Notes - Unit 3

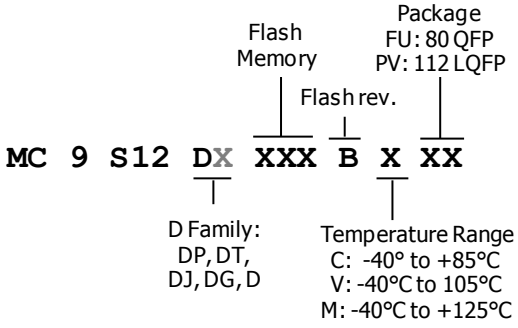
Overview of the HCS12 Microcontroller

- The HCS12 is a family of Freescale® microcontrollers (MCUs) targeted to automotive and process control applications. MCUs incorporate a processor (CPU) and peripherals (e.g., memory, serial communication interfaces, timer functions, A/D converter) in a single chip. Among the peripherals available, we can mention:
  - Input Capture (IC)
  - Output Compare (OC)
  - Pulse-Width Modulation (PWM)
  - Controller area network (CAN)
  - Byte data link control (BDLC)
  - 8-channel (10 bits per channel) ADC modules.
  - Serial peripheral interface (SPI)
  - Serial communication interface (SCI)
  - Interintegrated circuit (I^2C)

- Most HCS12 devices have a bus clock speed of 25 MHz and include on-chip SRAM (Static RAM) and EEPROM to hold data and/or programs. External memory can also be used.
- HCS12 Microcontrollers use on-chip flash memory to hold Program Memory.
  Flash memory: It can be erased and reprogrammed electrically. Most microcontrollers nowadays use on-chip flash memory as their program memory (In a PC, the BIOS is stored in a flash memory).
- HCS12 devices have a 16-bit CPU. With a 16-bit address line, the CPU can handle up to 64K memory positions. Usually (as a convention) each position holds one byte, which means that we can address up to 64 KB.

HCS12 Family Numbering System

There exist many HCS12 families (H,A,B,C,D,E,G,K,P,Q,X). We will be working with the 'D' family (**MC9S12D**) which is designed for automotive multiplexing applications. In the Dragon12-Light Board, we have:

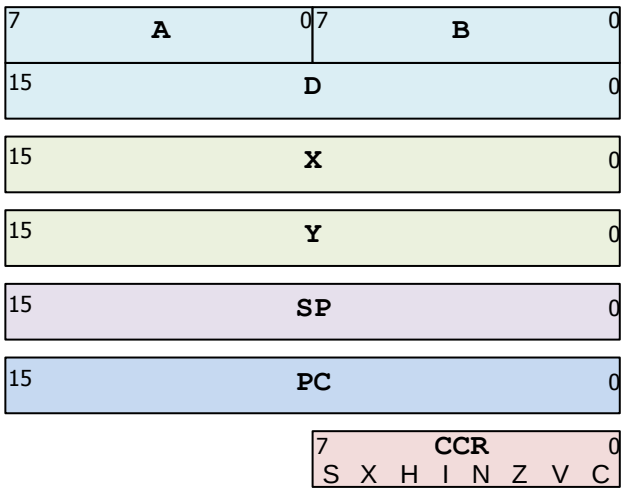


- **MC9S12DG256BCPV**: As per the MC9S12D family datasheet, this device belongs to the 'DG' family. The device has a temperature range of -40°C to 85°C and it comes in the package 112LQFP package. The device features: 256KB of Flash Memory, 12KB of RAM, 4KB of EEPROM, 2 CANs, 2 SCI ports, 2 SPI ports, 1 I^2C module, 2 ADC modules (8 channels per module), 8 PWMs, and 91 I/O pins.
- What about the **MC9S12DT128BCFU** device?

