65C02MMU

Manual

Version 1

22.1.2021

by Renato Bugge

Introduction

The 6502 is one of the most successful processors of all time. It is still produced in over 100 million units per year and sits in many embedded devices. Of the many variants that can be found, the basic CMOS 6502 (or 65C02) is still widely used without modification, but at some point you often want more memory to play with without having to go to a larger 16-bit core. This can be done with some simple logic and memory banking, but is often cumbersome as one has to split programs and use memory registers to move between the different banks.

The 65C02MMU

The 65C02MMU is meant to help with memory banking on the 65C02. It was first made as a stand-alone MMU to a NMOS 6502, but for a new product one wants everything integrated, so all the parts therefore ended up into a 65C02 verilog core.
The original 65C02 core was based on a 6502 core by Arlet Ottens, which was extended to a full 65C02 instruction set by David Banks and Ed Spittles (in 2016). Both were published on 6502.org, a site dedicated to 6502 hardware and software.

Flavours

The 65C02MMU is not much larger than the 65C02 for its smallest flavour which is with 20-bit addressing. This may not seem much, but it increases the addressable memory to 1MiB which is often enough. There are also flavours for 24-bit, 28-bit and 32-bit addressing.
It must be noted that addresses above the 28-bit address range can only be accessed in the 64KiB-bank mode.

Compability

The 65C02MMU core is compatible with the 65C02 core. It is also somewhat compatible with the 6502 core, but does not contain the non-NOP «illegal instructions». The 65C02MMU core can run 100% of all programs that the 65C02 run unless one starts poking random values into a byte following the $82 «NOP ZP» opcode (with can end in the MMU part being switched on).

The unused opcodes of 6502

Both the 65C02 and the 6502 contains unused opcodes. On the 6502 these opcodes are referred to as «illegal instructions» since they were not originally documented, but none the less were used by many programmers. On the 65C02, some of these opcodes were still unused, and simply made into NOP (no operation) instructions to prevent unwanted behaviour (mainly all $x3 and $xB opcodes). A few of these were NOPs on the original 6502 but were observed to «do» something even if the result were not used. For example the opcodes $FC and $DC are NOP opcodes, but instead of taking 2 cycles to finish (like the documented NOP on the 6502), they took 4 or 5 cycles to complete. This is true for both the 65C02 and 6502 processors, and was found to resembled the LDA/LDY/LDX $addr instructions, but with no change in registers. E.g. those NOP's use the bytes following the opcode as an address. That address is then used to read a memory location and the content discarded.

By looking at both the 65C02 and the 6502, there are a few NOP opcodes that overlap and that allows one or two bytes (immediately following the opcode) to be in the machine code program. These bytes are not used for anything, so why not use them for the MMU?

Of the NOP opcodes, the following are the same on both the 65C02 and the 6502:
$44,$54,$5C,$82,$C2,$D4,$DC,$E2,$F4 and $FC

I have therefore used these to pass parameters to the MMU so that it can be used for efficient handling of memory banking.

The instructions

The following is a list of instuctions with opcodes, with an explanation following further down.

```
$82 $xy       MMS #$xy     Memory Management System register
$44 ZP        MMB $ZP      Memory Management BLOCK register from Zero Page
$E2 $xx       MMB #$xx     Memory Management BLOCK register
$C2 $xx       MMZ #$xx     Memory Management Zeropage BANK register
$5C $xx $yy   MMI $yyxx    Memory Management Interrupt BANK register
$D4 ZP        MMP $ZP,X    Memory Management Protection Table from Indexed Zeropage
$FC $xx $yy   MMF $yyxx    Memory Management Fetch BANK register
$54 ZP        MMF $ZP,x    Memory Management Fetch BANK register from Indexed Zeropage
$DC $xx $yy   MMJ $yyxx    Memory Management Jump BANK register
$F4 ZP        MMJ $ZP,X    Memory Management Jump BANK register from Indexed Zeropage
```

All these instructions are derived from the 6502 illegal «NOP» codes that allow one or two bytes to follow the opcode. The CPU reads these bytes, but they do not result in anything as viewed on the CPU side. That makes them valuable for other things, and the following pages shows what the are used for by the MMU (and thus what to expect from them).

