- **More on addressing modes.**
- **9S12 cycles and execution time.**
- **AS12 Assembler Directives**
- Huang, Sections 1.6 through 1.10
  - Using X and Y registers as pointers
  - How to tell which branch instruction to use
  - How to hand assemble a program
  - Number of cycles and time taken to execute an 9S12 program

The HCS12 has 6 addressing modes

Most of the HC12's instructions access data in memory
There are several ways for the HC12 to determine which address to access

Effective Address:
    Memory address used by instruction

ADDRESSING MODE:
    How the HC12 calculates the effective address

HC12 ADDRESSING MODES:
    INH Inherent
    IMM Immediate
    DIR Direct
    EXT Extended
    REL Relative (used only with branch instructions)
    IDX Indexed (won't study indirect indexed mode)

Using X and Y as Pointers
• Registers X and Y are often used to point to data.
• To initialize pointer use

```
ldx #table
```

not

```
ldx table
```

• For example, the following loads the address of table ($2000) into X; i.e., X will point to table:

```
ldx #table ; Address of table => X
```

The following puts the first two bytes of table ($0C7A) into X. X will not point to table:

```
ldx table ; First two bytes of table => X
```

• To step through table, need to increment pointer after use

```
        ldaa 0,x
        inx
or
        ldaa 1,x+
```

table
0C
7A
D5
00
61
62
63
64

```
        org $2000
table:  dc.b 12,122,-43,0
        dc.b ‘a’,’b’,’c’,’d’
```

Which branch instruction should you use?

Branch if A > B
Is 0xFF > 0x00?

If unsigned, 0xFF = 255 and 0x00 = 0,
so 0xFF > 0x00

If signed, 0xFF = −1 and 0x00 = 0,
so 0xFF < 0x00

Using unsigned numbers: BHI (checks C bit of CCR)
Using signed numbers: BGT (checks V bit of CCR)

For unsigned numbers, use branch instructions which check C bit
For signed numbers, use branch instructions which check V bit

Hand Assembling a Program

To hand-assemble a program, do the following:

1. Start with the org statement, which shows where the first byte of the program will go into memory.
(E.g., org $2000 will put the first instruction at address $2000.)

2. Look at the first instruction. Determine the addressing mode used.
(E.g., ldab #10 uses IMM mode.)

3. Look up the instruction in the HCS12 Core Users Guide, find the appropriate Addressing Mode, and the Object Code for that addressing mode.
(E.g., ldab IMM has object code C6 ii.)
• Table 5.1 of the Core Users Guide has a concise summary of the instructions, addressing modes, op-codes, and cycles.

4. Put in the object code for the instruction, and put in the appropriate operand. Be careful to convert decimal operands to hex operands if necessary.
(E.g., ldab #10 becomes C6 0A.)

5. Add the number of bytes of this instruction to the address of the instruction to determine
the address of the next instruction.
(E.g., $2000 + 2 = $2002 will be the starting address of the next instruction.)

```
org $2000
ldab #10
loop: clra
dbne b,loop
swi
```

