HT8 MCU Extended Instruction Set Applications

D/N: AN0407E

Introduction

The Holtek Flash MCU extended instruction set is used to address the full data memory area. The extended instructions can directly access data memory without using indirect addressing when the data memory is located in different Banks expect for Bank 0, thus improving the CPU firmware performance. For MCUs that support extended instructions, each instruction will require an additional cycle when compared to corresponding general instruction for its execution. This application note will use the HT66F70A as an example MCU to show how to use extended instructions.

Accessing Different Banks of Data Memory Using Extended Instructions

Taking the HT66F70A device as an example, we will use extended instructions to access different data memory banks to carry out simple arithmetic operations. For extended instructions, they require a specific data memory address forma. For example for the address “A3H” in Bank 3, the data memory address would be “03A3H”, with the higher byte indicating the bank number and the lower byte indicating the specific address in the corresponding bank. The following example shows how to assign values to the data memory in Bank 1 and Bank 2 respectively and to get the values in these two Banks for add operation and also how to store the sum into Bank 3. It should be noted that the extended instructions begin with the letter “L”. A compiler error will occur if non-extended instructions are used in the program, as the non-extended instructions only perform operations on Bank 0 by default.

Direct Addressing Program Example Using Extended Instructions:

```
include HT66F70A.inc
ds .section AT 080H 'data'
cs .section 'code'
   ORG     00H          ; HT66F70A RESET VECTOR
MAIN:
   MOV     A, 11H
   LMOV    [0180H], A   ; Transfer the value ‘11H’ to the address pointed by ‘80H’ in Bank 1
   MOV     A, 22H
   LMOV    [0280H], A   ; Transfer the value ‘22H’ to the address pointed by ‘80H’ in Bank 2
   LMOV    A, [0180H]   ; Transfer the data pointed by ‘80H’ in Bank 1 to ACC
   LADD    A, [0280H]   ; Add operation with the values read from “80H” of Bank 1 and Bank 2 respectively
   LMOV    [0380H], A   ; Store the sum into the address defined by ‘80H’ in the Bank 3
```

The above example accesses the address directly. For users who are used to operating with variables, the following example shows how to access variables in different Banks. This example implements the same function with as the above one. It should be noted that using non-extended instructions in this example will not generate a compiler error, however the instructions only operate on variables located in Bank 0. For example, if changing the extended instruction “LMOV” to “MOV” in the second line of the main function “LMOV ADDRESS1,A”, the value of ADDRESS1 will not be changed, instead the first address of Bank 0 will be assigned with the value ‘11H’.

```
include HT66F70A.inc
RAMBank 1 DS1                     ; Declare that DS1 is located in RAM Bank 1
RAMBank 2 DS2                     ; Declare that DS2 is located in RAM Bank 2
RAMBank 3 DS3                     ; Declare that DS3 is located in RAM Bank 3
ds1 .section AT 080H 'data'       ; Declare the variable to be located in RAM Bank 1
ADDRESS1 DB ?
ds2 .section AT 080H 'data'       ; Declare the variable to be located in RAM Bank 2
ADDRESS2 DB ?
ds3 .section AT 080H 'data'       ; Declare the variable to be located in RAM Bank 3
ADDRESS3 DB ?
cs .section 'code'
   ORG     00H                    ; HT66F70A RESET VECTOR
MAIN:
   MOV     A,11H
   LMOV    ADDRESS1,A             ; Transfer the value '11H' to variable ADDRESS1 located in the '80H' of Bank 1
   MOV     A,22H
   LMOV    ADDRESS2,A             ; Transfer the value '22H' to variable ADDRESS2 located in the '80H' of Bank 2
   LMOV    A,ADDRESS1             ; Transfer the value of ADDRESS1 to Accumulator
   LADD    A,ADDRESS2             ; Add the values of ADDRESS1 and ADDRESS2, stores the sum into ACC
   LMOV    ADDRESS3,A             ; Store the sum into the variable ADDRESS3 in Bank3
   JMP     $
```

The value in ADDRESS3 will now be equal to 33H at the address of 0380H, which is address 80 in Bank 3. The example has shown how the MCU can implement direct addressing in every bank.

