The following document contains information on Cypress products. The document has the series name, product name, and ordering part numbering with the prefix “MB”. However, Cypress will offer these products to new and existing customers with the series name, product name, and ordering part number with the prefix “CY”.

How to Check the Ordering Part Number

1. Go to www.cypress.com/pcn.
2. Enter the keyword (for example, ordering part number) in the **SEARCH PCNS** field and click **Apply**.
3. Click the corresponding title from the search results.
4. Download the Affected Parts List file, which has details of all changes

For More Information

Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most innovative automotive, industrial, smart home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, go to www.cypress.com.

AN204829

F²MC-16FX Family, MB96300 Performance Considerations

This application note explains resources and guidelines that influence application execution performance directly.

1 Introduction

The 16FX family of MCUs is advanced 16-bit microcontroller architecture. This application note explains resources and guidelines that influence application execution performance directly.

Following resources are considered:

- CPU
  - Clock frequency
  - Instruction queue
- Flash memory
  - Data width
  - Code and data buffer
  - Wait states
  - Core voltage
- Bus system and according code and data alignment

2 CPU performance

This chapter explains influence of CPU clock frequency and instruction queue on performance.

2.1 CPU clock frequency

The CPU executes the application code. Most applications are written in C language. The compiler translates the code to assembly language instructions. The CPU executes only such assembly instructions. All available assembly instructions are shown in appendix of Hardware Manual as well as in Programming Manual. Each instruction takes a certain number of CPU clock cycles to be executed. The fastest instructions take only one CPU clock cycle, such as NOP. The slowest instructions (with fixed execution time) take 23 CPU clock cycles. The RETI (return from interrupt) instruction is an example of this.

There is also a set of instructions that operate on strings. The execution time of these instructions depends on the length of the string.

The CPU clock frequency has direct impact on the execution time of the instructions. This is because the clock cycle length is the inverse of the clock frequency. Hence, the faster the clock frequency is, the faster the program is executed. The relation is linearly.

2.2 Instruction queue

The 16FX family MCUs features an 8 byte instruction queue. Whenever an instruction is executed, the presumably next instructions are loaded into the instruction queue when the bus system is not occupied by current instruction. As an example, consider the MULUW instruction (Multiply Unsigned Word Data of Accumulator). This instruction takes 4 CPU clock cycles and is executed completely within the ALU. It does not need to access the bus system. In the meantime, next instructions can be loaded into the instruction queue. The CPU assumes linear code to determine next instructions.

The number of instructions that fit into instruction queue depends on instruction length. The shortest instructions take one byte length, e.g. NOP. Longer instructions take up to five byte such as certain SUBL instruction. Usually there will be at least two following instructions in the execution queue.

If the CPU has to execute a branch instruction, the instruction queue must be flushed and loaded again with next instructions from branch target address. In this sense, a branch instruction is every instruction that disrupts linear code execution. It includes taken conditional branches, jumps, function calls, interrupt calls, function and interrupt returns, etc.

The importance of the instruction queue becomes clear when considering the complete code fetch mechanism. The bus width of 16FX MCUs is 16-bit wide. This means that only instructions of one or two bytes length can be loaded by one bus access. If an instruction is longer and it has to be fetched from memory, there is a delay of at least one clock cycle due to the additional bus access. Depending on flash memory wait state settings, there may be additional delays. For more information on flash memory timing, please refer chapter 3.1 “Wait states and internal supply voltage”.

Rule 1: The instruction queue increases performance by minimizing the number of clock cycles to fetch the next instruction.

The instruction queue loads next instructions by increasing program address. The most benefit of instruction queue and best performance is achieved when using programming techniques to generate linear code. One such method is usage of on inline functions. Another method is to use a compiler optimization that makes use of loop unrolling.

3 Flash memory performance

The flash memory A and B of MB96300 series have certain features and configuration options that need to be considered when determining optimum performance settings. These are notably the data width, the code and data buffers, the wait states, and the core voltage.

3.1 Wait states and internal supply voltage

The CPU of 16FX family MCUs can operate with up to 56 MHz clock frequency, depending on the individual device. The CPU can access RAM and registers without wait states up to the maximum specified CPU clock frequency. However, the flash memory A and B need wait states. The corner frequency and number of wait states depend on CPU clock frequency and core voltage. The corner frequency also depends on the core voltage. Using the default core voltage of 1.8V, the maximum CPU frequency with 0 wait states is 25 MHz. Increasing the core voltage to 1.9V increases the maximum CPU frequency with 0 wait states to 28 MHz.

