Application Note 53

Configuring ARM Caches

Document number: ARM DAI 0053A

Issued: January 1998

ENGLAND

Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY

Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN

Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA

ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

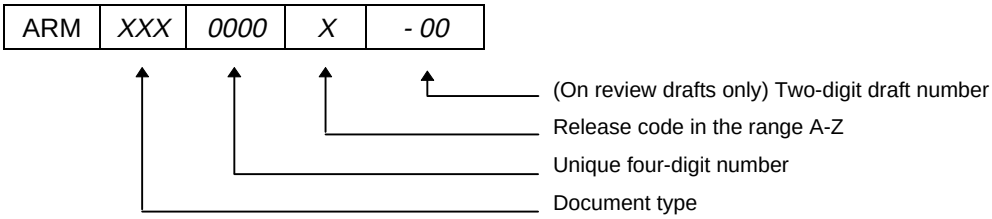
ARM
Advanced RISC Machines

Proprietary Notice

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.



Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	January 1998	SKW	Released

Table of Contents

1 Introduction

In order to get maximum performance from processors such as ARM710a and StrongARM SA-110 it is necessary to enable the cache. An application may have this done for it by, for example, the underlying microkernel on the system. However, if there is no such kernel an application will have to enable the caches itself.

There are two main cache systems used on ARM processors. These are:

- Memory Management Unit (for example ARM610, ARM710a, SA-110)
- Protection Unit (for example ARM940T)

This Application Note presents some sample code for enabling the cache on both types of ARM processor, and an overview of the `PageTables` module from ARMulator.

It also discusses:

- repetitive assembly in `armasm` and `tasm`
- `armasm` and `tasm` conditional assembly
- macros in `armasm` and `tasm`
- inline assembler with the C compiler
- inline functions

2 Overview

This section gives an overview of the cache systems used in ARM processors.

2.1 Memory Management Unit (MMU)

The memory management unit provides a full virtual memory system. For a fuller description refer to the *ARM Architecture Reference Manual* (ARM DDI 0100).

In brief, it uses off-chip page tables to describe to the processor:

- A virtual to physical address mapping
- Access permissions
- Cache and write-buffer control.

Three sizes of page (1MB, 64kB and 4kB) are supported. (Sub-pages of 16kB and 1kB are also provided for access control.) An additional system of "Domains" operates to provide efficient access protection in a multithreaded environment.

This system allows, for example, multiple virtual address spaces with demand paging and swapping. Flavors of the UNIX operating-system have been ported to ARM-powered computers using such a memory management unit.

The advantages of this system are:

- Full control over memory at a fine granularity
- Domain-based protection
- Virtual to physical address translation.

The main disadvantage is that it requires in-memory pagetables (the cache cannot be enabled without MMU enabled).

2.2 Protection Unit (PU)

The protection unit provides access and cache control, for a more embedded environment. For a fuller description refer to the *ARM940T Data Sheet* (ARM DDI 0092).

In brief, the protection unit has a set of on-chip registers, which hold descriptions of:

- Access permissions
- Cache and write-buffer control

for up to eight (programmable) regions of memory.

This system allows basic memory protection and cache control for use in, for example, an embedded application.

The advantages of this system are:

- Access control held entirely on-chip (no need for any off-chip tables)
- Provides four levels of access control, cache and write-buffer control.

The disadvantages are:

- Small number of regions
- Restrictions on region size and alignment.

3 Example

This section describes how to enable caches, using a simple example.

3.1 Memory layout

Although the two systems are different, both use coprocessor 15 to control the system. Both systems have enough common functionality to distinguish which is in use. As an example, consider the memory map shown in ***Figure 1: Example memory layout***.

