Memory Management Unit

What you will learn

Learn how to:

- Initialize a BAT register
- Set up the MMU for Page Translations
  - Invalidate TLBs
  - Define size and location of Hashed Page Table
  - Configure Segment registers for a task
  - Create the initial Hashed Page Table
  - Load PTEs into Hashed Page Table

Why have an MMU?

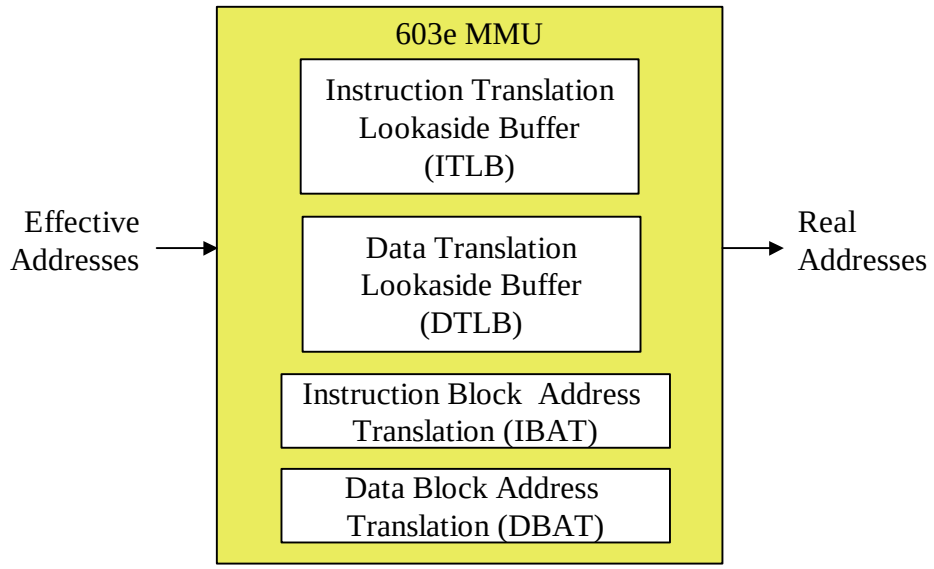
An MMU has several important uses:

- Privilege Control - prevents access of Supervisor areas by User (*Problem*) level programs.
- Cache Control - allows accesses to I/O devices to be non-cacheable while allowing other areas to be cacheable.
- Read Protection - prevents loss of data from speculative destructive reads (status flags), while allowing speculative reads from other memory areas
- Write Protection - allows selected memory areas to be read-only or treated like ROM.
- Memory Protection - restricts programs to accessing only those memory areas needed. Prevents one task from erroneously or maliciously disturbing another tasks memory area.
- Address Translation (*relocation*) - allows multiple programs that may have the same *logical* address range to reside in memory at the same time, by relocating them where convenient.

What is the 603e MMU?

Definition The 603e MMU assigns protection attributes to pages in memory and also implements address translation.

Block Diagram



TLBs and BATs

1. The TLBs are address caches (64 entry, 2-way set associative) that hold recently used 4K byte page entries.
2. The BATs are for large address ranges whose mappings don't change often.

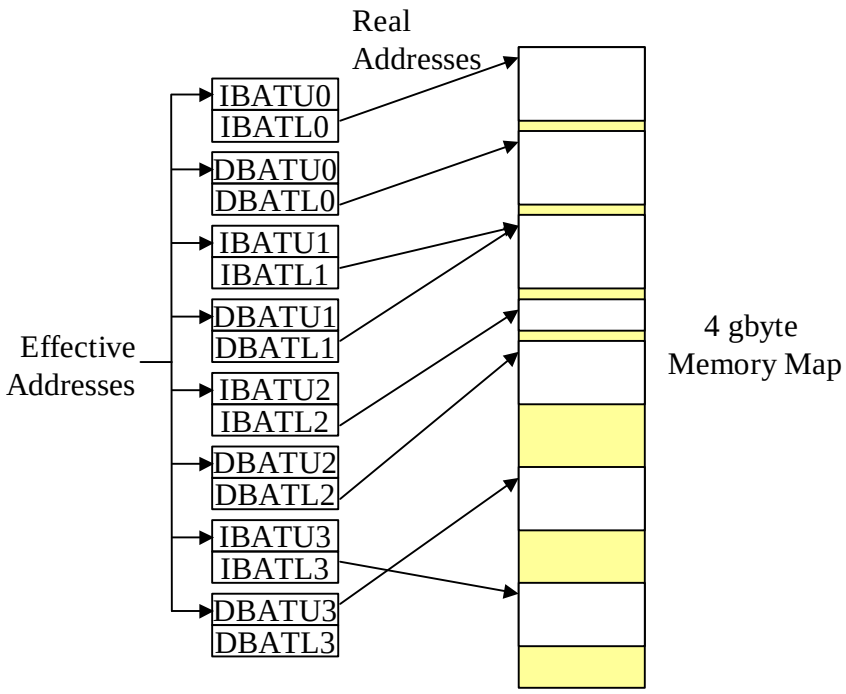
MMU Functions

- The core asserts effective addresses which are converted to real addresses by the MMU.
- The MMU also provides protection such as privilege-only access.
- Depending on the type of access and the way the MMU has been programmed, the effective address may be handled either by way of a TLB, page addressing, or a BAT.
- BAT registers are programmed from reset and changed infrequently or not at all.
- TLBs are loaded from hashed page tables and entries are changed out more frequently.

What is Block Address Translation?

Definition — If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address.

Block Diagram



Characteristics of Block Address Translation

- Block address translation defines up to 8 windows in the memory map, four for instructions and four for data.
- Data and instruction areas may overlap.
- When an effective address is asserted by the task, it is compared against the eight windows defined by the BAT registers. If there is a match, the associated real address is asserted. If there is no match, then page translation is executed.
- Blocks can vary in size from a minimum of 128K bytes to 256Mbytes.

BAT Programming Model

D or IBATxU - Upper BAT Registers, x=0-3 P. 7-25

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BEPI															Res

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Reserved			BL											V_S	V_P

D or IBATxL - Lower BAT Registers, x=0-3 P. 7-25

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BRPN															Res

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Reserved									WIMG*				Res	PP	

*Attempts to write one to W and G in the IBAT registers causes boundedly-undefined results.

Summary

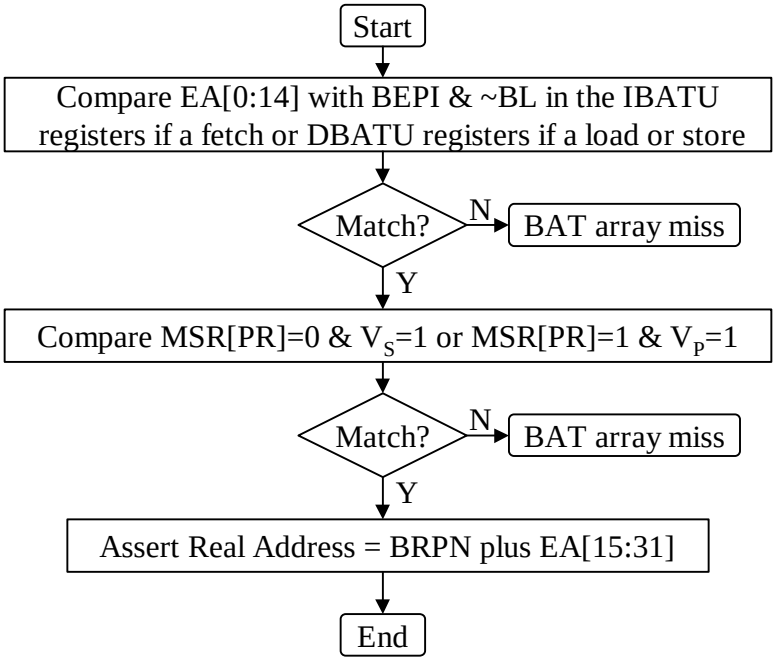
• The complete programming model for Block Address Translation consists of 8 register pairs, structured as shown, four for instructions and four for data.

How BAT Operates

Introduction

The diagram below shows the flow of BAT operation. It assumes the block protection bits, PP, are compatible. If not, a DSI or ISI exception occurs.

Flow Diagram



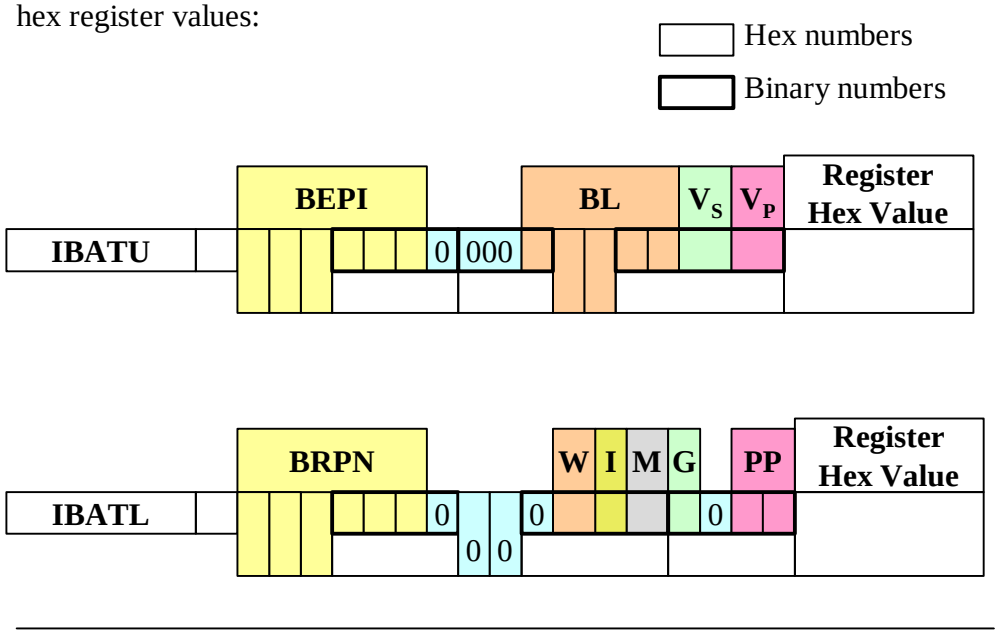
Description of Flow

1. Operation begins with the assertion of an effective address.
2. The top 15 bits of the effective address is compared with the top bits of the BEPI field in the upper BAT registers either instruction or data depending on the type of access. The number of bits that actually are compared is determined by the BL field in the upper BAT register.
3. If there is no match with any of the four BAT registers, the access is a BAT miss. Following a BAT miss the MMU attempts to translate the address using a TLB or page translation.
4. If there is a match, then the protection attributes are checked. If the protection attributes don't allow an this access, then the result is again a BAT miss. In this case, an xSI exception is taken.
5. If the protection matches, then BRPN is concatenated with bits 16-31 are used to form the real address.