Instruction: MMS, opcode: $82, one parameter byte: $xy

The Memory Management System register (MMS) is responsible for setting up the MMU flags and starting the MMU core. At 65C02 reset, the MMU is off, so if one wants to use it, it has to be switched on.

The following sequence is used to switch on the MMU:
$82 $AA MMS $AA
$82 $55 MMS $55

The reason for this sequence is to prevent accidentally switching on the MMU (or make it less likely). Passing any other sequence with the MMU off will not switch it on. You do not need to have the two MMS instructions following another, it is ok to have other instructions in between.

After the MMU is switched ON, the MMS instruction can be used to pass flags according to the following format:
$82 %01xxxxxx MMS %01xxxxxx

The bits7-6 have to be %01 unless you want to switch the MMU OFF or into STANDBY. For example, using the instruction MMS $80 will result in the MMU being switched off, which means that the cpu goes back to a standard 65C02 mode (with a maximum of 64KiB of memory). If you only want to temporarily switch off the MMU, the command MMS $AA will put the MMU back into standby (and it can be switched on again with MMS $55).

The bits0-5 are used to set the following flags once the MMU is ON:

Bit no	What it controls
5	Switches the BANK_REG_MEM flag ON (1) or OFF (0). When this flag is ON, the internal registers of the MMU are accessible in the $0000-$0005 and $0100-$01FF memory area. See further explanation below.
4	A «1» in this position will switch ON the IGNORE_LSB flag. Similarly a «0» in this bit will switch the IGNORE_LSB flag OFF. This flag affects the MMF ($FC) and MMJ ($DC) instructions so that only bit 4-15 of the address are used if IGNORE_LSB=ON. For the IGNORE_LSB=OFF state, the full 16-bit address field is used. The practical use of this flag is during the 4KiB memory banking mode. In this mode, address fields in the memory range that is banked have their upper 4 bits (bit 15-12) discarded, and their real address upper bits comes from the banking registers. Thus, by ignoring the lower 4 bits of the banking register, it becomes easier to handle the address field (as shown in section xxx).
3	The INBANK flag decides if the MMB instruction addresses the IRQ BANK register or the NMI BANK register. Setting this bit to «1» sets it to NMI mode, while a «0» sets it to IRQ mode.
2	Bit reserved for future use (1= use JBANK as target BANK when storing data with STA/STY/STX/STZ instead of FBANK. FBANK is still used for all the other instructions that load or modify memory.)
1	This bit is «1» to switch on the FULL_BLOCK_MOVE flag. When this flag is on, the full 64KB address range of the 65C02 is banked. When the FULL_BLOCK_MOVE flag is off, only a 4KB memory block (set by the BLOCK register) will be used by the MMU for banking memory. See section YY for further explanation of the two banking modes.
0	This bit controls the PROTECTED_MEMORY flag. Setting it to «1» starts the Protected

	Memory Table control which dictates how and when different memory banks can be accessed.

BANK_REG_MEM flag

The BANK_REG_MEM flag determines whether the internal MMU registers are accessible in the $0000-$01FF memory area. The registers can be read using LDA/LDX/LDY or written to using STA/STX/STY. (Please note that if writing to registers, there may be one extra clock cycle before the MMU register changes, compared to using a MMU instruction).

The following memory locations will affect the subsequent register

- $0000 = The FBANK register. This register controls the memory bank we will fetch to/from in the next instruction and it is usually not the same as where the current code executes (but it can be). The MMF instruction is usually used to write to this register.
- $0001 = The current ZBANK register in the ZBANK stack (pointed to by JBANK). This is were the current Zeropage and Stack resides for the current memory bank (=JBANK). It is normally written to using the MMZ instruction after a MMJ instruction.
- $0002 = ZBANK for previous JBANK (e.g. the Zeropage&Stack location after RTS or RTI)
- $0003 = IBANK register (usually written to using MMI when system register bit 3 is 0)
- $0004 = NBANK register (usually written to using MMI when system register bit 3 is 1)
- $0005 = Contains the JBPOINTER register (which is the JBANK stack pointer)
- $0100-$01FF = JBANK stack
- $0006-$00FF for future implementations

Please note that if you use LDA/LDX/LDY to read the $0000 location with BANK_REG_MEM ON before or after a MMF instruction, the next instruction to read a memory location > $01ff is going to be in the bank pointed to by the MMF. The reason for this is that the MMF instruction is not affected by memory accesses below $01FF when the BANK_REG_MEM register is ON.

