Application Note 51

ARMulator Cache Models

Document number: ARM DAI 0051A

Issued: January 1998

Copyright Advanced RISC Machines Ltd (ARM) 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

Proprietary Notice

ARM and the ARM Powered logo are trademarks of Advanced RISC Machines Ltd.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

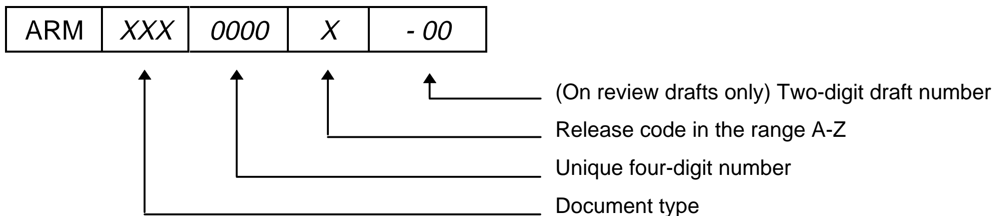
The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

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 **2**

2 Using the Cache Models **3**
- 2.1 Overview 3
- 2.2 MMUlator 3
- 2.3 StrongMMU 9

3 Writing a New Cache Model **10**
- 3.1 Initialization 10
- 3.2 Memory access 11
- 3.3 Example 12

Open Access

1 Introduction

This Application Note covers two areas:

- Using the cached processor models in the ARM Software Development Toolkit
- Writing your own cache model.

Open Access

2 Using the Cache Models

This section describes how to use and reconfigure the cache models supplied as part of ARMulator in the Software Development Toolkit.

2.1 Overview

The ARMulator supplied with the ARM Software Development Toolkit contains two cache models, which are used to model various cached processors.

The two models are the "generic" cache model—MMUlator—and the "harvard" cache model—StrongMMU. These model a wide range of Cached ARM processors, as shown in ***Table 1: Cache models and processors***.

Cache Model	Processors
MMUlator	ARM600, ARM610, ARM700, ARM710, ARM710a, ARM810
StrongMMU	SA-110

Table 1: Cache models and processors

Two models are needed, as the "generic" model does not model the split instruction/data cache architecture used by SA-110.

To support all these processors, the MMUlator model is configurable. It is possible to generate your own configurations for caches by editing `armul.cnf`.

Note *Reconfiguring a cache model under ARMulator is relatively easy; it is considerably more complex, and expensive, in silicon. Before changing cache sizes when developing and benchmarking code, ensure that a device is available with the parameters you have configured.*

2.2 MMUlator

2.2.1 Editing armul.cnf

Refer to *Application Note 52: The ARMulator Configuration File* (ARM DAI 0052) for details on how to edit `armul.cnf` to change these parameters. Essentially, there are two approaches:

- Create a new "processor", using the "alias" facility of the configuration file to form a basis (this is the recommended approach).
- Edit an existing processor's definition.

For example, to create your own cached processor model that is something like an ARM710a:

1. Locate `ARM710a` in the `Processors` section. It is defined as an alias for ARM7.
2. Add your own variant immediately below this (the new lines are marked with "|"), for example:

```
  ARM710a:Processor=ARM710a
| ARM7XX:Processor=ARM7XX
  ARM710a:Memory=ARM710a
| ARM7XX:Memory=ARM7XX
```


Open Access

3 Add “plumbing” for this new processor outside the definition of ARM7 (as for ARM710):

```
  ARM710a=ARM7
| ARM7XX=ARM7
```

4 Locate `ARM710` in the `Memories` section. It is defined as an alias for `MMUlator`.

5 Add two levels of plumbing for your new processor. Firstly at the outer-most level, alongside that for `ARM710`:

```
  ARM710a=MMUlator
| ARM7XX=MMUlator
```

and then inside the definition of `MMUlator`:

```
  ARM710a=ARM700
| ARM7XX=ARM700
```

6 Copy the configuration for `ARM710a` inside the `ARM700` block:

```
  ARM710a:NoCoprocessorInterface
  ARM710a:CacheWords=4
  ARM710a:CacheBlocks=128
  ARM710a:ChipNumber=0x711

| ARM7XX:NoCoprocessorInterface
| ARM7XX:CacheWords=4
| ARM7XX:CacheBlocks=128
| ARM7XX:ChipNumber=0x7ff
```

7 Edit these parameters, and add extra ones, as needed.

These parameters are explained in greater detail in the following sections.