HCS12 Registers

The HCS12 CPU includes:

- General-purpose accumulators (A and B): These 8-bit registers can be concatenated into a single 16-bit accumulator (D) for certain instructions (D_H=A, D_L=B).
- Index Registers (X and Y): These 16-bit registers are used for indexed addressing. The contents of these registers are usually added to another value to form an effective address.
- Stack Pointer (SP): This 16-bit register points to the top byte of the stack. A stack is a LIFO structure and can be located anywhere in the standard 64 KB address space.
- Program Counter (PC): This 16-bit register holds the address of the next instruction to be executed.
- Condition Code Register (CCR): This 8-bit register contain five status indicators, two interrupt masking bits (I,X), and a STOP instruction control bit (S). The status indicators are Carry/Borrow (C), Overflow (V), Zero (Z), Negative (N), and Half Carry (H).



An HCS12 microcontroller also includes I/O registers, subdivided into: data, data direction, control, and status registers. They occupy the memory space (or address space).

Memory Space: It is the amount of memory (both data and instructions in the HCS12) that the processor can handle. The 16-bit address line in the HCS12 can handle 64KB (assuming each memory content is a byte). A straightforward implementation would use a memory chip of 64 KB. However, in real-life applications, the memory space is filled by memory devices of different technologies (Flash, EEPROM, SRAM), the stack, and I/O registers. The whole memory space does not need to be populated.

HCS12 ADDRESSING MODES

- Addressing modes determine how the CPU instructions access operands (usually memory locations) to be operated upon.
- Format of HCS12 instruction: **OPCODE** (1 to 2 bytes) | **OPERANDS** (0 to five bytes)
- Usually a CPU instruction utilizes only one addressing mode during the course of execution. But sometimes, a CPU instruction might use more than one. Recall that the effective address is a 16-bit number.
- A multibyte number (more than 1 byte) is stored in memory from the most significant to the least significant byte, starting from a low address to high addresses.

INHERENT

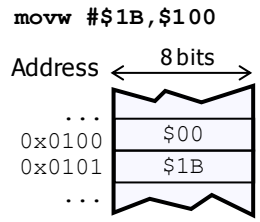
Instructions using this addressing mode have either no operands or the operands are CPU registers. CPU does not access memory locations. Examples of HCS12 instructions that use this mode:

```
nop          ; This instruction has no operands
iny          ; Y ← [Y] + 1
clrb         ; B ← 0
coma         ; A ← not(A)
tfr D,Y      ; Y ← [D]
```

IMMEDIATE

Here, the operand values are embedded in the instruction. The symbol '#' indicates an immediate value.

```
ldaa #$12          ; A ← $12. The symbol '$' indicates a hexadecimal value.
ldx #$4C32         ; X ← $4C32
ldy #$67           ; Y ← $0067
movb #$FE,$13A     ; m[$013A] ← $FE. movb: Move byte.
movw #$1B,$100     ; m[$0100] ← $00, m[$0101] ← $1B. movw: Move Word. The
                   assembler expects a 16-bit value in this instruction. Thus, the value
                   $001B is stored starting from address $0100.
```

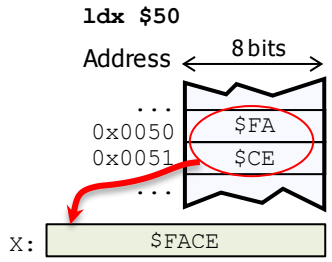


- `movb, movw`: In the examples, only the sources (`#$FE,#$1B`) use the immediate addressing mode. The destination (addresses) are actually using the Extended Mode (see below).

DIRECT

Here, instructions access operands from memory in the address range `$0000` to `$00FF`. We only indicate addresses from `$00` to `$FF` (8 bits are included as operand, saving program space and execution time). A program can be optimized by placing the most frequently accessed data on this area of memory. This mode is also called zero-page addressing.

```
ldaa $3F     ; A ← [$3F]. Note that since we do not use '#', the provided value
               indicates a memory address.
ldx $50      ; XH ← [$50], XL ← [$51]. A 16-bit value stored in memory
               (starting from $50) is placed on X.
```



EXTENDED

Similar to the Direct Mode, but we can access operands in the entire range from `0000` to `$FFFF`.

```
ldaa $E109   ; A ← [E109]. A gets the contents of the memory address E109
dec $910F    ; m[$910F] ← [$910F] - $01
```

RELATIVE

This addressing mode is used only by branch instructions. The distance of the branch (or jump) is called *offset*. The offset is specified as a signed number (2's complement). Branch instructions include an 8-bit OPCODE and an offset.