Instruction: MMB, opcode: $44, one parameter byte: $ZP (zero page location)

The Memory Management Block register (BLOCK) affects the location of the MMU handled address range in 4 KiB banking mode. The MMB instruction sets the BLOCK value to the byte content of $ZP. When this contains the value $xy, the memory range $y000-$yfff is set as MMU handled memory. Note that the value of x is currently ignored when setting BLOCK.

Example (this code starts from $B000):

```
*=$B000
     LDA #$0A
     STA $01
     MMB $01   ; Sets BLOCK register to $A
```

Please note that if code is running within the MMU memory range, setting MMB will result in the code suddenly being read from bank $0000 (and most likely crashing the program).

If system flag FULL_BLOCK_MOVE is ON, the full memory range $0000-$ffff is handled by the MMU as one 64KiB bank and the BLOCK register is not used.

If FULL_BLOCK_MOVE is OFF, any memory outside of the $y000-$yfff range (set by MMB $xy) is used as ordinary 6502 memory without being affected by the MMU.

NOTE! There is usually no need to change the location of the $y000-$yfff range once it has been set. This is because other 4KiB banks can be accessed using MMF and jumped into using MMJ.

NOTE! Branching out of the 4KiB memory using Bxx instructions will result in the program branching out of the MMU memory range. It is necessary to use the MMJ & JMP instructions if crossing a 4KiB page boundary.

IMPORTANT! If the system uses protected memory (PROTECTED_MEMORY flag is ON), only the memory that is affected by the MMU can be protected.

Instruction: MMZ, opcode: $Z2, one parameter byte: $xx

The Memory Management Zeropage BANK register (ZBANK) affects the location of the Zeropage. If FULL_BLOCK_MOVE is OFF, the MMU handles the memory as 4KiB blocks so that the ZBANK contains the 8 upper bits of the zero page memory range $xx100-$xx1ff. If FULL_BLOCK_MOVE is ON, the MMU handles the memory as 64KiB blocks so that the ZBANK contains the 8 upper bits of the zero page memory range $xx0100-$xx01ff.

IGNORE_LSB does not affect the MMZ instruction.

Instruction: MMF, opcode: $FC and $54, one or two parameter bytes: $xx $yy

Syntax 1: MMF $yyxx ($FC $xx $yy)
Syntax 2: MMF $ZP,x ($54 $ZP)

The Memory Management Fetch Bank (FBANK) register affect the reading/storing of memory from instructions. This is done by the MMF instruction that sets the upper bits of the next accesses to memory (that are read or stored to). E.g. LDA/STA/INC and so on, will be pointed to the memory bank given by MMF. The MMF has four modes of operation:

1. FULL_BLOCK_MOVE is OFF meaning the MMU memory blocks are 4KiB is size, and IGNORE_LSB is OFF meaning all bits of the FBANK register are used:

   The BLOCK is used to see if instruction memory address matches. For example with BLOCK=$A, we get that LDA $Azzz matches such and is requesting to access a MMU memory page. The MMF address is then used as upper bits so that the actual memory read is $yyxxzzz. E.g:

   MMF $0142
   LDA $A123 (with BLOCK register previously set to $A)

   which will result in the content of memory $142123 being loaded into the accumulator (if that memory location exists).

2. FULL_BLOCK_MOVE is OFF meaning the MMU memory blocks are 4KiB is size, and IGNORE_LSB is ON meaning that bits 15-4 of the FBANK register are used:

   The BLOCK is used to see if instruction memory address matches. For example with BLOCK=$A, we get that LDA $Azzz matches such and is requesting to access a MMU memory page. The MMF address is then used as upper bits so that the actual memory read is $0yyxzzz. E.g:

   MMF $0142
   LDA $A123 (with BLOCK register previously set to $A)

   which will result in the content of memory $14123 being loaded into the accumulator.

3. FULL_BLOCK_MOVE is ON meaning the MMU memory blocks are 64KiB is size, and IGNORE_LSB is OFF meaning all bits of the FBANK register are used:

   The BLOCK is not used under 64KiB block size since all instruction memory addresses matches. The MMF address is then used as upper bits so that the actual memory read is $yyxxzzzz. E.g:

   MMF $0142
   LDA $A123 (with BLOCK register previously set to $A)

   which will result in the content of memory $142A123 being loaded into the accumulator (if that memory location exists).