The upper limit frequencies to use with one wait state are 46 MHz and 50 MHz when using 1.8V and 1.9V core voltage respectively. The maximum CPU frequency of 56 MHz requires 2 or 3 wait states, depending on core voltage being 1.9V or 1.8V.

Please note that the number of wait states mentioned above assume that flash memory is read. This includes data access as well as code execution. If data is to be written into flash memory, e.g. application update or EEPROM emulation, a higher number of wait states applies. For details, please refer MB96300 Hardware Manual chapter “Flash Memory”.

A wait state means that an access takes longer than one CPU clock cycle. The access is extended by the number of wait states. If using 1 wait state, every read access takes 2 CPU clock cycles, with 2 wait states, every access takes 3 CPU clock cycles, and so on. During the wait time, the CPU is stalled.

The time the CPU is stalled due to code fetch accesses to flash memory is minimized because the instruction queue automatically fetches next instructions before the CPU will execute them. Hence, linear code shows least performance impact of flash memory wait states.

The more jumps, branches, etc. are used in code, the slower the execution is going to be. When using CPU clock frequencies that need flash memory wait states, this becomes evident because the instruction queue needs to be flushed and cannot perform a pre-fetch.

As an example, consider the following, simple count loop:

```
#pragma segment FAR_CODE=test_prog,locate=0xFE0003
void perf_test()
{
  unsigned int tw;
  unsigned int i = 0;

  __wait_nop();          // NOP only to place breakpoint for time
                         // measurement

  for (i=0; i<20; i++)
  {
    tw = i;
  }

  result = tw;           // second breakpoint on this line because we
                         // like to measure loop only

  __wait_nop();
}
```

This loop is compiled with optimization level 2 and loop unrolling disabled. The resulting assembly code looks like this:

```
    64:   __wait_nop();
FE0003: 00              NOP
    67:   for (i=0; i<20; i++)
FE0004: D0              MOVN    A,#0
FE0005: 9C              MOVW    RW4,A
FE0006: 6004            BRA     FE000C
    68:   {
    69:     tw = i;
FE0008: 7BA4            MOVW    RW5,RW4
    70:   }
FE000A: 7344            INCW    RW4
FE000C: 8C              MOVW    A,RW4
FE000D: 3B1400          CMPW    A,#0014
FE0010: F2F6            BC      FE0008
    72:   result = tw;
FE0012: 8D              MOVW    A,RW5
FE0013: 5B4022          MOVW    2240,A
    74:   __wait_nop();
FE0016: 00              NOP
    76: }
FE0017: 66              RETP
```

There were breakpoints placed on addresses 0xFE0003 and 0xFE0012. The clock cycles it takes to execute the loop depend on the selected wait states:

Wait states	Clock cycles
0	97
1	117
2	149

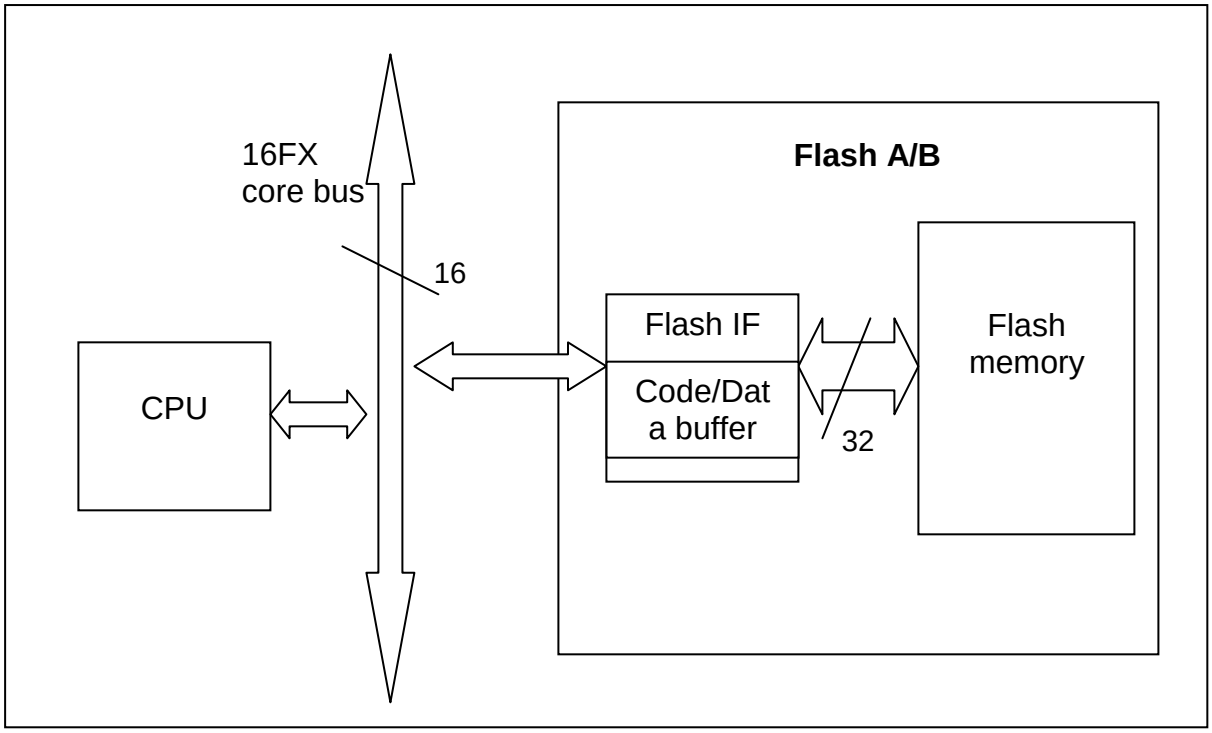
Here, it is evident that due to the very short loop of only five instructions the flush and reload of the instruction queue is affected by wait states.