2.2.1.1 Size of cache parameters

MMUlator models a simple cache consisting of lines arranged in sets. Each line consists of a number of words. For example, see ***Figure 1: Example cache layout***.

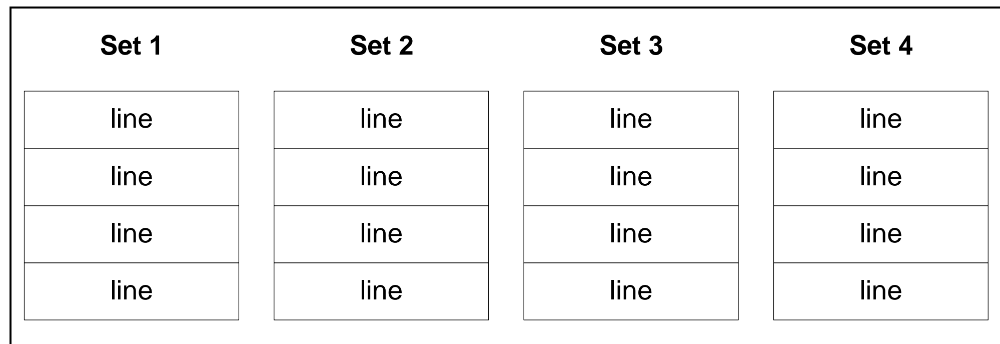


Figure 1: Example cache layout

This would be called a 256-byte, 4-way set-associative cache, as:

- There are 256 bytes in total (4 bytes/word * 4 words/line * 4 lines/set * 4 sets)
- There are four lines in each set (hence “4-way”).

An address is decoded to locate data in this sample cache, as shown in ***Figure 2: Decoding a location in the cache***.

Open Access

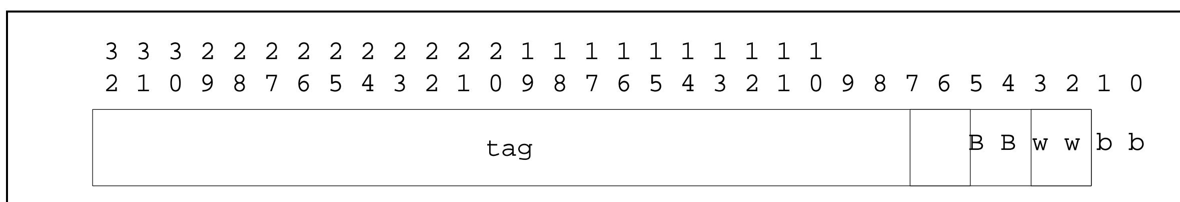


Figure 2: Decoding a location in the cache

- `BB` chooses one of the four sets
- `ww` chooses the word in the line
- `bb` chooses the byte in the word (alternatively, you might regard `wwbb` as a byte-address inside the line)
- `tag` is used to choose the line from the set. (This is the associative part of the cache. Each set contains a four-entry associative lookup array against which the tag is compared to find the line.)

The size and shape of the cache can be altered by changing the three parameters shown in ***Table 2: Cache size parameters***.

Tag	Description
`CacheWords`	The number of words in a cache line.
`CacheAssociativity`	The number of lines in a set (or block).
`CacheBlocks`	The number of sets (blocks) in the cache.

Table 2: Cache size parameters

For example, ARM610 (4kB, 64-way set-associative) defines these as:

```
CacheWords=4
CacheAssociativity=64
CacheBlocks=4
```

whereas ARM710 (8kB, 4-way set-associative) defines them as:

```
CacheWords=8
CacheAssociativity=4
CacheBlocks=64
```

To reduce the sample cache’s size from 8kB to 4kB, the number of sets in the cache could be halved:

```
ARM7XX:CacheWords=4
ARM7XX:CacheBlocks=64
```

This leaves the size at 4 * 4 * 4 * 64 = 4096 bytes.

