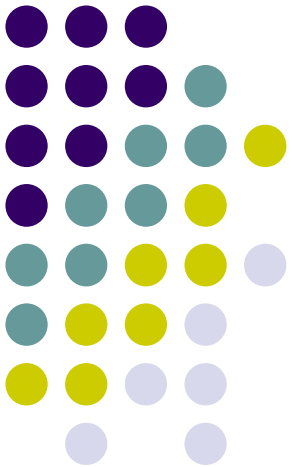
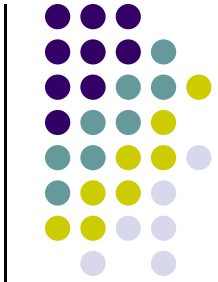
HCS12 Instruction Set

Chapter 7

Part 1

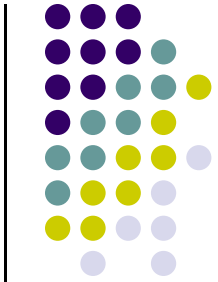


M68HCS12 Instruction Set



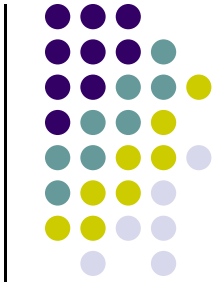
- Categories
  - Load and Store
  - Stack
  - Transfer
  - Decrement and Increment
  - Clear and Set
  - Shift and Rotate
  - Arithmetic Instructions

Arithmetic Instructions



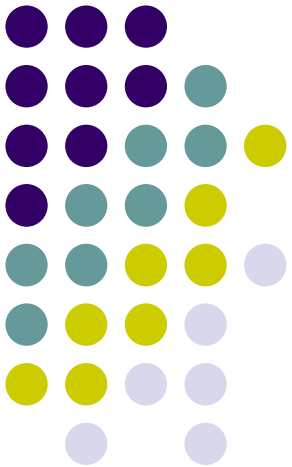
- Add and Subtract
- Decimal Arithmetic
- Negating Instruction
- Multiplication
- Division
- Logic
- Data Test
- Conditional Branch

Other Instructions



- Unconditional Jump and Branch
- Subroutines
- Interrupt
- Miscellaneous

HCS12 Instruction Set



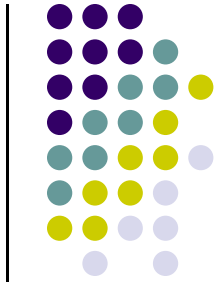


TABLE 7-2 HCS12 Instruction Set[a]

Mnemonic	Operation	Mnemonic	Operation
Load Registers (see Section 7.5)			
LDAA	(M) → A	LDAB	(M) → B
LDD	(M:M + 1) → D	LDS	(M:M + 1) → SP
LDX	(M:M + 1) → X	LDY	(M:M + 1) → Y
LEAS	EA → SP	LEAX	EA → X
LEAY	EA → Y		
PULA	(SP) → A	PULB	(SP) → B
PULD	(SP:SP + 1) → D	PULC	(SP) → CCR
PULX	(SP:SP + 1) → X	PULY	(SP:SP + 1) → Y
Store Registers (see Section 7.5)			
STAA	A → (M)	STAB	B → (M)
STD	D → (M:M + 1)	STS	SP → (M:M + 1)
STX	X → (M:M + 1)	STY	Y → (M:M + 1)
PSHA	A → (SP)	PSHB	B → (SP)
PSHD	D → (SP:SP + 1)	PSHC	CCR → (SP)
PSHY	Y → (SP:SP + 1)	PSHX	X → (SP:SP + 1)
Transfer/Exchange Registers (see Section 7.6)			
TFR	Any Reg → Any Reg	EXG	Any Reg ← → Any Reg
Move Memory Contents (see Section 7.7)			
MOVB	(M1) → (M2)	MOVW	(M1:M1 + 1) → (M2:M2 + 1)

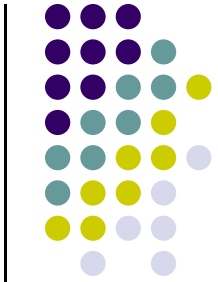


TABLE 7-2 Continued

Mnemonic	Operation	Mnemonic	Operation
Decrement/Increment (see Section 7.8)			
DEC	(M) − 1 → (M)	DECA	A − 1 → A
DECB	B − 1 → B	DES	SP − 1 → SP
DEX	X − 1 → X	DEY	Y − 1 → Y
INC	(M) + 1 → (M)	INCA	A + 1 → A
INCB	B + 1 → B	INS	SP + 1 → SP
INX	X + 1 → X	INY	Y + 1 → Y
Clear/Set (see Section 7.9)			
CLR	0 → (M)	CLRA	0 → A
CLRB	0 → B		
BCLR	0 → (M bits)	BSET	1 → (M bits)
Arithmetic (see Section 7.11)			
ABA	A + B → A	ABX	B + X → X (see LEAX)
ABY	B + Y → Y (see LEAY)	ADDA	A + (M) → A
ADDB	B + (M) → B	ADDD	D + (M:M + 1) → D
ADCA	A + (M) + C → A	ADCB	B + (M) + C → B
DAA	Decimal adjust		
SUBA	A − (M) → A	SBA	A − B → A
SUBD	D − (M:M + 1) → D	SUBB	B − (M) → B
SBCB	B − (M) − C → B	SBCA	A − (M) − C → A
NEG	Two's complement (M)	NEGA	Two's complement → A
NEGB	Two's complement B	SEX	Sign extend A,B,CCR
MUL	Unsigned A * B → D	EMUL	Unsigned D*Y → Y:D
EMULS	Signed D * Y → Y : D		
IDIV	Unsigned D/X → X,D	EDIV	Unsigned Y:D/X → Y,D
EDIVS	Signed Y : D/X → Y,D	IDIVS	Signed D/X → X,D
FDIV	Fractional D/X → X,D		
Logic (see Section 7.12)			
ANDA	A • (M) → A	ANDB	B • (M) → B
ANDCC	CCR • (M) → CCR		
EORB	B EOR (M) → B	EORA	A EOR (M) → A
ORAB	B OR (M) → B	ORAA	A OR (M) → A
ORCC	CCR OR (M) → CCR		
COM	Ones' complement (M)	COMA	Ones' complement A
COMB	Ones' complement B		
Rotates and Shifts (see Section 7.10)			
ROL	Rotate Left (M)	ROLA	Rotate Left A
ROLB	Rotate Left B	ROR	Rotate Right (M)
RORA	Rotate Right A	RORB	Rotate Right B
ASL	Arith Shift Left (M)	ASLA	Arith Shift Left A
ASLB	Arith Shift Left B	ASLD	Arith Shift Left D
ASR	Arith Shift Right (M)	ASRA	Arith Shift Right A
ASRB	Arith Shift Right B		
LSLA	Logic Shift Left A	LSL	Logic Shift Left (M)
LSLD	Logic Shift Left D	LSLB	Logic Shift Left B
LSRA	Logic Shift Right A	LSR	Logic Shift Right (M)
LSRD	Logic Shift Right D	LSRB	Logic Shift Right B
Data Test (see Section 7.13)			
BITA	Test bits in A	BITB	Test bits in B
CBA	A − B	CMPA	A − (M)
CMPB	B − (M)	CPD	D − (M : M + 1)

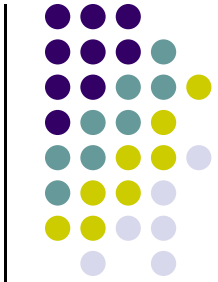


TABLE 7-2 Continued

Mnemonic	Operation	Mnemonic	Operation
CPX	X − (M : M + 1)	CPY	Y − (M : M + 1)
CPS	SP − (M : M + 1)		
TST	Test (M) = 0 or negative	TSTA	Test A = 0 or negative
TSTB	Test B = 0 or negative		
Fuzzy Logic and Specialized Math (see Chapter 19)			
MEM	Membership function	REV	MIN-MAX Rule Evaluation
REVW	Weighted rule evaluation	WAV	Weighted average
EMINM	MIN(D, (M : M + 1)) → (M : M + 1)	EMIND	MIN(D, (M : M + 1)) → D
MINM	MIN(A, (M)) → (M)	MINA	MIN (A, (M)) → A
EMAXM	MAX(D, (M : M + 1)) → (M : M + 1)	EMAXD	MAX(D, (M : M + 1)) → D
MAXM	MAX(A, (M)) → (M)	MAXA	MAX(A, (M)) → A
ETBL	16-bit table interpolate	EMACS	Multiply and accumulate
TBL	8-bit table interpolate		
Conditional Branch (see Section 7.14)			
BMI	Short branch minus	LBMI	Long branch minus
BPL	Short branch plus	LBPL	Long branch plus
BVS	Short branch two's complement overflow set	LBVS	Long branch two's-complement overflow set
BVC	Short branch two's-complement overflow clear	LBVC	Long branch two's-complement overflow clear
BLT	Short branch two's-complement less than	LBLT	Long branch two's complement less than
BGE	Short branch two's complement greater than or equal	LBGE	Long branch two's complement greater than or equal
BLE	Short branch two's complement less than or equal	LBLE	Long branch two's complement less than or equal
BGT	Short branch two's complement greater than	LBGT	Long branch two's complement greater than
BEQ	Short branch equal	LBEQ	Long branch equal
BNE	Short branch not equal	LBNE	Long branch not equal
BHI	Short branch higher	LBHI	Long branch higher
BLS	Short branch lower or same	LBLS	Long branch lower or same
BHS	Short branch higher or same	LBHS	Long branch higher or same
BLO	Short branch lower	LBLO	Long branch lower
BCC	Short branch carry clear	LBCC	Long branch carry clear
BCS	Short branch carry set	LBCS	Long branch carry set
Loop Primitive (see Section 7.15)			
DBEQ	Decrement and branch = 0	DBNE	Decrement and branch <> 0
IBEQ	Increment and branch = 0	IBNE	Increment and branch <> 0
TBEQ	Test and branch = 0	TBNE	Test and branch <> 0
Jump and Branch (see Section 7.16)			
JMP	Jump to address		
JSR	Jump to subroutine	Call	Call subroutine
RTS	Return to subroutine	RTC	Return from CALL
BSR	Branch to subroutine		
BRN	Short branch never	LBRN	Long branch never
BRA	Short branch always	LBRA	Long branch always
BRSET	Branch bits set		
BRCLR	Branch bits clear		

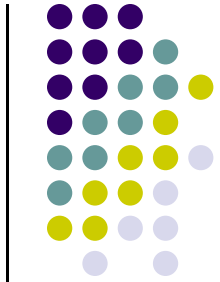
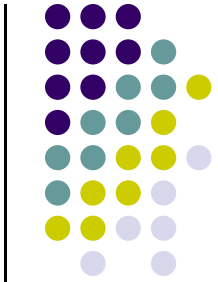


TABLE 7-2 Continued

Mnemonic	Operation	Mnemonic	Operation
Condition Code (see Section 7.17)			
ANDCC	Clear CCR Bits	ORCC	Set CCR Bits
Interrupt (see Chapter 12)			
CLI	Clear interrupt mask	SEI	Set interrupt mask
SWI	S/W interrupt	RTI	Return from interrupt
WAI	Wait for interrupt	TRAP	S/W interrupt
Miscellaneous (see Section 7.20)			
NOP	No operation	STOP	Stop clocks
BGND	Background debug mode		

[a] (M) indicates the instruction addresses memory using immediate, direct, extended, or index addressing. Register Name (A, B, D, X, Y, SP, PC) indicates the contents of that register. (SP) means on the stack. C denotes the contents of carry flag. CCR denotes the contents of the condition code register. EA means Effective Address.