How to Locate a Block of Memory (1 of 2)

Introduction — This example shows how to locate a block of memory.

Procedure — Use the following templates to fill in the individual fields, then convert to hex register values:

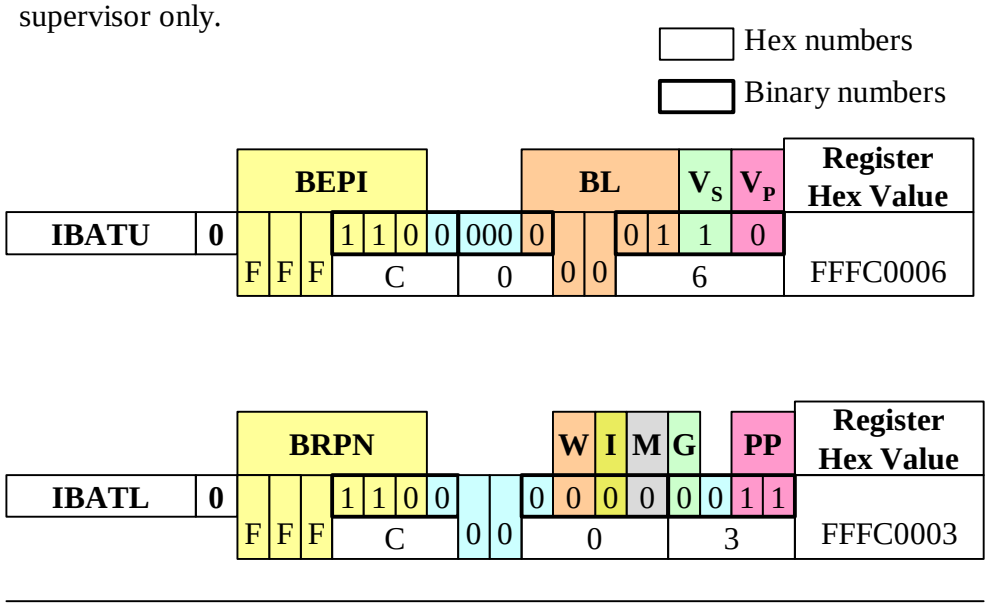


1. Because the bit fields of the BAT registers do not easily line up with the hex boundaries, these templates can be helpful.
2. The colors identify the fields within the register.
3. Boxes outlining various fields in a register contain bits if the box is outlined with a thick line or hex digits if outlined with a thin line.
4. Let’s try this out with an example.

How to Locate a Block of Memory (2 of 2)

Example

Locate a block of read-only memory at address 0xFFFC0000. The length of the memory is 256K. Assume effective address equals real address and only instructions are in the memory. Block should be accessible by the supervisor only.



1. The effective address is 0xFFFC0000 so in BEPI, the Block Effective Page Index, is initialized to 0xFFF followed by binary 110, the 3 most significant bits of 0xC.
2. Since the real address is the same as the effective address, BRPN, the Block Real Page number, is initialized similarly.
3. According to Table 7-9 on p. 7-26, the value for BL if the memory is 256K bytes is binary all zeroes except a one in the least significant bit.
4. This block is to be accessible by the supervisor only; therefore V_S is 1 and V_P is zero.
5. The problem doesn't state much about the WIMG bits, but since it is instruction-only, W and G must be zero. Nothing is said about an external master; therefore, M is assigned zero. Finally, since enabling the cache enhances performance, I is assigned zero.
6. According to Table 7-10 on p. 7-28, read-only permission is x1 so PP is assigned binary 11.
7. The final register values are shown.

How to Initialize the MMU for BAT (1 of 2)

Introduction — Here we describe the steps in initializing the MMU for BAT. Reset conditions are assumed.

Action — Here are the steps in initialization:

Step	Action	Example

1	Clear BAT registers	`li r22,0`
		`mtibatu 0,r22`
		`-`
		`mtibatu 3,r22`
		`mtdbatu 0,r22`
		`-`
		`mtdbatu 3,r22`

2	Init lower BAT register
	BRPN:real page addr[0:14]
	WIMG:attribute bits
	PP:access protection (7-25)
		`lis r22,0xFFFC`
		`ori r22,r22,3`
		`mtibatl 0,r22`

(Continued on next page)

1. First the valid bits of the upper BAT registers must be cleared..
2. Next, any required lower BAT registers are initialized.

How to Initialize the MMU for BAT (2 of 2)

Action

Step	Action	Example

3	Init upper BAT register
	BEPI:effective addr[0:14]
	BL:block length
	V_S:privilege mode valid
	V_P:problem mode valid
	(7-25)
		`lis r22,0xFFFC`
		`ori r22,r22,6`
		`mtibatu 0,r22`

4	Repeat steps 2 and 3 for each required lower-upper pair BAT registers.	
5	Execute isync	`isync`

Caution

It is the responsibility of the software to insure that an effective address is translated by only one IBAT and only one DBAT. If this is not done, results are undefined.

1. Then the lower BAT registers are initialized.
2. After all the registers are initialized, execute an isync instruction. This will insure that all previous instruction have been completed before proceeding.

Exercise-Initialize MMU for BAT (1 of 2)

A system is to have 2 memory areas for Block Address Translation. The features of each block are as follows:

	Instruction Block	Data Block
Block Start Address, Eff	0xC8000000	0xE4000000
Block Start Address, Real	0xC8000000	0xD2480000
Length	4 Mbytes	512 Kbytes
WIMG	M	WIG
Access Protection	R/O	R/W
Supervisor/User	Supervisor	Both

Write the routine to initialize this system. (see next page)

1. Here's a chance to check your understanding. Here we describe a configuration and on the next page, please complete the program.

Exercise-Initialize MMU for BAT (2 of 2)

```
li r22,0                /* init gpr to zero              */
mtibatu 0,r22           /* invalidate IBAT0              */
mtibatu 1,r22           /* invalidate IBAT1              */
mtibatu 2,r22           /* invalidate IBAT2              */
mtibatu 3,r22           /* invalidate IBAT3              */
mtdbatu 0,r22           /* invalidate DBAT0              */
mtdbatu 1,r22           /* invalidate DBAT1              */
mtdbatu 2,r22           /* invalidate DBAT2              */
mtdbatu 3,r22           /* invalidate DBAT3              */
lis r22,_______         /* init gpr for upper IBAT0L     */
ori r22,r22,____        /* init gpr for lower IBAT0L     */
mtibatl 0,r22           /* init IBAT0L                   */
lis r22,_______         /* init gpr for upper IBAT0U     */
ori r22,r22,____        /* init gpr for lower IBAT0U     */
mtibatu 0,r22           /* init IBAT0U                   */
lis r22,_______         /* init gpr for upper DBAT0L     */
ori r22,r22,____        /* init gpr for lower DBAT0L     */
mtdbatl 0,r22           /* init DBAT0L                   */
lis r22,_______         /* init gpr for upper DBAT0U     */
ori r22,r22,___         /* init gpr for lower DBAT0U     */
mtdbatu 0,r22           /* init DBAT0U                   */
isync                   /* context synchronize          */
```

How to Assign BAT Protection (1 of 2)

Notation: S:U = Supervisor:User
R/W:PT = Supervisor access is R/W:User access is Page Translation

Reference Table

If a block is to have this protection...	...then the BAT register valid bits must be...	...and PP in the BATL must be...
R/W:R/W or R/O:R/O or No access:No access	$V_S = 1$ and $V_P = 1$	10 for R/W:R/W X1 for R/O:R/O 00 for No access:No access
R/W:PT or R/O:PT or No access:PT	$V_S = 1$ and $V_P = 0$	10 for R/W:PT X1 for R/O:PT 00 for No access:PT
PT:R/W or PT:R/O or PT:No access	$V_S = 0$ and $V_P = 1$	10 for PT:R/W X1 for PT:R/O 00 for PT:No access
PT:PT	$V_S = 0$ and $V_P = 0$	

1. Here we can learn to assign the protection to a block that we desire. First of all, check the notation we use. A protection pair consists of protection in the supervisor mode, followed by a colon, followed by protection in the user mode.
2. So in the left hand column, we find the protection that we would like to give a particular block, and then use the other two columns to tell us what values go into the valid bits and the PP field.
3. For example, for a block in which we would like supervisor page translation and user read/write, we would assigned 0 to V_S, 1 to V_P, and 10 to PP.
4. No access means the DSI or ISI exception will be taken.

How to Assign BAT Protection (2 of 2)

Example

Three blocks are to be protected as follows: BAT0 is to be R/W:R/W, BAT1 is to be PT:R/O, and BAT2 is to be R/O:PT. Fill in the required protection values in the table below.

Block	V_S	V_P	PP
0	1	1	10
1	0	1	x1
2	1	0	x1

Exercise

Two blocks are to be protected as follows: BAT0 is to be No access:PT, and BAT1 is to be R/W:R/W. Fill in the required protection values in the table below.