Alternatively, `MMUlator` can work out the number of sets for itself, if you specify a cache size:

```
ARM7XX:CacheWords=4
ARM7XX:CacheSize=4kB
```

Note *Setting `Verbose=True` at the top of `armul.cnf` causes `MMUlator` to report, when started, the dimensions of the cache it is using.*

`CacheWords` and `CacheBlocks` **must** be a power of two. `CacheAssociativity` should also be a power of two for most implementations.

Open Access

2.2.1.2 Other size parameters

MMUlator also models the write-buffer (`WB`) and memory management unit (`MMU`). The sizes of these units are also controlled through `armul.cnf`, as shown in ***Table 3: MMU and write-buffer parameters***.

Tag	Description
`WriteBufferWords`	The depth of the write-buffer; that is, how many words it can hold.
`WriteBufferAddrs`	The depth of the “translated-address buffer”. This defines how many writes can be in the write-buffer. (A single write may be many sequential words, as from, for example, an `STM`.)
`TLBSize`	The number of entries in the MMU’s “translation-lookaside buffer”.

Table 3: MMU and write-buffer parameters

2.2.2 Cache replacement

MMUlator supports either random or cyclic replacement. (Least recently used is **not** supported.)

The type of replacement is controlled by the parameter `Replacement`, which may have the values shown in ***Table 4: Replacement algorithms***.

Value	Replacement Algorithm

`Random`
`PseudoRandom`
`RNG`
	Random replacement (controlled by `RNG` tag)

`RoundRobin`	Cyclic counter, 1 per set
`GlobalRoundRobin`	Cyclic counter, 1 per cache

Table 4: Replacement algorithms

For random replacement, a further option, `RNG`, controls the type of random number generator used. See ***Table 5: Random number generators***.

Value	Random Number Generator
`6`	33-bit Linear-Feedback Shift Register (LFSR), as used on ARM3, ARM600, ARM610 and so on
`7`	16-bit LFSR, as used in ARM7-series
`8`	LFSR with counter, for lock-down caches
`anything else`	C library `rand()` function

Table 5: Random number generators

Open Access

The cache-write policy is configurable via the `CacheWrite` tag (see ***Table 6: Cache write policies***).

Value	Description
`WriteThrough`	Write-through cache. Writes to cached locations are updated in the cache, and sent to memory, possibly using the write-buffer.
`WriteBack`	Write-back cache. Writes to cached locations go just to the cache, marking the line as “dirty”. “Dirty” lines are written out to main memory entirely when the line gets replaced.

Table 6: Cache write policies

2.2.3 Clocks

Various processors have different clock models. Whereas `MMUlator` does support some of these models, only “Synchronous Clock” is accurately modeled.

Conceptually, there are three clocks (two clocking domains), shown in ***Table 7: Clocks on a cached processor***.

Clock	Description
`MCLK`	the memory system clock
`FCLK`	the fast processor clock
`GCLK`	the clock used inside the processor, which is switched between `FCLK` and `MCLK` on demand

Table 7: Clocks on a cached processor

Synchronous clocking has `FCLK` as an exact multiple of `MCLK`. Asynchronous clocking has the two clocks unrelated. (The distinction is made for technical reasons, to do with synchronization between the clocks, which are beyond the scope of this document.)

2.2.3.1 Clocking parameters

Clocking is controlled by a number of configuration options, shown in ***Table 8: Clocking parameters***.

Value	Description
`PLLClock`	Chip has on-board PLL clock source
`IdleCycles`	Whether to model idle cycles (see below)
`MCCfg`	Ratio of `FCLK` to `MCLK` (synchronous)
`sNa`	Synchronous/Not asynchronous

Table 8: Clocking parameters

In practice synchronous clocking should always be used in MMUlator.

Open Access

MMUlator uses the clock speed provided by the debugger as FCLK and derives MCLK from it, based on the value of MCCfg. If no clock speed is provided, MCCfg is nonetheless used to determine the ratio of internal to external clock cycles, which affects cycle counts shown in $statistics.

IdleCycles controls whether to call the external memory cycle on those cycles where the processor is idle on its external bus (because, perhaps, it is running code out of cache). Not doing so speeds up simulation, but causes incorrect values to be reported for the simulated execution time. Therefore this option should not be set when benchmarking code.