Access Different Banks of Data Memory using Indirect Addressing with Memory Pointers MP1L/MP1H, MP2L/MP2H

Five Memory Pointers, known as MP0, MP1L, MP1H, MP2L and MP2H, are provided in the HT66F70A. These Memory Pointers are physically implemented in the Data Memory and can be manipulated in the same way as normal registers providing a convenient way with which to address and track data. When any operation to the relevant Indirect Addressing Registers is carried out, the actual address that the microcontroller is directed to is the address specified by the related Memory Pointer. MP0, together with Indirect Addressing Register, IAR0, are used to access data from Bank 0, while MP1L/MP1H together with IAR1 and MP2L/MP2H together with IAR2 are used to access data from all Banks according to the values in the corresponding MP1H or MP2H register. Note that the function of MP1L and MP1H here is similar to the program memory bank pointer BP. Taking the HT66F70A as an example, we will use indirect addressing instructions to transfer the value ‘11H’ to Bank1 and the value ‘22H’ to Bank2, then exchange the value of these two banks. If the MCU does not have extended instructions, then it will need to change the value of BP. For the value in Banks except for Bank 0, use MP1L/MP1H and IAR1, MP2L/MP2H and IAR2 to access the data memory:

```
include HT66F70A.inc
RAMBank 1 DS1                  ; Declare DS1 to be located in RAM Bank 1
RAMBank 2 DS2                  ; Declare DS2 to be located in RAM Bank 2
ds .section AT 080H 'data'     ; Preset the variable located in Bank 0
BLOCK DB ?
NUM DB ?
ds1 .section AT 080H 'data'    ; Declare the variable to be located in Bank 1
ADDRESS1 DB ?
ds2 .section AT 080H 'data'    ; Declare the variable to be located in Bank 2
ADDRESS2 DB ?
cs .section 'code'
   ORG 00H                     ; HT66F70A RESET VECTOR
MAIN:
   MOV     A,080H
   MOV     BLOCK,A             ; One bank of data memory has 128 bytes
   MOV     A,01H
   MOV     MP1H,A              ; MP1H=01H, setup MP1L/MP1H and IAR1 to indirectly address Bank 1
   MOV     A,02H
   MOV     MP2H,A              ; MP2H=01H, setup MP2L/MP2H and IAR2 to indirectly address Bank 2
   MOV     A,OFFSET ADDRESS1
   MOV     MP1L,A              ; Obtain the address of variable ADDRESS1 in Bank 1, namely the first address
                               ; of Bank1 – "0180H"
   MOV     A,OFFSET ADDRESS2
   MOV     MP2L,A              ; Obtain the address of variable ADDRESS2 in Bank 2, namely the first address
                               ; of Bank2 – "0280H"
LOOP:                          ; Assign the value "011H" to the whole Bank 1 data memory and assign the value
                               ; "022H" to the whole Bank 2data memory using a LOOP function
   MOV     A,011H
   MOV     IAR1,A              ; Assign the value "011H" to Bank 1
   MOV     A,022H
   MOV     IAR2,A              ; Assign the value "022H" to Bank 2
   INC     MP1L                ; Bank 1 pointer incremented by 1
   INC     MP2L                ; Bank 2 pointer incremented by 1

   SDZ     BLOCK
   JMP     LOOP
   MOV     A,080H
   MOV     BLOCK,A
   DEC     MP1L
   DEC     MP2L
LOOP1:                         ; Exchange the values of Bank1 and Bank2 using a LOOP1 function
   LMOV    A,IAR1
   LMOV    NUM,A               ; Transfer the value of Bank 1 to the middle variable NUM
   LMOV    A,IAR2
   LMOV    IAR1,A              ; Transfer the value of Bank 2 to Bank 1
   LMOV    A,NUM
   LMOV    IAR2,A              ; Transfer the value in middle variable to Bank 2
   DEC     MP1L                ; Bank 1 pointer decremented by 1
   DEC     MP2L                ; Bank 2 pointer decremented by 1
   SDZ     BLOCK
   JMP     LOOP1
   JMP     $                   ; Stop
```

After executing the LOOP routine, the values in the whole of Bank 1 is 11H and the values in Bank 2 is 22H. After executing the LOOP1 exchange routine, the values in Bank is 22H and the values in Bank 2 is 11H.