4. FULL_BLOCK_MOVE is ON meaning the MMU memory blocks are 64KiB is size, and IGNORE_LSB is ON meaning that bits 15-4 of the FBANK register are used:

The BLOCK is not used under 64KiB block size since all instruction memory addresses matches. The MMF address is then used as upper bits so that the actual memory read is $0yyxzzzz. E.g:

MMF $0142
LDA $A123 (with BLOCK register previously set to $A)

which will result in the content of memory $14A123 being loaded into the accumulator.

IGNORE_LSB flag

The reason for the IGNORE_LSB flag is to align upper memory bits and lower memory bits so that they become easier to manipulate in assembly during 4KiB access. E.g. if you want to access $yyxzzz, it will take fewer instructions to put together the middle byte ($xz) as they do not have to be bit shifted. This code is a good example on how to do that:

```
; Read 24-bit location with address stored in $01,$02 and $03
; (with MSB in $03 and LSB in $01)
     MMS #$AA       ; MMU to STANDBY
     MMS #$55       ; MMU to ON
     MMS #%11110000 ; Switch IGNORE_LSB=ON, FULL_BLOCK_MOVE=OFF.
     LDA #$56
     STA $01        ; Set up $01-$03 address
     LDA #$34
     STA $02        ; Set up $01-$03 address
     LDA #$12
     STA $03        ; Set up $01-$03 address

; All is now set up, so do the actual read:
     LDX #1
     MMB $02        ; Contains $34 --> BLOCK to range $3000-$3fff
     MMF $02,X      ;Set fetch bank to $123
     LDA ($00,X)    ;Read $01&$02 --> reads from address $123456
```

The MMF $02,x instruction takes memory location $02 and $02+x, so when x=1 we get the address from $02 (LSB) and $03 (MSB) which is stored into FBANK (Fetch Bank register). Since IGNORE_LSB is switched on, FBANK=$0123.
LDA will retrieve address $3456 from $01 and $02 (we reuse x=1 to not touch the y-register). Since BLOCK was set to $3, this is within the MMU banked range. It will therefore not read the content of address $3456, but from $456 in bank $123, e.g. Address $123456.
The code actually reading the address $123456 is only the MMF and LDA, so if you want to extend this to a loop (for copying or alike), that could be done by adding some INC instructions and branches:

```
; All is now set up, so do the actual read:
     LDX  #1
     LDY  #0
loop
     MMB  $02       ; Contains $34 --> BLOCK to range $3000-$3fff
     MMF  $02,X     ;Set bank to $123
innerloop
     LDA  ($00,X)   ;Read $01&$02 --> reads from address $123456
```

```
st   STA  $A000,Y
     INY
     INC  $01
     BNE  innerloop
     INC  st+2
     INC  $02
     BNE  loop
     RTS
```

As can be seen, this inner loop is quite compact, and it could even be extended to use a 24-bit address for storage (with for example $04-$06 as target address) with few efforts. If one only needs to copy a few bytes, this example is faster:

```
; A shorter and faster copy loop for small
     MMS  #$AA       ; MMU to STANDBY
     MMS  #$55       ; MMU to ON
     MMS  #%11100000 ; Switch IGNORE_LSB=OFF, FULL_BLOCK_MOVE=OFF.
     MMB  #$34       ; Contains $34 --> BLOCK to range $3000-$3fff

     MMF  $0123      ; Set bank to $123
     LDY  #0
     LDX  #8
innerloop
ld   LDA  $3456,Y    ; Reads from address $123456+Y
st   STA  $A000,Y    ; Store at $A000+Y
     INY
     BNE  innerloop
     INC  ld+2
     INC  st+2
     DEX
     BNE  innerloop
     RTS
```

This example is faster since the inner loop is smaller than the previous (omitting INC $ZP), but one needs to be careful not to INC the MSB of «LDA $3456» to above $3F since a value of $40 would mean that the MMU would not handle it like a bank. That is were FULL_BLOCK_MOVE = ON comes in.

Note that other accesses to the range $3000-$3fff after the above code will also use the FBANK register at MSB for storage. If this is not intended, you should change the FBANK register to another location or set the MMU on standby (MMU #$AA) before accessing the $3000-$3FFF memory.