- Short branch instructions support an 8-bit offset, which allows a range between -128 (`$80`) to 127 (`$7F`).
- Long branch instructions support a 16-bit offset, which allows a range between -32768(`$8000`) to 32767(`$7FFF`).
- Loop primitive instructions (e.g.: `DBEQ, DBNE`) support a 9-bit offset, which allows a range of -256(`$100`) to 255 (`$0FF`).

✓ A programmer usually specifies a label of the instruction to branch to. The assembler will calculate the offset value. If the offset is zero, the CPU executes the instruction immediately following the branch instruction.

```
start     …
          …
          …
          bmi start;  if flag N (of CCR) =1 then PC ← PC + offset, else PC points to next instruction
          …
```

* Note that the next instruction is NOT necessarily located at `PC ← PC+1`

INDEXED

- Here, two components are used to compute the effective address of: an operand or the target of a branch instruction
  - **Base Address** (stored in a base register: `X,Y,SP, or PC`)
  - **Offset**: Distance of the target from the base address. It is a signed number (2's complement)
  - **Effective address = Base Address + Offset.** Instruction Syntax: offset, Base Address Register
- We now present the variations of the indexed addressing mode. In the following examples, we utilize the notation `[]` to indicate the contents of either a register or a memory location. The symbol `':'` denotes concatenation of numbers.

CONSTANT OFFSET

The offset is a signed 5-bit, 9-bit, or 16-bit constant.

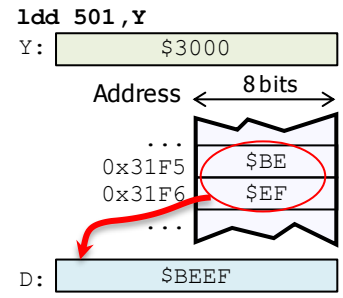
Offset	Range
5-bit	-16 to 15
9-bit	-256 to 255
16-bit	-32768 to 32767

`ldaa $0E,X` ; A ← [$0E + [X]]. If X=$2000, then A gets the contents of the memory address $200E

`ldab -20,Y` ; B ← [-20 +[Y]]. If Y=$1000, then B gets the contents of the memory address $1000-$14

`jmp $7A,PC` ; PC ← [$7A+[PC]]. We do not get the contents of the effective address, but rather jump to the effective address.

`ldd 501,Y` ; D_H ← [501+[Y]], D_L ← [501+[Y]+1]. If Y=$3000, then D gets the contents of the memory addresses $31F5 and $31F6.



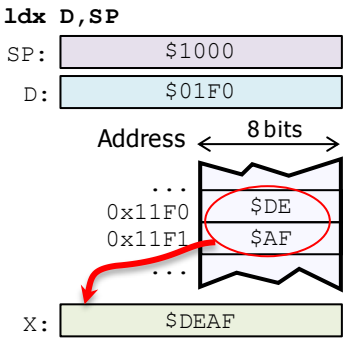
ACCUMULATOR OFFSET

Here, the offset is in an accumulator (which can be A, B, or D).
Operand Address = Base Address + Accumulator

`ldaa B,X` ; A gets the value of memory address B+X. B an X are unchanged

`staa B,X` ; m([B]+[X]) ← A. The contents of A are stored at address B+X.

`ldx D,SP` ; X ← [[D]+[SP]:[D]+[SP]+1]. X gets the value starting at memory address D+SP. X_H gets the value at address D+SP. X_L gets the value at the address D+SP+1.