Differences between Extended Instructions and Non-extended Instructions – Efficiency Enhancements using Multiple Banks

As the Holtek C V3 has a different architecture with Holtek C V2, extended instructions affect them in different ways. Here we introduce Holtek C V3.

Non-extended instructions can only access bank 0, however extended instructions can directly access all the data memory banks.

Non-instructions	Corresponding Extended Instructions
MOV, ADD, SUB …… Each instruction size: 1 word	LMOV, LADD, LSUB …… Each instruction size: 2 words
Can only access data memory bank 0	Can directly access all data memory banks

Access Data Memory Using Non-extended Instructions

For MCUs that do not support extended instructions, addressing banks other than Bank 0 can only be accessed using indirect addressing. As the following example shows, indirect addressing has 5 more instructions when compared to direct addressing. When the data memory Bank 0 overflows, the user can define the variables used less in other banks and define variables which are frequently used in Bank 0.

Non-extended Instructions Direct addressing Bank 0	Non-extended Instructions Indirect Addressing Except for Bank 0
Rambank 0 ds ds .section 'data' _var0 db ?	Rambank 1 ds ds .section 'data' _var1 db ?

MOV A, 40H
MOV _var0, A
	MOV A, BANK _var1
	OR A, ROM_BANK FUNC
	MOV BP, A
	MOV A, OFFSET _var1
	MOV MP1, A
	MOV A, 40H
	MOV IAR1, A

Code size : 2 words	Code size : 7 words

Comparison of Accessing Data Memory Banks Expect for Bank0 Using Extended Instructions and Non-extended Instructions

- For device that do not support extended instructions: indirectly address the data memory Banks except for Bank 0.
- For device that support extended instructions: can directly address the data memory Banks apart from Bank 0 and require smaller code size than if using indirect addressing. There follows an example using the “LMOV” assembly language instruction:

Non-extended Instructions Indirectly Addressing Except for Bank 0	Extended instructions Directly Addressing Except for Bank 0
Rambank 1 ds ds .section ‘data’ _var1 db ?	Rambank 1 ds ds .section ‘data’ _var1 db ?

MOV A, BANK _var1
OR A, ROM_BANK FUNC
MOV BP, A
MOV A, OFFSET _var1
MOV MP1, A
MOV A, 40H
MOV IAR1, A
	MOV A, 40H
	LMOV _var1, A

Code size : 7 words	Code size : 3 words

Extended Instructions will Affect the C Compiler Results

Holtek C V3 behavior when the MCU does not support extended instructions

Holtek C V2 uses indirect addressing to access variables, no matter where it is located, which is less efficient.

Holtek C V3 has changed its architecture to configure all variables in Bank0, which is convenient for direct addressing.

Advantages:

- Directly addressing code is more efficient

Disadvantages:

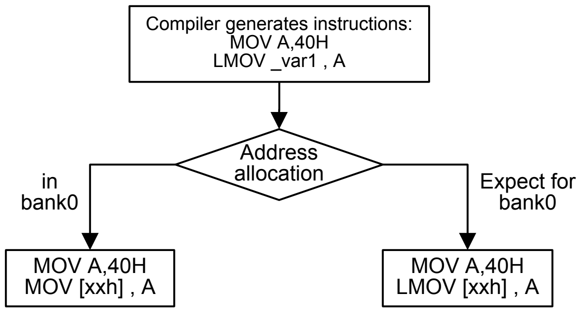
- If there are too many variables and Bank 0 is used up, the user should manually re-configure some variables to other banks. This condition will be improved if the MCU supports extended instructions.
- For accessing variables in banks other than Bank 0, the only way is to use indirect addressing.

Holtek C V3 Behavior when the MCU supports extended instructions

The user does not have to manually adjust the variable configurations if the MCU supports extended instructions.

- Variables can be completely arranged by the Compiler or Linker
- Non-extended instructions can be used for directly addressing variables in Bank 0.

  Extended instructions can be used for directly addressing variables in banks other than Bank 0.



Benefit Estimation when Holtek C V3 Adds Extended Instructions into its Architecture

- In C programs, each variable access instruction can save 4 words
- For an actual application program and for an MCU with a 4K word program memory, an 86 times variable access will have a 20% reduction in code size. The more variables that are used, the more obvious the benefits will be.