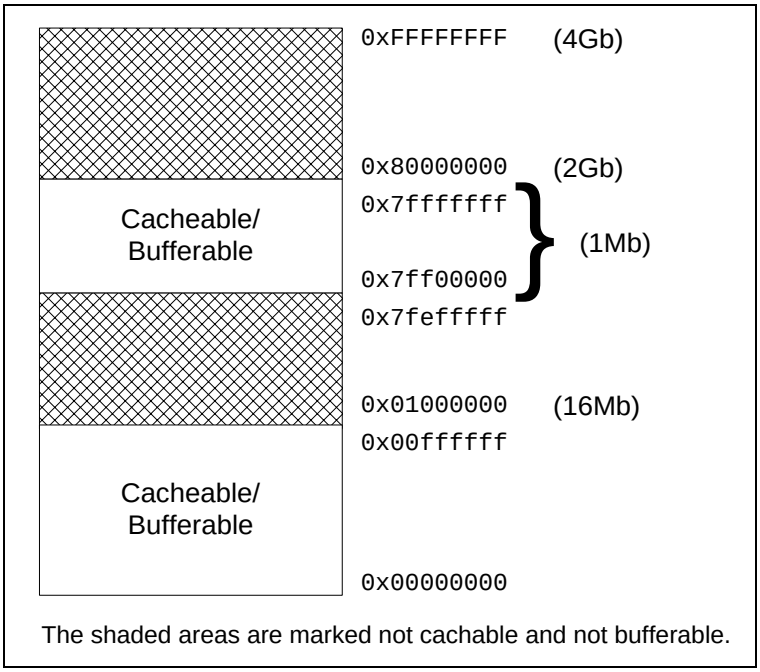


Figure 1: Example memory layout

- The bottom 16MB of memory are marked as cacheable. This is where application code and the heap would be based.
- One megabyte of memory immediately below 2GB is also marked as cacheable, as this is where the stack is placed.
- The rest of memory is neither cacheable nor bufferable. A real system may also mark that memory as “no access” (abort generating).

Code to set up this memory map is given in C and ARM Assembler.

The MMU architecture requires the top-level pagetable to be aligned to a 16kB boundary. As it is not possible to specify alignment from the C compiler, an assembler file is needed.

The sample C code makes use of the inline assembler feature of the compiler to control coprocessor 15. This code can be used in full, or stripped down for a specific processor.

3.2 Assembler source

By default, this code defines an empty pagetable, which the C code will later fill in with the pagetable entries.

The empty table uses the `ALIGN` and `NOINIT` AREA directives to ensure that the area is aligned to a 16kB (2^{14}) boundary and is allocated at runtime.

A different approach is to generate a fixed pagetable at compile-time using the assembler. This code is also supplied, but is not assembled.

The following section describes some of the techniques used to generate this table. For a complete description of the assembler, refer to the *Software Development Toolkit Reference Guide* (ARM DUI 0041).

3.2.1 Repetitive assembly

The second approach uses repetitive assembly to generate a pagetable. The following code fragment (from `pagetab.s` on ***3.2.3 The pagetab.s file*** on page 7) demonstrates this idea:

```
        GBLA    counter
counter SETA 0

        ; area 1 - 0->16MB, cacheable
        WHILE counter < 16
        L1Entry SECTION, (counter:SHL:20), 0, \
                        C_BIT+B_BIT, ALL_ACCESS
counter SETA counter + 1
        WEND
```

Where:

- `GBLA counter` declares a global numeric variable called `counter` which is initialized to zero using the `SETA` directive.
- The `WHILE ... WEND` construct is used to assemble the lines repeatedly between `WHILE` and `WEND`.
- In this example, the loop body is assembled for `counter` = `0`, `1`, `2` ... `14` and `15`, but because the looping condition is checked at the top of the loop, it is possible for the code between a `WHILE` and a `WEND` never to be assembled. For example, if `counter` were initialized to `16`, the body of the `WHILE ... WEND` loop would not be assembled at all.
- Each time around the loop the macro `L1Entry` is called (with five arguments), and `counter` is incremented.

3.2.2 Macro usage and conditional assembly

The following code fragment for `L1Entry` is also taken from `pagetab.s`:

```
MACRO
L1Entry $type, $addr, $dom, $ucb, $acc
[ $type = SECTION
  DCD ( (($addr) :AND: &FFF00000) :OR: \
        (($acc) :SHL: 10) :OR: \
        (($dom) :SHL: 5) :OR: $ucb :OR: $type )
  MEXIT
]
[ $type = PAGE
  DCD ( (($addr) :AND: &FFFFFC00) :OR: \
        (($dom) :SHL: 5) :OR: \
        (($ucb) :AND: U_BIT) :OR: $type )
|
  DCD 0 ; Invalid Level 1 Page Table Entry
]
MEND
```

Note that a backslash breaks a logical line of assembly language across two physical lines. However, there must be no character after the backslash on the line.

