Application Note

Advanced Data Memory Paging for the CARMEL DSP

June 2001

http://www.infineon.com/dsp
carmel.support@infineon.com

Introduction

The increasing demand for more data memory especially in the 3rd Generation of Wireless applications requires cost effective paging approaches to extend the address space on CARMEL. Three different approaches for advanced RAM paging will be introduced in this application note. The classic paging method is the simplest and fast to implement. The other two more sophisticated approaches are pointer oriented and focus primarily on the needs of firmware developers.

A low power and low size implementation of the Carmel DSP 20xx enhanced with a paging mechanism of choice is a highly competitive solution addressing the needs for both modem and wireless applications in integrated wireless baseband devices.

Three Paging Approaches

In this context the term paging is used to describe a method for extending a fixed address space by overlaying data memory areas. There are several ways to implement paging. More advanced methods reduce the impact on the assembly programmer but they require more hardware support to be added to the processor. We present the following three methods:

1. Classic Memory Block Paging
2. Pointer oriented Paging
3. Pointer Arithmetic supporting Paging

1. Classic Memory Block Paging

Most common mechanisms to extend the address space have special hardware, which we define as **Memory Paging Unit (MPU)**. This unit sits between the DSP and the memory unit. Whenever the DSP wants to access memory (whether it is to read or write data), it sends the desired memory address to the MPU, which translates it to another address before passing it to the memory unit. The address generated by the DSP, after any indexing or other addressing-mode arithmetic, is called a

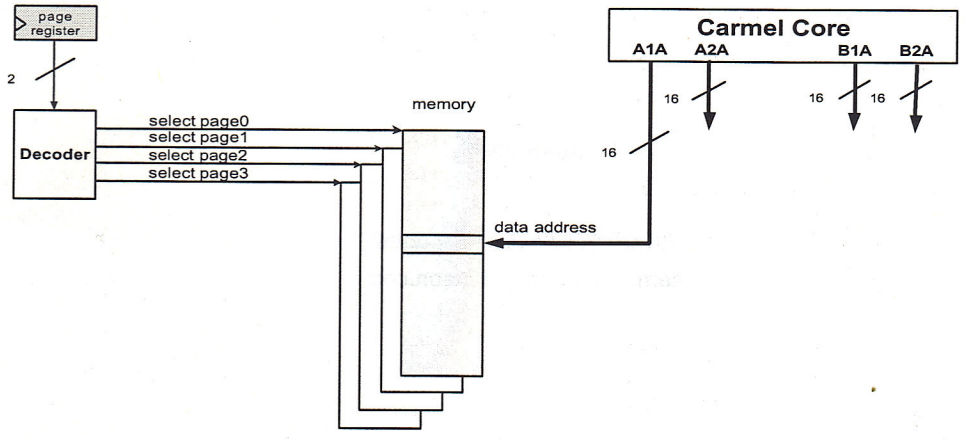

Figure 1: Page Selection

virtual address. The address translated by the MPU is called a physical address. Normally, the translation is done at the granularity of a *page*. In other words, each page is mapped to a contiguous region of physical memory called a page frame. Today's applications do not require a continuous Data

Page (memory address space) of more than 64kB. For example the CARMEL 20xx DSP Generation platform offers 128kB continuous data memory address space and by a simple MPU add-on the total data address space can be extended in steps of 8kB.

The address size of the CARMEL™ 20xx DSP core is 16 bit for accessing 64k words data memory. The use of page registers extends the address space beyond 64K. A page register functions like an extension to normal address size, thus the resulting address size is the size of the page register plus the size of the regular address. To address a total space of 256K word data memory, the extended address size has to be 18 bits and thus the page registers will contain 2-bits of additional address information.

Assuming the application code swaps page frames of size 4K words, then complete 64K address space is divided into 16 pages. For each page there exists one page register with 2 bits. In general the number of page registers depends on the granularity of the paged memory blocks. The granularity is chosen in a fixed way according to the size of physical memory blocks.

The approach presented in this chapter is to put the page table entries in special registers in the external address space. Each external register will have 2 page bits to indicate for each memory block which page is currently in use. In order to switch one of the 16 pages in memory segment number X the user just sets the relevant external register called "pageX".

```
(ext)page4=2;//set 4th page content to <BIT1page4, Bit0page4> = <1, 0>
```

Consequently any 16 bit address register which is used to access data within the 16-bit address range 0x4000 ... 0x4fff will be mapped into the extended address range 0x24000 ... 0x24fff. The hardware detects which 4K block is used and decides to extend the address with the corresponding page register (Figure 1).

The classic memory block paging is used efficiently for multiple large ROM tables. For example there could be 4 different ROM tables and only one of those tables is mapped into the data address space at runtime. In this case the page register content is the number of the currently active ROM table.