Note: The variable access time depends on the product. The figures for the 86 times access above is calculated according to the samples collected.

Benefit estimation example:

	Total ROM	Code Size	Benefit Description
Use non-extended instructions indirect addressing	4096 words	1668 words	When using extended instruction, the code size will be reduced to 80% of the code that uses indirect addressing (1324/1668)
Use extended instructions direct addressing	4096 words	1668-86×4 =1324 words	

Extended Instruction Set Summary

Table Conventions

x: Bits immediate data

m: Data Memory address

A: Accumulator

i: 0~7 number of bits

addr: Program memory address

Mnemonic	Description	Cycles	Flag Affected
Arithmetic Operation			
LADD A,[m]	Add Data Memory to ACC	2	Z, C, AC, OV, SC
LADDM A,[m]	Add ACC to Data Memory	2[Note]	Z, C, AC, OV, SC
LADC A,[m]	Add Data Memory to ACC with Carry	2	Z, C, AC, OV, SC
LADCM A,[m]	Add ACC to Data memory with Carry	2[Note]	Z, C, AC, OV, SC
LSUB A,[m]	Subtract Data Memory from ACC	2	Z, C, AC, OV, SC, CZ
LSUBM A,[m]	Subtract Data Memory from ACC with result in Data Memory	2[Note]	Z, C, AC, OV, SC, CZ
LSBC A,[m]	Subtract Data Memory from ACC with Carry	2	Z, C, AC, OV, SC, CZ
LSBCM A,[m]	Subtract Data Memory from ACC with Carry, result in Data Memory	2[Note]	Z, C, AC, OV, SC, CZ
LDAA [m]	Decimal adjust ACC for Addition with result in Data Memory	2[Note]	C
Logical Operation			
LAND A,[m]	Logical AND Data Memory to ACC	2	Z
LOR A,[m]	Logical OR Data Memory to ACC	2	Z
LXOR A,[m]	Logical XOR Data Memory to ACC	2	Z
LANDM A,[m]	Logical AND ACC to Data Memory	2[Note]	Z
LORM A,[m]	Logical OR ACC to Data Memory	2[Note]	Z
LXORM A,[m]	Logical XOR ACC to Data Memory	2[Note]	Z
LCPL [m]	Complement Data Memory	2[Note]	Z
LCPLA [m]	Complement Data Memory with result in ACC	2	Z
Increment & Decrement			
LINCA [m]	Increment Data Memory with result in ACC	2	Z
LINC [m]	Increment Data Memory	2[Note]	Z
LDECA [m]	Decrement Data Memory with result in ACC	2	Z
LDEC [m]	Decrement Data Memory	2[Note]	Z
Rotate			
LRRA [m]	Rotate Data Memory right with result in ACC	2	None
LRR [m]	Rotate Data Memory right	2[Note]	None
LRRCA [m]	Rotate Data Memory right through Carry with result in ACC	2	C
LRRC [m]	Rotate Data Memory right through Carry	2[Note]	C
LRLA [m]	Rotate Data Memory left with result in ACC	2	None
LRL [m]	Rotate Data Memory left	2[Note]	None
LRLCA [m]	Rotate Data Memory left through Carry with result in ACC	2	C
LRLC [m]	Rotate Data Memory left through Carry	2[Note]	C
Data Move			
LMOV A,[m]	Move Data Memory to ACC	2	None
LMOV [m],A	Move ACC to Data Memory	2[Note]	None
Bit Operation			
LCLR [m].i	Clear bit of Data Memory	2[Note]	None
LSET [m].i	Set bit of Data Memory	2[Note]	None
Branch			
LSZ [m]	Skip if Data Memory is zero	2[Note]	None
LSZA [m]	Skip if Data Memory is zero with data movement to ACC	1[Note]	None
LSNZ [m]	Skip if Data Memory is not zero	2[Note]	None
LSZ [m].i	Skip if bit i of Data Memory is zero	2[Note]	None
LSNZ [m].i	Skip if bit i of Data Memory is not zero	2[Note]	None
LSIZ [m]	Skip if increment Data Memory is zero	2[Note]	None
LSDZ [m]	Skip if decrement Data Memory is zero	2[Note]	None
LSIZA [m]	Skip if increment Data Memory is zero with result in ACC	2[Note]	None
LSDZA [m]	Skip if decrement Data Memory is zero with result in ACC	2[Note]	None
Table Read Operation			
LTABRD [m]	Read table to TBLH and Data Memory	3[Note]	None
LTABRDL [m]	Read table (last page) to TBLH and Data Memory	3[Note]	None
LITABRD [m]	Increment table pointer TBLP first and Read table to TBLH and Data Memory	3[Note]	None