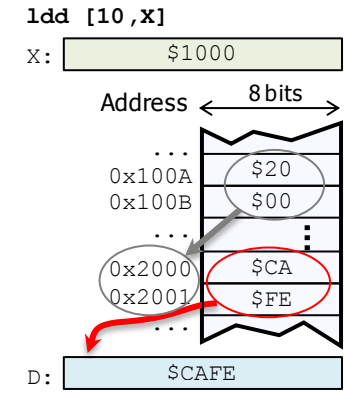


16-BIT OFFSET INDIRECT

The offset is a 16-bit constant. The sum of the offset and the base address is called the pointer (because it points to the address where the contents will be accessed). The memory contents of this pointer represent the effective address (actually, effective address = `[pointer]:[pointer+1]` as a memory location is 8-bits wide).

`ldaa [$10C3,X]` ; A ← [[$10C3+X]:[$10C3+X+1]]. A gets the value pointed by the contents of addresses $10C3+X and $10C3+X+1

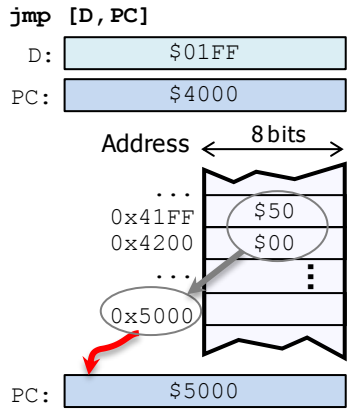
`ldd [10,X]` ; D gets the value pointed by the contents of memory addresses $0A+X and $0A+X+1. If X=$1000, we first get the 16-bit pointer value located at addresses $100A and $100B, which is $2000. Then, we read the contents of the memory addresses $2000 and $2001 and place it on D.



ACCUMULATOR D OFFSET INDIRECT

Similar to the 16-bit offset indexed indirect. With the difference that the offset is in D.

`jmp [D,PC]` ; If PC=$4000 and D=$01FF, we first read the contents of address $41FF. Since this is a jump instruction, the target to jump to is a 16-bit value, which we read starting from address $41FF. If we assume the 16-bit value to be $5000, then we jump to this address (or PC ← $3000). The jump instruction is special, because all it needs to get is the target address (not the contents of that address).



AUTO PREDECREMENT, PREINCREMENT, POSTDECREMENT, OR POSTINCREMENT

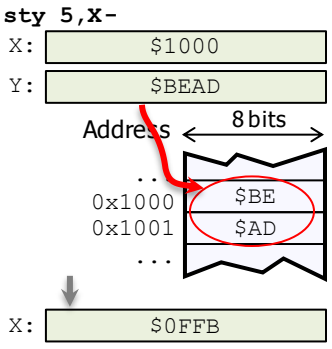
Base index registers may be X, Y, or SP (does not make sense for PC as it will change the instruction flow). Tee CPU allows the index register to be incremented or decremented by any integer value in the ranges of -8 to -1 or 1 to 8.

Pre-decrement/pre-increment: adds the base address and the offset to get an effective memory address. The effective memory address is written into the base index register. Then, it accesses the contents of the memory location specified by the effective address. In the examples, assume that X and Y were initially `$1000`.

```
staa 4,-X     ; X ← $1000-4=$0FFC. The instructions then stores the contents of A on memory location $0FFC.
ldaa 2,+Y     ; Y ← $1000+2=$1002. Then, A gets the contents of memory location $1002.
```

Post-decrement/post-increment: The effective address is the value on the base index register. The instruction access the contents of the memory location specified by this memory address. Then, a new effective address is created by adding the value of the base index register and the offset, this new effective address is stored in the base index register. In the examples, assume that X was initially `$1000` and Y was initially `$BEAD`.

```
ldab 8,X+     ; B gets the memory contents of address $1000. Then $1008 is written
                into X
sty  5,X-     ; YH is stored in memory location $1000, YL is stored in memory location
                $1001. Then X receives the new value of $0FFB.
```