2.2.4 Miscellaneous parameters

Tag	Description
ReplaceTicks	Extra-delay (in cycles) for each refill.
CacheType	Controls interlocks on sequential accesses to the same cache region. For example, CacheType=ARM6; CacheType=ARM8
HasRFlag	MMU has the “R” bit in the control register.
Has26BitConfig	Control register has “P” and “D” bits.
HasBranchPrediction	Control register has “Z” bit.
HasUpdateable	Page table entries have the “U” bit.
BufferedSwap	Write half of a SWP instruction uses the writebuffer.
CacheBlockInterlock CacheWritebackInterlock	Whether there are cycle interlocks between accesses to the same cache set.
Architecture ChipNumber Revision	Controls the ID register contents, and behavior of coprocessor 15 (which varies between architectures). For example Architecture=4; ChipNumber=0x710; Revision=0xf
LockDownCache	Cache supports “lock down”.
LockDownTLB	TLB supports “lock down”.
InvalidP15AccessesUndefined	Whether illegal coprocessor 15 instructions are bounced.
HasWriteBuffer	Processor has a write-buffer.

Table 9: Miscellaneous parameters

Open Access

2.3 StrongMMU

The StrongMMU cache model has fewer configuration options, listed in ***Table 10: StrongMMU options***.

Tag	Description
CCLK CCCFG	CCLK and CCCFG pins on SA-110. (Refer to the *SA-110 Technical Reference Manual*.)
Time_Scale_Factor	Ratio of modeled time to reported time (see below).
Icache_Lines	Total number of lines in the I-Cache.
Icache_Associativity	Associativity of the I-Cache.
Dcache_Lines	Total number of lines in the D-Cache.
Dcache_Associativity	Associativity of the D-Cache.
ClockSwitching	Whether (unlike hardware) clock switching is enabled on reset.
Config=Enhanced Config=Standard	Controls the bus mode. (CONFIG pin on the SA-110–refer to the *SA-110 Technical Reference Manual*.)

Table 10: StrongMMU options

Internally, the SA-110 uses a PLL to control clocking. This can only operate at one of a small set of frequencies. It is controlled by the CCLK and CCCFG pins.

However all of these frequencies are large (between 85.7MHz and 287MHz), and the clock modeled by ARMulator has a (relatively) large resolution. (Microseconds internally, centiseconds to an application running on ARMulator.) To model one centisecond at 287MHz involves executing about two million instructions, which takes around five to ten seconds on a typical workstation. Modeling ten seconds of execution would take hours.

For accurate benchmarking this would not be acceptable. StrongMMU therefore provides the Time_Scale_Factor parameter. This declares a value by which to divide the modeled clock speed. For example, if running at 287MHz with Time_Scale_Factor set to 100, StrongMMU will instead model a SA-110 running at 2.87MHz, increasing the accuracy of the timer by a factor of 100.

Note *For simplicity, it is recommended that powers of ten are used for Time_Scale_Factor.*

Open Access

3 Writing a New Cache Model

Both the MMUlator and StrongMMU cache emulators are part of the core ARMulator, and are therefore not supplied in source form. However, writing a simple cache model of your own is not very complicated.

Essentially a cache model is a memory model like any other, except that it sits both beneath the processor model and above the real memory model.

Examples of memory models that do this are provided with the toolkit—for example `watchpnt.c`. These models are known as “veneer memory models”.

As with any memory model, it splits into two main parts:

- Initialization
- Memory Access

An example of a simple cache model is given at the end of this section.

See *Application Note 32: Rebuilding the ARMulator* (ARM DAI 0032) for details of adding this model to the ARMulator.

3.1 Initialization

In addition to the standard initialization functions of allocating a state, setting up the interface, and so on, a cache model should also:

- Use `ToolConf_Lookup(config, ARMulCnf_Memory)` to find the name of the memory model which will sit underneath the cache
- Use `ARMul_FindMemoryInterface` to locate this memory model and initialize it. To do this, the model must have its own `ARMul_MemInterface` block.