Mnemonic	Description	Cycles	Flag Affected
LITABRDL [m]	Increment table pointer TBLP first and Read table (last page) to TBLH and Data Memory	3[Note]	None
Miscellaneous			
LCLR [m]	Clear Data Memory	2[Note]	None
LSET [m]	Set Data Memory	2[Note]	None
LSWAP [m]	Swap nibbles of Data Memory	2[Note]	None
LSWAPA [m]	Swap nibbles of Data Memory with result in ACC	2	None

Note: 1. For these extended skip instructions, if the result of the comparison involves a skip then up to four cycles are required, if no skip takes place two cycles are required.

2. Any extended instruction which changes the contents of the PCL register will also require three cycles for execution.

Extended Instructions Definition

The extended instructions are used to directly access the data stored in any data memory sectors.

LADC A, [m] Add Data Memory to ACC with Carry

Description: The contents of the specified Data Memory, Accumulator and the carry flag are added. The result is stored in the Accumulator.

Operation: ACC ← ACC + [m] + C

Affected flag(s): OV, Z, AC, C, SC

LADCM A, [m] Add ACC to Data Memory with Carry

Description: The contents of the specified Data Memory, Accumulator and the carry flag are added. The result is stored in the specified Data Memory.

Operation: [m] ← ACC + [m] + C

Affected flag(s): OV, Z, AC, C, SC

LADD A, [m] Add Data Memory to ACC

Description: The contents of the specified Data Memory and the Accumulator are added. The result is stored in the Accumulator.

Operation: ACC ← ACC + [m]

Affected flag(s): OV, Z, AC, C, SC

LADDM A, [m] Add ACC to Data Memory

Description: The contents of the specified Data Memory and the Accumulator are added. The result is stored in the specified Data Memory.

Operation: [m] ← ACC + [m]

Affected flag(s): OV, Z, AC, C, SC

LAND A, [m] Logical AND Data Memory to ACC

Description: Data in the Accumulator and the specified Data Memory perform a bitwise logical AND operation. The result is stored in the Accumulator.

Operation: ACC ← ACC“AND”[m]

Affected flag(s): Z

LANDM A, [m] Logical AND ACC to Data Memory

Description: Data in the specified Data Memory and the Accumulator perform a bitwise logical AND operation. The result is stored in the Data Memory.

Operation: [m] ← ACC“AND”[m]

Affected flag(s): Z

LCLR [m] Clear Data Memory

Description: Each bit of the specified Data Memory is cleared to 0.

Operation: [m] ← 00H

Affected flag(s): None

LCLR [m].i	Clear bit of Data Memory
Description	Bit i of the specified Data Memory is cleared to 0.
Operation	[m].i ← 0
Affected flag(s)	None

LCPL [m]	Complement Data Memory
Description	Each bit of the specified Data Memory is logically complemented (1_s complement). Bits which previously contained a 1 are changed to 0 and vice versa.
Operation	[m] ← $\overline{[m]}$
Affected flag(s)	Z

LCPLA [m]	Complement Data Memory with result in ACC
Description	Each bit of the specified Data Memory is logically complemented (1's complement). Bits which previously contained a 1 are changed to 0 and vice versa. The complemented result is stored in the Accumulator and the contents of the Data Memory remain unchanged.
Operation	ACC ← $\overline{[m]}$
Affected flag(s)	Z