Load Instructions



- Mnemonic Operation
  - LDAA – Load Accumulator A
  - LDAB – Load Accumulator B
  - LDD - Load Accumulator D
  - LDS - Load Stack Pointer
  - LDX - Load Index X Register
  - LDY - Load Index Y Register
- Addressing modes: All except INH
- Condition codes: N,Z and V=0

Addressing Modes

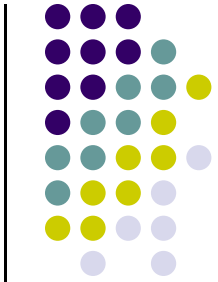
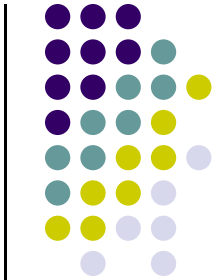


TABLE 7-5 Load Register Instructions

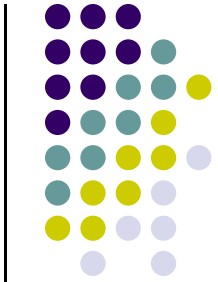
Function	Opcode	Symbolic Operation	Addressing Modes[a]						Condition Codes[b]			
			IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Load Accumulator A	LDAA	(M) → A	x	x	x	x	x		↕	↕	0	–
Load Accumulator B	LDAB	(M) → B	x	x	x	x	x		↕	↕	0	–
Load Accumulator D	LDD	(M : M + 1) → D	x	x	x	x	x		↕	↕	0	–
Load Stack Pointer	LDS	(M : M + 1) → SP	x	x	x	x	x		↕	↕	0	–
Load Index Register X	LDX	(M : M + 1) → X	x	x	x	x	x		↕	↕	0	–
Load Index Register Y	LDY	(M : M + 1) → Y	x	x	x	x	x		↕	↕	0	–
Load SP Effective Address	LEAS	EA → SP				x			–	–	–	–
Load X Effective Address	LEAX	EA → X				x			–	–	–	–
Load Y Effective Address	LEAY	EA → Y				x			–	–	–	–
Pull A from Stack	PULA	(SP) → A						x	–	–	–	–
Pull B from Stack	PULB	(SP) → B						x	–	–	–	–
Pull CCR from Stack	PULC	(SP) → CR						x	↕	↕	↕	↕
Pull D from Stack	PULD	(SP : SP + 1) → D						x	–	–	–	–

Address Modes



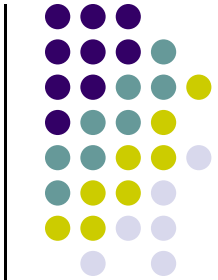
- Immediate Mode
  - LDAA #$02
- Direct Addressing Mode
  - LDAA $02
- Extended Addressing Mode
  - LDAA $1002
- Register Index
  - LDAA $02, X
- Register Indirect
  - LDAA [1,X]

Store Instructions



- Mnemonic Operation
  - STAA – Store Accumulator A
  - STAB – Store Accumulator B
  - STD - Store Accumulator D
  - STS - Store Stack Pointer
  - STX - Store Index X Register
  - STY - Store Index Y Register
- Addressing modes: All except IMM and INH
- Condition codes: N,Z and V=0

Address Modes



- Immediate Mode
  - STAA #$02 → (Illegal operation)
- Direct Addressing Mode
  - STAA $02
- Extended Addressing Mode
  - STAA $1002
- Register Index
  - STAA $02, X
- Register Indirect
  - STAA [1,X]

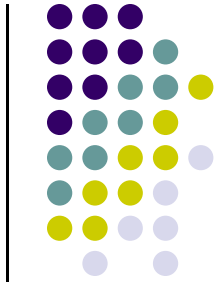
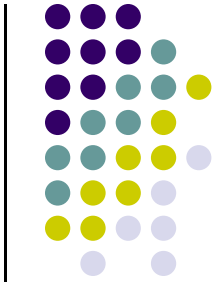


TABLE 7-6 Store Register Instructions

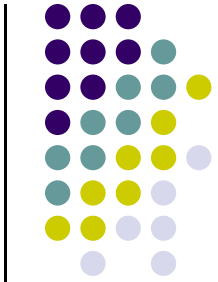
Function	Opcode	Symbolic Operation	Addressing Modes						Condition Codes			
			IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Store Accumulator A	STAA	A → (M)		x	x	x	x		↕	↕	0	–
Store Accumulator B	STAB	B → (M)		x	x	x	x		↕	↕	0	–
Store Accumulator D	STD	D → (M : M + 1)		x	x	x	x		↕	↕	0	–
Store Stack Pointer	STS	SP → (M : M + 1)		x	x	x	x		↕	↕	0	–
Store Index Register X	STX	X → (M : M + 1)		x	x	x	x		↕	↕	0	–
Store Index Register Y	STY	Y → (M : M + 1)		x	x	x	x		↕	↕	0	–
Push A to Stack	PSHA	A → (SP)						x	–	–	–	–
Push B to Stack	PSHB	B → (SP)						x	–	–	–	–
Push CCR to Stack	PSHC	CCR → (SP)						x	–	–	–	–
Push D to Stack	PSHD	D → (SP : SP + 1)						x	–	–	–	–
Push X to Stack	PSHX	X → (SP : SP + 1)						x	–	–	–	–
Push Y to Stack	PSHY	Y → (SP + SP + 1)						x	–	–	–	–

Notation: Memory Locations



- $C000: 12 34 56 78 9A BC DE F0
- This is read: memory location
- $C000=$12, $C001=$34, $C002=$56, $C003=$78, $C004=$9A, $C005=$BC, $C006=$DE, $C007=$F0

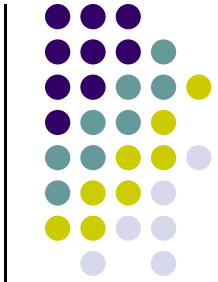
Notation: Memory Locations



- $C000: 12 34 56 78 9A BC DE F0

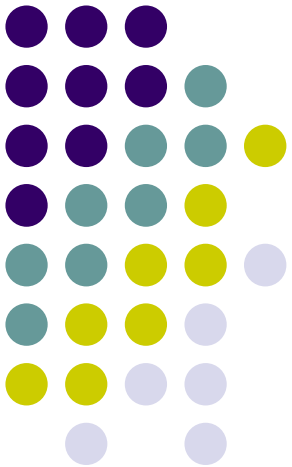
Address	Data
$C000	$12
$C001	$34
$C002	$56
$C003	$78
$C004	$9A
$C005	$BC
$C006	$DE
$C007	$F0

16-bit Load and Store

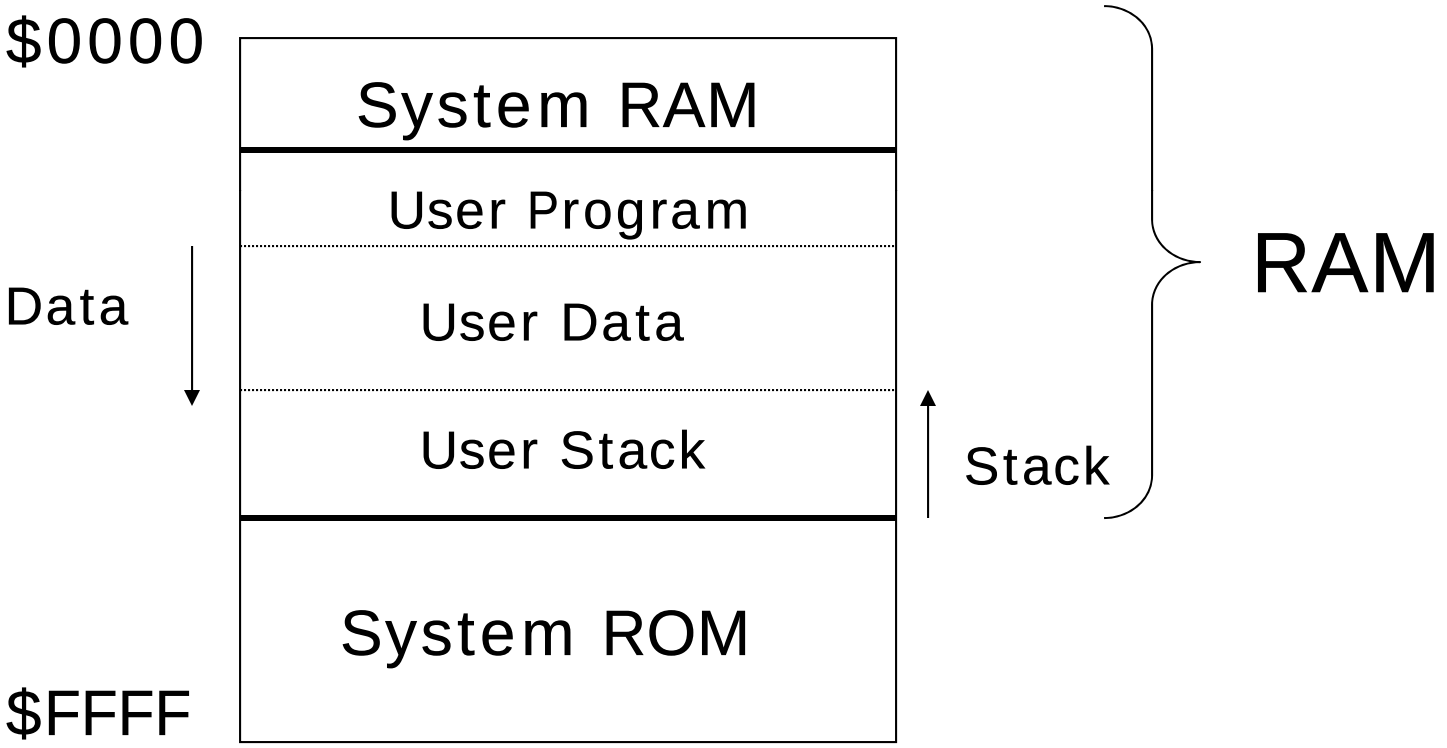


- For a 16-bit (2 byte) load and store, the high byte is stored at the lower address and the low byte is stored at the higher address.
- Ex: $C000: 12 34 56 78 9A BC DE
- LDAA $C001 A ← ($C001) = $34
- LDX $C002 X ← $5678
- STX $C004 ($C004) ← $5678
- $C000: 12 34 56 78 56 78

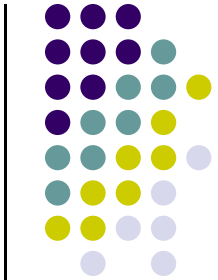
Stack Memory



Sample Memory Map

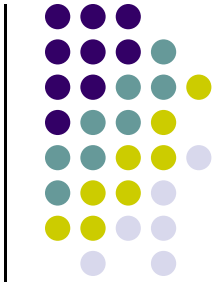


Stack Memory



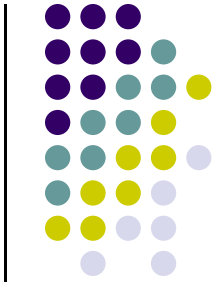
- The Stack is a LIFO-Last In First Out Buffer
  - Stack is stored in RAM
  - Stack Instructions
    - PSH – Push register data onto Stack
      - PSHA: Push Reg A on Stack
      - PSHB: Push Reg B on Stack
      - PSHD: Push Reg D on Stack
      - PSHX: Push Reg X on Stack
      - PSHY: Push Reg Y on Stack
    - PUL - Pull register data from Stack
      - PULA: Pull Reg A from Stack
      - PULB: Pull Reg B from Stack
      - PULD: Pull Reg D from Stack
      - PULX: Pull Reg X from Stack
      - PULY: Pull Reg y from Stack

Notation- Stack (Indirect Addressing)

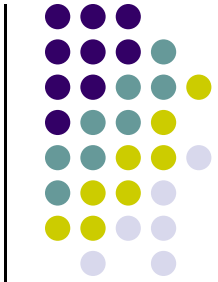


- Stack Pointer (SP) is a 16 bit address that points to “top” of the Stack
- Notation
  - **Sp =** the value or address stored in the stack pointer register
  - (Sp) = the contents of the memory location pointed to by Sp.

