to the data area where the procedure stack is stored for the handler, program or task currently in execution. The stack segment is selected for all stack pushes and pops and for any memory reference that uses the ESP (stack pointer) and EBP (base pointer) as a base pointer (for further details, refer to Ref [4]).

Assuming that the non-zero segment base values are cache aligned, the introduced penalty in the memory access varies depending on the particular segment being used.

The maximum throughput of a MOV operation using the Data Segment (DS) is reduced to one every two cycles (instead of one every cycle).

On the other hand, the price of using the stack segment is much higher. The maximum throughput of a MOV operation using the Stack Segment (SS) is reduced to one every nine cycles.

Data Segment vs Stack Segment: Write Memory Access Case Study

The purpose of this section is to show the importance of the segmentation details using simple practical examples.

For simplicity, only the write memory access case study is covered, but similar examples might be derived using the same methodology.

Using the Intel processor's internal Time Stamp Counter (TSC) and the related instructions for time reading, it is quite straightforward to measure the various memory access delays.

In the following sections, a read operation of the current value of the processor's TSC is represented by calling the generic C function

```
extern uint64_t getTscValue(void);
```

Refer to the available documentation Ref [2] Ref [3] for the details on how to implement the above function using the Intel instruction set.

Write Access on Stack Segment

To reproduce a *write access* to a variable in the *stack segment*, the code snippet that can be used is trivial:

```
#define LOOP_CYCLES_NUMBER 100000

uint64_t StackMemoryWrite (void)
{
    volatile long a;
    int           i;
    uint64_t      elapsedTime;
    uint64_t      tscTicksBefore;
    uint64_t      tscTicksAfter;

    tscTicksBefore = getTscValue();

    for (i = 0; i < LOOP_CYCLES_NUMBER; i++)
    {
      /* Dummy write accesses on 'a' memory variable
         using stack segment */
```

```
        a = 1;    a = 2;    a = 3;    a = 4;    a = 5;
        a = 6;    a = 7;    a = 8;    a = 9;    a = 10;
    }

    tscTicksAfter = getTscValue();

    elapsedTime = tscTicksAfter - tscTicksBefore;

    return elapsedTime;
}
```

Basically, the `StackMemoryWrite()` function returns the number of TSC ticks needed to carry out a given number of write accesses to the dummy variable `a` (100000 * 10 in the example).

The use of the `volatile` attribute prevents the compiler from performing code optimizations on the access to the variable `a`, which is clearly allocated on the stack segment.

For instance, this is how the Microsoft Visual Studio* 2010 compiler translates the code in the `for()` loop in assembly language, generating the expected number of write memory accesses on the stack segment:

```
   …
   mov   edx, 9
   mov   eax, 100000          ; 000186a0H
   lea   ecx, DWORD PTR [edx+1]
$LL3@StackMemor:

; 253   :
; 254   :     for (i = 0; i < LOOP_CYCLES_NUMBER; i++)

   dec   eax

; 255   :     {
; 256   : /* Dummy write accesses on 'a' memory variable
; 257   :    using stack segment */
; 258   :          a = 1;    a = 2;    a = 3;    a = 4;    a = 5;

   mov   DWORD PTR _a$[esp+16], 1
   mov   DWORD PTR _a$[esp+16], 2
   mov   DWORD PTR _a$[esp+16], 3
   mov   DWORD PTR _a$[esp+16], 4
   mov   DWORD PTR _a$[esp+16], 5

; 259   :          a = 6;    a = 7;    a = 8;    a = 9;    a = 10;

   mov   DWORD PTR _a$[esp+16], 6
   mov   DWORD PTR _a$[esp+16], 7
   mov   DWORD PTR _a$[esp+16], 8
   mov   DWORD PTR _a$[esp+16], edx
   mov   DWORD PTR _a$[esp+16], ecx
   jne   SHORT $LL3@StackMemor
```

```
    …
```