Another application could be the reduction of CARMEL data space needed for accessing large data buffers dedicated to acceleration hardware. Let's assume we have an acceleration hardware that uses a buffer of size 16K words. For the sake of a simple example let's assume the task for the processor is to sum up all the content of this buffer. Without paging we would use the following CARMEL code:

```
a0l=16*1024;
     r0=0x0000;  //implementation without paging
     clr(a1) || rep(a0l)single{
          a1+=*r0++;}
```

The same task can be performed using only one data page of 4K words with 4 pages. Then we have to map the 16K words buffer into the extended data address space. One possible buffer mapping is for example:

1st 4K= <0x0000,0x0000>...<0x0000,0x0FFF>
2nd 4K= <0x0001,0x0000>...<0x0001,0x0FFF>
3rd 4K= <0x0002,0x0000>...<0x0002,0x0FFF>
4th 4K= <0x0003,0x0000>...<0x0003,0x0FFF>

The following CARMEL code executes the same task with just three cycle's overhead every 4096 cycles.

```
a0l=4096;
r0=0x0000;              //implementation with simple 4K block paging
a0h=0;            // set page to 0 first
clr(a1) ||rep(4)block{
```

```
(ext)page0=a0h;     //update page register
inc(a0h);           //increment to next page
rep(a0l)single{
 a1+=*r0++;}
}
```

The advantage of this method is a small amount of hardware between the CARMEL and the memory. The limitations of this method are the fixed size (4K) and the alignment of memory blocks.

An alternative example for the classical approach is given below in Figure 2 showing a 4 banks x 128 kBytes constellation.

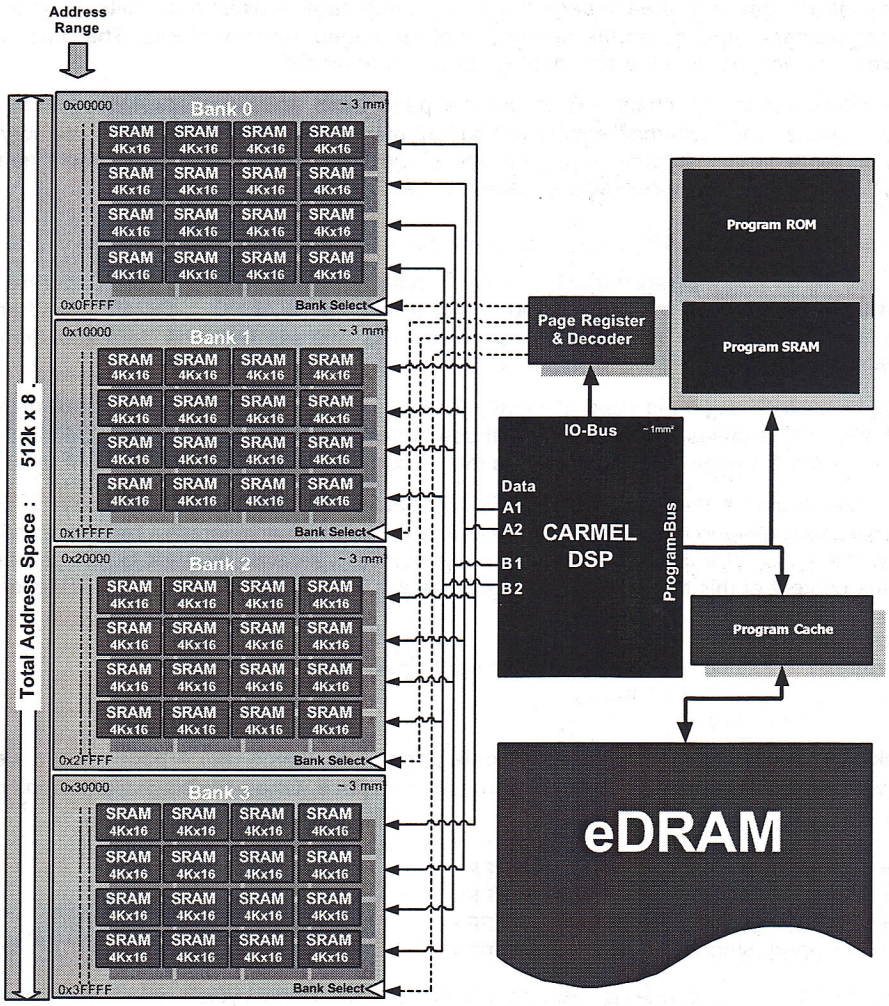

Figure 2: Classic Paging example for 4 banks x 128 kBytes with optional cache and eDRAM

In the first example the data allocation must fit into 4K granularity. We address this limitation of memory block paging in a more flexible way using one of the subsequent two advanced paging mechanisms.