3.2 Bus width and buffers

16FX family MCU features a 16-bit architecture. The flash memory A and B are designed with a 32-bit wide interface. The flash interface adapts this memory width to the 16-bit bus width of 16FX family MCUs. This means that whenever the CPU reads 16-bit of code or data from flash memory A or B, the flash memory performs internally an aligned 32-bit read. 16-bit of such access is delivered to CPU. The other 16 bit are stored in a buffer in the flash memory interface. Most code is executed in linear way. This results in next (part of) instruction being already stored in flash memory interface. There is no need to fetch it directly from flash memory. This avoids wait states because only accesses directly to flash memory are subject to wait states.

There are separate buffers for code and data. Both buffers can be disabled individually. They are enabled by default after a reset.

Figure 1. 16FX bus system CPU – Flash memory



The example given in 3.1 “Wait states and internal supply voltage” uses enabled code and data buffers. Running the same example above again with disabled code buffer yields following results:

Wait states	Clock cycles (code buffer enabled)	Clock cycles (code buffer disabled)
0	97	97
1	117	149
2	149	212

As expected, the code buffer improves performance when using wait states. In this example, the advantage is about 21% when 1 wait state is used and nearly 30% when 2 wait states are used.

Disabling the data buffer has no effect in this example. This is because the example does not access data, e.g. constants in flash memory.

4 Programming hints

Most applications are written in C language. The tool chain consisting of compiler, assembler, and linker will take care that a working application is generated. However, some points should be considered when selecting options of the tools.

4.1 Memory model

The 16FX MCUs offer a 24-bit wide address range. The CPU and instruction set is structured such that the Program Counter (PC) contains only the lower 16-bit of the next instruction address. The upper most 8-bit are given in the Program Counter Bank register (PCB). The PCB is updated automatically by CPU if the corresponding instructions are used. The compiler can take care of this automatically.

Similarly, when an instruction operates on a certain data address, the instruction contains the lower 16-bit of the data address. The upper most 8-bit are taken from Data Bank Register (DTB) or Additional Data Bank Register (ADB), depending on the instruction. If data is to be accessed in a different bank, the bank register needs to be updated by software. The compiler can take care of this automatically. Data in this sense are peripheral registers, variables in RAM as well as constants.

The compiler can generate code required to update the bank registers. Whether the compiler generates this code depends on the selected memory model. There are 4 memory models available: Small, Medium, Compact, and Large. They differ in which bank registers are updated by compiler automatically.

Memory Model	Data bank registers	PCB updated
Small	Maintained	Maintained
Medium	Maintained	Updated
Compact	Updated	Maintained
Large	Updated	Updated

Whenever a bank register is updated, the code size increases and performance decreases due to additional instructions. Therefore, one should select the memory that fits the application best. Memory model Large works with least consideration of side effects but leads to largest and slowest code.

In most applications, memory model Medium can be selected. The Medium model assumes that data is stored in bank 0x00 and there is more than 64 KB of code. Because data comprise also constants, special care needs to be applied to these. There are 3 main options to deal with constants:

- Constants are copied to RAM at application start and are treated similar to initialized variables. Depending on the number of constants, this route may not be viable because the RAM may be too small.
- Constants are accessed by 24-bit addresses. This means that either memory models Compact or Large are selected or that all constants are declared with qualifier `__far`. This increases code size and decreases application performance.
- Constants are stored only in flash memory and are accessed with 16-bit addresses by usage of ROM Mirror peripheral. In this case, the Medium model may be used with no additional software overhead. The hardware of the MCU takes care to access the correct address in flash memory. However, it must be noted that the linker setting must be adjusted to place constants in the upper half of a 64 KB sector of flash memory. Also, the ROM Mirror must be enabled and configured to access the correct flash memory bank.