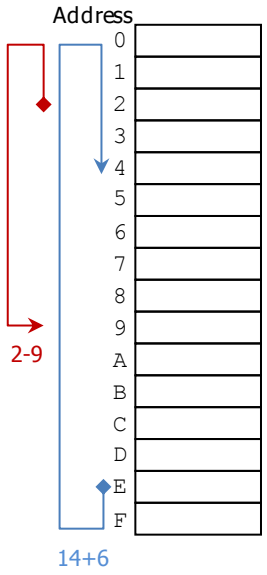
WRAPAROUND FOR EFFECTIVE ADDRESSES

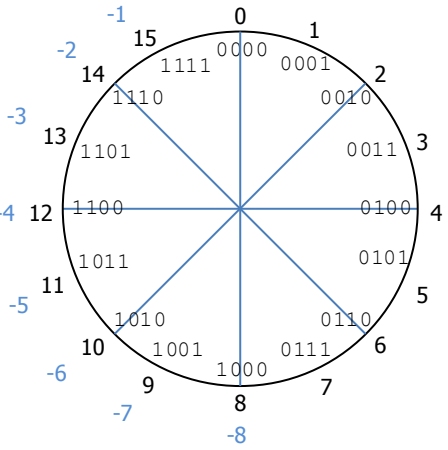
The effective address is a positive number whose range is from `$0000` to `$FFFF`. When computing the effective address we can end up with a number that is out of bounds (`$0000` to `$FFFF`) or a negative number.
An efficient way to deal with this issue is to wraparound the resulting numbers when they are out of bounds (as in a clock).

For simplicity's sake, let's assume we are dealing with 4 bits. The effective address range is then `$0` to `$F`.

If for example, we add the numbers 14(`$E`) and $6, the results is 20(`$14`). 20 is out of bounds, so we wraparound, and the answer is `$4`.

Likewise, if we subtract `9` from `2`, the result is -7, which is out of bound. So, we wraparound (this time in the opposite direction), and the answer is 9.





Addition of unsigned integers

To implement it, let's use the example of 20+6 in binary form. We notice that if we only keep the 4 LSBs (i.e., we omit the carry), the remaining answer is the one we are looking for.

	c_4=1	c_3=1	c_2=1	c_1=1	c_0=0	
14 =		1	1	1	0	+
7 =		0	1	1	1	
21 =	1	0	1	0	1	
					$5	

Subtraction of unsigned integers

Here, if we use the example of 2 - 9 in binary, we assume we have a borrow, and the answer would be 1001 with a borrow for the next stage. If we just use the 4 LSBs and discard the borrow, the remaining answer is the one we are looking for.

	b_4=1	b_3=0	b_2=0	b_1=1	b_0=0	
2 =	0	0	1	0		−
9 =	1	0	0	1		
9 =	1	0	0	1		

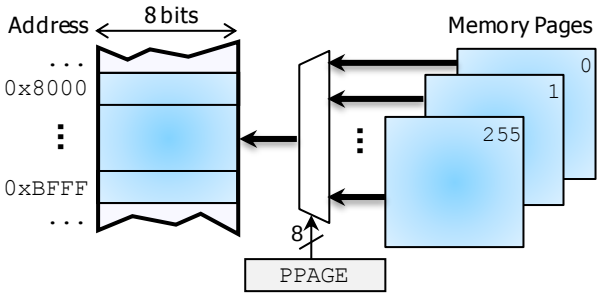
About implementing the subtraction of unsigned numbers using 2's complement representation

The arithmetic circuits inside the HCS12 support unsigned and signed operations. In particular the subtraction of unsigned numbers can be seen as a 2's complement operation. In the '2 - 9' example, we notice that after sign-extending (this also makes sure we do not overflow), if we just keep the 4 LSBs, we get the answer we are looking for (if we think of the result as an unsigned number). Moreover, we can use the MSB (which is NOT the carry out) as our borrow bit.

	c_5=0	c_4=0	c_3=1	c_2=1	c_1=1	c_0=0	
2 =	0	0	0	1	0		+
-9 =	1	0	1	1	1		
-7 =	1	1	0	0	1		
						$5	

In general, the addition/subtraction of unsigned/signed numbers uses the same hardware, it's just that we interpret the result in a different way. The only difference though, is that the overflow in unsigned numbers is the carry out, this is not the case of 2's complement numbers.