Write Access on Data Segment

To reproduce the same type of *write access* to a variable in the *data segment*, the above code snippet can be modified as follows:

```
#define LOOP_CYCLES_NUMBER 100000

volatile long a;

uint64_t DataSegmentMemoryWrite (void)
{
    int            i;
    uint64_t       elapsedTime;
    uint64_t       tscTicksBefore;
    uint64_t       tscTicksAfter;

    tscTicksBefore = getTscValue();

    for (i = 0; i < LOOP_CYCLES_NUMBER; i++)
    {
        /* Dummy write accesses on 'a' memory variable
           using data segment */
        a = 1;     a = 2;     a = 3;     a = 4;     a = 5;
        a = 6;     a = 7;     a = 8;     a = 9;     a = 10;
    }

    tscTicksAfter = getTscValue();

    elapsedTime = tscTicksAfter - tscTicksBefore;

    return elapsedTime;
}
```

The new `DataSegmentMemoryWrite()` function returns the number of TSC ticks needed to carry out a given number of write accesses to the dummy variable `a` (100000 * 10 in the example).

The use of the `volatile` attribute prevents the compiler from performing code optimizations on the access to the variable `a`, which is clearly allocated on the data segment.

For instance, this is how the Microsoft Visual Studio* 2010 compiler translates the code in the `for()` loop in assembly language, generating the expected number of write memory accesses on the data segment:

```
    …
    mov    edx, 9
    mov    eax, 100000          ; 000186a0H
```

```
   lea   ecx, DWORD PTR [edx+1]
   npad  5
$LL3@DataSegmen:

; 280  :
; 281  :     for (i = 0; i < LOOP_CYCLES_NUMBER; i++)

   dec   eax

; 282  :     {
; 283  :         /* Dummy write accesses on 'a' memory variable
; 284  :            using data segment */
; 285  :         a = 1;    a = 2;    a = 3;    a = 4;    a = 5;

   mov   DWORD PTR ?a@@3JC, 1       ; a
   mov   DWORD PTR ?a@@3JC, 2       ; a
   mov   DWORD PTR ?a@@3JC, 3       ; a
   mov   DWORD PTR ?a@@3JC, 4       ; a
   mov   DWORD PTR ?a@@3JC, 5       ; a

; 286  :         a = 6;    a = 7;    a = 8;    a = 9;    a = 10;

   mov   DWORD PTR ?a@@3JC, 6       ; a
   mov   DWORD PTR ?a@@3JC, 7       ; a
   mov   DWORD PTR ?a@@3JC, 8       ; a
   mov   DWORD PTR ?a@@3JC, edx     ; a
   mov   DWORD PTR ?a@@3JC, ecx     ; a
   jne   SHORT $LL3@DataSegmen
```

Write Memory Access: Experimental Data

To show the importance of the implementation of the segmentation-based protection mechanism on Intel® Atom™ architecture, a simple test has been carried out.

The above `StackMemoryWrite()` and `DataSegmentMemoryWrite()` functions have been executed on the Intel® Atom™ Processor N450 with the Intel® 82801HM I/O Controller Hub Customer Reference Board (CRB), using two different run-time environments: *Windows* 7* and a *commercial RTOS* implementing the segmentation-based memory protection mechanism using non-zero segment base addresses.

The code has been compiled in both *release* and *debug* modes using the two development environments. The *debug* configuration is a build option typically used in the software development phase, as the generated binary includes additional symbolic debug information and the generated code is not optimized for execution (the idea is to facilitate source code debugging). The *release* configuration is a build option especially designed for the final product release. The code is optimized to reduce the execution time and the size of the generated binary (no extra symbols are included).

Due to the simplicity of the original C code, there were no significant differences between the assembly code generated by the two compilers.

The following table reports the experimental results for the function simulating the write accesses to the stack segment.

Table 1. StackMemoryWrite() Average Execution Time, Calculated After 100 Consecutive Executions

	Windows* [TSC ticks]	Commercial RTOS [TSC ticks]	Ratio (RTOS/Win)
Release	1,684,050	15,029,565	8.92
Debug	2,357,713	19,685,439	8.34

The experimental data correlates to the one-to-nine ratio reported in Ref [1] when trying to access variables on the stack segment with segment base addresses other than zero.

The following table refers to the function performing write access to the data segment.

Table 2. DataSegmentMemoryWrite() Average Execution Time, Calculated After 100 Consecutive Executions

	Windows* [TSC ticks]	Commercial RTOS [TSC ticks]	Ratio (RTOS/Win)
Release	2,374,306	3,851,909	1.6
Debug	3,221,116	4,671,192	1.4

These results correlates to the one-to-two ratio reported in Ref [1], when trying to access variables on the data segment with segment base addresses other than zero.