For example:

```
/* Find the name of the child memory interface */
child_name = (tag_t)ToolConf_Lookup(config, ARMulCnf_Memory);
/* Now locate it using ARMul_FindMemoryInterface. This also locates
 * its configuration for us */
if (child_name != NULL)
  child_init = ARMul_FindMemoryInterface(
                               state, child_name, &child_config);
if (child_name == NULL ||
    child_init == NULL || child_init == MemInit)
  return ARMul_RaiseError(state, ARMulErr_NoMemoryChild, ModelName);
/* Initialize the child model */
child_interf = &cache->child;
err = child_init(state, child_interf, type, child_config);
if (err != ARMulErr_NoError) {
  free(cache);
  return err;
}
```

One further consideration is that the other memory interface functions—`ReadClock`, `ReadCycles` and so on—must be passed on to the external memory model. You cannot, though, simply copy the functions over to our `ARMul_MemInterface`, as they would then be called with the wrong `handle`. Instead you have to create thin veneer functions.

Open Access

3.2 Memory access

In a simple cache model, the memory access function has to:

- Search the cache for the data being accessed.
- For a read:

  If found:

  - Read from cache;
  - Perform an idle cycle on the external bus.

  If not found:

  - Replace a line in the cache.
- For a write:

  If found:

  - Write the value to the cache;
  - Perform the write externally.

  If not found:

  - Write to external memory.

The exact details might change if you are modeling, for example, a write-back cache, a write-buffer, memory protection, and so on.

See ***3.3 Example*** on page 12 for the sample source code.

Open Access

3.3 Example

```
/* excache.c - Memory veneer that models a simple cache.
 * Copyright (C) Advanced RISC Machines Limited, 1997.
 * All rights reserved.
 */

#include "armdefs.h"
#include "armcnf.h"

#define ModelName (tag_t)"ExampleCache"

#define CACHELINES 128
#define LINESIZE 4

#define TAGMASK ( ~((LINESIZE-1) << 2) )

typedef struct {
  unsigned int bigendSig;
  /* fetchingLine contains the line and current word
   * being fetched */
  unsigned int fetchingLine;
#define FETCHINGFLAG 0x80000000L
  struct {
    ARMword tag;
    ARMword word[LINESIZE];
  } line[CACHELINES];
  ARMul_MemInterface child;
} cache_state;

#define INVALIDTAG 0xffffffffL

/*
 * ARMulator callbacks
 */
static void ConfigChange(
    void *handle, ARMword old, ARMword new)
{
  cache_state *cache = (cache_state *)handle;
  IGNORE(old);
  cache->bigendSig = ((new & MMU_B) != 0);
}

static void Interrupt(void *handle,unsigned int which)
{
  cache_state *cache = (cache_state *)handle;
  if (which & ARMul_InterruptUpcallReset) {
    /* On reset, invalidate all the cache entries */
    unsigned int i;
    for (i = 0; i < CACHELINES; i++)
      cache->line[i].tag = INVALIDTAG;
    /* Stop any pending line fetch */
    cache->fetchingLine = 0;
  }
}
```


Open Access

```
/*
 * Cache model functions
 */
static int CacheLineReplace(cache_state *cache)
{
  /* start/continue a line fetch */
  unsigned int fetching = cache->fetchingLine;
  /* extract word/line from 'fetching' */
  unsigned int word = fetching % LINESIZE;
  unsigned int line =(fetching / LINESIZE) % CACHELINES;
  ARMword addr, *ptr;
  ARMul_acc acc;

  /* construct address from cache tag */
  addr = cache->line[line].tag | (word << 2);

  /* first fetch is non-sequential */
  if (word == 0)
    acc = acc_LoadWordN;
  else
    acc = acc_LoadWordS;

  /* Call the external memory system */
  ptr = &cache->line[line].word[word];
  switch (cache->child.x.basic.access(
              cache->child.handle, addr, ptr, acc)) {
  case 1:
    /* fetch successful */
    if (word == LINESIZE-1)
      cache->fetchingLine = 0;
    else
      cache->fetchingLine = fetching+1;
    return 0;

  case 0:
    /* wait - try again next cycle */
    return 0;

  case -1: default:
    /* abort line fetch, abort core */
    cache->line[line].tag = INVALIDTAG;
    cache->fetchingLine = 0;
    return -1;
  }
}
```