The macro definition is enclosed between `MACRO` and `MEND`. The first line of the definition gives the macro's name and lists its parameters.

The body of the macro illustrates the use of `[` (`IF`) ... `]` (`ENDIF`) and `[` (`IF`) ... `|` (`ELSE`) ... `]` (`ENDIF`) to assemble different code conditional on a value known at assembly-time. In this example, the controlling expressions of the `IF`s involve a macro parameter (`$type`) which gets its value when the macro is used.

This macro definition also uses the `MEXIT` directive to exit processing of a macro before the `MEND` directive is reached. (The `MEXIT` directive is similar to a `return` statement in a C function.)

3.2.3 The pagetab.s file

```
; pagetab.s - pagetable definition
; Copyright (C) 1997 Advanced RISC Machines Limited.
; All Rights Reserved

                        GBLL     BlankPageTable
BlankPageTable          SETL     {FALSE}

; Generate either a blank page table for the C code to fill in, or
; generate a full page table at assemble-time

        IF BlankPageTable

        AREA            pagetab, ALIGN=14, DATA, NOINIT

        ; Generate a blank page table with the % directive

        EXPORT          pagetable
pagetable
        % 16*1024

        ELSE

        ; Generate a full page table, using repetitive assembly

; Access Permissions - not shifted into position.
NO_ACCESS  * 0 ; Depending on the 'R' and 'S' bit, 0
SVC_R      * 0 ;  represents one of these access
ALL_R      * 0 ;  permissions (see A.R.M.)
SVC_RW     * 1
NO_USR_W   * 2
ALL_ACCESS * 3

; U, C and B bits in their correct positions (see A.R.M.)
U_BIT      * 16
C_BIT      * 8
B_BIT      * 4

; Entry type
SECTION    * 2
PAGE       * 1
INVALID    * 0

        MACRO
        L1Entry $type, $addr, $dom, $ucb, $acc
        [ $type = SECTION
          DCD ( (($addr) :AND: &FFF00000) :OR: \
                (($acc) :SHL: 10) :OR: \
                (($dom) :SHL: 5) :OR: $ucb :OR: $type )
          MEXIT
        ]
        [ $type = PAGE
          DCD ( (($addr) :AND: &FFFFFC00) :OR: \
                (($dom) :SHL: 5) :OR: \
                (($ucb) :AND: U_BIT) :OR: $type )
        |
          DCD 0 ; Invalid Level 1 Page Table Entry
        ]
        MEND
```

Example

```
        AREA        pagetab, ALIGN=14, DATA

        EXPORT      pagetable
pagetable

; Create the pagetable using repetitive assembly
        GBLA    counter
counter SETA 0

        ; area 1 - 0->16MB, cacheable
        WHILE counter < 16
        L1Entry SECTION, (counter:SHL:20), 0, \
                U_BIT+C_BIT+B_BIT, ALL_ACCESS
counter SETA counter + 1
        WEND

        ; area 2 - up to 2GB, not-cacheable
        WHILE counter < 2047
        L1Entry SECTION, (counter:SHL:20), 0, \
                0, ALL_ACCESS
counter SETA counter + 1
        WEND

        ; area 3 - last page under 2GB, cacheable
        L1Entry SECTION, (2048 :SHL: 20), 0, \
                U_BIT+C_BIT+B_BIT, ALL_ACCESS
counter SETA counter + 1

        ; area 4 - rest of memory, not cacheable
        WHILE counter < 4096
        L1Entry SECTION, (counter:SHL:20), 0, \
                0, ALL_ACCESS
counter SETA counter + 1
        WEND

        ENDIF

        END

; end of file pagetab.s
```

3.3 C source

`pagetab.c` contains code for enabling the cache and creating a page table.

Since the cache is controlled via coprocessor 15, and there is no mechanism in standard C to access coprocessor registers, this code uses the `__asm` extension to do so. Refer to the *Software Development Toolkit Reference Guide* (ARM DUI 0041) for complete details of the C compiler.

The code exports a function, `initMMU,` which examines coprocessor 15 to determine the processor type and initialize the cache appropriately.

Since this code accesses coprocessor 15 it must be called only from a privileged mode. If called from User Mode, the accesses are ignored by the MMU or PU, and the ARM faults the instructions as undefined.