Block	V_S	V_P	PP
0			
1			

---

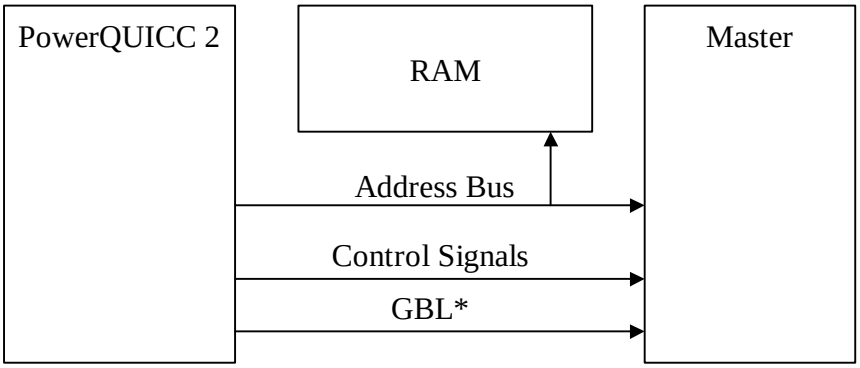
1. In the example, BAT0 is to be R/W:R/W. Looking at the table on the previous page, we see this requires the valid bits to each be one, and the PP bits to be 10.
2. BAT1 is to be PT:R/O. Again, according to the previous table, V_S must be zero, V_P must be one, and PP can be 01 or 11.
3. Finally, block 2 is to be R/O:PT requiring V_S to be 1, V_P to be zero, and PP again to be 01 or 11.
4. Try the exercise to check your understanding.

What Are the WIMG Bits?

Definition The WIMG bits are attributes assigned to blocks and pages.

Attribute	0	1
W	Write-back	Write-through
I	Caching enabled	Caching inhibited
M	Local access	Global access
G	Unguarded	Guarded

Snooping Block Diagram, M bit



If the asserted address is in a page or block with M=1, the GBL* signal is asserted. This notifies other masters to snoop their data cache(s). If the page or block has M=0, GBL* is not asserted.

1. Here's a summary of the WIMG bits. We're already familiar with W (write-through or write-back) and I (cache enabled or disabled) from the cache chapter.
2. The diagram explains the M bit. M stands for Memory Coherency bit. It is useful only in systems which can have more than one bus master. If a page with the M bit set is accessed, the PowerQUICC 2 asserts the GBL, global, pin. This notifies other bus masters that the PowerQUICC 2 is accessing data that they all share.
3. If the PowerQUICC is accessing data that the other bus master has cached, and if that data has been modified, the access of the PowerQUICC 2 must be held off until the other bus master can write the updated data to memory.

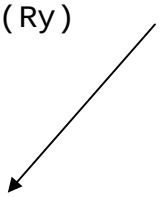
What is the Guarded Bit?

Definition

The Guarded attribute prevents out-of-order loading and pre-fetching from the addressed memory location.

Example

```
loop: lbz Rx,0(Ry)
      ------
      ------
      ------
      ------
      bc loop
```



1. “bc loop” enters sequencer
2. Branch unit predicts branch to loop
3. Sequencer pre-fetches lbz instruction
4. If 0(Ry) is not guarded, data is loaded. If it is guarded, data is not loaded until the branch is decided.

Add’l Comments

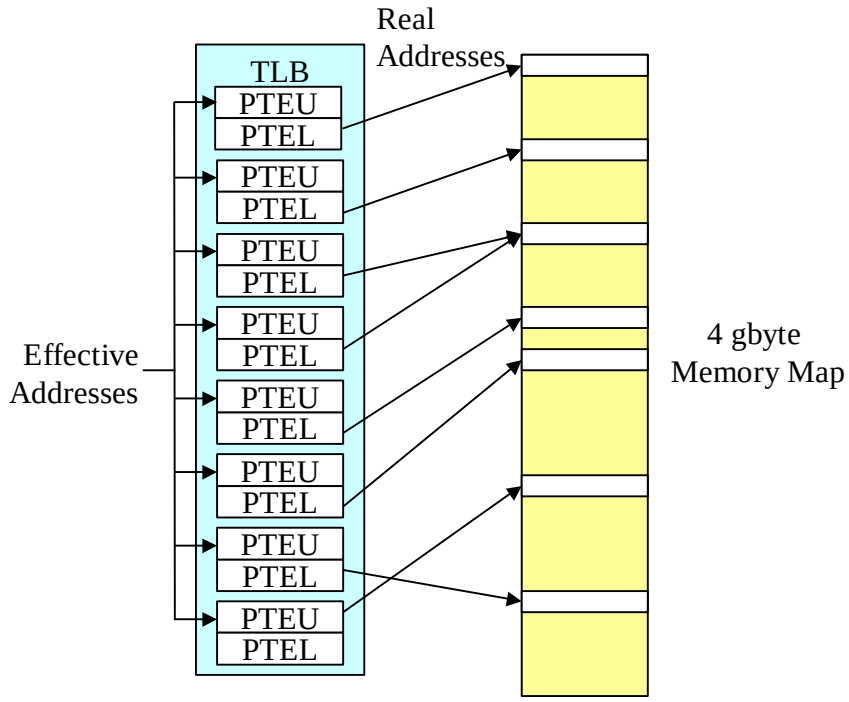
- A page should be guarded if it is subject to destructive reads.
- If the lbz instruction is in a guarded page, it is not fetched until the branch is decided.
- If the guarded instruction or data is in cache, the guarded bit has no effect.

What is Page Translation?

Definition

If an effective address matches the corresponding field of a TLB (Translation Lookaside Buffer) entry, the information in the TLB entry is used to generate the physical address.

Block Diagram



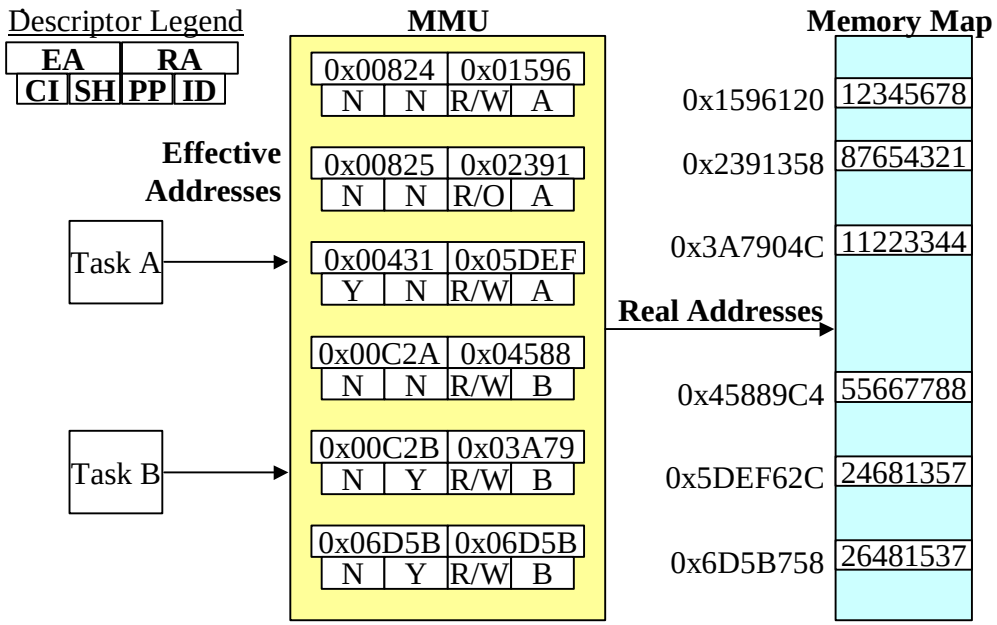
Characteristics of Page Translation

- Page translation allocates memory in 4K blocks.
- Data and instruction areas may overlap.
- When an effective address is asserted by the task, it is compared against the page entries in the TLB. If there is a match, the associated real address is asserted. If there is no match, then a page table search occurs.

Exercises - Page Translation

Introduction — The diagram below shows two tasks accessing the memory map through the MMU. A descriptor legend is shown along with six descriptors in the MMU.

Exercise



TLB Entries — TLB entries can be either:
1. Initialized directly using the instructions tlbli or tlbld or
2. Loaded with the result of a page table search.

What is the result when:

1. Task A asserts a read to 0x824120? _____.
2. Task A asserts a write to 0x825358? _____.
3. Task A asserts a read to 0x43162C? _____.
4. Task A asserts a read to 0xC2A9C4? _____.
5. Task A asserts a read to 0xC2B04C? _____.
6. Task B asserts a read to 0x6D5B758? _____.
7. Task A asserts a read to 0xC2C158? _____.

Load TLB Direct Programming Model (1 of 2)

PTEU - Page Table Entry Upper P. 7-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	VSID														

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
VSID									H	API					

PTEL - Page Table Entry Lower P. 7-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RPN															

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RPN				0	0	0	R	C	WIMG				0	PP	

DCMP - Data PTE Compare Register P. 5-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	VSID														

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
VSID									H	API					

Introduction

The next two pages are the programming model for loading the TLB directly.

PTE, Page Table Entry

- The PTE consists of two fields: the upper word, PTEU, and the lower word, PTEL.
- PTEU contains the virtual segment ID which consists of the task number and the segment number. It also contains the field API which is the most significant six bits of the page index field of the effective address. The H bit indicates whether this is a hash primary or hash secondary PTE.
- PTEL contains the real page number and the protection bits, WIMG and PP. In addition, the R bit is used to record that this page has been accessed and the C bit, that this page has been changed.

DCMP

This register contains the value to be compared with PTEU in searching the TLB or the page tables for a match. For a load TLB, this register specifies the PTEU to go into the entry. This is for data accesses only.

Load TLB Direct Programming Model (2 of 2)

ICMP - Instruction PTE Compare Register P. 5-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	VSID														

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
VSID									H	API					

RPA - Required Physical Address Register P. 5-38

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RPN															

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RPN				0	0	0	R	C	WIMG				0	PP	

ICMP This register contains the value to be compared with PTEU in searching the TLB or the page tables for a match. This is for instruction accesses only.

RPA On a TLB load, this register specifies the PTEL value to go into the entry. For a table search, RPA will be loaded with the real address before returning from the exception.

How to Load a Page into a TLB (1 of 3)

Assumption - Reset conditions exist.

Action These are the steps in initialization:

Step	Action	Example

1	Clear BAT registers	`li r22,0`
		`mtibatu 0,r22`
		`-`
		`mtibatu 3,r22`
		`mtdbatu 0,r22`
		`-`
		`mtdbatu 3,r22`

2	Invalidate the TLBs	`for(i=0;i<32;i++)`
		`  invalidate(i<<12);`
		`.`
		`.`
		`void invalidate(i)`
		`int i;`
		`{`
		`  asm(" tlbie r3");`
		`}`

(Continued on next page)

Introduction Typically a TLB entry is loaded as a result of a miss exception routine doing a table search, finding the PTE and loading it into the TLB. Sometimes, it may be advantageous to load an entry prior to execution, perhaps as part of initialization in a context switch. This procedure shows how to do such a load.