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
LBMI *rel16*	Long branch if minus If N=1, then (PC)+4+rel⇒PC	REL	18 2B qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBNE *rel16*	Long branch if not equal to 0 If Z=0, then (PC)+4+rel⇒PC	REL	18 26 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBPL *rel16*	Long branch if plus If N=0, then (PC)+4+rel⇒PC	REL	18 2A qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBRA *rel16*	Long branch always	REL	18 20 qq rr	OPPP	– – – – – – – –
LBRN *rel16*	Long branch never	REL	18 21 qq rr	OPO	– – – – – – – –
LBVC *rel16*	Long branch if V clear If V=0,then (PC)+4+rel⇒PC	REL	18 28 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBVS *rel16*	Long branch if V set If V=1,then (PC)+4+rel⇒PC	REL	18 29 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LDAA #*opr8i* LDAA *opr8a* LDAA *opr16a* LDAA *oprx0_xysppc* LDAA *oprx9,xysppc* LDAA *oprx16,xysppc* LDAA [D,*xysppc*] LDAA [*oprx16,xysppc*]	Load A (M)⇒A or imm⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	86 ii 96 dd B6 hh ll A6 xb A6 xb ff A6 xb ee ff A6 xb A6 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
LDAB #*opr8i* LDAB *opr8a* LDAB *opr16a* LDAB *oprx0_xysppc* LDAB *oprx9,xysppc* LDAB *oprx16,xysppc* LDAB [D,*xysppc*] LDAB [*oprx16,xysppc*]	Load B (M)⇒B or imm⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C6 ii D6 dd F6 hh ll E6 xb E6 xb ff E6 xb ee ff E6 xb E6 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
LDD #*opr16i* LDD *opr8a* LDD *opr16a* LDD *oprx0_xysppc* LDD *oprx9,xysppc* LDD *oprx16,xysppc* LDD [D,*xysppc*] LDD [*oprx16,xysppc*]	Load D (M:M+1)⇒A:B or imm⇒A:B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CC jj kk DC dd FC hh ll EC xb EC xb ff EC xb ee ff EC xb EC xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPf fIPRPf	– – – – Δ Δ 0 –
LDS #*opr16i* LDS *opr8a* LDS *opr16a* LDS *oprx0_xysppc* LDS *oprx9,xysppc* LDS *oprx16,xysppc* LDS [D,*xysppc*] LDS [*oprx16,xysppc*]	Load SP (M:M+1)⇒SP or imm⇒SP	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CF jj kk DF dd FF hh ll EF xb EF xb ff EF xb ee ff EF xb EF xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPf fIPRPf	– – – – Δ Δ 0 –
LDX #*opr16i* LDX *opr8a* LDX *opr16a* LDX *oprx0_xysppc* LDX *oprx9,xysppc* LDX *oprx16,xysppc* LDX [D,*xysppc*] LDX [*oprx16,xysppc*]	Load X (M:M+1)⇒X or imm⇒X	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CE jj kk DE dd FE hh ll EE xb EE xb ff EE xb ee ff EE xb EE xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPf fIPRPf	– – – – Δ Δ 0 –
LDY #*opr16i* LDY *opr8a* LDY *opr16a* LDY *oprx0_xysppc* LDY *oprx9,xysppc* LDY *oprx16,xysppc* LDY [D,*xysppc*] LDY [*oprx16,xysppc*]	Load Y (M:M+1)⇒Y or imm⇒Y	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CD jj kk DD dd FD hh ll ED xb ED xb ff ED xb ee ff ED xb ED xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPf fIPRPf	– – – – Δ Δ 0 –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
BRCLR *opr8a, msk8, rel8* BRCLR *opr16a, msk8, rel8* BRCLR *oprx0_xysppc, msk8, rel8* BRCLR *oprx9,xysppc, msk8, rel8* BRCLR *oprx16,xysppc, msk8, rel8*	Branch if bit(s) clear; if (M)•(mask byte)=0, then (PC)+2+rel⇒PC	DIR EXT IDX IDX1 IDX2	4F dd mm rr 1F hh ll mm rr 0F xb mm rr 0F xb ff mm rr 0F xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	– – – – – – – –
BRN *rel8*	Branch never	REL	21 rr	P	– – – – – – – –
BRSET *opr8, msk8, rel8* BRSET *opr16a, msk8, rel8* BRSET *oprx0_xysppc, msk8, rel8* BRSET *oprx9,xysppc, msk8, rel8* BRSET *oprx16,xysppc, msk8, rel8*	Branch if bit(s) set; if ($\overline{M}$)•(mask byte)=0, then (PC)+2+rel⇒PC	DIR EXT IDX IDX1 IDX2	4E dd mm rr 1E hh ll mm rr 0E xb mm rr 0E xb ff mm rr 0E xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	– – – – – – – –
BSET *opr8, msk8* BSET *opr16a, msk8* BSET *oprx0_xysppc, msk8* BSET *oprx9,xysppc, msk8* BSET *oprx16,xysppc, msk8*	Set bit(s) in M (M) | mask byte⇒M	DIR EXT IDX IDX1 IDX2	4C dd mm 1C hh ll mm 0C xb mm 0C xb ff mm 0C xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	– – – – Δ Δ 0 –
BSR *rel8*	Branch to subroutine; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (PC)+2+rel⇒PC	REL	07 rr	SPPP	– – – – – – – –
BVC *rel8*	Branch if V clear; if V=0, then (PC)+2+rel⇒PC	REL	28 rr	PPP (branch) P (no branch)	– – – – – – – –
BVS *rel8*	Branch if V set; if V=1, then (PC)+2+rel⇒PC	REL	29 rr	PPP (branch) P (no branch)	– – – – – – – –
CALL *opr16a, page* CALL *oprx0_xysppc, page* CALL *oprx9,xysppc, page* CALL *oprx16,xysppc, page* CALL [D,*xysppc*] CALL [*oprx16, xysppc*]	Call subroutine in expanded memory (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (PPG)⇒M_{SP} pg⇒PPAGE register subroutine address⇒PC	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	4A hh ll pg 4B xb pg 4B xb ff pg 4B xb ee ff pg 4B xb 4B xb ee ff	gnSsPPP gnSsPPP gnSsPPP fgnSsPPP fIignSsPPP fIignSsPPP	– – – – – – – –
CBA	Compare A to B; (A)–(B)	INH	18 17	OO	– – – – Δ Δ Δ Δ
CLCSame as ANDCC #$FE	Clear C bit	IMM	10 FE	P	– – – – – – – 0
CLISame as ANDCC #$EF	Clear I bit	IMM	10 EF	P	– – – 0 – – – –
CLR *opr16a* CLR *oprx0_xysppc* CLR *oprx9,xysppc* CLR *oprx16,xysppc* CLR [D,*xysppc*] CLR [*oprx16,xysppc*] CLRA CLRB	Clear M; $00⇒M Clear A; $00⇒A Clear B; $00⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	79 hh ll 69 xb 69 xb ff 69 xb ee ff 69 xb 69 xb ee ff 87 C7	PwO Pw PwO PwP PIfw PIPw O O	– – – – 0 1 0 0
CLVSame as ANDCC #$FD	Clear V	IMM	10 FD	P	– – – – – – 0 –
CMPA #*opr8i* CMPA *opr8a* CMPA *opr16a* CMPA *oprx0_xysppc* CMPA *oprx9,xysppc* CMPA *oprx16,xysppc* CMPA [D,*xysppc*] CMPA [*oprx16,xysppc*]	Compare A (A)–(M) or (A)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	81 ii 91 dd B1 hh ll A1 xb A1 xb ff A1 xb ee ff A1 xb A1 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ Δ Δ
CMPB #*opr8i* CMPB *opr8a* CMPB *opr16a* CMPB *oprx0_xysppc* CMPB *oprx9,xysppc* CMPB *oprx16,xysppc* CMPB [D,*xysppc*] CMPB [*oprx16,xysppc*]	Compare B (B)–(M) or (B)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C1 ii D1 dd F1 hh ll E1 xb E1 xb ff E1 xb ee ff E1 xb E1 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ Δ Δ