3.3.1 Inlined functions

For example, the function, from `pagetab.c`:

```
__inline unsigned long readCP15R0(void)
{
  unsigned long id;

  __asm { MRC P15, 0, id, c0, c0; }

  return id;
}
```

uses both the `__inline` and `__asm` directives of the compiler.

`__inline` instructs the compiler to generate the code at point of use, rather than creating a function and calling it.

As `__asm` is a statement block rather than an expression, it cannot easily be made into a macro.

Later `__inlined` functions, such as `mmuFlushCache`, contain conditional statements. However, these are optimized away by the compiler, as the function is called with a known argument.

Note *Inlining of functions is disabled by the compiler when the `-g` (generate debug tables) command line option is given.*

3.3.2 The inline assembler

`__asm` introduces an assembler statement block. This is used as there is no other way for the compiler to generate an `MCR` (read coprocessor) instruction. The compiler is left to allocate a register for `id` and generate an appropriate instruction for it. This demonstrates one use of `__asm`.

Other `__asm` statements, such as in `puWriteRegion0`, use more complicated expressions as arguments in the inline assembler. The compiler generates the code for this expression along with the inline assembler code. In these examples, however, the result of the expression is computed by the compiler at compile-time, so the expression is optimized away.

3.3.3 The pagetab.c file

```
/* pagetab.c - Generic code for setting up pagetables. */
/* Copyright (C) 1997 Advanced RISC Machines Limited.
 * All Rights Reserved. */

/* pagetable declared external -
 * it needs to be in an assembler file so it can be aligned
 * to a 16kB boundary.
 * Note: extern unsigned long *pagetable; is NOT the same
 * thing (see ANSI C standard), and will not work in this
 * instance.
 */
extern unsigned long pagetable[4096];

#define BlankPageTable 1

#if BlankPageTable
/* pagetab.s exports a blank page table - use code to
 * fill it in.
 */
enum access {
  NO_ACCESS,
  NO_USR_W,
  SVC_RW,
  ALL_ACCESS
  };

enum entrytype {
  INVALID,
  PAGE,
  SECTION
  };

#define U_BIT 16
#define C_BIT 8
#define B_BIT 4

#define L1Entry(type,addr,dom,ucb,acc) \
  ( (type == SECTION) ? ( ((addr) & 0xfff00000) |        \
                          ((acc) << 10) | ((dom) << 5) | \
                          (ucb) | (type) ) :             \
    (type == PAGE) ? ( ((addr) &0xfffffc00) |            \
                       ((dom) << 5) |                    \
                       ((ucb) & U_BIT) | (type) ) :      \
    0)
```

```
static unsigned long const *createPageTable(void)
{
  int i;
  unsigned long *p = pagetable;
  unsigned long entry;

  /* first region - 16MB of cacheable/bufferable */
  entry = L1Entry(SECTION, 0, 0,
                  U_BIT|C_BIT|B_BIT, ALL_ACCESS);
  for (i = 0; i < 16; i++) {
    *p++ = entry | (i << 20);
  }

  /* second region - upto 1MB short of 2GB,
   * not cacheable/not bufferable */
  entry = L1Entry(SECTION, 0, 0, 0, ALL_ACCESS);
  for (; i < 2048 - 1; i++) {
    *p++ = entry | (i << 20);
  }

  /* third region - up to 2GB cacheable/bufferable */
  entry = L1Entry(SECTION, 0, 0,
                  U_BIT|C_BIT|B_BIT, ALL_ACCESS);
  for (; i < 2048; i++) {
    *p++ = entry | (i << 20);
  }

  /* rest of memory - not cacheable/not bufferable */
  entry = L1Entry(SECTION, 0, 0, 0, ALL_ACCESS);
  for (; i<4096; i++) {
    *p++ = entry | (i << 20);
  }

  return pagetable;
}
#else  /* BlankPageTable */
/* pagetab.s exports a complete page table */
#define createPageTable() ((unsigned long const *)pagetable)
#endif

typedef enum {
  arch_3,
  arch_4
} architecture;

typedef enum { FALSE, TRUE } bool;

__inline unsigned long readCP15R0(void)
{
  unsigned long id;

  __asm { MRC P15, 0, id, c0, c0; }

  return id;
}
```