LDAA [m]	Decimal-Adjust ACC for addition with result in Data Memory
Description	Convert the contents of the Accumulator value to a BCD (Binary Coded Decimal) value resulting from the previous addition of two BCD variables. If the low nibble is greater than 9 or the AC flag is set, then a value of 6 will be added to the low nibble. Otherwise the low nibble remains unchanged. If the high nibble is greater than 9 or the C flag is set, then a value of 6 will be added to the high nibble. Essentially, the decimal conversion is performed by adding 00H, 06H, 60H or 66H depending on the Accumulator and flag conditions. Only the C flag may be affected by this instruction which indicates that if the original BCD sum is greater than 100, it allows multiple precision decimal addition.
Operation	[m] ← ACC + 00H or [m] ← ACC + 06H or [m] ← ACC + 60H or [m] ← ACC + 66H
Affected flag(s)	C

LDEC [m]	Decrement Data Memory
Description	Data in the specified Data Memory is decremented by 1.
Operation	[m] ← [m] – 1
Affected flag(s)	Z

LDECA [m]	Decrement Data Memory with result in ACC
Description	Data in the specified Data Memory is decremented by 1. The result is stored in the Accumulator. The contents of the Data Memory remain unchanged.
Operation	ACC ← [m] – 1
Affected flag(s)	Z

LINC [m]	Increment Data Memory
Description	Data in the specified Data Memory is incremented by 1.
Operation	[m] ← [m] + 1
Affected flag(s)	Z

LINCA [m]	Increment Data Memory with result in ACC
Description	Data in the specified Data Memory is incremented by 1. The result is stored in the Accumulator. The contents of the Data Memory remain unchanged.
Operation	ACC ← [m] + 1
Affected flag(s)	Z

LMOV A, [m]	Move Data Memory to ACC
Description	The contents of the specified Data Memory are copied to the Accumulator.
Operation	ACC ← [m]
Affected flag(s)	None

LMOV [m], A Move ACC to Data Memory
Description The contents of the Accumulator are copied to the specified Data Memory.
Operation [m] ← ACC
Affected flag(s) None

LOR A, [m] Logical OR Data Memory to ACC
Description Data in the Accumulator and the specified Data Memory perform a bitwise logical OR operation. The result is stored in the Accumulator.
Operation ACC ← ACC“OR”[m]
Affected flag(s) Z

LORM A, [m] Logical OR ACC to Data Memory
Description Data in the specified Data Memory and the Accumulator perform a bitwise logical OR operation. The result is stored in the Data Memory.
Operation [m] ← ACC“OR”[m]
Affected flag(s) Z

LRL [m] Rotate Data Memory left
Description The contents of the specified Data Memory are rotated left by 1 bit with bit 7 rotated into bit 0.
Operation [m].(i+1) ← [m].i (i=0~6)
[m].0 ← [m].7
Affected flag(s) None

LRLA [m] Rotate Data Memory left with result in ACC
Description The contents of the specified Data Memory are rotated left by 1 bit with bit 7 rotated into bit 0. The rotated result is stored in the Accumulator and the contents of the Data Memory remain unchanged.
Operation ACC.(i+1) ← [m].i (i=0~6)
ACC.0 ← [m].7
Affected flag(s) None

LRLC [m] Rotate Data Memory Left through Carry
Description The contents of the specified Data Memory and the carry flag are rotated left by 1 bit. Bit 7 replaces the Carry bit and the original carry flag is rotated into bit 0.
Operation [m].(i+1) ← [m].i (i=0~6)
[m].0 ← C
Affected flag(s) C ← [m].7
C

LRLC A [m] Rotate Data Memory left through Carry with result in ACC
Description Data in the specified Data Memory and the carry flag are rotated left by 1 bit. Bit 7 replaces the Carry bit and the original carry flag is rotated into the bit 0. The rotated result is stored in the Accumulator and the contents of the Data Memory remain unchanged.
Operation ACC.(i+1) ← [m].i (i=0~6)
ACC.0 ← C
Affected flag(s) C ← [m].7
C

LRR [m] Rotate Data Memory right
Description The contents of the specified Data Memory are rotated right by 1 bit with bit 0 rotated into bit 7.
Operation [m].i ← [m].(i+1) (i=0~6)
[m].7 ← [m].0
Affected flag(s) None

LRRA [m] Rotate Data Memory right with result in ACC
Description Data in the specified Data Memory are rotated right by 1 bit with bit 0 rotated into bit 7. The rotated result is stored in the Accumulator and the contents of the Data Memory remain unchanged.
Operation ACC.i ← [m].(i+1) (i=0~6)
ACC.7 ← [m].0
Affected flag(s) None