ADDRESSING MORE THAN 64 KB



The HCS12 incorporates hardware that supports addressing a larger memory space than the standard 64 Kbytes. The expanded memory system is accessed using a bank-switching scheme.

The HCS12 treats the 16 Kbytes of memory space from `$8000` to `$BFFF` as the program memory window. The system includes an 8-bit page register (`PPAGE`), which allows up to 256 16KB program memory "pages" to be switched into and out of the memory window. This provides up to $256 \times 16KB = 2^{12}KB = 4MB$ of paged program memory.

✓ A bank-switching scheme is preferred over a large linear address space. In systems with large linear address spaces, instructions require more bits for the address line, thereby increasing CPU overhead.

A SAMPLE OF HCS12 INSTRUCTIONS

Here, we provide examples on the first four group of HCS12 instructions. Throughout the course, we will be referring to the HCS12 CPU Reference Manual Rev. 4.0, which contains the entire instruction set, machine code, supported addressing modes, and which CCR bits are affected (if any).

LOAD AND STORE INSTRUCTIONS

- Load instructions copy memory contents into an accumulator or register. Load instructions (except `LEA_` instructions) affect CCR bits (`Z, N, V`).
- Store instructions copy the content of a CPU register to memory. Store instructions update the N an Z bits of CCR.
- Table 5.1 of the HCS12 CPU Reference Manual lists the available instructions. Here are some examples:

Instruction	Operation	Addressing Mode
`ldx #1023`	`X ← $3FF`	Immediate
`ldab $04FA`	`B ← [$04FA]`	Extended
`ldd 6,Y`	`D ← [6+[Y]]:[6+[Y]+1]`	Indexed - constant offset (5 bits)
`lds D,X`	`SP ← [[D]+[X]]`	Indexed - accumulator offset
`ldy $FE`	`Y ← [$FE]:[$FF]` (Y_H ← [$FE], Y_L ← [$FF])	Direct
`ldaa [$BED,X]`	`A ← [ [$BED+[X]] ]`	Indexed - 16 bit offset indirect
`ldd [$FEE,Y]`	`D ← [ [$FEE+[X]]:[$FEE+[X]+1] ]`	Indexed - 16-bit offset indirect
`leax -17,X`	`X ← [X]-17` *X gets the effective address, `CCR` is not modified	Indexed - constant offset (9 bits)
`staa 256`	`m[$100] ← A`	Extended
`stx [$BEE,SP]`	`m[ptr]:m[ptr+1] ← X, ptr=[$BEE+[SP]]:[$BEE+[SP]+1]`	Indexed - 16-bit offset indirect
`std [D,X]`	`m[ptr]:m[ptr+1] ← D, ptr=[[D]+[X]]:[[D]+[X]+1]`	Indexed - Accumulator D offset indirect
`stab [$C0,A]`	`m[ [$C0+[A]] ] ← B`	Indexed - 16-bit offset indirect
`sts $FAD,Y`	`m[ $FAD+[Y]:$FAD+[Y]+1 ] ← SP`	Indexed - constant offset (16 bits)

TRANSFER AND EXCHANGE INSTRUCTIONS

- Transfer instructions copy the contents of a register of accumulator into another register or accumulator. Source content is not changed by the operation.
  - Transfer register to register (`TFR`) is a universal transfer instruction. A `TFR` instruction DOES NOT affect the CCR bits (except when `CCR` is the destination register). The same goes for `TSX, TSY, TXS, TYS,` and `TPA`. `TAP` (transferring from `A` to `CCR`) does affect the `CCR` bits.
  - Transfer A to B (`TAB`) and Transfer B to A (`TBA`) instructions do affect the N,Z, V bits of CCR.
  - **Sign extension**: When transferring from a 8-bit register into a 16-bit register, the 8-bit number is sign-extended to 16 bits. Here, the 8-bit number is assumed to be represented in 2's complement.
  - The sign extend 8-bit operand (`SEX`) instruction is a special case of the universal transfer instruction. This instruction sign-extends an 8-bit number to 16 bits. The 8-bit number is copied from `A`, `B`, or `CCR` into `D`, `X`, `Y`, or `SP`. This instruction does not affect the CCR bits.