Example

```
/*
 * Inline functions for MMU functions
 */

__inline void mmuSetPageTabBase(unsigned long const *pagetab)
{
  __asm { MCR  P15, 0, pagetab, c2, c0; }
}

__inline void mmuSetDomainAccessControl(unsigned long flags)
{
  __asm { MCR  P15, 0, flags, c3, c0; }
}

__inline void mmuFlushCache(architecture arch)
{
  unsigned long dummy;


  switch (arch) {
  case arch_3:
    __asm { MCR  P15, 0, dummy, c7, c0, 0 }
    break;
  case arch_4:
  default:
    __asm { MCR  P15, 0, dummy, c7, c7, 0 }
    break;
  }
}

__inline void mmuFlushTLB(unsigned arch)
{
  unsigned long dummy;

  switch (arch) {
  case arch_3:
    __asm { MCR  P15, 0, dummy, c5, c0; }
    break;
  case arch_4:
  default:
    __asm { MCR  P15, 0, dummy, c8, c7, 0; }
    break;
  }
}
```

```
#define MMU_I 0x1000
#define MMU_Z 0x0800
#define MMU_F 0x0400
#define MMU_R 0x0200
#define MMU_S 0x0100
#define MMU_B 0x0080
#define MMU_L 0x0040
#define MMU_D 0x0020
#define MMU_P 0x0010
#define MMU_W 0x0008
#define MMU_C 0x0004
#define MMU_A 0x0002
#define MMU_M 0x0001

__inline void enable(unsigned long flags)
{
  __asm { MCR P15, 0, flags, c1, c0; }
}

/* StrongARM clock switching */

__inline void saClockSwitch(bool enable)
{
  unsigned long dummy;

  if (enable) {
    __asm { MCR  P15, 0, dummy, c15, c1, 2; }
  } else {
    __asm { MCR  P15, 0, dummy, c15, c2, 2; }
  }
}
```

Example

```
/*
 * Protection unit
 */

typedef enum {
  size_4k=11,size_8k,   size_16k,  size_32k,
  size_64k,  size_128k, size_256k, size_512k,
  size_1M,   size_2M,   size_4M,   size_8M,
  size_16M,  size_32M,  size_64M,  size_128M,
  size_256M, size_512M, size_1G,   size_2G,
  size_4G
} regionSize;

__inline void puWriteRegion0(
    unsigned long base, regionSize size, bool enable)
{
  __asm {
    MCR  P15, 0, base + (size << 1) + enable, c6, c0, 0;
  }
}

__inline void puWriteRegion1(
    unsigned long base, regionSize size, bool enable)
{
  __asm {
    MCR  P15, 0, base + (size << 1) + enable, c6, c1, 0;
  }
}

__inline void puWriteRegion2(
    unsigned long base, regionSize size, bool enable)
{
  __asm {
    MCR  P15, 0, base + (size << 1) + enable, c6, c2, 0;
  }
}

__inline void puSetCacheable(unsigned long regions)
{
  __asm { MCR  P15, 0, regions & 0xffUL, c2, c0, 0 }
}

__inline void puSetBufferable(unsigned long regions)
{
  __asm { MCR  P15, 0, regions & 0xffUL, c3, c0, 0 }
}

__inline void puSetProtection(unsigned long regions)
{
  __asm { MCR  P15, 0, regions & 0xffffUL, c5, c0, 0 }
}

__inline void puFlushCache(void)
{
  mmuFlushCache(arch_3);    /* same as for arch_3 */
}
```

```
unsigned long initMMU(void)
{
  unsigned long id;

  id = readCP15R0();

  switch (id & 0xff00) {
  case 0x0600:                      /* ARM6xx */
  case 0x7000:                      /* ARM70x */
  case 0x7100:                      /* ARM71x */
    /* architecture v. 3 */
    mmuSetPageTabBase(createPageTable());
    mmuSetDomainAccessControl(3);
    mmuFlushCache(arch_3);
    mmuFlushTLB(arch_3);
    enable(MMU_D+MMU_P+MMU_W+MMU_C+MMU_M);
    break;

  case 0x8100:                      /* ARM810 */
    /* architecture v. 4 with branch prediction */
    mmuSetPageTabBase(createPageTable());
    mmuSetDomainAccessControl(3);
    mmuFlushCache(arch_4);
    mmuFlushTLB(arch_4);
    enable(MMU_Z+MMU_D+MMU_P+MMU_W+MMU_C+MMU_M);
    break;

  case 0xa100:                      /* StrongARM SA-110 */
    /* architecture v. 4 with I cache */
    /* enable I cache - doesn't need MMU enabling */
    enable(MMU_I+MMU_D+MMU_P);
    saClockSwitch(TRUE);
    mmuSetPageTabBase(createPageTable());
    mmuSetDomainAccessControl(3);
    mmuFlushCache(arch_4);
    mmuFlushTLB(arch_4);
    enable(MMU_I+MMU_D+MMU_P+MMU_W+MMU_C+MMU_M);
    break;

  case 0x9400:                      /* ARM940 */
    /* protection unit */
    /* region 0 will cover the whole memory, then two
     * regions are used to cover the cacheable parts */
    puSetCacheable(~1ul);   /* not region 0 */
    puSetBufferable(~1ul);  /* not region 0 */
    puSetProtection(~0ul);  /* all regions */
    /* background region - not cacheable */
    puWriteRegion0(0x0, size_4G, TRUE);
    /* first region - 16MB */
    puWriteRegion1(0x0, size_16M, TRUE);
    /* second region - 1MB */
    puWriteRegion2((1ul << 31) - (1ul << 20), size_1M, TRUE);
    puFlushCache();
    enable(MMU_W+MMU_C+MMU_M);
    break;
  }

  return id;
}

/* end of file pagetab.c */
```