LRRC [m]	Rotate Data Memory right through Carry
Description	The contents of the specified Data Memory and the carry flag are rotated right by 1 bit. Bit 0 replaces the Carry bit and the original carry flag is rotated into bit 7.
Operation	[m].i ← [m].(i+1) (i=0~6) [m].7 ← C C ← [m].0
Affected flag(s)	C

LRRCA [m]	Rotate Data Memory right through Carry with result in ACC
Description	Data in the specified Data Memory and the carry flag are rotated right by 1 bit. Bit 0 replaces the Carry bit and the original carry flag is rotated into bit 7. The rotated result is stored in the Accumulator and the contents of the Data Memory remain unchanged.
Operation	ACC.i ← [m].(i+1) (i=0~6) ACC.7 ← C C ← [m].0
Affected flag(s)	C

LSBC A, [m]	Subtract Data Memory from ACC with Carry
Description	The contents of the specified Data Memory and the complement of the carry flag are subtracted from the Accumulator. The result is stored in the Accumulator. Note that if the result of subtraction is negative, the C flag will be cleared to 0, otherwise if the result is positive or zero, the C flag will be set to 1.
Operation	ACC ← ACC – [m] – C
Affected flag(s)	OV、Z、AC、C、SC、CZ

LSBCM A, [m]	Subtract Data Memory from ACC with Carry and result in DataMemory
Description	The contents of the specified Data Memory and the complement of the carry flag are subtracted from the Accumulator. The result is stored in the Data Memory. Note that if the result of subtraction is negative, the C flag will be cleared to 0, otherwise if the result is positive or zero, the C flag will be set to 1.
Operation	[m] ← ACC – [m] – C
Affected flag(s)	OV、Z、AC、C、SC、CZ

LSDZ [m]	Skip if Decrement Data Memory is 0
Description	The contents of the specified Data Memory are first decremented by 1. If the result is 0 the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is not 0 the program proceeds with the following instruction.
Operation	[m] ← [m]－1 Skip if [m] = 0
Affected flag(s)	None

LSDZA [m]	Skip if decrement Data Memory is zero with result in ACC
Description	The contents of the specified Data Memory are first decremented by 1. If the result is 0, the following instruction is skipped. The result is stored in the Accumulator but the specified Data Memory contents remain unchanged. As this requires the insertion of a dummy instruction hile the next instruction is fetched, it is a two cycle instruction. If the result is not 0, the program proceeds with the following instruction.
Operation	ACC ← [m]－1 Skip if ACC = 0
Affected flag(s)	None

LSET [m]	Set Data Memory
Description	Each bit of the specified Data Memory is set to 1.
Operation	[m] ← FFH
Affected flag(s)	None

LSET [m].i	Set bit of Data Memory
Description	Bit i of the specified Data Memory is set to 1.
Operation	[m].i
Affected flag(s)	None

LSIZ [m] Skip if increment Data Memory is 0

Description: The contents of the specified Data Memory are first incremented by 1. If the result is 0, the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is not 0 the program proceeds with the following instruction.

Operation: [m] ← [m] + 1
Skip if [m] = 0

Affected flag(s): None

LSIZA [m] Skip if increment Data Memory is zero with result in ACC

Description: The contents of the specified Data Memory are first incremented by 1. If the result is 0, the following instruction is skipped. The result is stored in the Accumulator but the specified Data Memory contents remain unchanged. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is not 0 the program proceeds with the following instruction.

Operation: ACC ← [m] + 1
Skip if ACC = 0

Affected flag(s): None

LSNZ [m].i Skip if bit i of Data Memory is not 0

Description: If bit i of the specified Data Memory is not 0, the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is 0 the program proceeds with the following instruction.

Operation: Skip if [m].i≠0

Affected flag(s): None

LSNZ [m] Skip if Data Memory is not 0

Description: If the content of the specified Data Memory is not 0, the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is 0 the program proceeds with the following instruction.

Operation: Skip if [m]≠0

Affected flag(s): None

LSUB A, [m] Subtract Data Memory from ACC

Description: The specified Data Memory is subtracted from the contents of the Accumulator. The result is stored in the Accumulator. Note that if the result of subtraction is negative, the C flag will be cleared to 0, otherwise if the result is positive or zero, the C flag will be set to 1.