Stack Pointer



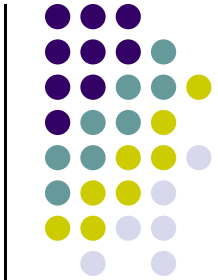
Stack Example (8-bit)



- Push (PSH?) Operation
  1. SP ← SP – 1
  2. (Sp) ← Reg (A or B)

- Ex: Let A=$23, B=$1D, SP=$2000, ($2000) = $AA
- Execute: PSHA
- SP=SP-1=$1FFF (SP) = ($1FFF)=$23,
- Execute: PSHB
- SP=SP-1=$1FFE, ($1FFE)=$1D,

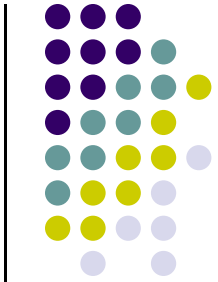
Stack Example (8-bit)



- Pull (PUL?) Operation
  1. Reg (A or B) ← (SP)
  2. SP ← SP + 1

- Ex: Let A=$23, B=$1D, SP=$1FFE,
- ($1FFE) = $AA, ($1FFF) = $55, ($2000) = $3F
- Execute: PULA
- A ← ($1FFE)=$AA, SP=SP+1=$1FFF,
- Execute: PULB
- We have: B ← ($1FFE)=$55, SP=SP+1=$2000,

Stack Example (16 bit)



- Push (PSH?) Operation
  1. SP ← SP – 1
  2. (SP) ← low byte of Reg (X,Y, or D)
  3. SP ← SP – 1
  4. (SP) ← high byte of Reg (X ,Y or D)

  Ex: Let X=$1234, Y=$FEDC, SP=$2000
- Execute: PSHX
- SP ← SP-1 = $1FFF
- (SP)=($1FFF) = $34 (low byte of X)
- SP ← SP – 1 = $1FFE
- (SP) = ($1FFE) = $12 (high byte of X)

Stack Example (16 bit)

Ex: Let X=$1234, Y=$FEDC, SP=$1FFE

- Execute: PSHY
- SP ← SP-1 = 1FFD
- (SP) = ($1FFD) = $DC (low byte of Y)
- SP ← SP – 1 = $1FFC
- (SP)=($1FFC) = $FE (high byte of Y)

Stack Example (16-bit)

- Pull (PUL?) Operation
  1. High Byte of Reg (X or Y) ← (SP)
  2. SP ← SP + 1
  3. Low byte of Reg (X or Y) ← (SP)
  4. SP ← SP + 1

- Ex: Let X=$1234, Y=$FEDC, SP=$1FFE,
- ($1FFE) = $AA, ($1FFF) = $55, ($2000) = $3F
- Execute: PULX
- (XH)←($1FFE)= $AA High byte of X (XH)
- SP=SP+1=$1FFF,
- (XL)←($1FFF) = $55 Low byte of X (XL)
- SP←SP+1=$2000,
- X ← $AA55

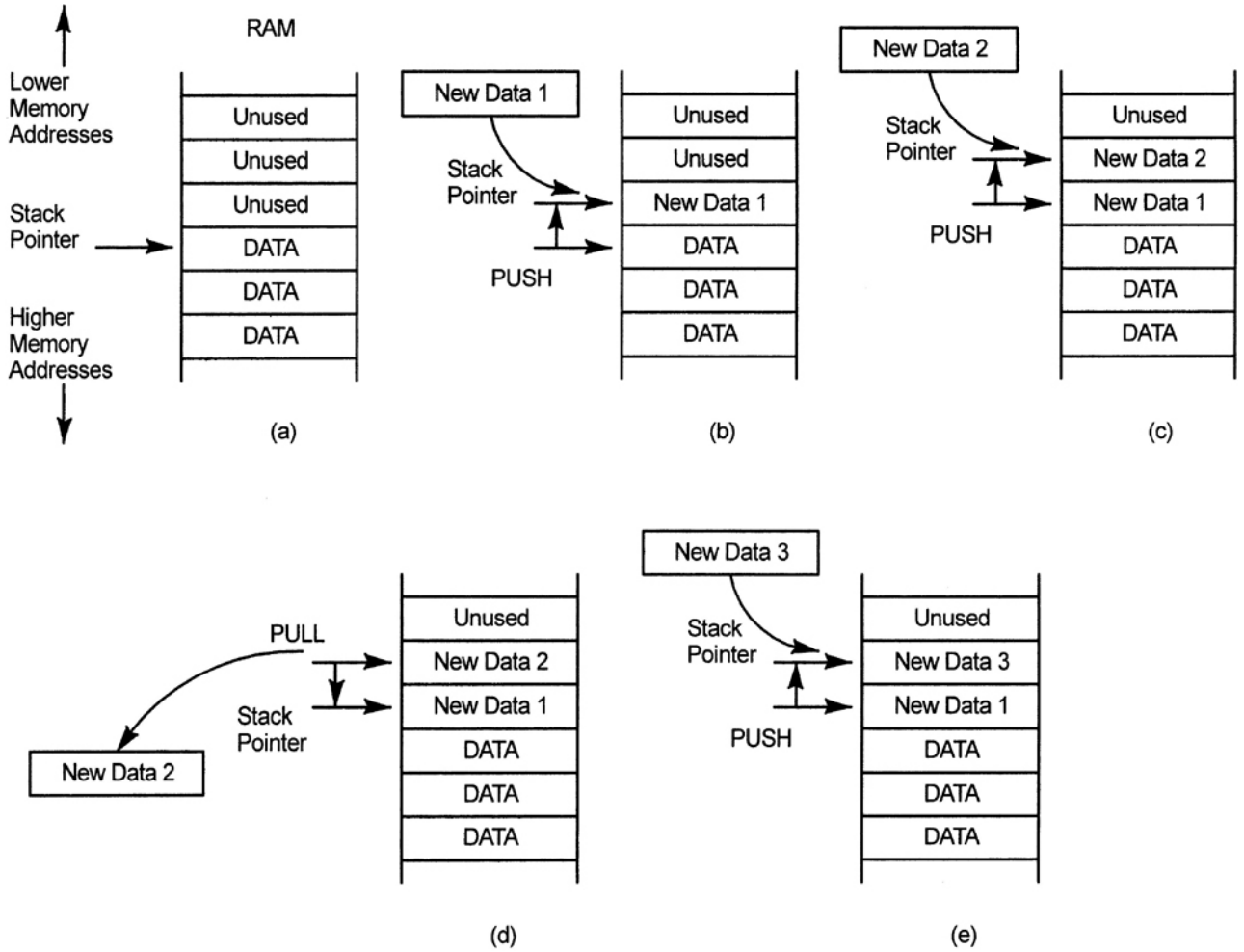
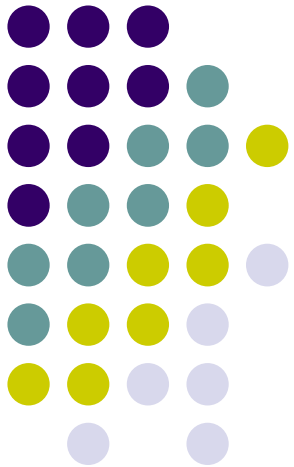
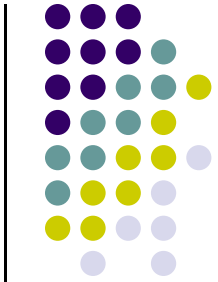


Figure 7-3 Stack operations. (a) Stack pointer before stack operations. (b) Stack pointer after a push. (c) Stack pointer after a second push. (d) Stack pointer after a pull. (e) Stack pointer after a third push.

Example Code

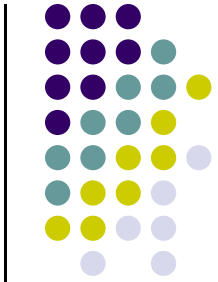


Example Code #1
What's wrong with this code?



```
Mysub:  PSHA        ;  Save registers on stack
        PSHB
        {subroutine code}
        PULA
        PULB
        RTS         ;   Return from subroutine
```

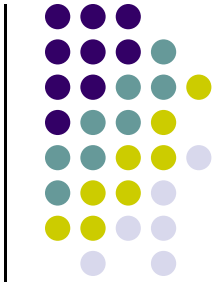
Example Code #1
What's wrong with this code?



```
Mysub:  PSHA        ;  Save registers on stack
        PSHB
        {subroutine code}
        PULA
        PULB
        RTS         ;   Return from subroutine
```

On exit from the subroutine, registers A and B have been exchanged
You need PUL in the reverse order of a PSH

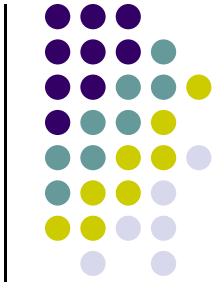
Example Code -Corrected
What's wrong with this code?



```
Mysub:  PSHA        ;  Save registers on stack
        PSHB
        {subroutine code}
        PULB
        PULA
        RTS         ;   Return from subroutine
```

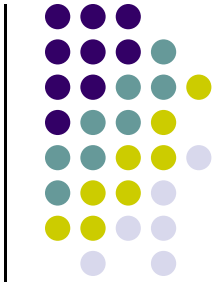
On exit from the subroutine, registers A and B have been exchanged
You need PUL in the reverse order of a PSH

Example Code #2
What's wrong with this code?



```
        LDX  #Data
        LDAA  #MAX
Loop:   LDAA 0,X
        BEQ Spin
        PSHA
        ANDA #$3F
        STAA 1,X+
        DBNE  A, Loop
Done:   BRA Spin
```

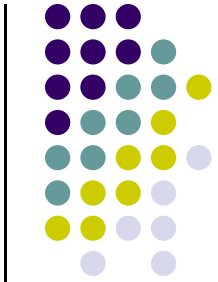
Example Code #2
What's wrong with this code?



```
        LDX  #Data
        LDAA  #MAX
Loop:   LDAA 0,X
        BEQ Spin
        PSHA
        ANDA #$3F
        STAA 1,X+
        DBNE  A, Loop
Done:   BRA Spin
```

Unbalanced PSH and PUL instructions. We PSHA within the Loop but never PULA . You must balance PSH and PUL instructions or you will have a **STACK OVERFLOW.**

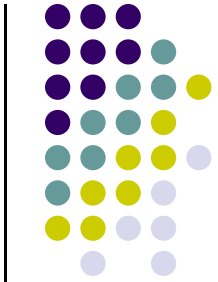
Example Code #2 -Corrected
What's wrong with this code?



```
        LDX  #Data
        LDAA  #MAX
Loop:   LDAA 0,X
        BEQ Spin
        PSHA
        ANDA #$3F
        STAA 1,X+
        PULA
        DBNE  A, Loop
Done:   BRA Spin
```

Unbalanced PSH and PUL instructions. We PSHA within the Loop but never PULA again. You must balance PSH and PUL instructions or you will have a **STACK OVERFLOW.**

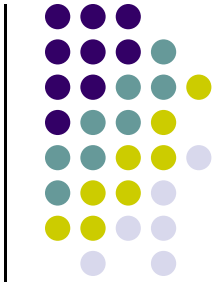
Example Code #3
What data are in the A,B, X, and Y registers after this code fragment executes?



```
LDS  #$0500
LDX  #$1234
LDY  #$4567
PSHX
PSHY
PULB
PULA
PULX
```

A=? B=? X=? Y=?