Step 1 Clearing the BAT registers following reset is SOP because any and all of the BATUs could have their valid buts set.

Step 2 Invalidating the TLBs following reset is SOP because any entry can come up falsely valid.

How to Load a Page into a TLB (2 of 3)

Action

Step	Action	Example

3	Init DCMP with PTEU
	V: valid bit
	VSID:virtual segment ID
	H: hash bit
	API:upper 6 bits, page index
	(5-37)
		`lis r22,0x8045`
		`ori r22,r22,0x0009`
		`mtspr 977,r22`

4	Init RPA with PTEL
	RPN:real page number
	R:reference bit
	C:change bit
	WIMG:attributes
	PP:page protection
	(5-38)
		`lis r22,0x9876`
		`ori r22,r22,0x0002`
		`mtspr 982,r22`

(Continued on next page)

1. Next, DCMP (or ICMP) must be initialized with the desired PTEU word.
2. Then RPA must be initialized with the required PTEL word.

How to Load a Page into a TLB (3 of 3)

Action

Step	Action	Example

5	Do a TLB load
	ea: effective address
	(2-47)
		`lis r22,0x1234`
		`ori r22,r22,0x5678`
		`tlbli r22`

6	Enable MMU	`mfmsr r22`
		`ori r22,r22,0x30`
		`mtmsr r22`
		`isync`

1. Step 5, the tlb load instruction must be executed. The operand is the desired effective address.
2. Finally, the MMU must be enabled.
3. Let’s take a look at an example.

Example - Loading a Page into a TLB (1 of 4)

```
/* THIS PROGRAM CHECKS THE USE OF THE  tlbld INSTRUC-  */
/* TION. THIS INSTRUCTION AND  tlbli ARE USEFUL IN     */
/* LOADING TLB ENTRIES DIRECTLY RATHER THAN GOING THRU*/
/* EXCEPTION PROCESSING. THE PROGRAM FIRST INITIALIZES*/
/* A TLB ENTRY AND SUCCESSFULLY WRITES TO THE LOCA-   */
/* TION. IT THEN INVALIDATES THE ENTRY, AND TRIES TO  */
/* WRITE TO THE LOCATION AGAIN, BUT NOW A DATA STORE  */
/* TRANSLATION EXCEPTION OCCURS.                      */
/* IMPORTANT PARAMETERS ARE:                          */
/* PTEU=0x80001000,PTEL=0x22082, EFFECTIVE ADDRESS IS */
/* 0x24xxx, REAL ADDRESS IS 0x22xxx.                  */

main()
{
   int *tptr;             /* TEST POINTER             */
   void invbat();         /* DECLARE INVBAT FUNCTION  */
   void invalidate();     /* DECLARE INVALIDATE FUNC  */
   int i;                 /* GENERAL VARIABLE         */
```

Example - Loading a Page into a TLB (2 of 4)

```
    invbat();              /* INVALIDATE BAT REGS      */
    for(i = 0; i < 32; i++) /* INVALIDATE THE TLBS     */
       invalidate(i<<12);
    initDCMP(0x80001000);  /* INIT DCMP WITH PTEU      */
    initRPA(0x22082);      /* INIT RPA WITH PTEL       */
    filldataentry(0x24000); /* INIT TLB ENTRY FOR EA   */
    tptr = (int *) 0x22100; /* CLEAR TEST LOCATION     */
    *tptr = 0;
    asm(" mfmsr r22");      /* ENABLE DATA MMU         */
    asm(" ori r22,r22,0x10");
    asm(" mtmsr r22");
    asm(" isync");
    asm(" li r22,0x20");   /*INIT TASK 2,SEG 0,NO PROT */
    asm(" mtsr sr0,r22");
    tptr = (int *) 0x24100;    /* ACCESS PAGE          */
    *tptr = 0x12345678;
    invalidate(0x24100);
    *tptr = 0x9ABCDEF0;     /* ACCESS PAGE             */
}
```

Example - Loading a Page into a TLB (3 of 4)

```
void invbat()
{
   asm(" li r22,0");       /* INVALIDATE BAT REGS      */
   asm(" mtibatu 0,r22");
   asm(" mtibatu 1,r22");
   asm(" mtibatu 2,r22");
   asm(" mtibatu 3,r22");
//   asm(" mtdbatu 0,r22");
   asm(" mtdbatu 1,r22");
   asm(" mtdbatu 2,r22");
   asm(" mtdbatu 3,r22");
}

void invalidate(i)
int i;
{
   asm(" tlbie r3");     /* INV TLB ENTRY FOR EA IN r3*/
}
```

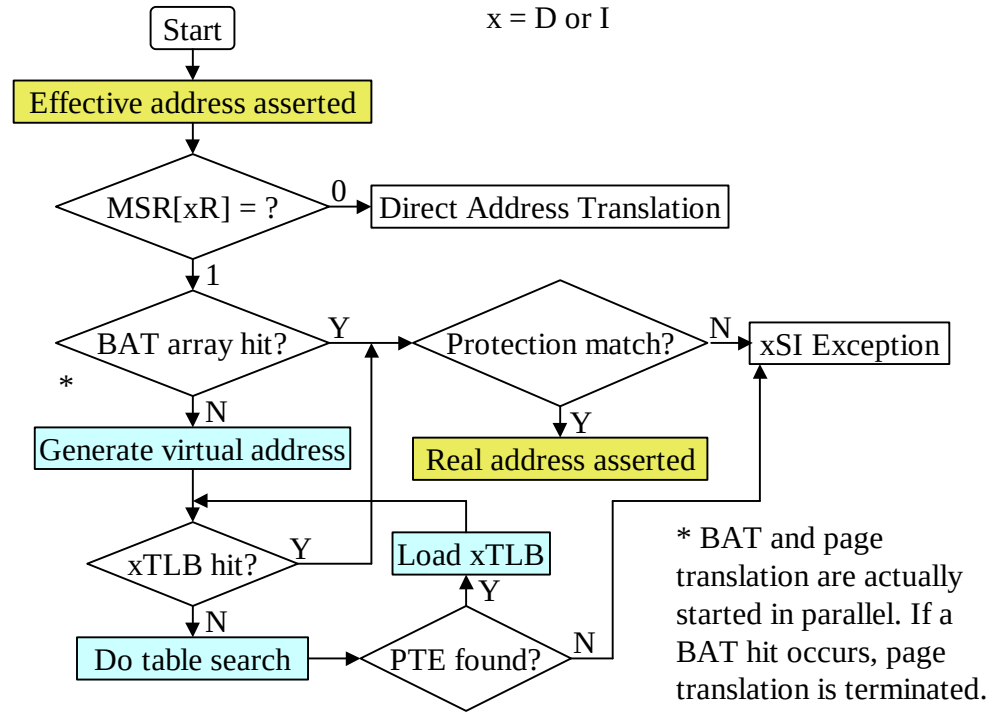
Example - Loading a Page into a TLB (4 of 4)

```
initDCMP(pteu)
int pteu;
{
   asm(" mtspr 977,r3"); /* 1ST WORD OF PTE TO DCMP  */
}

initRPA(ptel)
int ptel;
{
   asm(" mtspr 982,r3"); /* 2ND WORD OF PTE TO RPA   */
}

filldataentry(ea)
int ea;
{
   asm(" tlbld r3");     /* LOAD DATA TLB ENTRY      */
}
```

How an Effective Address is Translated

Introduction — The diagram below shows the flow in determining how an address will be translated.

Flow Diagram

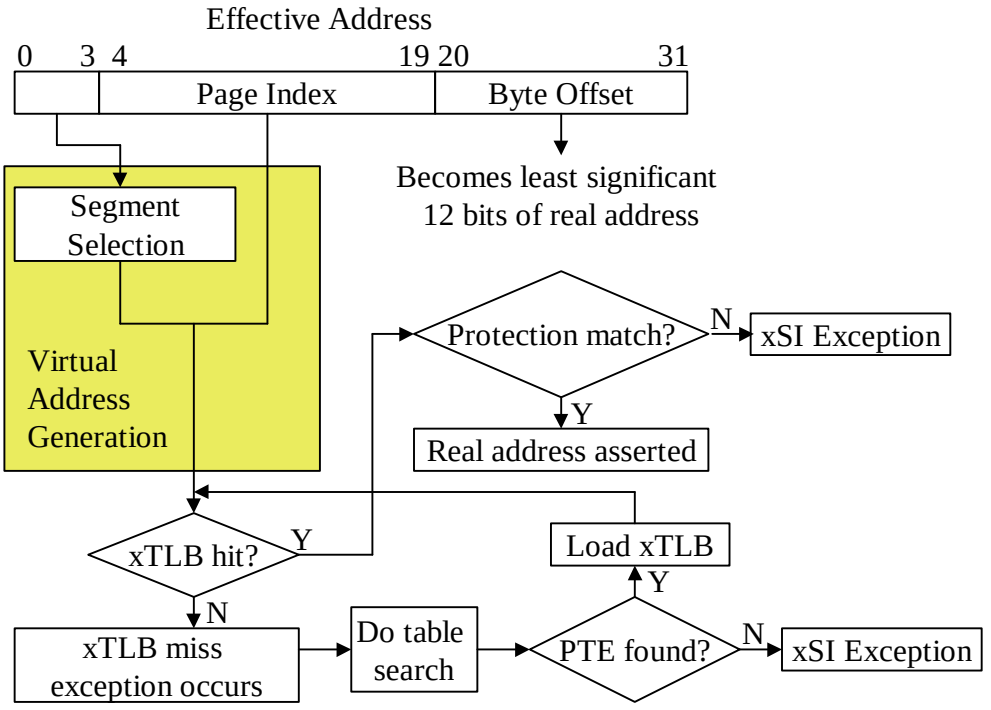


1. The translation process begins with the assertion of an effective address. If the associated MMU, instruction or data, is not enabled, then memory is accessed directly.
2. If it is enabled, then BAT and page translation are begun.
3. If a BAT hit occurs, page translation is terminated, and a check is made for protection compatibility. If there is no compatibility, an xSI exception is taken.
4. If a BAT miss occurs, translation continues with the generation of a virtual address. The MMU then checks for a TLB hit. If there is a hit, a check is made for protection compatibility. If there is no compatibility, an xSI exception is taken.
5. If there is no hit, a table search is executed. If the page table entry is found, it is loaded in the TLB and then a hit occurs. If no PTE is found, execution goes to the xSI vector via a branch.
6. This is the overall picture. Next, we want to learn what virtual address generation is.