In both cases, the measured ratio is not exactly equal to the theoretical value, since the portion of the code under test (based on a `for()` loop) also includes instructions different from the *pure* memory store.

Lesson Learned

The purpose of the functions described earlier was to deliberately stress the processor memory access with non-zero segment base addresses, therefore the results are not representative of the code execution time in a real software application.

However some general guidelines can be extrapolated.

Guideline 1: Paging-based memory protection mechanisms are preferred over segmentation when deploying software applications explicitly for the Intel® Atom™ microarchitecture.

This conclusion comes from the analysis of the delays reported in the memory access when operating with non-zero segment base values. Unfortunately, this rule often cannot be followed, since the segmentation-based protection mechanism is needed by design and/or the platform based on the Intel® Atom™ processor is not the only supported platform. Consequently, further code optimizations may be required.

Guideline 2: When using the segmentation-based protection mechanism, limit the number and size of automatic variables inside C functions.

If the number and size of automatic variables is limited, an optimized compiler can generate assembly code that performs almost no access operations on the stack (all the access operations are to/from CPU registers). Whenever the handling of big data structures is required, those structures should be allocated in the data segment, limiting in this way the performance penalty to be paid for the read/write access.

When using non-zero segment base addresses, code snippets such as the following should be avoided when developing for the Intel® Atom™ microarchitecture. Since the `buffer` array will be allocated on the stack, access to this array will be sub-optimal.

```
#define BUFFER_SIZE 1024

void Function (void)
{
    unsigned char buffer[BUFFER_SIZE];
   …
```

Guideline 3: When using the segmentation-based protection mechanism, the use of C compilers explicitly optimized for Intel® Atom™ architecture is preferred, avoiding the *debug* build options.

The reduction of the number of memory access operations (using transfers to/from CPU registers as an alternative) is a general principle valid for every application that needs performance enhancements. However, this becomes even more critical on Intel® Atom™ microarchitecture when the segmentation-based memory protection mechanism is adopted.

The use of an optimized compiler and a *release* build configuration can significantly reduce the performance difference between zero and non-zero segment base memory addressing.

The binary compiled with typical *debug* configurations does not introduce code optimizations and results in almost all the access operations to/from system memory, causing higher execution times when segments base registers are non-zero.

Conclusion

The details on the memory protection mechanism implemented by the OS and/or the run-time execution environment can have a significant impact on performance when deploying software applications for the Intel® Atom™ microarchitecture.

Whenever possible, paging-based protection mechanisms should be preferred over solutions based on segmentation, since the same code running with non-zero segment base registers values can experience a delay in the access operations on the system memory.

However, when segmentation is needed by design, good coding techniques and the use of compilers optimized for the Intel® Atom™ microarchitecture can reduce the performance gap, significantly improving performance.

Reference list

[1] Intel® 64 and IA-32 Architectures Optimization Reference Manual
http://www.intel.com/Assets/PDF/manual/248966.pdf

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A & 2B): Instruction Set Reference, A-Z
http://www.intel.com/Assets/PDF/manual/325383.pdf

[3] How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures
http://download.intel.com/embedded/software/IA/324264.pdf

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture
http://www.intel.com/Assets/PDF/manual/253665.pdf

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

Author

Leonardo Potenza is a Software Engineer with the Embedded and Communications Group at Intel Corporation.

Contributors

Adrian Hoban is a Software Engineer with the Embedded and Communications Group at Intel Corporation.

Maryam Tahhan is a Software Engineer with the Embedded and Communications Group at Intel Corporation.

James McGinley is a Solution Architect with the Embedded and Communications Group at Intel Corporation.

Peter Mangan is an Engineering Manager with the Embedded and Communications Group at Intel Corporation.

Kevin Finucane is a Software Program Manager with the Embedded and Communications Group at Intel Corporation.

Acronyms

OS	Operating System
RTOS	Real-Time Operating System
AGU	Address Generation Unit
CS	Code Segment
SS	Stack Segment
DS	Data Segment
TSC	Time Stamp Counter
CRB	Customer Reference Board

§