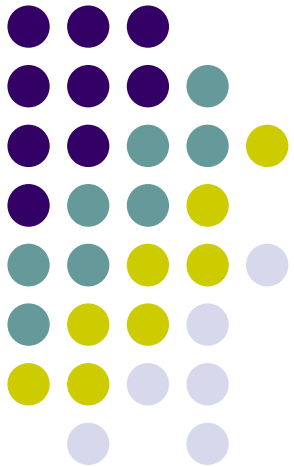
Example Code #3 - Solution
What data are in the A,B, X, and Y registers after this code fragment executes?



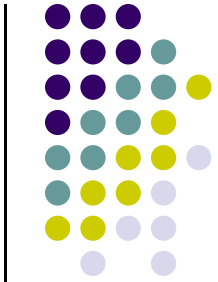
```
LDS  #$0500      ;  Sp ← $0500
LDX  #$1234      ;   X ← $1234
LDY  #$4567      ;   Y ← $4567
PSHX             ;   ($04FE:$04FF)←$1234:SP←SP-2 = $04FE
PSHY             ;   ($04FC:$04FD) ←$4567 :SP←SP-2 = $04FC
PULB             ;   B←($04FC) = $45 : SP←SP+1 = $04FD
PULA             ;   A←($04FD) = $67  : SP←SP+1 =$04FE
PULX             ;    X ←($04FE:$04FF) = $1234
```

A=$45 B=$67 X=$1234 Y=$4567

Transfer Instructions



Transfer Register Instructions



- Mnemonic Operation
  - TAB : Transfer A to B: B ← (A)
  - TBA : Transfer B to A: A ← (B)
  - TAP : Transfer A to CCR: (CCR) ← A
  - TPA : Transfer CCR to A: (A) ← CCR
  - TFR : Transfer Reg1 to Reg2: (Reg2) ← Reg1
  - EXG : Exchange Reg1 and Reg2: Reg1 ←→ Reg2
- Addressing modes: INH Only
- Condition codes: Check Table

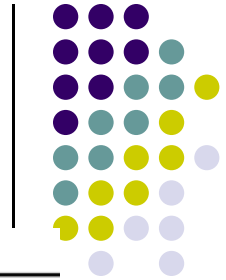


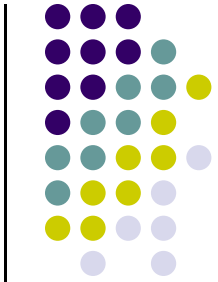
TABLE 7-7 Transfer Register Instructions

Function	Opcode	Symbolic Operation	Addressing Modes: IMM	IDIR	IEXT	IIDX	IIDR	IINH	Condition Codes: N	Z	V	C
Transfer A to B	TAB	A → B						x	↕	↕	0	–
Transfer A to CCR	TAP	A → CCR						x	↕	↕	↕	↕
Transfer B to A	TBA	B → A						x	↕	↕	0	–
Transfer Registers	TFR Reg,Reg	Reg → Reg						x	–	–	–	–
Transfer Reg to CCR	TFR Reg,CCR	Reg → CCR						x	↕	↕	↕	↕
Transfer CCR to A	TPA	CCR → A						x	–	–	–	–
Exchange Registers	EXG Reg,Reg	Reg ↔ Reg						x	–	–	–	–

TABLE 7-23 The 8-bit and 16-bit Transfer and Exchange Instructions

Transfer	8-bit → 16-bit	The 8-bit source is transferred to the low byte of the destination; the sign of the source is extended into the high byte of the destination; see the `SEX` instruction.
	16-bit → 8-bit	The low byte of the 16-bit source is transferred to the 8-bit destination.
Exchange	8-bit ↔ 16-bit	The low bytes of the registers are exchanged and the high byte of the 16-bit register is set to $00.

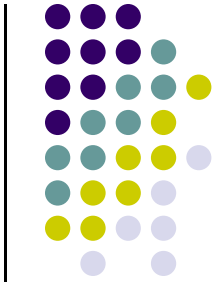
Transfer Examples



Example 7-12 Transfer Register Instructions

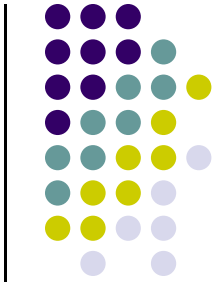
```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

 Rel. Loc    Obj. code Source line
 ---- ------ --------- -----------

    4
    5 000000 180E            tab        ; A -> B
    6 000002 180F            tba        ; B -> A
    7 000004 B701            tfr   a,b  ; A -> B
    8 000006 B710            tfr   b,a  ; B -> A
    9 000008 B781            exg   a,b  ; A <-> B
   10 00000A B750            tfr   x,a  ; Low byte X -> A
   11 00000C B705            tfr   a,x  ; A sign extended -> X
   12 00000E B796            exg   b,y  ; B <-> Low Byte Y, High
                                        Byte Y = $00
```

Load Effective Address



- LEAS: Load Effective Address into SP: SP←EA
- LEAX: Load Effective Address into X: X←EA
- LEAY: Load Effective Address into Y: Y←EA
- Addressing modes: Register Index only
- Condition Codes: none changed

Load Effective Address Example

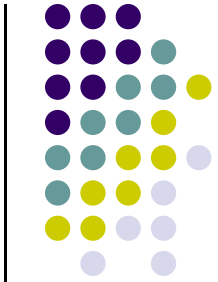


Example 7-11 Load Effective Address Instructions

Assume X = $1234, Y = $1000, and SP = $0A00. Give the contents of each affected register after the following instructions are executed:

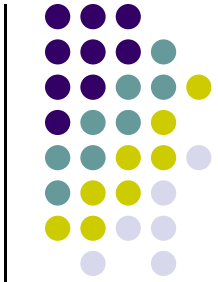
Instruction	Result
`leax 10,X`	X = X + 10_{10} = $1234 + $000A = $123E
`leax $10,Y`	X = Y + $10 = $1000 + $0010 = $1010
`leas -10,SP`	SP = SP - 10_{10} = $0A00 − $000A = $09F6

Move Byte or Word in Memory



- MOVB: Move Byte: (M2) ← (M1)
- MOVW:Move Word:(M2:M2+1)←(M1:M1+1)
- Addressing modes: Ext and Register Index
- Condition Codes: none changed

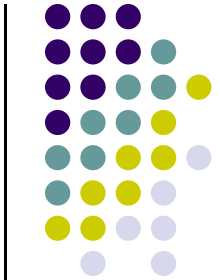
MOVB and MOVW Examples



Example 7-13 MOVB and MOVW Instructions

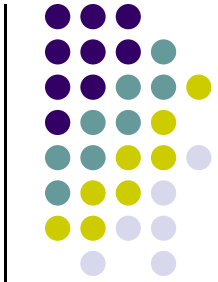
```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

 Rel. Loc    Obj. code Source line
 ---- ------ --------- -----------
    4
    5                  ; Initialize 8-bit memory
    6 000000 180B 64xx          movb   #$64,Data1
      000004 xx
    7                  ; Initialize 16-bit memory
    8 000005 1803 1234          movw   #$1234,Data2
      000009 xxxx
    9 00000B 1804 xxxx          movw   Data2,Data3
      00000F xxxx
   10 000011           Data1:   ds.b   1
   11 000012           Data2:   ds.w   1
   12 000014           Data3:   ds.w   1
```

Increment Instructions



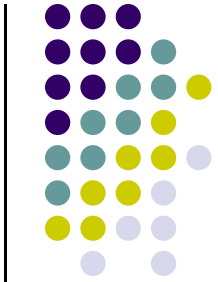
- Mnemonic Operation
  - INCA : Increment A: A ← (A) + 1
  - INCB : Increment B: B ← (B) + 1
  - INS : Increment SP: SP ← (SP) + 1
  - INX : Increment X : X ← (X) + 1
  - INY : Increment Y : Y ← (Y) + 1
- Addressing modes: INH Only
- Condition codes:
  - INCA,INCB: N,Z and V
  - INX, INY: Z
  - INS: none

Increment Instructions



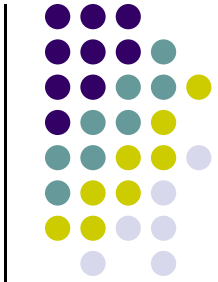
- Mnemonic Operation
  - INC : Increment Mem: (M) ← (M) +1
- Addressing modes: All but INH
- Condition codes: N,Z and V

Decrement Instructions



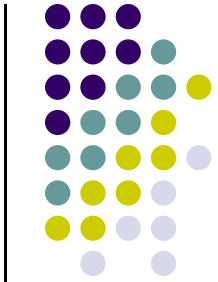
- Mnemonic Operation
  - DECA : Decrement A: A ← (A) – 1
  - DECB : Decrement B: B ← (B) – 1
  - DES : Decrement SP: SP ← (SP) -1
  - DEX : Decrement X : X ← (X) - 1
  - DEY : Decrement Y : Y ← (Y) – 1
- Addressing modes: INH Only
- Condition codes:
  - DECA,DECB: N,Z and V
  - DEX, DEY: Z
  - DES: none

Decrement Instructions



- Mnemonic Operation
  - DEC : Decrement Mem: (M) ← (M) -1
- Addressing modes: All but INH
- Condition codes: N,Z and V

Clear Instructions



- Mnemonic Operation
  - CLR : Clear Memory: (M) ← 0
  - CLRA : Clear A: A ← 0
  - CLRB : Clear B: B ← 0
- Addressing modes:
  - CLR: EXT, IDX, INH
  - CLRA, CLRB: INH only
- Condition codes: N=0,Z=1,V=0,C=0

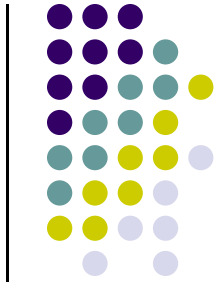
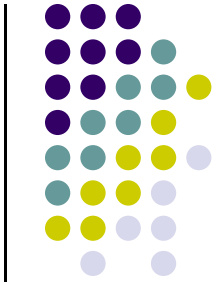


TABLE 7-9 Decrement and Increment Instructions

Function	Opcode	Symbolic Operation	Addressing Modes						Condition Codes			
			IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Decrement Memory	DEC	(M) − 1 → (M)			x	x	x		↕	↕	↕	–
Decrement A	DECA	A − 1 → A						x	↕	↕	↕	–
Decrement B	DECB	B − 1 → B						x	↕	↕	↕	–
Decrement X	DEX	X − 1 → X						x	–	↕	–	–
Decrement Y	DEY	Y − 1 → Y						x	–	↕	–	–
Decrement SP	DES[a]	S − 1 → S						x	–	–	–	–
Increment Memory	INC	(M) + 1 → (M)			x	x	x		↕	↕	↕	–
Increment A	INCA	A + 1 → A						x	↕	↕	↕	–
Increment B	INCB	B + 1 → B						x	↕	↕	↕	–
Increment X	INX	X + 1 → X						x	–	↕	–	–
Increment Y	INY	Y + 1 → Y						x	–	↕	–	–
Increment SP	INS[a]	S + 1 → S						x	–	–	–	–

[a] DES and INS are equivalent to LEAS −1,S and LEAS 1,S.