How a Virtual Address is Generated

Introduction — The diagram below shows the flow in determining how a virtual address is generated.

Flow Diagram



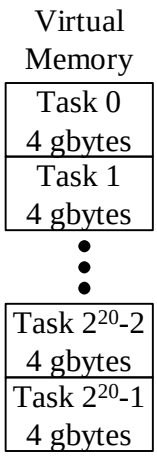
1. A virtual address is generated by combining the page index, bits 4-19 of the effective address, and the segment selection.
2. Next, what is segment selection?

What is Virtual Memory?

Definition

Virtual memory refers to the ability of the 603e MMU to allocate 4 gigabytes of memory for to up to a million tasks.

603ev Virtual Memory Map



Comments

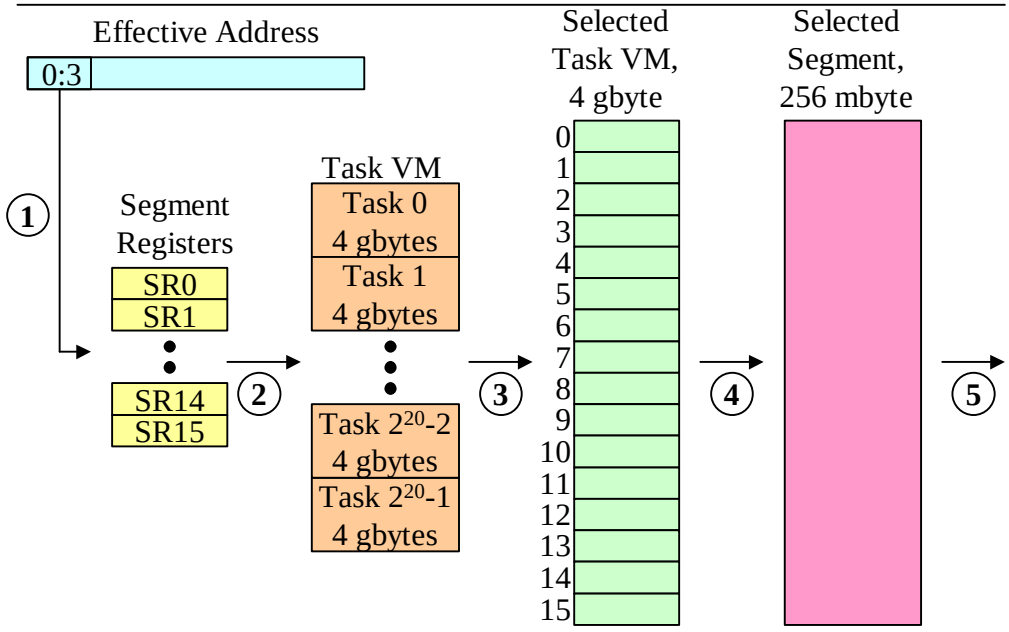
- At any point in time, only one task can be running; therefore, only one 4 gbyte memory space is in use.
- The pages of a particular task may reside in physical memory or on disk or both.
- When a page that is needed is on disk, the OS must move it into memory.

1. First of all, let's review the meaning of virtual memory on the 603ev.
2. Virtual memory consists of 1 million 4 gigabyte memory spaces each assigned to a specific task number.
3. Since there is only one 4 gigabyte memory available, it's apparent that if all tasks are in physical memory at once, then probably there are few tasks in the system.
4. Virtual memory operation allows that some pages can be in memory and some on disk.

How a Segment is Selected (1 of 2)

Introduction — The diagram below shows the flow in determining how a segment is selected.

Flow Diagram

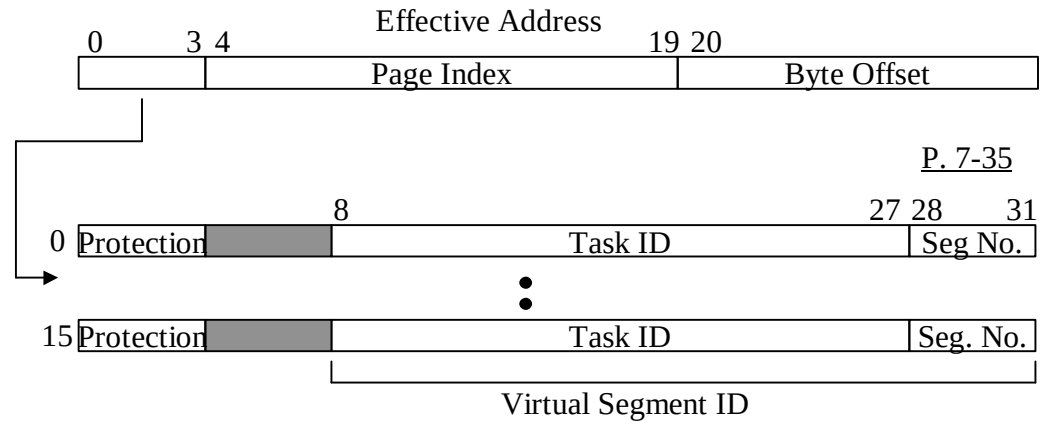


The virtual segment ID from the Segment Register determines the selected 4 gbyte space and further, determines the segment within that space.

1. Segment selection begins with bits 0-3 of the effective address being used to select one of 16 segment registers.
2. The segment register, among other things, contains the task ID which selects 1 of 1 million virtual memory spaces.
3. The virtual memory space is divided into 16 memory segments, 256Mbytes each.
4. The segment register also contains a segment number which selects one of the sixteen segments.
5. This segment selection then becomes part of the virtual address.

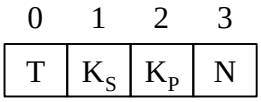
How a Segment is Selected (2 of 2)

Segment Translation



• Two sets of segment registers: one for data and a shadow set for instruction accesses

Protection



Prot. Bit	Description
T	Must always be 0
K_S	Supervisor state protection key
K_P	User state protection key
N	No-execute protection bit

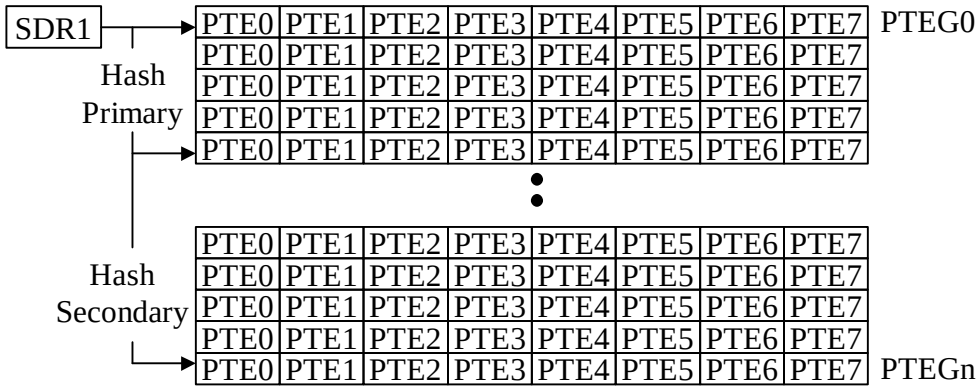
1. The segment register consists basically of three fields: protection, task ID, and segment number.
2. It is the job of the operating system to program the segment registers on a task switch. At this time it will put the task number in a least one segment register.
3. It puts the number of a segment to be used in the segment register. If the task is using only one segment register, the segment number could be any number 0-15. If the task is using all the segment registers, then the segment number field might contain the number of the segment register.
4. There are four protection bits: T, K_S, K_P, and N.
5. T must always be zero. If it is one, execution gets directed to a feature that is no longer implemented on PowerPC.
6. There are two protection keys, one for supervisor and one for user.
7. N can specify that a segment be used for data only. In this case the shadow instruction register is disabled.

What is the 603e Page Table Search?

Definition

The 603e page table search is the process of searching through a hashed table of page entries for a match to a requested effective address.

Page Table Structure



Calculations

```
HP = ((EA>>12) & 0xFFFF) ^ (getSRn() & 0x7FFFF);
HS = ~HP;
PTEGptr = (int *) 0;
PTEGptr = (int *)((getSDR1() & 0xFE000000) + (((HP>>10) & (getSDR1()
          & 0x1FF)) | (setSDR1()  & (0x1FF<<16)) + ((HP &
          0x3FF)<<6)));
```

1. Here we define a page table search.
2. The page table is in memory at a location specified by the register, SDR1 (by the way, there is no other register SDRn except SDR1).
3. A page table entry (PTE) is located in one of two page table groups (PTEG) which is selected by one of two hash functions, primary and secondary.
4. The calculations for the hash functions and the PTEGs are shown.

Page Programming Model (1 of 3)

SDR1 - Storage Description Register P. 7-50

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HTABORG															

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	HTABMASK								

PTEU - Page Table Entry Upper P. 7-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	VSID														

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
VSID									H	API					

PTEL - Page Table Entry Lower P. 7-37

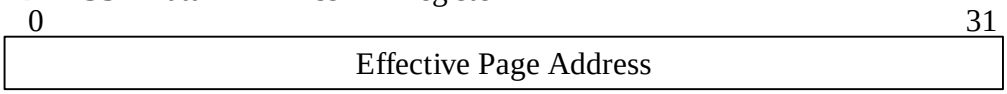
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RPN															

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RPN				0	0	0	R	C	WIMG				0	PP	

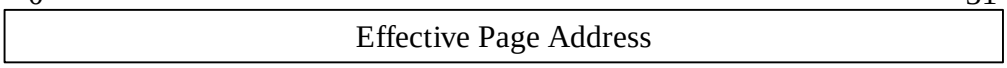
1. The next three pages are the programming model for page translation.
2. SDR1 contains an originating address (HTABORG) for the page table and a mask field (HTABMASK) to specify the length.
3. The PTE consists of two fields: the upper word, PTEU, and the lower word, PTEL.
4. PTEU contains the virtual system ID from the segment register. It also contains the field API which is the most significant six bits of the page index field of the effective address. The H bit indicates whether this is a hash primary or hash secondary PTE.
5. PTEL contains the real page number and the protection bits, WIMG and PP. In addition, the R bit is used to record that this page has been accessed and the C bit, that this page has been changed.