Open Access

```
/* Read and write to the cache */

/*
 * Read a word from cache, doing an idle cycle on
 * the external bus
 */
static int ReadFromCacheLine(
    cache_state *cache, ARMword addr, ARMword *word,
    ARMul_acc acc, unsigned int line)
{
  /* addr has been found in line 'line' - perform access
   * and return */
  ARMword *ptr;

  ptr = &(cache->line[line].word[(addr >> 2) % LINESIZE]);

  /* do an idle cycle on the external bus */
  if (acc_ACCOUNT(acc))
    cache->child.x.basic.access(
              cache->child.handle, addr, word, acc_Icycle);

  /* Standard read code [from armflat.c] */
  switch (acc & WIDTH_MASK) {
  case BITS_8:              /* read byte */
    if (HostEndian != cache->bigendSig)
      addr ^= 3;
    *word = ((unsigned8 *)ptr)[addr & 3];
    break;

  case BITS_16: {           /* read half-word */
    /* extract half-word */
#ifndef HOST_HAS_NO_16BIT_TYPE
    /*
     * unsigned16 is always a 16-bit type, but if there is
     * no native 16-bit type (e.g. ARM!) then we can do
     * something a bit more cunning.
     */
    if (HostEndian != cache->bigendSig)
      addr ^= 2;
    *word = *((unsigned16 *)(((char *)ptr)+(addr & 2)));
#else
    unsigned32 datum;
    datum=*ptr;
    if (HostEndian != cache->bigendSig)
      addr ^= 2;
    if (addr & 2) datum <<= 16;
    *word = (datum >> 16);
#endif
  }
    break;

  case BITS_32:             /* read word */
    *word = *ptr;
    break;

  default:
    return -1;
  }

  return 1;
}
```

Open Access

```
/*
 * Write to data that's in the cache.
 * Update the cache entry, but don't do any external
 * activity (done at the higher level).
 */
static int WriteToCacheLine(
    cache_state *cache, ARMword addr, ARMword *word,
    ARMul_acc acc, unsigned int line)
{
  /* addr has been found in line 'line' - update cache
   * and return */
  ARMword *ptr;

  ptr = &(cache->line[line].word[(addr >> 2) % LINESIZE]);

  /* Standard 'write' code [from armflat.c] */
  switch (acc & WIDTH_MASK) {
    /* extract byte */
  case BITS_8:                /* write_byte */
    if (HostEndian != cache->bigendSig)
      addr ^= 3;
    ((unsigned8 *)ptr)[addr & 3] = (unsigned8)(*word);
    break;

  case BITS_16:               /* write half-word */
    if (HostEndian != cache->bigendSig)
      addr ^= 2;
    *((unsigned16 *)(((char *)ptr) + (addr & 2))) =
                                    (unsigned16)(*word);
    break;

  case BITS_32:               /* write word */
    *ptr = *word;
    break;

  default:
    return -1;
  }

  return 1;
}
```

```
/*
 * Memory veneer functions
 */
static int MemAccess(
    void *handle, ARMword addr, ARMword *word,
    ARMul_acc acc)
{
  cache_state *cache = (cache_state *)handle;
  ARMword tag;
  unsigned int line;

  /* check if we're in the middle of a line fetch */
  if (acc_ACCOUNT(acc) && cache->fetchingLine != 0)
    return CacheLineReplace(cache);

  if (acc_MREQ(acc)) {
    /* search through the cache tags */
    tag = addr & TAGMASK;
    for (line = 0; line < CACHELINES; line++)
      if (cache->line[line].tag == tag) {
        /* cache hit! */
        if (acc_READ(acc))
          return ReadFromCacheLine(
                            cache, addr, word, acc, line);
        else
          WriteToCacheLine(cache, addr, word, acc, line);
        /* and fall through to update external memory */
      }

    /* fall through from searching cache */
    if (acc_READ(acc)) {
      if (acc_ACCOUNT(acc)) {
        /* cache miss - choose a victim to replace */
        /* Unix rand() has the bad property that
         * rand() % 4 is cyclic */
        line = ((rand() >> 2) % CACHELINES);
        /* set up fetchingLine with the line number.
         * FETCHINGFLAG is used to ensure the result
         * is non-zero. start with word 0.
         */
        cache->fetchingLine = ( (line * LINESIZE) +
                                FETCHINGFLAG );
        cache->line[line].tag = tag;
        /* start a line fetch */
        return CacheLineReplace(cache);
      }
      /* else read from memory - fall through */
    }
    /* else write to physical memory - cache already
     * updated */
    /* fall through */
  }
  /* else do idle cycle on external bus - fall through */
  return cache->child.x.basic.access(
                      cache->child.handle, addr, word, acc);
}
```