2. Pointer-oriented Paging

First we summarize the different ways in which CARMEL generates addresses for data access. Either the stack pointer register addresses memory or one of the eight address registers R0...R7 is used to compute an address. Finally the third possibility is a direct 16 bit memory address encoded as immediate address in the instruction. Those are 10 different base address methods, which we can use to tailor the paging mechanism to the needs of a programmer.

The *pointer-oriented paging* introduces one page register for each of these 10 base address registers.

The content of those page registers is now a base pointer specific address extension. If pointer Rx generates a memory access then the page register page_Rx is used for extending the address space.

The advantage of this method is that at any given time in the program execution there are now up to 10 different address spaces available simultaneously. For example let page_R3 = 0x0010, then the register R3 points to an arbitrary extended address within <0x0003, 0x0000>...<0x0003,0xffff>. In order to implement the Pointer oriented approach we need to have 10 page registers mapped to the CARMEL external memory space:

- 8 page registers for Rx (X: 0..7)
- 1 page register for SP
- 1 page register for direct memory addressing

When the address is driven on the address bus during a memory access - the page select bits are decoded to select the appropriate page register. That page register is concatenated to the original 16 address bits of the address bus.

The only difference between pointer oriented paging and full 32 bit addressing are the pointer computations. For example consider page_R3 is loaded with 0x0010 and R3 is set to 0xFE00. Then let's see what the actual 32 bit address looks like after a couple of address computations like "*(r3+=0x100)=a0l;".

```
(ext)page_R3=0x0010;
nop;                    //set page register two cycles before using *r3
r3=0xfe00;              //current 32-bit address is: 0x0010 fe00
*(r3+=0x100)=a0l; //current 32-bit address is: 0x0010 ff00
*(r3+=0x100)=a0l; //current 32-bit address is: 0x0010 0000
```

In this case any pointer computation crossing the 64K boundary will not change the page register. Thus the programmer must avoid allocating arrays across 64K boundaries.

The pointer-oriented paging allows an easy implementation of the initial example (the sum of 16K external buffer values):

```
a0l=16*1024;
(ext)page_R0=0x02;      //R0 addresses page starting at 0x020000
r0=0x0000;              //implementation with pointer oriented paging
clr(a1) || rep(a0l)single{
      a1+=*r0++;}
```

Pointer Arithmetic supporting Paging

The only remaining limitation is that any address computation must not cross the 64K boundary. This difference to natural 32 bit addressing is eliminated in the third paging method.

3. Pointer Arithmetic supporting Paging

This is the most advanced paging method, which extends the concept of pointer oriented paging. There are still the same 10 pointer specific page registers called page_Rx. Additionally a sophisticated hardware extension keeps the complete 32-bit address up to date during pointer calculations. Coming back to the address computation example, this means:

```
(ext)page_R3=0x0010;
nop;                        //set page register two cycles before using *r3
r3=0xfe00;                  //current 32-bit address is: 0x0010 fe00
*(r3+=0x100)=a0l; //current 32-bit address is: 0x0010 ff00
*(r3+=0x100)=a0l; //current 32-bit address is: 0x0011 0000
```

During each pointer calculation the value of the address extension in the page register increments, decrements or remains unchanged. The paging hardware decides on the fly for each memory access, if an update of page registers is required.

A special case is address computations without memory access. The hardware needs to take special precautions for those instructions:

- Change pointer register
- Change register group
- Moving between registers
- Moving to registers
- Poping values from stack to registers (relevant page is not automatically poped)
- Pushing registers into stack (relevant page is not automatically pushed)

The initial example may now be written in a very convenient way. It also allows the 16K word buffer to cross the 64K boundary.

```
a0l=16*1024;
(ext)page_R0=0x02;      //R0 addresses page starting at 0x02F000
r0=0xF000;              //buffers is allowed to cross 64K boundary
clr(a1) || rep(a0l)single{
      a1+=*r0++;} //page_R0 will be updated automatically from 2 to 3
```

Another special case is the addressing of long memory operands with one base address register. For long memory access it could happen, that page boundary separates the upper and the lower half word. Usually the programmer can align memory allocations to even address in order to avoid this case. For example:

```
(ext)page_R0=0x02;
nop;                        //set page register two cycles before using *r0
r0=0xFFFD;                  //odd start address
a2=*(long*)r0++;            //current   32-bit address is: 0x0002 fffd
a3=*(long*)r0++;            //current   32-bit address is: 0x0002 ffff
a4=*(long*)r0++;            //ATTENTION 32-bit address is: 0x0003 0001
```

```
//in one cycle a0l=*0x0002ffff and a0h=*0x00030000 with just one pointer
//requires extra hardware support
```

Together with enhanced support for paging in the CARMEL tool set, this paging method is almost as flexible as a complete 32-bit processor. One difference to a processor with complete 32 bit pointer arithmetic is the maximum size of address offset calculations, because the 16 bit RNx registers allow only the usual offset values between –32768 and + 32767.