Page Programming Model (2 of 3)

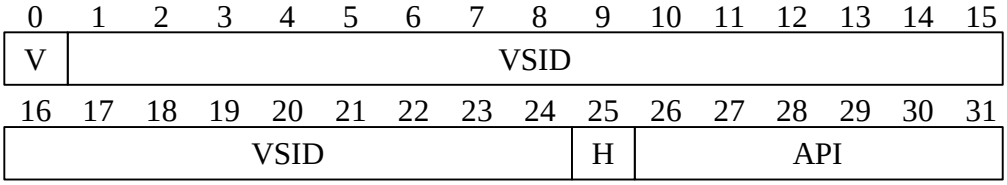
DMISS - Data TLB Miss EA Register P. 5-36



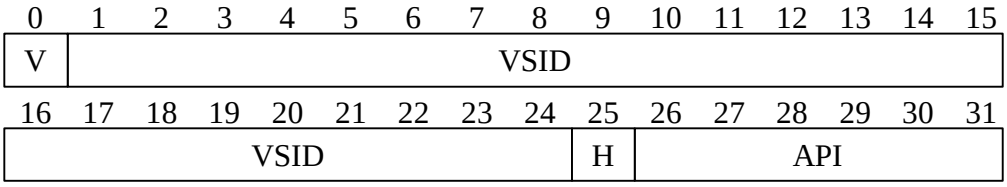
IMISS - Instruction TLB Miss EA Register P. 5-36



DCMP - Data PTE Compare Register P. 5-37



ICMP - Instruction PTE Compare Register P. 5-37



1. These registers contain information about the miss that can be used in the miss service routine.
2. The value in ICMP or DCMP is the word to be searched for in the page table.

Page Programming Model (3 of 3)

HASH1 - Primary PTEG Address P. 5-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HTABORG							Hashed Page Address								

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Hashed Page Address										0	0	0	0	0	0

HASH2 - Secondary PTEG Address P. 5-37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HTABORG							Hashed Page Address								

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Hashed Page Address										0	0	0	0	0	0

RPA - Required Physical Address Register P. 5-38

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RPN															

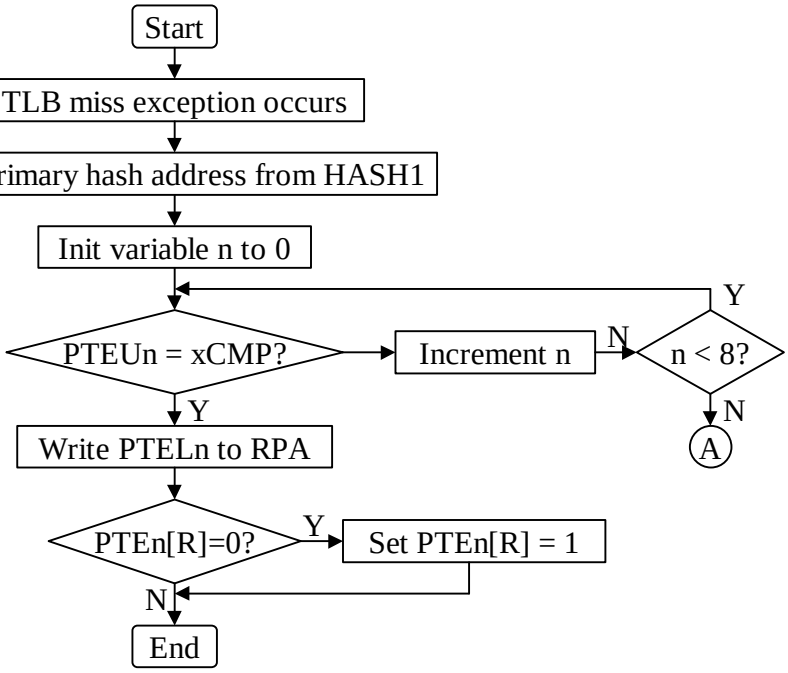
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RPN				0	0	0	R	C	WIMG				0	PP	

1. The hash registers are used by the miss routine as pointers to where the search should begin.
2. Once the PTE has been found, PTE1 of the PTE can be moved to RPA and the search is complete.

How the Page Table is Searched? (1 of 2)

Introduction The diagram below shows the flow in searching the page table.

Flow Diagram

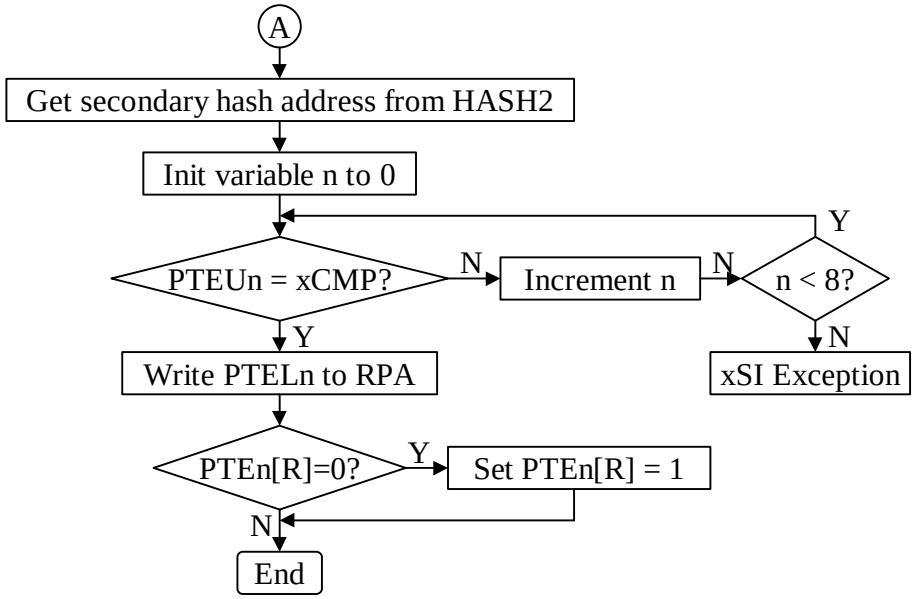


1. PTEUn is the upper word of page table entry n at the PTEG address.
2. PTELn is the lower word.

1. The table is searched as shown. After the miss exception occurs, the service routine uses the pointer in HASH1, which points to the right PTEG, to begin the search.
2. The service routine searches through PTEs until it either finds a match or it has searched all 8 entries in the PTEG.
3. If a match was found, then PTE1 is written to RPA and the R bit is updated.
4. If a match was not found, then a second search is started.

How the Page Table is Searched? (2 of 2)

Flow Diagram (cont'd)



1. PTEUn is the upper word of page table entry n at the PTEG address.
2. PTELn is the lower word.

1. The second search is done in the same way, except it uses HASH2.
2. If a match is not found in this search, it is an error, and execution goes to the xSI exception vector via a branch.

How to Assign Page Protection (1 of 2)

Notation: S:U = Supervisor:User
R/W:R/O = Supervisor access is R/W:User access is R/O

Reference Table

If a page is to have this protection...	...then the page protection must be...	...and PP in the PTE must be...
R/W:R/W or R/O:R/O	K_S = N/A and K_P = N/A	10 for R/W:R/W and 11 for R/O:R/O
R/W:No access or R/W:R/O	K_S = 0 and K_P = 1	00 for R/W:No access and 01 for R/W:R/O
No access:R/W or R/O:R/W	K_S = 1 and K_P = 0	00 for No access:R/W and 01 for R/O:R/W
No access:No access or R/O:R/O	K_S = 1 and K_P = 1	00 for No access:No access and 01 for R/O:R/O

1. Here we can learn to assign the protection to a page that we desire. Once again, check the notation we use. A protection pair consists of protection in the supervisor mode, followed by a colon, followed by protection in the user mode.
2. So in the left hand column, we find the protection that we would like to give a particular block, and then use the other two columns to tell us what values go into the valid bits and the PP field.
3. For example, for a page in which we would like supervisor no access and user read/write, we would assigned 1 to K_S, 0 to K_P, and 00 to PP.

How to Assign Page Protection (2 of 2)

Example

A segment is to have three valid pages as follows: page 1 is to be R/W:R/W, page 2 is to be R/W:R/O, and page 3 is to be No access:R/W. Fill in the required protection values in the table below.

K_S	K_P
0	1

Page	PP
1	10
2	01
3	*

* Page 3 must be located in another segment.

Exercise

A segment is to have two valid pages as follows: page 1 is to be No access:R/W, and page 2 is to be R/W:R/W. Fill in the required protection values in the table below.

K_S	K_P

Page	PP
1	
2	

---

1. In the example, page 1 is to be R/W:R/W. Looking at the table on the previous page, we see this requires only that the PP bits be 10.
2. Page 2 is to be R/W:R/O. Again, according to the previous table, K_S must be zero, K_P must be one, and PP must be 01.
3. Finally, page 3 is to be No Access:R/W. This requires K_S to be 1, K_P to be zero; therefore, this page must go in another segment.
4. Try the exercise to check your understanding.

What are the TLB Miss Exceptions? (1 of 2)

Definition

If a valid PTE is not found in the TLB, a TLB miss exception occurs. Software must then find and load the PTE into the TLB. Additional registers provide some hardware assist.

Types of Miss Interrupts

	Type	Vector
1	Instruction TLB Miss	0x1000
2	Data Load TLB Miss	0x1100
3	Data Store Translation Miss or C=0	0x1200

- Exception type 2 occurs if a data load address cannot be translated by the DTLB
- Exception type 3 occurs if:
  1. A data store address cannot be translated by the DTLB or
  2. The C bit must be changed.