Clear and Set Bit Instructions



- Mnemonic Operation
  - BCLR : Clear Bits
  - BSET : Set Bits
- Addressing modes: EXT, IDX,IDR
- Condition codes: N,Z,V=0
- Example:
  - BCLR ADDRESS MASK(8 bits)
    - BCLR $33 $AA
    - Clear bits 2,4,6, and 8 at memory add $33

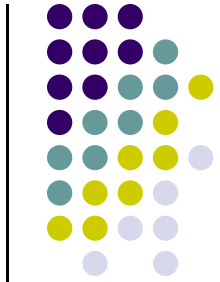


TABLE 7-10 Clear and Set Instructions

			Addressing Modes						Condition Codes			
Function	**Opcode**	**Symbolic Operation**	**IMM**	**DIR**	**EXT**	**IDX**	**IDR**	**INH**	**N**	**Z**	**V**	**C**
Clear Memory	CLR	$0 \rightarrow (M)$			x	x	x		0	1	0	0
Clear A	CLRA	$0 \rightarrow A$						x	0	1	0	0
Clear B	CLRB	$0 \rightarrow B$						x	0	1	0	0
Clear Bits in Memory	BCLR				x	x	x		↕	↕	0	–
Set Bits in Memory	BSET				x	x	x		↕	↕	0	–

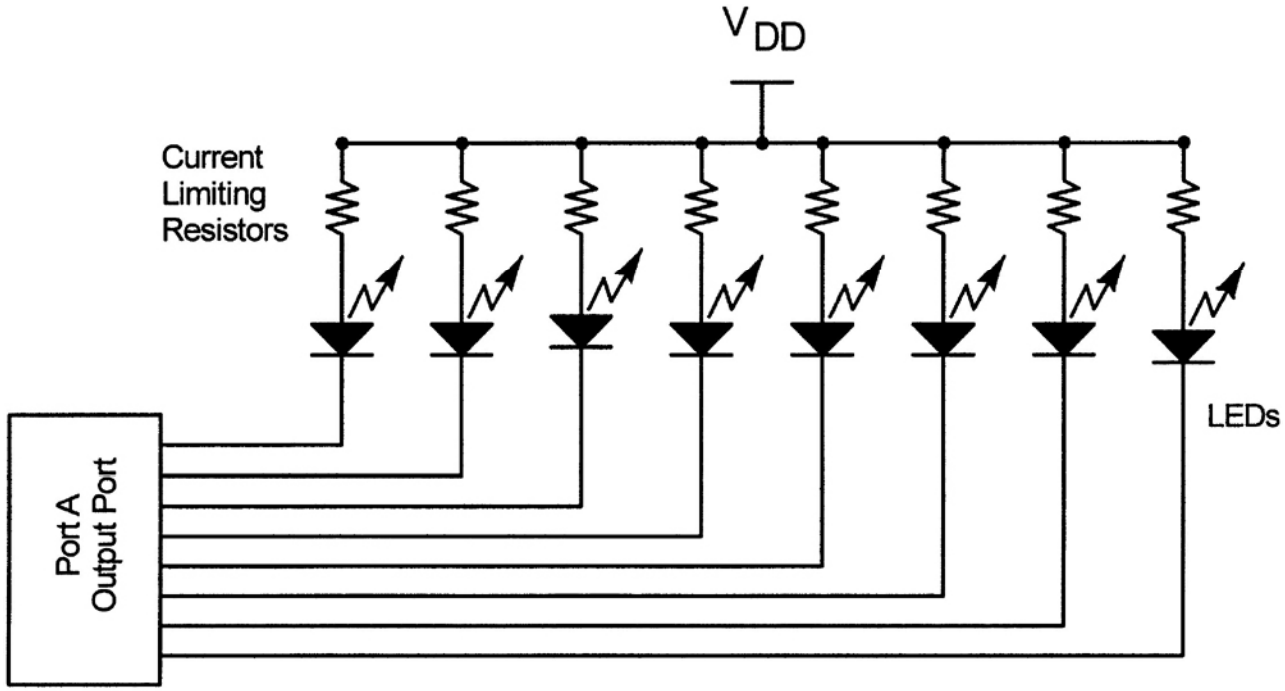
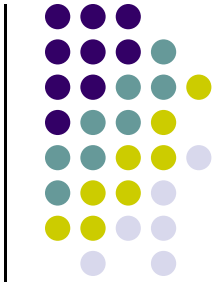


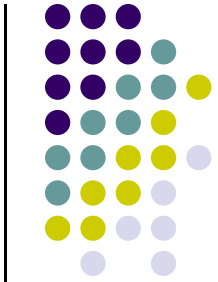
Figure 7-4 BCLR and BSET used for turning on and off LEDs.



Example 7-20 LED Program Using BCLR and BSET

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003
 Rel. Loc    Obj. code Source line
 ---- ------ --------- -----------
  4
  5          0000 0002 DDRA:  EQU   $0002  ; Data direction
                                            register
  6          0000 0000 PTA:   EQU   $0000  ; Port A register
  7                    ; . . .
  8   000000 4C02 FF          bset  DDRA,%11111111 ; Make all lines
                                                   output
  9                    loop:
 10   000003 4C00 FF          bset  PTA,%11111111  ; Set all bits,
                                                   LEDs off
 11   000006 4D00 01          bclr  PTA,%00000001  ; Clear bit 0,
                                                   LED on
 12   000009 4D00 02          bclr  PTA,%00000010  ; Clear bit 1
 13   00000C 4D00 04          bclr  PTA,%00000100  ; Clear bit 2
 14   00000F 4D00 08          bclr  PTA,%00001000  ; Clear bit 3
 15   000012 4D00 10          bclr  PTA,%00010000  ; Clear bit 4
 16   000015 4D00 20          bclr  PTA,%00100000  ; Clear bit 5
 17   000018 4D00 40          bclr  PTA,%01000000  ; Clear bit 6
 18   00001B 4D00 80          bclr  PTA,%10000000  ; Clear bit 7
 19   00001E 20E3             bra   loop           ; Do it forever
```

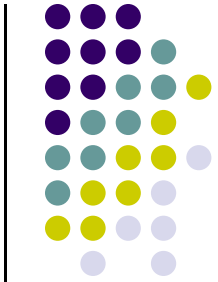
Example Code #4
What data are in the A and B registers after this code fragment executes?