DBNE Decrement and Branch if Not Equal to Zero DBNE

Operation (counter) – 1 ⇒ counter
If (counter) not = 0, then (PC) + $0003 + rel ⇒ PC

Subtracts one from the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter register does not reach zero. Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DBNE *abdxysp, rel9*	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)

Loop Primitive Postbyte (lb) Coding				
Source Form	Postbyte[1]	Object Code	Counter Register	Offset
DBNE A, *rel9*	0010 X000	04 20 rr	A	Positive
DBNE B, *rel9*	0010 X001	04 21 rr	B	
DBNE D, *rel9*	0010 X100	04 24 rr	D	
DBNE X, *rel9*	0010 X101	04 25 rr	X	
DBNE Y, *rel9*	0010 X110	04 26 rr	Y	
DBNE SP, *rel9*	0010 X111	04 27 rr	SP	
DBNE A, *rel9*	0011 X000	04 30 rr	A	Negative
DBNE B, *rel9*	0011 X001	04 31 rr	B	
DBNE D, *rel9*	0011 X100	04 34 rr	D	
DBNE X, *rel9*	0011 X101	04 35 rr	X	
DBNE Y, *rel9*	0011 X110	04 36 rr	Y	
DBNE SP, *rel9*	0011 X111	04 37 rr	SP	