Hardware Assist Registers

When execution begins at the service routine, the following registers contain useful information:

- D,IMISS = EA that caused the miss
- HASH1 = primary PTEG address; HASH2 = secondary PTEG address
- D,ICMP = word to be compared against first word of PTEs
- RPA = ISR loads with second word of matching PTE

What are the TLB Miss Exceptions? (2 of 2)

Temporary Scratch Registers

- When execution begins at the service routine, MSR[TGPR] = 1 and four, 32-bit, temporary registers, GPR0-3, are available.
- If an access of GPR4-31 is attempted while MSR[TGPR] = 1, results are undefined.

CR0

- When execution begins at the service routine, CR0 has been saved in SRR1[0:3].
- The ISR must restore CR0 from SRR1[0:3] before executing rfi.

```
        .
        .
mfspr r3,srr1
mtcrf 0x80,r3
rfi
```

Service Routines

The programs for the TLB miss exception service routines are in the 603UM, p. 5-44.

What are the MMU Error Exceptions?

Definition

An MMU error exception occurs if a page search results in: 1) no PTE was found or 2) the protection associated with the PTE is not compatible.

Data Storage Interrupt (DSI)

- Occurs for an error due to a data access
- Uses exception vector 0x300
- Error information can be found in:
  - SRR0 - effective address that caused the exception
  - SRR1 - copy of MSR
  - DSISR - protection violation and read/write status
  - DAR - effective address of protected memory byte

Instruction Storage Interrupt (ISI)

- Occurs for an error due to an instruction access
- Uses exception vector 0x400
- Error information can be found in:
  - SRR0 - effective address of the next instruction
  - SRR1 - copy of MSR; bit 4 indicates a protection violation

How to Initialize a Page Table Entry (1 of 2)

Assumption
- The segment registers and SDR1 have been initialized.
- The page table has been cleared.
- The TLBs have been invalidated.

Action

These are the steps in initialization:

Step	Action	Example
1	Calculate the primary hash function	`See earlier page`
2	Calculate the secondary hash function	`hs = ~hp;`
3	Calculate the pointer to the PTEG	`See earlier page`
4	Search the primary PTEG for an unused entry	`while(((PTEPptr->PTEU<0) && (i++ < 8)) PTEGptr++;`

(Continued on next page)

How to Initialize a Page Table Entry (2 of 2)

Action

Step	Action	Example
5	If no available entry in primary PTEG, then search secondary PTEG	
6	Initialize the entry with the new page	`PTEGptr->PTEU = pteu;` `PTEGptr->PTEL = ptel;`
7	Execute sync	`asm(“ sync”);`

Exercise - Initializing a Page Table Entry (1 of 9)

```
/* THIS PROGRAM GENERATES A PAGE TABLE ENTRY AND THEN */
/* ACCESSES THE PAGE TO CHECK THE RESULT. SINCE THE   */
/* SERVICE ROUTINE FOR A DATA MISS IS NOT INCLUDED,   */
/* WHEN A DATA MISS OCCURS, THE HASHX REGISTER IS     */
/* CHECKED TO VERIFY IT HAS THE SAME VALUE AS THE LO- */
/* CATION OF THE PAGE ENTRY.                          */

struct PTE {
   int PTEU;              /* FIRST WORD OF PTE        */
   int PTEL;              /* SECOND WORD OF PTE       */
   };
#define FALSE 0
#define TRUE  1
```

Exercise - Initializing a Page Table Entry (2 of 9)

```
main()
{
   int *pt;                /* PT POINTER FOR CLEAR     */
   void invbat();          /* DECLARE INVBAT FUNCTION  */
   void invalidate();      /* DECLARE INVALIDATE FUNC  */
   int i;                  /* GENERAL VARIABLE         */

   invbat();               /* INVALIDATE BAT REGS      */
   asm(" lis r22,0");       /* INIT SR0                */
   asm(" ori r22,r22,0x20"); /* TASK 2,SEG 0, NO PROT */
   asm(" mtsr SR0,r22");
   asm(" lis r22,0x3");    /* INIT SDR1                */
   asm(" mtsdr1 r22");     /* LOCATE PT AT 0x30000     */
   pt = (int *) 0x30000;   /* INIT PNTR TO PAGE TABLE */
   for(i = 0; i < _____; i++)  /* CLEAR PT             */
      *pt++ = 0;
   for(i = 0; i < 32; i++)
      invalidate(i<<12);
```

Exercise - Initializing a Page Table Entry (3 of 9)

```
    add_a_page(0x24000,0x80001000,0x00022082);
                              /*ADD A PAGE,EA=0x24000,   */
                              /*RA=0x22000,TASK 2,SEG 0  */
    pt = (int *) 0x22100;     /* CLEAR TEST LOCATION     */
    *pt = 0;
    asm(" mfmsr r22");        /* ENABLE DATA MMU         */
    asm(" ori r22,r22,____");
    asm(" mtmsr r22");
    asm(" isync");
    pt = (int *) 0x24100;     /* ACCESS PAGE             */
    *pt = 0x12345678;
    remove_a_page(0x24000,0x80001000,0x00022082);
                              /* REMOVE PAGE             */
    *pt = 0x9ABCDEF0;         /* ACCESS PAGE             */
}
```

Exercise - Initializing a Page Table Entry (4 of 9)

```
void invbat()
{
   asm(" li r22,0");      /* INVALIDATE BAT REGS      */
   asm(" mtibatu 0,r22");
   asm(" mtibatu 1,r22");
   asm(" mtibatu 2,r22");
   asm(" mtibatu 3,r22");
//   asm(" mtdbatu 0,r22");
   asm(" mtdbatu 1,r22");
   asm(" mtdbatu 2,r22");
   asm(" mtdbatu 3,r22");
}

void invalidate(i)
int i;
{
   asm(" _____ r3");
}
```

Exercise - Initializing a Page Table Entry (5 of 9)

```
add_a_page(ea,pteu,ptel)
int ea,pteu,ptel;
{
   int hp,hs;              /* PRIMARY AND SECONDARY HASH */
   struct PTE *PTEGptr;    /* POINTER TO PTEG GROUP      */
   int i;                  /* GENERAL VARIABLE           */

   hp = ((ea>>12) & 0xFFFF) ^ (getSR0() & 0x7FFFF);
                           /* DETERMIN PRIMARY HASH VALUE*/
   hs = ___;               /* DETRMN SECONDARY HASH VALUE*/
   PTEGptr = (struct PTE *) 0;   /* INIT POINTER TO ZERO*/
   PTEGptr = (struct PTE *)((getSDR1() & 0xFE000000) +
                    (((hp>>10) & (getSDR1() & 0x1FF)) |
                    (getSDR1() & (0x1FF<<16)) +
                    ((hp & 0x3FF)<<6)));
   i = 0;
   while((PTEGptr->PTEU < 0) && (i++ < 8))
     PTEGptr++;
```

Exercise - Initializing a Page Table Entry (6 of 9)

```
if(PTEGptr->PTEU < 0)
  {
     pteu |= 0x________;
     PTEGptr = (struct PTE *)((getSDR1() & 0xFE000000) +
                    (((hs>>10) & (getSDR1() & 0x1FF)) |
                    (getSDR1() & (0x1FF<<16)) +
                    ((hs & 0x3FF)<<6)));
     i = 0;
     while((PTEGptr->PTEU < 0) && (i++ < 8))
        PTEGptr++;
  }
  if(PTEGptr->PTEU >= 0)
  {
     PTEGptr->PTEU = pteu;
     PTEGptr->PTEL = ptel;
     asm(“ sync”);
     return(0);
  }
  else
     return(1);
}
```

Exercise - Initializing a Page Table Entry (7 of 9)

```
remove_a_page(ea,pteu,ptel)
int ea,pteu,ptel;
{
   int hp,hs;              /* PRIMARY AND SECONDARY HASH */
   struct PTE *PTEGptr;    /* POINTER TO PTEG GROUP      */
   int i;                  /* GENERAL VARIABLE           */
   char found;             /* BOOLEAN FLAG               */

   hp = ((ea>>12) & 0xFFFF) ^ (getSR0() & 0x7FFFF);
                           /* DETERMIN PRIMARY HASH VALUE*/
   hs = ~hp;               /* DETRMN SECONDARY HASH VALUE*/
   PTEGptr = (struct PTE *) 0;   /* INIT POINTER TO ZERO*/
   PTEGptr = (struct PTE *)((getSDR1() & 0xFE000000) +
                    (((hp>>10) & (getSDR1() & 0x1FF)) |
                    (getSDR1() & (0x1FF<<16)) +
                    ((hp & 0x3FF)<<6)));
   i = 0;
   found = FALSE;
```

Exercise - Initializing a Page Table Entry (8 of 9)

```
do
     if(PTEGptr->PTEU == pteu && PTEGptr->PTEL == ptel)
        found = TRUE;
  while (found == FALSE && i++ < 8);
  if(found == FALSE)
  {
     pteu |= 0x00000040;
     PTEGptr = (struct PTE *)((getSDR1() & 0xFE000000) +
                    (((hs>>10) & (getSDR1() & 0x1FF)) |
                    (getSDR1() & (0x1FF<<16)) +
                    ((hs & 0x3FF)<<6)));
     i = 0;
     do
        if(PTEGptr->PTEU == pteu && PTEGptr->PTEL == ptel)
           found = TRUE;
     while (found == FALSE && i++ < 8);
  }
```

Exercise - Initializing a Page Table Entry (9 of 9)

```
 if(found == TRUE)
   {
      PTEGptr->PTEU = 0;
      asm(“ sync”);
      return(0);
   }
   else
      return(1);
}

getSDR1()
{
   asm(" mfsdr1 r3");
}

getSR0()
{
   asm(" mfsr r3,0");
}
```

Example - Configuring the MMU for a System (1 of 4)

Example Physical Memory Map

This is an example of a mix of different devices and the address range from which they are accessed.