4 ARMulator

This section describes setting up the cache in ARMulator, using the Page Tables model.

4.1 Introduction

The ARMulator provides a debugging and benchmarking platform for software development. For full details on the ARMulator, refer to:

- *Software Development Toolkit Reference Guide* (ARM DUI 0041), ***Chapter 9 The ARMulator***
- *Software Development Toolkit User Guide* (ARM DUI 0040), ***Chapter 5 The ARMulator***
- *Application Note 32: The ARMulator* (ARM DAI 0032)
- *Application Note 51: ARMulator Cache Models* (ARM DAI 0051)

This Application Note refers to the Page Tables model.

In releases up to and including release 2.11 of the toolkit, the Page Tables model only supports MMU-based processors. However, it defines memory in terms of regions, in a similar way to the Protection Unit, with the following differences:

- Regions must be multiples of 1MB in size (but may be *any* integer multiple)
- Regions must lie on *any* megabyte boundary, rather than one dictated by the region size
- The priority of regions applies (`Region[1]` has priority over `Region[0]`, and so on).

On every reset the Page Tables model:

- Writes a set of page tables to memory
- Sets up the MMU, configuring
  - the page tables base address
  - the Domain Access Control register
- Flushes the cache and TLB
- Enables the cache, write-buffer and MMU (or not, according to the configuration).

4.2 The armul.cnf file

The Page Tables model is configured using the ARMulator configuration file `armul.cnf`. For fuller details of `armul.cnf` refer to *Application Note 52: The ARMulator Configuration File* (ARM DAI 0052).

To modify the configuration, locate the `PageTables` region of the file. This declares the default options for the MMU, and several regions of memory. Details on configuring the model are given in the *Software Development Toolkit User Guide* (ARM DUI 0040), ***5.11 Supplied Models: Page Table Manager***.

A sample configuration is given for the memory map defined in ***3.1 Memory layout*** on page 4. The sample regions are generated using the same three regions as the Protection Unit example.

4.2.1 Configuration

```
;; Page tables
{ Pagetables
MMU=Yes
AlignFaults=No
Cache=Yes
WriteBuffer=Yes
Prog32=Yes
Data32=Yes
LateAbort=Yes
BigEnd=No
BranchPredict=Yes
ICache=Yes
PageTableBase=0xa0000000
DAC=0x00000003

; region 0 will cover the whole memory, then two
; regions are used to cover the cacheable parts
{ Region[0]
; background region - not cacheable
VirtualBase=0
PhysicalBase=0
Pages=4096
Cacheable=No
Bufferable=No
Updateable=No
Domain=0
AccessPermissions=3
Translate=Yes
}

{ Region[1]
; first region - 16MB
VirtualBase=0
PhysicalBase=0
Pages=16
Cacheable=Yes
Bufferable=Yes
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}

{ Region[2]
; second region - 1MB
VirtualBase=0x7ff00000
PhysicalBase=0x7ff00000
Pages=1
Cacheable=Yes
Bufferable=Yes
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}
}
```