Operation: ACC ← ACC – [m]

Affected flag(s): OV、Z、AC、C、SC、CZ

LSUBM A, [m] Subtract Data Memory from ACC with result in Data Memory

Description: The specified Data Memory is subtracted from the contents of the Accumulator. The result is stored in the Data Memory. Note that if the result of subtraction is negative, the C flag will be cleared to 0, otherwise if the result is positive or zero, the C flag will be set to 1.

Operation: [m] ← ACC – [m]

Affected flag(s): OV、Z、AC、C、SC、CZ

LSWAP [m] Swap nibbles of Data Memory

Description: The low-order and high-order nibbles of the specified Data Memory are interchanged.

Operation: [m].3~[m].0 ↔ [m].7~[m].4

Affected flag(s): None

LSWAPA [m] Swap nibbles of Data Memory with result in ACC

Description: The low-order and high-order nibbles of the specified Data Memory are interchanged. The result is stored in the Accumulator. The contents of the Data Memory remain unchanged.

Operation: ACC.3~ACC.0 ← [m].7~[m].4
ACC.7~ACC.4 ← [m].3~[m].0

Affected flag(s): None

LSZ [m]	Skip if Data Memory is 0
Description	If the content of the specified Data Memory is 0, the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is not 0 the program proceeds with the following instruction.
Operation	Skip if [m] = 0
Affected flag(s)	None
LSZA [m]	Skip if Data Memory is 0 with data movement to ACC
Description	The contents of the specified Data Memory are copied to the Accumulator. If the value is zero, the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is not 0 the program proceeds with the following instruction.
Operation	ACC ← [m] Skip if [m] = 0
Affected flag(s)	None
LSZ [m].i	Skip if bit i of Data Memory is 0
Description	If bit i of the specified Data Memory is 0, the following instruction is skipped. As this requires the insertion of a dummy instruction while the next instruction is fetched, it is a two cycle instruction. If the result is not 0, the program proceeds with the following instruction.
Operation	Skip if [m].i = 0
Affected flag(s)	None
LTABRD [m]	Move the ROM code to TBLH and data memory
Description	The program code addressed by the table pointer (TBHP and TBLP) is moved to the specified Data Memory and the high byte moved to TBLH.
Operation	[m] ← program code (low byte) TBLH ← program code (high byte)
Affected flag(s)	None
LTABRDL [m]	Read table (last page) to TBLH and Data Memory
Description	The low byte of the program code (last page) addressed by the table pointer (TBLP) is moved to the specified Data Memory and the high byte moved to TBLH.
Operation	[m] ← program code (low byte) TBLH ← program code (high byte)
Affected flag(s)	None
LITABRD [m]	Increment table pointer low byte first and read table to TBLH and data memory
Description	Increment table pointer low byte, TBLP, first and then the program code addressed by the table pointer (TBHP and TBLP) is moved to the specified Data Memory and the high byte moved to TBLH.
Operation	[m] ← program code (low byte) TBLH ← program code (high byte)
Affected flag(s)	None
LITABRDL [m]	Increment table pointer low byte first and read table(last page)to TBLH and data memory
Description	Increment table pointer low byte, TBLP, first and then the low byte of the program code (last page) addressed by the table pointer (TBLP) is moved to the specified Data Memory and the high byte moved to TBLH.
Operation	[m] ← program code (low byte) TBLH ← program code (high byte)
Affected flag(s)	None
LXOR A, [m]	Logical XOR Data Memory to ACC
Description	Data in the Accumulator and the specified Data Memory perform a bitwise logical XOR operation. The result is stored in the Accumulator.
Operation	ACC ← ACC“XOR”[m]
Affected flag(s)	Z
LXORM A, [m]	Logical XOR ACC to Data Memory
Description	Data in the specified Data Memory and the Accumulator perform a bitwise logical XOR operation. The result is stored in the Data Memory.
Operation	[m] ← ACC“XOR”[m]
Affected flag(s)	Z

Version and Modify Information

Date	Author	Issue
2015.12.17	Fengyi Tao (馮毅韜)	First Version

Disclaimer