4. Hardware with pointer arithmetic support

The hardware needs to detect which pointer was used for a memory access. Therefor we introduce the page select bus consisting of 4 bits called SEL[4] representing the 10 different addressing possibilities Rx, SP, Direct.

The address unit generates the information, if an underflow or overflow occurred during an address calculation. This information has to be provided to the MPU in 2 bits called MOD[2]. If the computed address result crosses the upper page boundary MOD=1, then the page_Rx register is incremented (resp. decremented MOD=2 for lower boundary). Otherwise the page_Rx is not modified MOD=0.

The 6 bit information (MOD and SEL) is stored in a delay line with a depth of two pipeline stages. In the CARMEL pipeline the memory write access is executed two pipeline stages later than the read memory access. In case of a memory write access the page decision is evaluated based on the delayed version of the MOD and SEL signals. Each data memory address bus (A1A, A2A, B1A, B2A) should have its own SEL[4] and MOD[2] outputs from the addressing unit (see Figure 3).

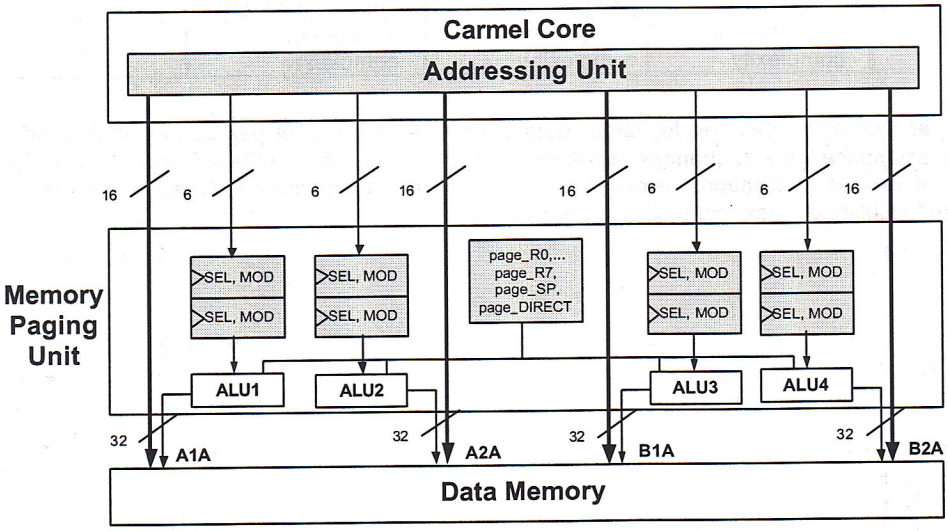

Figure 3: Memory Paging Unit

A special case for the hardware is the addressing mode indexed addressing. In this mode the offset is added before accessing the memory and the pointers are not updated. Consequently for the

expanded memory address needs to be the content of (page_Rx+/-1) and the register page_Rx is not updated in this mode.

In this case the extended address (page_Rx+/-1) is calculated and used in the same address cycle. This might be critical in time, but if large amounts of memories are on-chip, then the memory read data cycle will be the critical path, and the logic described before will not limit the chip frequency further. Otherwise a wait cycle is inserted in case of indexed addressing across page boundaries. But this overhead occurs only in a very rare scenario.

5. Summary and trade-off

The choice of the paging mechanisms is a trade-off between the hardware complexity and the programming flexibility. The following table summarizes the results:

Trade-off	1.Classic Memory Block Paging	2.Pointer oriented Paging	3.Pointer Arithmetic supporting Paging	Full 32-bit processor
Simultaneous data pages	e.g. 16 x 4K words	10 x 64K words	10 x 4G words	10 x 4G words
Buffer allocation	aligned to 4K	aligned to 64K	non-aligned 4G	non-aligned 4G
Pointer arithmetic	inside 4K blocks	inside 64K blocks	inside 4G blocks	inside 4G blocks
one address offset	–4095....+ 4095	–32768....+ 32767	–32768....+ 32767	–2G....+2G
Hardware implementation complexity	only 16 simple page registers => standard complexity	detect which pointer was used => medium complexity	detect & compute pointer arithmetic => higher complexity	Carmel 3xxx

In times of ever increasing demand for larger data memories, a powerful paging mechanism offers extended address space without changes inside the CARMEL core. Three different solutions fulfill the requirements of various applications, allowing a cost-effective implementation. Offering the user the same software flexibility as provided by 32-bit processors.