4.2 Alignment

The linker places and aligns the different sections generated by the compiler automatically. One may override the defaults using `#pragma` instructions in C code or `.SECTION` instructions in assembly. Care should be exercised when doing so. The following explains more details for code and data sections.

4.2.1 Data alignment

The linker places data in just one section of a type (for memory models small and medium) or in separate sections per source module (for memory models compact and large). The rule for placing the data within a section is same: the large variables, e.g. `long`, are placed first such that the variables are aligned to a 32-bit boundary. This makes sure that only the minimum number of 2 bus accesses is used to access the variable. Next, 16-bit variables are located. As they are placed adjacent to long variables or are placed at the very beginning of the section, it is ensured that these are placed on a 16-bit aligned address. This means that these variables can be accessed by just one bus access. Lastly, the byte variables are placed. Here, the alignment is not of concern because these variables can always be accessed by just one bus access.

It should be noted that there are several data sections in general: at least separate sections for initialized and un-initialized data. These are both placed in RAM. These sections (or in compact and large model all additional sections), there may be a gap of one byte. This is inserted automatically to achieve an alignment on 32-bit or 16-bit boundary for the next section. If one uses special configuration to avoid such gaps, it should be noted that this may result in a performance decrease.

4.2.2 Code alignment

Code is placed on any address by the linker. There is no special alignment exercised. However, if one is concerned about execution speed e.g. of interrupt service routines, it may be worth to use `#pragma` instructions such that the beginning of an ISR is placed on an even address if not on a 32-bit boundary. This can be determined by looking at the generated assembly code and applying knowledge on bus width, buffers, and instruction queue as discussed already.

The linker generally uses Byte alignment to place code. If an alignment of 2 to an even address is desired, the following `#pragma` instruction may be used:

```
#pragma segment FAR_CODE=my_code_segment;align=2
```

If one likes to start a certain function always on the same address, the following following `#pragma` instruction can be used:

```
#pragma segment FAR_CODE=my_code_segment,locate=0xADDRESS
```

Please note that the default section name FAR_CODE is valid only for memory model Medium and Large. For memory models small and Compact, please use CODE instead.

The above mentioned `#pragma segment` instruction is valid until the next one is observed. To switch back to default behaviour, use following instruction:

```
#pragma segment FAR_CODE=FAR_CODE
```

Once again, the example code shown in 3.1 “Wait states and internal supply voltage” is considered. The code is shifted byte by byte and execution time is observed:

Start address	Clock cycles (0 wait states)	Clock cycles (1 wait state)	Clock cycles (2 wait states)
0xFE0003	97	117	149
0xFE0004	108	133	150
0xFE0005	97	138	180
0xFE0006	108	139	181

These results can be explained as follows:

The first three instructions have a length of one byte each. Hence, there is no effect due to code alignment.

The instruction at address 0xFE0006 is a branch instruction that flushes the instruction queue (`BRA FE000C`). Next instruction is 32-bit aligned. It is only one byte long. The following instruction (`CMPW A,#0014`) takes three bytes. Due to the code buffer, these can be fetched without any wait states.

This is followed by a branch instruction to the beginning of the loop (`BC FE0008`). The branch instruction is 32-bit aligned and subject to wait states. The branch target is also 32/bit aligned. Because of this and because the instruction queue had to be flushed after the branch, this instruction fetch is subject to wait states.

As the start address is increased, the alignment of the instruction changes to mostly starting instructions on odd addresses. This results in the increased number of cycles to execute the loop. In addition, wait states may apply more often.

If the start address is shifted even further, one will observe that the clock cycles repeat every 4 bytes, e.g. start address 0xFE0007 shows same cycle count as start address 0xFE0003. This shows the 32-bit width of the flash memory.

5 Additional Information

Information about Cypress Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

Documentation on 16FX family MCUs is available on their respective webpage:

http://www.cypress.com/16lx

6 Document History

Document Title: AN204829 - F²MC-16FX Family, MB96300 Performance Considerations

Document Number: 002-04829

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	11/04/2008	Initial release
*A	5084254	NOFL	01/14/2016	Migrated Spansion Application Note MCU-AN-300244-E-V10 to Cypress format

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless
Spansion Products	spansion.com/products

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums |Blogs | Video |Training

Technical Support

cypress.com/go/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com