NOTES:
1. Bits 7:6:5 select DBEQ or DBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
STY *opr8a*	Store Y	DIR	5D dd	PW	– – – – Δ Δ 0 –
STY *opr16a*	$(Y_H:Y_L)\Rightarrow M:M+1$	EXT	7D hh ll	PWO	
STY *oprx0_xysppc*		IDX	6D xb	PW	
STY *oprx9,xysppc*		IDX1	6D xb ff	PWO	
STY *oprx16,xysppc*		IDX2	6D xb ee ff	PWP	
STY [D,*xysppc*]		[D,IDX]	6D xb	PIfW	
STY [*oprx16,xysppc*]		[IDX2]	6D xb ee ff	PIPW	
SUBA #*opr8i*	Subtract from A	IMM	80 ii	P	– – – – Δ Δ Δ Δ
SUBA *opr8a*	$(A)-(M)\Rightarrow A$	DIR	90 dd	rPf	
SUBA *opr16a*	or $(A)-imm\Rightarrow A$	EXT	B0 hh ll	rPO	
SUBA *oprx0_xysppc*		IDX	A0 xb	rPf	
SUBA *oprx9,xysppc*		IDX1	A0 xb ff	rPO	
SUBA *oprx16,xysppc*		IDX2	A0 xb ee ff	frPP	
SUBA [D,*xysppc*]		[D,IDX]	A0 xb	fIfrPf	
SUBA [*oprx16,xysppc*]		[IDX2]	A0 xb ee ff	fIPrPf	
SUBB #*opr8i*	Subtract from B	IMM	C0 ii	P	– – – – Δ Δ Δ Δ
SUBB *opr8a*	$(B)-(M)\Rightarrow B$	DIR	D0 dd	rPf	
SUBB *opr16a*	or $(B)-imm\Rightarrow B$	EXT	F0 hh ll	rPO	
SUBB *oprx0_xysppc*		IDX	E0 xb	rPf	
SUBB *oprx9,xysppc*		IDX1	E0 xb ff	rPO	
SUBB *oprx16,xysppc*		IDX2	E0 xb ee ff	frPP	
SUBB [D,*xysppc*]		[D,IDX]	E0 xb	fIfrPf	
SUBB [*oprx16,xysppc*]		[IDX2]	E0 xb ee ff	fIPrPf	
SUBD #*opr16i*	Subtract from D	IMM	83 jj kk	PO	– – – – Δ Δ Δ Δ
SUBD *opr8a*	$(A:B)-(M:M+1)\Rightarrow A:B$	DIR	93 dd	RPf	
SUBD *opr16a*	or $(A:B)-imm\Rightarrow A:B$	EXT	B3 hh ll	RPO	
SUBD *oprx0_xysppc*		IDX	A3 xb	RPf	
SUBD *oprx9,xysppc*		IDX1	A3 xb ff	RPO	
SUBD *oprx16,xysppc*		IDX2	A3 xb ee ff	fRPP	
SUBD [D,*xysppc*]		[D,IDX]	A3 xb	fIfRPf	
SUBD [*oprx16,xysppc*]		[IDX2]	A3 xb ee ff	fIPRPf	
SWI	Software interrupt; $(SP)-2\Rightarrow SP$ $RTN_H:RTN_L\Rightarrow M_{SP}:M_{SP+1}$ $(SP)-2\Rightarrow SP$; $(Y_H:Y_L)\Rightarrow M_{SP}:M_{SP+1}$ $(SP)-2\Rightarrow SP$; $(X_H:X_L)\Rightarrow M_{SP}:M_{SP+1}$ $(SP)-2\Rightarrow SP$; $(B:A)\Rightarrow M_{SP}:M_{SP+1}$ $(SP)-1\Rightarrow SP$; $(CCR)\Rightarrow M_{SP}$; $1\Rightarrow I$ (SWI vector)⇒PC	INH	3F	VSPSSPSsP*	– – – 1 – – – –
*The CPU also uses VfPPP for hardware interrupts and unimplemented opcode traps.					
TAB	Transfer A to B; $(A)\Rightarrow B$	INH	18 0E	OO	– – – – Δ Δ 0 –
TAP	Transfer A to CCR; $(A)\Rightarrow CCR$ Assembled as TFR A, CCR	INH	B7 02	P	Δ ⇓ Δ Δ Δ Δ Δ Δ
TBA	Transfer B to A; $(B)\Rightarrow A$	INH	18 0F	OO	– – – – Δ Δ 0 –
TBEQ *abdxysp,rel9*	Test and branch if equal to 0 If (counter)=0, then (PC)+2+rel⇒PC	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
TBL *oprx0_xysppc*	Table lookup and interpolate, 8-bit $(M)+[(B)\times((M+1)-(M))]\Rightarrow A$	IDX	18 3D xb	ORfffP	– – – – Δ Δ – Δ
TBNE *abdxysp,rel9*	Test and branch if not equal to 0 If (counter)≠0, then (PC)+2+rel⇒PC	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
TFR *abcdxysp,abcdxysp*	Transfer from register to register $(r1)\Rightarrow r2$ r1 and r2 same size $00:(r1)\Rightarrow r2$ r1=8-bit; r2=16-bit $(r1_L)\Rightarrow r2$ r1=16-bit; r2=8-bit	INH	B7 eb	P	– – – – – – – – or Δ ⇓ Δ Δ Δ Δ Δ Δ
TPA Same as TFR CCR ,A	Transfer CCR to A; $(CCR)\Rightarrow A$	INH	B7 20	P	– – – – – – – –