The FULL_BLOCK_MOVE flag

In order to more efficiently manage memory copying, the FULL_BLOCK_MOVE flag can be switched on. This will give a full 64KiB banked memory so that the full address is used without the BLOCK register affecting the range. Before switching this on, be sure that the current program runs within the same bank as set by MMJ (See notes under the MMJ opcode for detail on this).

Copying larger block of data under FULL_BLOCK_MOVE = ON:

```
; (some previous code with MMJ $xxxx to a non-zero bank)
```

```
     MMS  #%11100010 ; FULL_BLOCK_MOVE=ON.

     MMF  $0012      ; Set bank to $12
     LDY  #0
     LDX  #$80
innerloop
ld   LDA  $3456,Y    ; Reads from address $123456+Y
st   STA  $0000,Y    ; Store at $120000+Y
     INY
     BNE  innerloop
     INC  ld+2
     STX  st+2
     INX
     BNE  innerloop
     RTS
```

Here we copy 32KiB of data with just 4 instructions over 14 CPU cycles per byte in the inner loop.

Instruction: MMJ opcode: $DC and $F4, one or two parameter bytes: $xx $yy or $ZP

Syntax 1: MMJ $yyxx ($DC $xx $yy)
Syntax 2: MMJ $ZP,x ($F4 $ZP)

The Memory Management Jump Bank (JBANK) register can affect the target address of the next JMP or JSR instruction. There are two different ways the bank setting works according to wether FULL_BLOCK_MOVE is ON or OFF.

For FULL_BLOCK_MOVE OFF (4KiB bank size):

The MMJ $yyxx instruction sets the JBANK address to $yyxx (IGNORE_LSB is OFF) or $0yyx (IGNORE_LSB is ON). The 4 most significant bits of the JMP $zzzz or JSR $zzzz address is ignored if these bits have the same value as BLOCK register (e.g. the target is withing the MMU bank). This means that the JMP or JSR will be to the address $yyxxzzz or $0yyxzzz.

Example 1:

```
MMB $A0         ; $Axxx is now handled by MMU
MMJ $1234       ; Set J(u)MP target bank
JMP $A567       ; JMP with MMU
```

This sequence will jump to address $1234567. The code does not need to reside at any special location and can be either in a different bank or outside of the 4KiB memory BLOCK address range.

Example 2:

```
MMB  $A0        ; $Axxx is now handled by MMU
LDA  #$34
STA  $01
LDA  #$12
STA  $02
LDX  #1
MMJ $01,x       ; Set J(u)MP target bank to $1234
JMP $A567       ; JMP with MMU
```

In this sequence, MMJ reads address $01 and $01+x=$02, and sets the block to that ($1234). It therefore jumps to address $1234567.

Please note that if a JMP occurs to outside the 4KiB memory BLOCK adress range (even after a MMJ instruction), it will not jump to that bank but to the normal address. In that case, the next JMP to within the BLOCK address range will use the parameters of the last unused MMJ.

For FULL_BLOCK_MOVE ON (64KiB bank size):

If you set MMJ to some non-zero value and run the program outside of the BLOCK range (through a JMP there), the program actually runs in bank $0000 even with MMJ indicating some other value. This would not cause concern if FULL_BLOCK_MOVE is off, but if you then switch FULL_BLOCK_MOVE on, the program would suddenly shift to another bank and another address. That may be intentional, but if its not, then the program would crash.

To prevent such crashes, the safest way to set FULL_BLOCK_MOVE=ON is with the following code:

(Code must reside in bank zero, e.g. base 6502 memory range $0000-$ffff. Here we use $1000)

```
*=$1000
     MMJ  $0000
     JMP  safejump
safejump
     MMS  #%11100010 ; FULL_BLOCK_MOVE=ON.
     MMJ  $0012      ; bank $12
     JMP  $3456      ; Jump to $123456
```

The next best way is to ensure that the running code is within the BLOCK register setting. Its then safe to swith FULL_BLOCK_MOVE=ON since the memory range in the BLOCK register is not affected. E.g.:

```
*=$1000
     MMB  $A0        ; BLOCK is $Axxx
     MMJ  $0123      ; bank $12
     JMP  $A456      ; Jump to $123456

*=$3456              ; Code has previously been copied to $123456
     MMS  #%11100010 ; FULL_BLOCK_MOVE=ON. Still in $123 bank
```