- Exchange instructions exchange the contents of pairs of registers or accumulators. When the first operand in an EXG instruction is 8-bits and the second operand is 16 bits, the 8-bit operand is zero-extended to 16 bytes as it is copied into the 16-bit register. These instructions (`EXG, XGDX, XGDY`) do not affect the CCR bits.

- Table 5.2 of the HCS12 CPU Reference Manual lists the available instructions. The addressing mode of this instruction is Inherent. Here are some examples:

Instruction	Operation	Comments
tfr CCR,X	X ← [CCR]	Sign extension to CCR
tfr Y,D	D ← [Y]	
tfr X,A	A ← [X_L], X_L=X(7…0)	Only the lower byte of X is transferred to A
tap	CCR ← [A]	
tsx	X ← [SP]	
tfr SP,A	A ← [SP_L], SP_L=SP(7…0)	Only the lower byte of SP is transferred to A
sex A,D	D ← [A]	A is sign-extended to 16 bits
exg Y,B	B ← [Y_L], Y ← $00:[B]	Zero extension to B
xgdy	D ← Y, Y ← D	
exg CCR,D	D ← $00:[CCR], CCR ← [$D_L$]	Zero extension to CCR

MOVE INSTRUCTIONS

Move instructions allow the transferring of data from:

- Memory location to another memory location.
- CPU registers or values to memory (and vice versa).

There are two instruction: `movb <src>,<dest>` (move byte) and `movw <src>,<dest>` (move word). Six combinations of immediate, extended, and indexed addressing are allowed to specify source and destination addresses:

- Extended to Extended
- Immediate to Extended
- Extended to Indexed
- Indexed to Extended
- Indexed to Indexed
- Immediate to Index

Note that using the Immediate addressing mode for the destination would not make sense as the Extended mode provides an instant address. Also, we can also use the Direct mode, as it is a version of Extended mode. Here are some examples:

Instruction	Operation	Addressing Mode Source , Destination
movb $1F00, $4000	m[$4000] ← [$1F00]	Extended , Extended
movb #$FA, $3FFF	m[$3FFF] ← $FA	Immediate, Extended
movb $2000, 4,-X	m[[X]-4] ← [$2000]	Extended, Indexed (pre-decrement)
mobw [$FF,X] ,$1FFF	m[$1FFF] ← [[$FF+[X]]:[$FF+[X]+1]]	Indexed(16-bit offset indirect), Extended
mobw 2,X, 0,Y	m[Y] ← [X+2]	Indexed (constant offset), Indexed (constant offset)
mobw #$09, D,Y	m([D]+[Y]) ← $00, m([D]+[Y]+1) ← $09	Immediate, Indexed (Acc. D offset)

ADD AND SUBTRACT INSTRUCTIONS

- 8- and 16-bit addition and subtraction can be performed between registers or between registers and memory. Special instructions support index calculation. Instructions that add the carry bit (or borrow in subtraction) in the condition code register (CCR) facilitate multiple precision computation.
- Table 5.4 of the HCS12 CPU Reference Manual lists the available instructions. Here are some examples:

Instruction	Operation	Comments	Addressing Mode
adda #$F1	A ← [A]+$F1		Immediate
abx	X ← $00:[B] + [X]	B is zero extended	Inherent
adca $1001	A ← [A] + [$1001] + C	C: carry bit of CCR	Extended
addd $DEED	D ← [D] + [$DEED]:[$DEED+1]		Extended
suba 2,X	A ← [A]-[2+[X]]		Indexed - constant offset
subd $FE,Y	D ← [D] - [$FE+[Y]]:[$FE+[Y]+1]		Indexed - constant offset
sbcb D,X	B ← [B]-[[D]+[X]]-C	C: carry bit of CCR (interpreted as borrow)	Indexed - Acc. offset

INSTRUCTION QUEUE

- The HCS12 uses an instruction queue to increase execution speed. The queue operation is automatic, and generally transparent to the user.
- Queue logic prefetches program information and positions it for execution (instructions are executed sequentially, one at a time). The HCS12 has the advantages of independent fetches (unlike a pipelined CPU that can execute more than one instruction at the same time), and maintains a straightforward relationship between bus and execution cycles, this facilitates program tracking and debugging.
- There are three 16-bit stages in the instruction queue. Instructions enter the queue at stage 1 and roll out after stage 3. We can select individual bytes in the queue. An OPCODE prediction algorithm determines the location of the next OPCODE in the instruction queue.
- Each instruction refills the queue by fetching as many bytes as the instruction has. Program information is fetched in aligned 16-bit words.

EXAMPLES

- Write an instruction sequence to swap the contents of memory locations `$0F00` and `$0F0A`.
```
ldaa $F00
movb $F0A, $F00
staa $F0A
```

- Write an instruction sequence to subtract the number stored at `$2000` from that stored in `$2001`, and store the difference at the address `X` (i.e., at the address whose value is `X`)
```
ldaa $2001
suba $2000
staa 0,X
```

- Write an instruction sequence to add B to the 16-bit word stored in memory locations `$A000` and `$A001`. Treat the value in B as a signed number. Store the result starting at a memory address whose value is located at address `Y+21`.
```
sex B,D
addd $A000
std [Y,21]
```

- Variables in memory. In the following high-level language sequence of instructions, the variables i, j, and k are located in memory locations `$A000`, `$A005`, and `$A00A`. Write the equivalent assembly instruction sequence.
```
i = 21; j = 50
k = i + j - 5
```

  Assembly instructions:
```
ldaa $A000        ; A ← $15
adda $A005        ; A ← A + $32 = $47
suba #$5          ; A ← A - $5 = $42
staa $A00A        ; m[$A00A] ← $42
```

- Give an instruction that can store the contents of Y starting at a memory location with an address smaller than X-9.
```
sty -10,X
```

- The numbers `$15`, `$78`, `$FA`, and `$E4` are stored at memory locations `$1000`, `$1001`, `$1002`, `$1003` respectively. Write an instruction sequence that adds up all these numbers and store the result in the address whose value is the contents of `Y`. Treat the numbers as unsigned integers and use 16 bits for the results to ensure the correctness of the answer.
```
ldaa $1000 ; A ← [$1000]
exg A,X    ; X ← $00:[A], A ← XL
ldab $1001 ; B ← [$1001]
abx        ; X ← $00:[B] +[X]
ldab $1002
abx
ldab $1003
abx
stx 0,Y    ; m[Y] ← [X]
```

- Redo the previous example, but treat the numbers `$15`, `$78`, `$FA`, and `$E4` as signed.

```
ldaa $1000    ; A ← [$1000]
sex A,D       ; D ← [A] (A is sign-extended)
std 0,Y

ldab $1001    ; B ← [$1001]
sex B,X       ; X ← [B] (B is sign-extended)
stx $BFFF
ldd 0,Y
addd $BFFF
std 0,Y
```

```
ldab $1002    ; B ← [$1002]
sex B,X       ; X ← [B] (B is sign-extended)
stx $BFFF
ldd 0,Y
addd $BFFF
std 0,Y

ldab $1003    ; B ← [$1003]
sex B,X       ; X ← [B] (B is sign-extended)
stx $BFFF
ldd 0,Y
addd $BFFF

std 0,Y       ; m[Y] ← [D]
```