68HC12 Cycles

• 68HC12 works on 48 MHz clock
• A processor cycle takes 2 clock cycles – P clock is 24 MHz
• Each processor cycle takes 41.7 ns (1/24 μs) to execute
• An instruction takes from 1 to 12 processor cycles to execute
• You can determine how many cycles an instruction takes by looking up the CPU cycles for that instruction in the Core Users Guide.
– For example, LDAA using the IMM addressing mode shows one CPU cycle (of type P).
– LDAA using the EXT addressing mode shows three CPU cycles (of type rPf).
– Section A.27 of the Core Users Guide explains what the HCS12 is doing during each of the different types of CPU cycles.

```
2000                    org $2000     ; Inst        Mode  Cycles
2000  c6 0a             ldab #10      ; LDAB        (IMM) 1
2002  87          loop: clra          ; CLRA        (INH) 1
2003  04 31 fc          dbne b,loop   ; DBNE        (REL) 3
2006  3f                swi           ; SWI               9
```

The program executes the ldab #10 instruction once (which takes one cycle). It then goes through loop 10 times (which has two instructions, on with one cycle and one with three cycles), and finishes with the swi instruction (which takes 9 cycles).
Total number of cycles:

$1 + 10 \times (1 + 3) + 9 = 50$

50 cycles = 50×41.7 ns/cycle = 2.08 μs

LDAB Load B LDAB

Operation (M) ⇒ B
or
imm ⇒ B

Loads B with either the value in M or an immediate value.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
V: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDAB #*opr8i*	IMM	C6 ii	P
LDAB *opr8a*	DIR	D6 dd	rPf
LDAB *opr16a*	EXT	F6 hh ll	rPO
LDAB *oprx0_xysppc*	IDX	E6 xb	rPf
LDAB *oprx9,xysppc*	IDX1	E6 xb ff	rPO
LDAB *oprx16,xysppc*	IDX2	E6 xb ee ff	frPP
LDAB [D,*xysppc*]	[D,IDX]	E6 xb	fIfrPf
LDAB [*oprx16,xysppc*]	[IDX2]	E6 xb ee ff	fIPrPf

HC12 Assembly Language Programming

Programming Model
Addressing Modes
Assembler Directives
HC12 Instructions
Flow Charts

Assembler Directives

• In order to write an assembly language program it is necessary to use assembler directives.
• These are not instructions which the HC12 executes but are directives to the assembler program about such things as where to put code and data into memory.
• All of the assembler directives can be found in as12.html on the EE 308 home page.
• We will use only a few of these directives. (Note: In the following table, [] means an optional argument.) Here are the ones we will need:

Directive Name	Description	Example
equ	Give a value to a symbol	len: equ 100
org	Set starting value of location counter where code or data will go	org $1000
dc[.size]	Allocate and initialize storage for variables. Size can be b (byte) or w (two bytes) If no size is specified, b is used	var: dc.b 2,18
ds[.size]	Allocate specified number of storage spaces. size is the same as for dc directive	table: ds.w 10
fcc	Encodes a string of ASCII characters. The first character is the delimiter. The string terminates at the next occurrence of the delimiter	table: fcc "Hello"

Using labels in assembly programs

A label is defined by a name followed by a colon as the first thing on a line. When the label is referred to in the program, it has the numerical value of the location counter when the label was defined.
Here is a code fragment using labels and the assembler directives dc and ds:

```
        org     $2000
table1: dc.b    $23,$17,$f2,$a3,$56
table2: ds.b    5
var:    dc.w    $43af
```

The as12 assembler produces a listing file (.lst) and a symbol file (.sym). Here is the listing file from the assembler:

```
as12, an absolute assembler for Motorola MCU's, version 1.2e

2000                                org   $2000
2000 23 17 f2 a3 56     table1:     dc.b  $23,$17,$f2,$a3,$56
2005                    table2:     ds.b  5
200a 43 af              var:        dc.w  $43af

Executed: Sat Jan 06 13:19:23 2007
Total cycles: 0, Total bytes: 7
Total errors: 0, Total warnings: 0
```

Note that table1 is a name with the value of $2000, the value of the location counter defined in the org directive. Five bytes of data are defined by the dc.b directive, so the location counter is increased from $2000 to $2005. table2 is a name with the value of $2005. Five bytes of data are set aside for table2 by the ds.b 5 directive. The as12 assembler initialized these five bytes of data to all zeros. var is a name with the value of $200a, the first location after table2.