	Address Space	Device Type	Size
1	0x00000000 - 0x03FFFFFF	SDRAM	64M
2	0x04000000 - 0x041FFFFF	SRAM	2M
3	0x04200000 - 0x045FFFFF	SDRAM - local bus	4M
4	0x04700000 - 0x04700FFF	Board config regs	4K
5	0x04900000 - 0x04900FFF	ATM PHY	4K
6	0x05000000 - 0x0500FFFF	PQ2 internal space	64K
7	0xFE000000 - 0xFFFFFFFF	Flash ROM	16M

Function Memory Area Assignment

Function	AS
EVT in ROM	**7**
EVT in RAM	1
Internal memory map	6
ATM PHY device	5
Board control & status	4
FCC BDs & buffers	3
MCC BDs & buffers	3

Function	AS
All other comm dev buffers	1
System stack	2
MMU page table	2
System scratchpad	2
System program area	1
User program area	1
User data area	1

Example - Configuring the MMU for a System (2 of 4)

Attribute Assignment

		Instruction					Data				
Function	**AS**	**W**	**I**	**M**	**G**	**Pr**	**W**	**I**	**M**	**G**	**Pr**
EVT in ROM	**7**	0	0	0	0	R/O:NA	0	1	0	0	R/O:NA
EVT in RAM	1	0	0	0	0	R/O:R/O	0	1	0	0	R/W:R/W
Internal memory map	6	0	1	0	0	NA:NA	0	1	0	0	R/W:NA
ATM PHY device	5	0	1	0	0	NA:NA	0	1	0	1	R/W:NA
Board control & status	4	0	1	0	0	NA:NA	0	1	0	1	R/W:NA
FCC BDs & buffers	3	0	1	0	0	NA:NA	0	1	0	0	R/W:R/W
MCC BDs & buffers	3	0	1	0	0	NA:NA	0	1	0	0	R/W:R/W
All other comm dev buffers	1	0	1	0	0	NA:NA	0	0	0	0	R/W:R/W
System stack	2	0	1	0	0	NA:NA	0	1	0	0	R/W:NA
MMU page table	2	0	1	0	0	NA:NA	0	1	0	0	R/W:NA
System scratchpad	2	0	1	0	0	NA:NA	1	0	0	0	R/W:NA
System program area	1	0	0	0	0	R/O:NA	0	1	0	0	NA:NA
User program area	1	0	0	0	0	R/O:R/O	0	1	0	0	NA:NA
User data area	1	0	0	0	0	NA:NA	1	0	0	0	R/W:R/W

Example - Configuring the MMU for a System (3 of 4)

Sorting by Address Space

		Instruction					Data				
Function	**AS**	**W**	**I**	**M**	**G**	**Pr**	**W**	**I**	**M**	**G**	**Pr**
EVT in RAM	1	0	0	0	0	R/O:R/O	0	1	0	0	R/W:R/W
All other comm dev buffers	1	0	1	0	0	NA:NA	0	1	0	0	R/W:R/W
System program area	1	0	0	0	0	R/O:NA	0	1	0	0	NA:NA
User program area	1	0	0	0	0	R/O:R/O	0	1	0	0	NA:NA
User data area	1	0	0	0	0	NA:NA	1	0	0	0	R/W:R/W
System stack	2	0	1	0	0	NA:NA	0	1	0	0	R/W:NA
MMU page table	2	0	1	0	0	NA:NA	0	1	0	0	R/W:NA
System scratchpad	2	0	1	0	0	NA:NA	1	0	0	0	R/W:NA
FCC BDs & buffers	3	0	1	0	0	NA:NA	0	1	0	0	R/W:R/W
MCC BDs & buffers	3	0	1	0	0	NA:NA	0	1	0	0	R/W:R/W
Board control & status	4	0	1	0	0	NA:NA	0	1	0	1	R/W:NA
ATM PHY device	5	0	1	0	0	NA:NA	0	1	0	1	R/W:NA
Internal memory map	6	0	1	0	0	NA:NA	0	1	0	0	R/W:NA
EVT in ROM	**7**	0	0	0	0	R/O:NA	0	1	0	0	R/O:NA

- AS3 & 7 can be block address translation.
- The system program area and all other commdev buffers can also be blocks.

Example - Configuring the MMU for a System (4 of 4)

Sorting by Address Space

Function	AS	Address
EVT in RAM	1	0x00000000 - 0x003FFFFF
All other comm dev buffers		0x00400000 - 0x007FFFFF
System program area		0x00800000 - 0x00BFFFFF
User program area		0x00C00000 - 0x02FFFFFF
User data area		0x03000000 - 0x03FFFFFF
System stack	2	0x04000000 - 0x0407FFFF
MMU page table		0x04080000 - 0x040FFFFF
System scratchpad		0x04100000 - 0x041FFFFF
FCC BDs & buffers	3	0x04200000 - 0x044FFFFF
MCC BDs & buffers		0x04500000 - 0x045FFFFF
Board control & status	4	0x04700000 - 0x04700FFF
ATM PHY device	5	0x04900000 - 0x04900FFF
Internal memory map	6	0x05000000 - 0x0500FFFF
EVT in ROM	7	0xFE000000 - 0xFFFFFFFF

- AS3 & 7 can be block address translation.
- The system program area and all other commdev buffers can also be blocks.

How to Initialize the MMU (1 of 4)

Assumption - Reset conditions exist.

Action These are the steps in initialization:

Step	Action	Example

1	Clear BAT registers	`li r22,0`
		`mtibatu 0,r22`
		`-`
		`mtibatu 3,r22`
		`mtdbatu 0,r22`
		`-`
		`mtdbatu 3,r22`

2	Init lower BAT register
	BRPN:real page addr[0:14]
	WIMG:attribute bits
	PP:access protection (7-25)
		`lis r22,0xFFFC`
		`ori r22,r22,3`
		`mtibatl 0,r22`

(Continued on next page)

How to Initialize the MMU (2 of 4)

Action

Step	Action	Example

3	Init upper BAT register
	BEPI:effective addr[0:14]
	BL:block length
	V_S:privilege mode valid
	V_P:problem mode valid
	(7-25)
		`lis r22,0xFFFC`
		`ori r22,r22,6`
		`mtibatu 0,r22`

4	Repeat steps 2 and 3 for each required lower-upper pair BAT registers.	

5	Initialize SR0-15
	T: must be 0
	K_S:Supervisor key
	K_P:User state key
	N:no-execute protection
	Task ID:task number
	Seg No.:segment number
	(7-35)
		`lis r22,0x6000`
		`ori r22,r22,0x0010`
		`mtsr SR0,r22`
		
		Also: `mtsrin`

How to Initialize the MMU (3 of 4)

Action

Step	Action	Example

6	Init SDR1
	HTABORG: PT base addr
	HTABMASK:PT addr mask
	(7-50)
		`lis r22,0xE044`
		`ori r22,r22,1`
		`mtsdr1 r22`

7	Initialize the PT to all invalid	

8	Invalidate the TLBs	`for (i=0; i < 32; i++)`
		`  invalidate(i<<12);`
		`  .`
		`  .`
		`void invalidate(i)`
		`int i;`
		`{`
		`  asm(" tlbie r3");`
		`}`

How to Initialize the MMU (4 of 4)

Action

Step	Action	Example
8	Initialize PT entries (7-37)	

9	Enable the MMU	`asm(“ mfmsr r22”);`
		`asm(“ ori r22,r22,0x30”);`
		`asm(“ mtmsr r22”);`
		`asm(“ isync”);`

Allocating the Blocks (1 of 2)

Hex numbers

Binary numbers

		BEPI					BL		V_S	V_P	Register Hex Value
IBATU	0	008	000	0	000	0	01	11	1	0	0x0080007E
	1	---	---	0	000	-	--	--	0	0	-
	2	---	---	0	000	-	--	--	0	0	-
	3	FE0	000	0	000	0	0F	11	1	0	0xFE0000FE

		BRPN					W	I	M	G		PP	Register Hex Value
IBATL	0	008	000	0	00	0	0	0	0	0	0	01	0x00800001
	1	---	---	0	00	0	-	-	-	-	0	--	-
	2	---	---	0	00	0	-	-	-	-	0	--	-
	3	FE0	000	0	00	0	0	0	0	0	0	11	0xFE000003

Allocating the Blocks (2 of 2)

		BEPI					BL		V_S	V_P	Register Hex Value
DBATU	0	008	000	0	000	0	01	11	1	1	0x0080001F
	1	004	000	0	000	0	01	11	1	1	0x0040001E
	2	042	000	0	000	0	01	11	1	1	0x0420001F
	3	FE0	000	0	000	0	0F	11	1	0	0xFE0000FE

		BRPN				W	I	M	G		PP	Register Hex Value
DBATL	0	008	000	0	00	0	0	1	0	0	10	0x00800022
	1	004	000	0	00	0	0	1	0	0	10	0x00400022
	2	042	000	0	00	0	0	1	0	0	10	0x04200022
	3	FE0	000	0	00	0	0	1	0	0	11	0xFE000023

Initializing the Segment Registers

		T	K_S	K_P	N		Task ID	Seg. No.	Register Hex Value
Segment Register	0	0	0	1	1	0	00001	0	0x30000010
	1	0	1	0	0	0	00000	0	0
	2	0	1	0	0	0	00000	0	0
	3	0	1	0	0	0	00000	0	0
	4	0	1	0	0	0	00000	0	0
	5	0	1	0	0	0	00000	0	0
	6	0	1	0	0	0	00000	0	0
	7	0	1	0	0	0	00000	0	0
	8	0	1	0	0	0	00000	0	0
	9	0	1	0	0	0	00000	0	0
	10	0	1	0	0	0	00000	0	0
	11	0	1	0	0	0	00000	0	0
	12	0	1	0	0	0	00000	0	0
	13	0	1	0	0	0	00000	0	0
	14	0	1	0	0	0	00000	0	0
	15	0	1	0	0	0	00000	0	0

Initializing SDR1

1. Determine total memory size to be allocated in pages.

AS	Size	Allocation
1	64M	8M-BAT
2	2M	Page
3	4M	BAT
4	4K	Page
5	4K	Page
6	64K	Page
7	16M	BAT

Memory to be allocated in pages = 58M

2. See page 7-52 to determine required page table size.

Required page table size = 512K; therefore,

HTABORG = x xxxx x000 and HTABMASK = 0 0000 0111

	HTABORG						HTABMASK	Reg Value
SDR1	0408	0	0	0	0	0	07	0x04080007