```
MASK  EQU  $AA
LDX   #$00FF
STX   $0000
BSET $00 MASK
BCLR $01 MASK
LDD  $0000
```

A= B= X= Y=

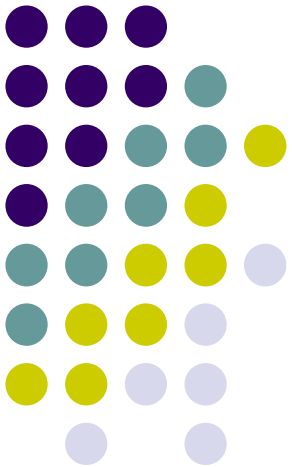
Example Code #4 - Solution
What data are in the A and B registers after this code fragment executes?



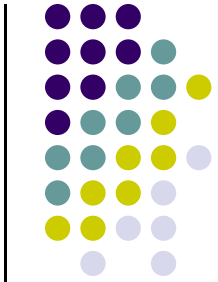
```
MASK  EQU  $AA
LDX   #$00FF       ;  X ← $00FF
STX   $0000        ;  ($0000:$0001) = $00FF
BSET $00 MASK   ; ($0000) OR $AA = $00 OR $AA = $AA
BCLR $01 MASK   ; ($0001) AND ($55) = $FF AND $55 = $55
LDD  $0000         ;  D=(A,B) ;  A←($00) = $AA B←($01) = $55
```

A=$AA B=$55

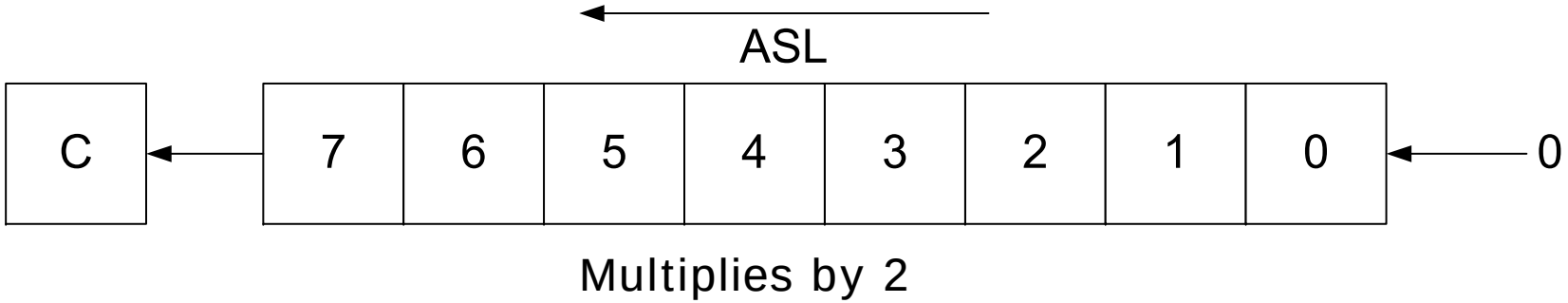
Shifting Instructions



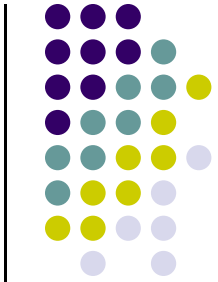
Shift Bit Instructions - ASL



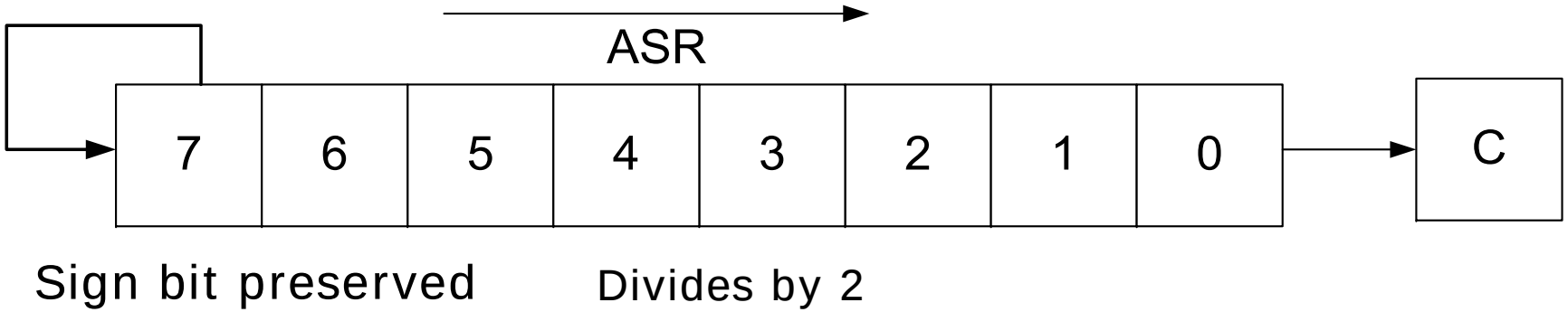
- Mnemonic Operation
  - ASL Arithmetic Shift Left Memory
  - ASLA: Arithmetic Shift Left A
  - ASLB: Arithmetic Shift Left B
  - ASLD: Arithmetic Shift Left D (16 bits)
- Addressing modes: INH or Direct, Index
- Condition codes: N,Z,V,C



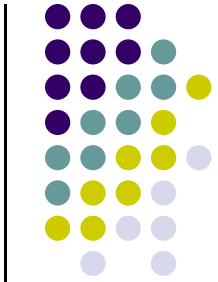
Shift Bit Instructions - ASR



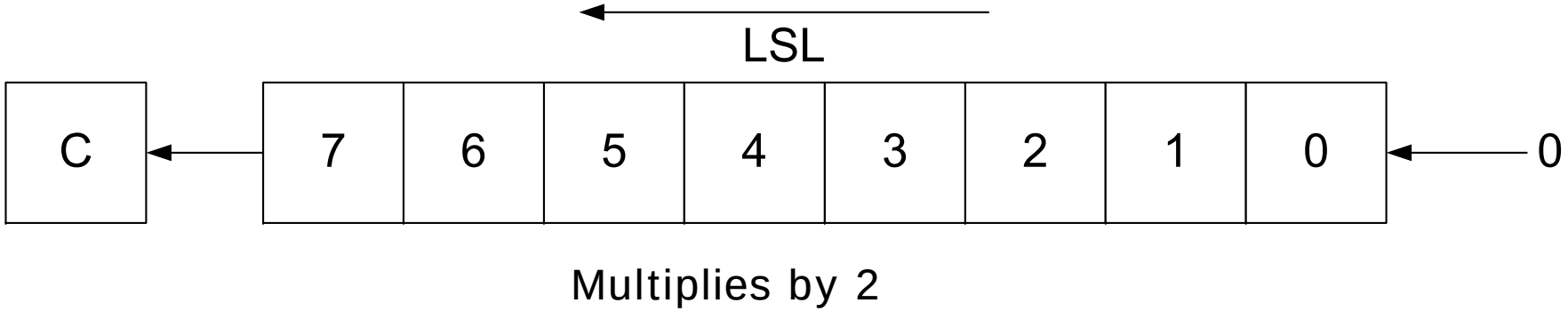
- Mnemonic Operation
  - ASR Arithmetic Shift Right Memory
  - ASRA: Arithmetic Shift Right A
  - ASRB: Arithmetic Shift Right B
  - ASRD: Arithmetic Shift Right D (16 bits)
- Addressing modes: INH or Direct, Index
- Condition codes: N,Z,V,C



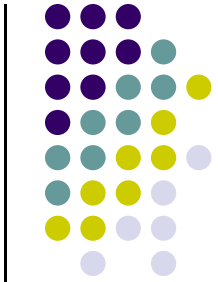
Shift Bit Instructions - LSL



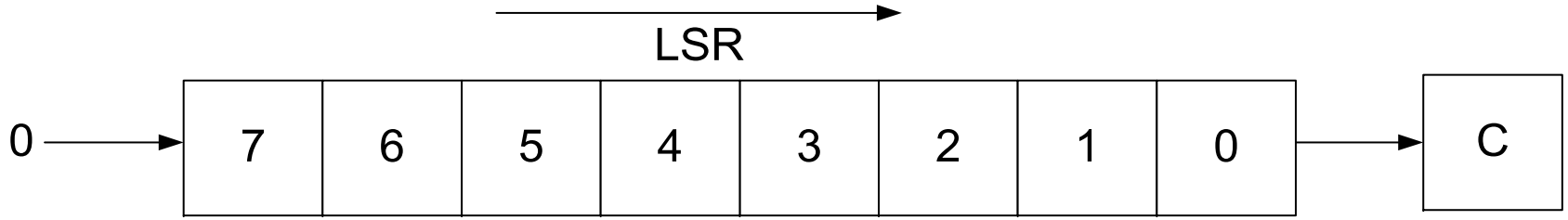
- Mnemonic Operation
  - LSL Logical Shift Left Memory
  - LSLA: Logical Shift Left A
  - LSLB: Logical Shift Left B
  - LSLD: Logical Shift Left D (16 bits)
- Addressing modes: INH or Direct, Index
- Condition codes: N,Z,V,C



Shift Bit Instructions - LSR

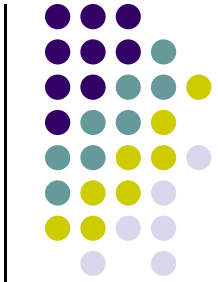


- Mnemonic Operation
  - LSR Logical Shift Right Memory
  - LSRA: Logical Right Left A
  - LSRB: Logical Right Left B
  - LSRD: Logical Right Left D (16 bits)
- Addressing modes: INH or Direct, Index
- Condition codes: N=0,Z,V,C

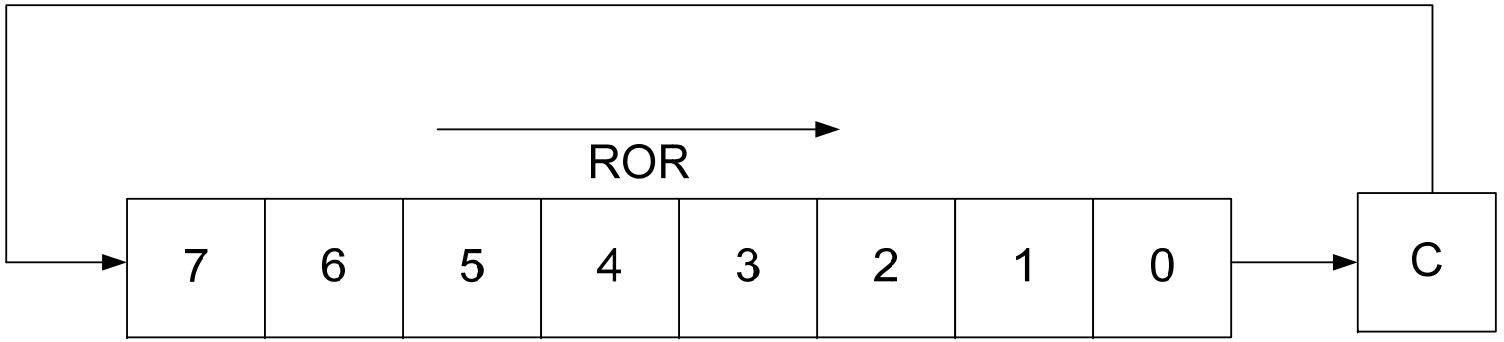


Divides by 2

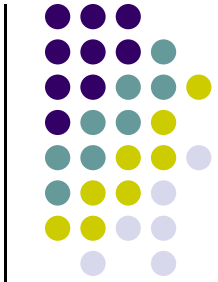
Shift Bit Instructions - ROR



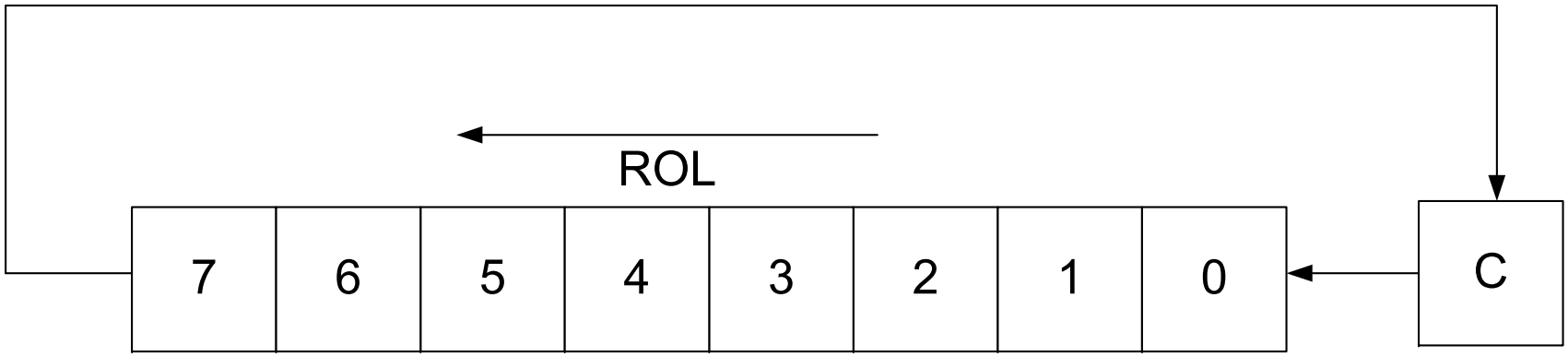
- Mnemonic Operation
  - ROR Rotate Right Memory
  - RORA: Rotate Right A
  - RORB: Rotate Right B
- Addressing modes: INH or Direct, Index
- Condition codes: N,Z,V,C



Shift Bit Instructions - ROL



- Mnemonic Operation
  - ROL Rotate Left Memory
  - ROLA: Rotate Left A
  - ROLB: Rotate Left B
- Addressing modes: INH or Direct, Index
- Condition codes: N,Z,V,C



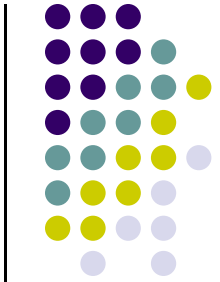
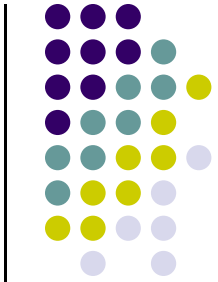


TABLE 7-11 Shift and Rotate Instructions

			Addressing Modes						Condition Codes			
Function	Opcode	Symbolic Operation	IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Arithmetic Shift Left Memory	ASL	Figure 7-5			x	x	x		↕	↕	↕	↕
Arithmetic Shift Left A	ASLA							x	↕	↕	↕	↕
Arithmetic Shift Left B	ASLB							x	↕	↕	↕	↕
Arithmetic Shift Left D (16-bit)	ASLD							x	↕	↕	↕	↕
Arithmetic Shift Right Memory	ASR	Figure 7-6			x	x	x		↕	↕	↕	↕
Arithmetic Shift Right A	ASRA							x	↕	↕	↕	↕
Arithmetic Shift Right B	ASRB							x	↕	↕	↕	↕
Arithmetic Shift Right D (16-bit)	ASRD							x	↕	↕	↕	↕
Logical Shift Left Memory	LSL	Figure 7-7			x	x	x		↕	↕	↕	↕
Logical Shift Left A	LSLA							x	↕	↕	↕	↕
Logical Shift Left B	LSLB							x	↕	↕	↕	↕
Logical Shift Left D (16-bit)	LSLD							x	↕	↕	↕	↕
Logical Shift Right Memory	LSR	Figure 7-8			x	x	x		0	↕	↕	↕
Logical Shift Right A	LSRA							x	0	↕	↕	↕
Logical Shift Right B	LSRB							x	0	↕	↕	↕
Logical Shift Right D (16-bit)	LSRD							x	0	↕	↕	↕
Rotate Left Memory	ROL	Figure 7-9			x	x	x		↕	↕	↕	↕
Rotate Left A	ROLA							x	↕	↕	↕	↕
Rotate Left B	ROLB							x	↕	↕	↕	↕
Rotate Right Memory	ROR	Figure 7-10			x	x	x		↕	↕	↕	↕
Rotate Right A	RORA							x	↕	↕	↕	↕
Rotate Right B	RORB							x	↕	↕	↕	↕



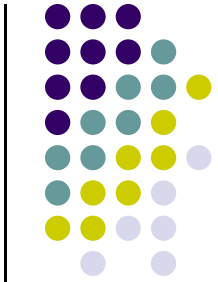
Example 7-23

Assume the A value is $A8. What is the result of each of the following instructions: ASLA, ASRA, LSLA, LSRA, ROLA, RORA?

Solution: The easiest way to look at these instructions is to show the values in binary. Before each instruction is executed, A contains %1010 1000. After each instruction, then, we find the following:

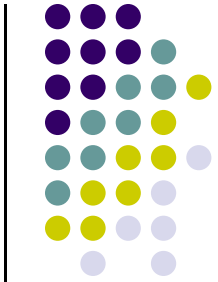
Before			After		
C	A Reg	Instruction	C	A Reg	Comments
x	1010 1000	ASLA	1	0101 0000	Zero shifted into bit 0.
x	1010 1000	ASRA	0	1101 0100	Sign bit is preserved.
x	1010 1000	LSLA	1	0101 0000	Same result as ASLA.
x	1010 1000	LSRA	0	0101 0100	Different from the ASRA.
C	1010 1000	ROLA	1	0101 000C	Carry bit is rotated into bit 0.
C	1010 1000	RORA	0	C101 0100	Carry bit is rotated into bit 7.

What is the value of D (in hex) after this code fragment executes?