Open Access

```
/* Dummy veneer functions */
/* These functions pass on their calls directly to the
 * child model. We can't use the child functions directly,
 * as we need to lookup the child's handle
 */
static unsigned long GetCycleLength(void *handle)
{
  cache_state *cache = (cache_state *)handle;
  return cache->child.x.basic.get_cycle_length(
                                     cache->child.handle);
}

static unsigned long ReadClock(void *handle)
{
  cache_state *cache = (cache_state *)handle;
  return cache->child.read_clock(cache->child.handle);
}

static const ARMul_Cycles *ReadCycles(void *handle)
{
  cache_state *cache = (cache_state *)handle;
  return cache->child.read_cycles(cache->child.handle);
}


/*
 * Initialize the memory interface
 */
static ARMul_Error MemInit(
    ARMul_State *state, ARMul_MemInterface *interf,
    ARMul_MemType type, toolconf config)
{
  cache_state *cache;
  ARMul_MemInterface *child_interf;
  armul_MemInit *child_init = NULL;
  tag_t child_name;
  toolconf child_config;
  ARMul_Error err;

  /* Find the name of the child memory interface */
  child_name = (tag_t)ToolConf_Lookup(config, ARMulCnf_Memory);

  /* Now locate it using ARMul_FindMemoryInterface.
   * This also locates it's config for us */
  if (child_name)
    child_init = ARMul_FindMemoryInterface(
                              state, child_name, &child_config);
  if (child_name == NULL ||
      child_init == NULL || child_init == MemInit)
    return ARMul_RaiseError(state, ARMulErr_NoMemoryChild,
                            ModelName);

  /* Allocate the cache state */
  cache = (cache_state *)malloc(sizeof(cache_state));
  if (cache == NULL)
    return ARMul_RaiseError(state,ARMulErr_OutOfMemory);
```


Open Access

```
  /* Initialize the child model */
  child_interf = &cache->child;

  err = child_init(state, child_interf, type, child_config);
  if (err != ARMulErr_NoError) {
    free(cache);
    return err;
  }

  /* Set up our interface with the ARMulator */
  interf->handle = cache;
  interf->read_clock = (child_interf->read_clock != NULL)
                        ? ReadClock
                        : NULL;
  interf->read_cycles = (child_interf->read_cycles != NULL)
                        ? ReadCycles
                        : NULL;

  switch (type) {
  case ARMul_MemType_Basic:
  case ARMul_MemType_16Bit:
  case ARMul_MemType_Thumb:
    /* supported */
    interf->x.basic.access=MemAccess;
    interf->x.basic.get_cycle_length =
         (child_interf->x.basic.get_cycle_length != NULL)
                        ? GetCycleLength
                        : NULL;
    break;

  default:
    *interf = *child_interf;
    free(cache);
    ARMul_ConsolePrint(state,"\
Cannot use cache on this type of memory interface.\n");
    return ARMulErr_NoError;
  }

  ARMul_PrettyPrint(state,", Simple cache");

  /* Install callbacks */
  ARMul_InstallExitHandler(state, free, cache);
  ARMul_InstallInterruptHandler(state, Interrupt, cache);
  ARMul_InstallConfigChangeHandler(
                           state, ConfigChange, cache);

  /* Initialize the model - we should get a reset anyway,
   * but better safe than sorry */
  Interrupt((void *)cache, ARMul_InterruptUpcallReset);

  return ARMulErr_NoError;
}

ARMul_MemStub ARMul_ExampleCache = {
  MemInit,
  ModelName
};

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

Open Access