Special case for the MMJ:

By using bit 2 of the system register (accessed using MMS) it is possible to use MMJ to set the BANK for storing data with STX/STY/STA and STZ. The reason for this special way of using MMJ is to increase the rate of moving data between different BANK's:

```
     MMS  #$AA       ; MMU to STANDBY
     MMS  #$55       ; MMU to ON
     MMS  #%11100110 ; FULL_BLOCK_MOVE=ON, MMJ_STORE=ON.

     MMF  $0012      ; Set bank to $12
     MMJ  $0104      ; Set target bank to $104
     LDY  #0
     LDX  #$80
innerloop
ld   LDA  $3456,Y    ; Reads from address $123456+Y
st   STA  $0000,Y    ; Store at $1040000+Y
     INY
     BNE  innerloop
     INC  ld+2
     STX  st+2
     INX
     BNE  innerloop
     RTS
```

Here we copy 32KiB of data between different BANK's with just 4 instructions over 14 CPU cycles per byte in the inner loop (ikke implementer enda!)

Instruction: MMI opcode: $5C, two parameter bytes: $xx $yy

Syntax: MMI $yyxx ($5C $xx $yy)

The register is set to the value $yyxx.

The MMI opcode sets the interrupt bank registers. There are two such registers, one for the IRQ (IBANK) and one for the NMI (NBANK). Both are accessed by this command, but only one at a time. The system register flag INBANK controls which register is set. If INBANK is ON, the NMI register (NBANK) is set, while if INBANK is OFF, the IRQ register (IBANK) will be set.

The IBANK and NBANK acts more or less like the JBANK register in that the program execution is moved to that bank. Since they are interrupt registers, the IBANK contains which bank to jump to in the case of a IRQ interrupt, while the NBANK contains the same for a NMI interrupt.

Please note that the target address of the IRQ and NMI vectors must be within the current banked range if the MMU is set to 4KiB banks (e.g. same as the BLOCK register). If the MMU uses 64KiB banking, the full address range is available for the IRQ/NMI target vectors.

Memory Management Protection Table setup

Instruction: MMB opcode: $D4, one parameter byte: $ZP

Syntax: MMP $ZP,X ($D4 $ZP)

MMP is always followed by an MMI $ADDR in which $ADDR is the MSB address of the memory page that is handled. Tables are the size of the maximum numbers of pages.

Target byte contains contains a register code in which the bits7-0 are used to set the attributes of the memory page being targeted. If all bits are zero, the page is unprotected memory (=memory executable, readable & writeable).

Bit no	What it controls
0	Write protect on/off. Memory can only be written to if bit is 0.
1	Jump protect on/off. Code in page can not be jumped to if bit is 1.
2	Read protect on/off. Memory can't be read if this bit is 1.
3	Supervisor mode on/off. If this bit is 1, code running in the memorypage can't jump out of that page, change memory in other pages or change MMU settings.
4	IRQ page. The page will be used for the IRQ interrupt routine.
5	NMI page. The page will be used for the NMI interrupt routine.
6	Reserved for future use
7	Reserved for future use

JBANK[0] is always 0 (starts with bank 0)
We set JBANK:
MMJ 01 --> JBANK[JBPOINTER+1]=01
JBANK[JBPOINTER] is current BANK memory at which we run
JSR --> JBPOINTER+1 (last stage)
JMP--> JBANK[JBPOINTER]=JBANK[JBPOINTER+1]

ZP0--> FETCH --> DECODE
1c 2c
ZP0--> MMJ/MMF --> DECODE

```
reg  [1:0] MMU_STATE;      //=0 for off, =1 for standby =3 for ON.
reg  SWAP;                 //=1 to swap $44 and $54 instructions (MMJ/MMF)
reg  IGNORE_LSB;   //=0 to ingnore bit 3-4 of the bank register
reg  FULL_BLOCK_MOVE;          //=1 if full 64KB block is moved with BANK, =0 if 4KB block is moved with BANK
reg  PROTECTED_MEMORY;       //=1 if using PTABLE, =0 if not used
reg  NIBANK;                   //=0 for IBANK (IRQ) , =1 for NBANK (NMI)
```