```
        LDD  #100
        PSHD
        ASLD
        ASLD
        STD   vTemp
        PULD
        ADDD vTemp
Spin:   BRA Spin
vTemp   DS.W 1
```

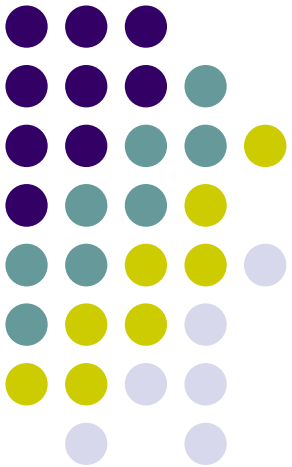
What is the value of D (in hex) after this code fragment executes?



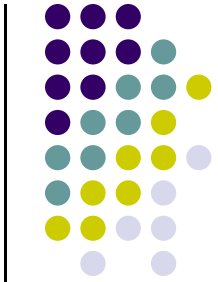
```
        LDD  #100       ; D ← 100
        PSHD            ; Save D on stack
        ASLD            ; D←D*2 = 200 (or origD*2)
        ASLD            ; D←D*2 = 400 (or origD*4)
        STD vTemp       ; Save D in memory (400)
        PULD            ; Pull original D from Stack (100)
        ADDD vTemp      ; D←D+vTemp, but this is just
                        ; D←origD+origD*4 = 5*origD
                        ; D ← 500
Spin:   BRA Spin
vTemp   DS.W 1
```

D= 500 = $01F4

Logic Instructions

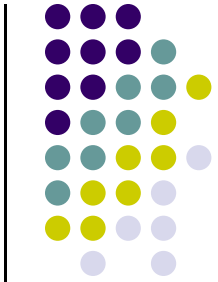


Logic Instructions



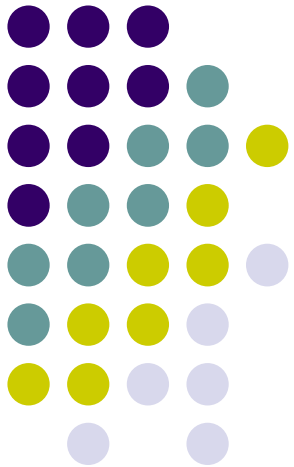
- Mnemonic Operation
  - ANDA: Logical A and mem: A ← A AND (M)
  - ANDB: Logical B and mem: B ← B AND (M)
  - ANDCC:Logical CCR and mem: CCR ← CCR AND (M)
  - EORA: Exclusive OR A and mem: A←A XOR (M)
  - EORB: Exclusive OR B and mem: B←B XOR (M)
  - ORAA: OR A or mem: A← A OR (M)
  - ORAB: OR B or mem: B← B OR (M)
  - ORCC: OR CCR or mem: CCR ← CCR OR (M)
- Addressing modes: All except IHN
- Condition codes: N,Z,V,C

Complement Instructions

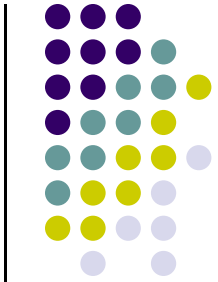


- Mnemonic Operation
  - COM: 1's Complement of mem: (M) ← (M)'
  - COMA: 1's Comp of A: A ← (A)'
  - COMB: 1's Comp of B: B←(B)'
- Addressing modes: INH or EXT, X, Y (mem)
- Condition codes: N,Z,V=0,C=1

Arithmetic Operations

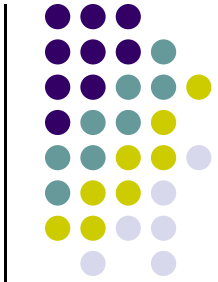


Arithmetic Instructions - ADD



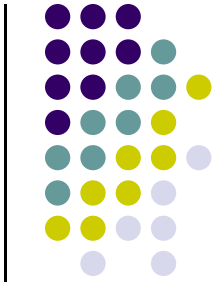
- Mnemonic Operation
  - ABA: Add B to A: A ← (A) + (B)
  - ABX: Add B + X : X ← B + X
  - ABY: Add B+Y: Y ←B + Y
  - ADDA: Add memory to A: A ← (A) + (M)
  - ADDB: Add memory to B: B ← (B) + (M)
  - ADDD: Add memory to D: D ← (D) + (M:M+1)
  - ADCA: Add memory to A and C: A← (A)+ (M) +C
  - ADCB: Add memory to B and C: B← (B)+ (M) +C
- Addressing modes: INH or All except IHN
- Condition codes: N,Z,V,C (except X and Y)

Arithmetic Instructions - Sub

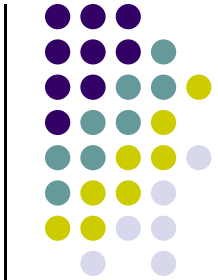


- Mnemonic Operation
  - SBA: Sub B from A: A ← (A) - (B)
  - SUBA: Sub memory from A: A ← (A) - (M)
  - SUBB: Sub memory from B: B ← (B) - (M)
  - SUBD: Sub memory from D: D ← (D) - (M:M+1)
  - SBCA: Sub memory and C from A: A←(A)-(M)-C
  - SBCB: Sub memory and C from B: B← (B)-(M)-C
- Addressing modes: INH or All except IHN
- Condition codes: N,Z,V,C

Arithmetic Instructions - Neg



- Mnemonic Operation
  - NEG: 2's comp memory: (M) ← -1*(M)
  - NEGA: 2's comp A : A ← -1*(A)
  - NEGB: 2's comp B : B ← -1*(B)
- Addressing modes: INH **or** Ext, Ix, Iy
- Condition codes: N,Z,V,C



Example 7-27 Negating Data

Assume A contains the following data before the HCS12 executes the NEGA instruction. What are the results in A and the N, Z, V, and C bits after the negation of each byte?
A = $00, $7F, $FF, $80.

Solution:

		After				
Before	**A Reg**	**N**	**Z**	**V**	**C**	**Comments**
$00	$00	0	0	0	0	Negating zero gives us zero
$7F	$81	1	0	0	0	Negating +127 give us −127
$01	$FF	1	0	0	0	Negating +1 gives us −1
$FF	$01	0	0	0	0	Negating −1 gives us +1
$80	$80	1	0	1	1	Negating −128 gives us −128 and overflow and carry

Explanation: The logic for the condition code register bits for all NEG instructions is as follows:

N: The N bit is set if the most significant bit of the result is set; it is cleared otherwise.
Z: Z is set if the result is $00; it is cleared otherwise.
V: V is set if there is a two's-complement overflow from the implied subtraction from zero; it is cleared otherwise. Two's-complement overflow occurs only if the data being negated is $80.
C: C is set if there is a borrow in the implied subtraction from zero and thus is set in all cases except when the data being negated is $100.

Arithmetic Instructions – DAA Decimal Adjust

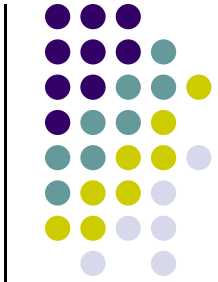


TABLE 7-24 DAA Instruction Correction Values

Initial C Bit	Value of A[7:4]	Initial H Bit	Value of A[3:0]	Correction Value	Corrected C Bit
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-2	1	0-3	66	1

Arithmetic Instructions – DAA Decimal Adjust

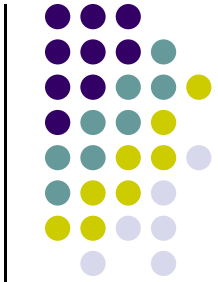


TABLE 7-24 DAA Instruction Correction Values

Initial C Bit	Value of A[7:4]	Initial H Bit	Value of A[3:0]	Correction Value	Corrected C Bit
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-2	1	0-3	66	1

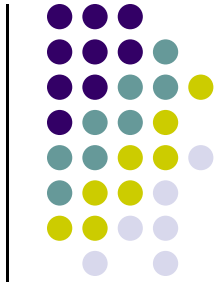
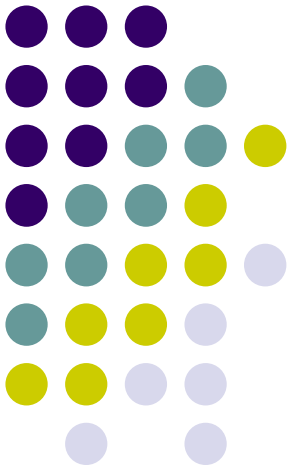


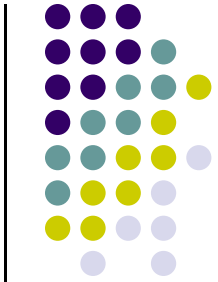
TABLE 7-12 Arithmetic Instructions

Function	Opcode	Symbolic Operation	Addressing Modes						Condition Codes			
			IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Add B to A	ABA	A + B → A						x	↕	↕	↕	↕
Add Memory to A	ADDA	A + (M) → A	x	x	x	x	x		↕	↕	↕	↕
Add Memory to B	ADDB	B + (M) → B	x	x	x	x	x		↕	↕	↕	↕
Add Memory to D (16-bit)	ADDD	D + (M : M +1) → D	x	x	x	x	x		↕	↕	↕	↕
Add with Carry to A	ADCA	A + (M) + C → A	x	x	x	x	x		↕	↕	↕	↕
Add with Carry to B	ADCB	B + (M) + C → B	x	x	x	x	x		↕	↕	↕	↕
Decimal Adjust	DAA							x	↕	↕	↕	↕
Subtract B from A	SBA	A − B → A						x	↕	↕	↕	↕
Subtract Memory from A	SUBA	A − (M) → A	x	x	x	x	x		↕	↕	↕	↕
Subtract Memory from B	SUBB	B − (M) → B	x	x	x	x	x		↕	↕	↕	↕
Subtract with Carry from A	SBCA	A − (M) − C → A	x	x	x	x	x		↕	↕	↕	↕
Subtract with Carry from B	SBCB	B − (M) − C → B	x	x	x	x	x		↕	↕	↕	↕
Subtract Memory from D (16-bit)	SUBD	D − (M : M +1) → D	x	x	x	x	x		↕	↕	↕	↕
Sign Extend A, B, CCR	SEX	See Example 7-30						x	—	—	—	—
Two's-Complement Memory	NEG	−(M) → (M)			x	x	x		↕	↕	↕	↕

Multiplication Instructions

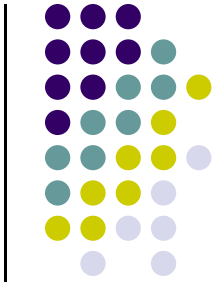


MUL Instruction



- MUL – Multiply Instruction
  - Multiplies 8-bit **unsigned** numbers loaded into the A and B registers
  - D ← A*B
  - Max value = $FF*$FF = $FE01 = 65,020

EMUL Instruction

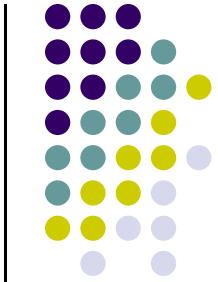


- EMUL – Multiply Instruction
  - Multiplies 16-bit **unsigned** numbers loaded into the D and Y registers
  - D:Y ← D*Y
  - Max value = $FFFF*$FFFF = $FFFE0001 = 4,294,839,225

IDIV Instruction

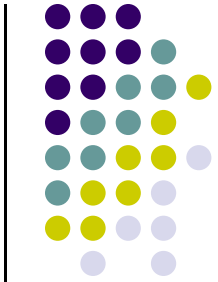
- IDIV – Integer Divide Instruction
  - Divides 16-bit **unsigned** D register by the 16-bit **unsigned** X register
  - Quotient is stored in the X register
    - X ← Integer portion of D/X
  - Remainder is stored in the D register
    - D ← D – Integer portion of D/X
  - If X is $0000, C← 1, and D ← $FFFF
  - For signed division
    - Determine sign of A and B
      - (-)/(-)=+ (+)/(+)=+ (-)/(+)=(-) (+)/(-)=-
      - NEG negative numbers
      - IDIV, NEG (if, needed)

Fractional Numbers



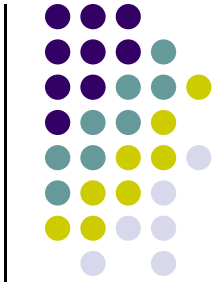
- Recall $2^0=1$, $2^1=2$, $2^2=4$, ….
- But, we can also go the other way
  - $2^{-1}=0.5$, $2^{-2}=0.25$, $2^{-3}=0.125$, $2^{-4}=0.0625$,…
- In general, we have $2^{-n}=1/2^n$
- So, for example, to represent
  - 2.5 = %10.10 and 5.25 = %101.01
    - Check: 2+0+0.5+0=2.5
    - Check: 4+0+1+0+0.25=5.25

Fractional Numbers



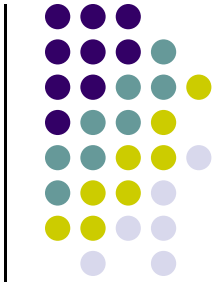
- We have
  - 2.5 = %10.10 and 5.25 = %101.01
- As 8 bit numbers, these become
  - %000010.10 and %000101.01
- Now, we really DON'T have a way to represent the decimal point in our numbers, so we have to **remember** where it is located.
- So really, we have
- 2.5 = %00001010 = $0A
- 5.25 = %00010101 = $15
- And we **remember** that we are using 2 bits for the fractional part.

Fractional Numbers



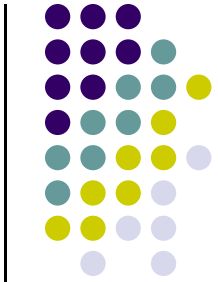
- Let's use 16-bit numbers where 4-bits (or one nybble) are reserved for the fractional part
- So, Y = $NNN.N
- Example, What is 375.875 in Hex?
- 375 → $177
- 0.875 = 1*0.5+1*0.25+1*0.125+0*0.0625
  - = %1110 = $E
- Or, 375.875 = $177E

Fractional Numbers



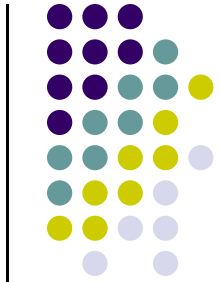
- Let's use 16-bit numbers where 4-bits (or one nybble) are reserved for the fractional part
- So, Y = $NNN.N
- Example, What is $24E3 as a fractional decimal?
- $24E → 590
- $3 = 0*0.5+0*0.25+1*0.125+1*0.0625=0.1875
- Or, $24E3 = 590.1875

FDIV Instruction



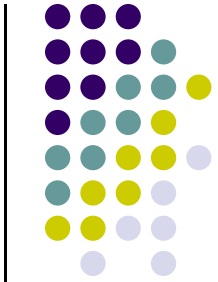
- FDIV – Fractional Divide Instruction
  - Divides 16-bit **unsigned** D register by the 16-bit **unsigned** X register
  - D register is assumed less than X register. (Answer is less than one)
  - Radix point is assumed to be the same
  - Fractional quotient is stored in the X register. Ex: %0.1010….
  - Remainder is stored in the D register
  - If X is $0000, C← 1, and D ← $FFFF

EDIV Instruction



- EDIV – Extended Fractional Divide Instruction
  - Divides 32-bit **unsigned** Y:D register by the 16-bit **unsigned** X register
  - Y←Quotient
  - D←Remander

EDIVS Instruction



- EDIV – Extended Fractional Divide Instruction **(Signed)**
  - Divides 32-bit **signed** Y:D register by the 16-bit **signed** X register
  - Y←Quotient
  - D←Remander

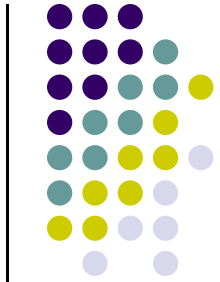
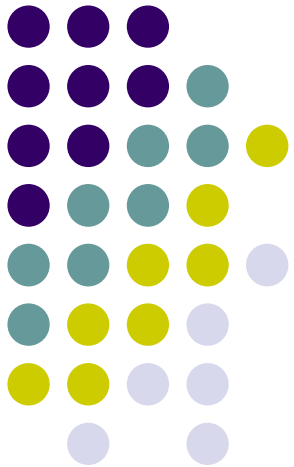


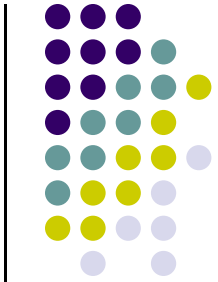
TABLE 7-12 Continued

Function	Opcode	Symbolic Operation	Addressing Modes						Condition Codes			
			IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Two's Complement A	NEGA	$-A \rightarrow A$						x	↕	↕	↕	↕
Two's Complement B	NEGB	$-B \rightarrow B$						x	↕	↕	↕	↕
Unsigned 8-bit Multiply A * B	MUL	$A * B \rightarrow D$						x	—	—	—	—
Unsigned 16-bit Multiply	EMUL	$D * Y \rightarrow Y : D$						x	↕	↕	—	↕
Signed 16-bit Multiply	EMULS	$D * Y \rightarrow Y : D$						x	↕	↕	—	↕
Unsigned 32/16-bit Division	EDIV	$Y : D/S \rightarrow Y,D$						x	↕	↕	↕	↕
Signed 32/16-bit Division	EDIVS	$Y : D/X \rightarrow Y,D$						x	↕	↕	↕	↕
Unsigned 16/16-bit Division	IDIV	$D/X \rightarrow X,D$						x	—	↕	0	↕
Signed 16/16-bit Division	IDIVS	$D/X \rightarrow X,D$						x	↕	↕	↕	↕
Fractional Division	FDIV	$D/X \rightarrow X,D$						x	—	↕	↕	↕

Branch Instructions

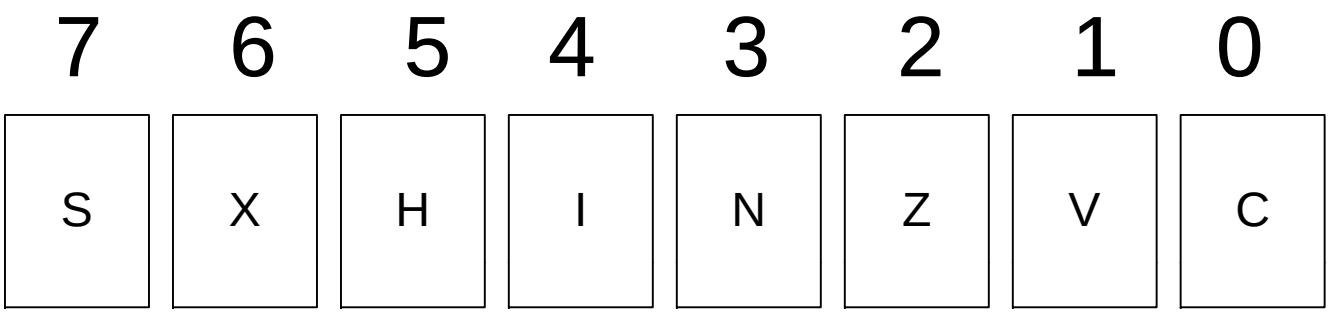


Branch Instructions



- A Branch Instruction typically follows a Compare instruction. The Branch instruction will branch if certain CCR bits are set or clear

Condition Code Register



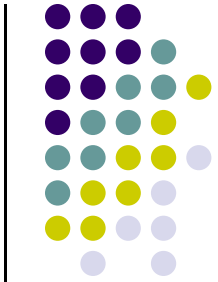
S = Stop
X = X Interrupt Bit
I = Interrupt Mask

Control Bits

N = Negative
Z = Zero
V = Overflow
C = Carry
H = Half Carry

Arithmetic Bits

Compare Instructions - CMP



- Mnemonic Operation
  - CBA: Compare B to A: A - B
  - CMPA: Compare memory to A: A - (M)
  - CMPB: Compare memory to B: B - (M)
  - CMPD: Compare memory to D: D - (M:M+1)
  - CMPX: Compare memory to X: X - (M:M+1)
  - CMPY: Compare memory to Y: Y – (M:M+1)
- Addressing modes: INH or All except IHN
- Condition codes: N,Z,V,C
  - Note: Only Condition Code Register (CCR) is changed by these instructions. All other registers unaffected.

Compare Instructions

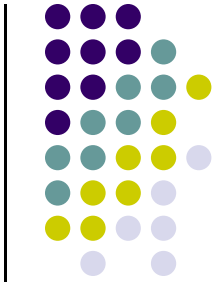
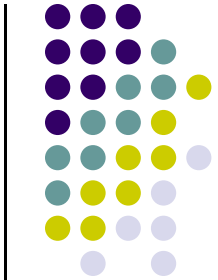


TABLE 7-15 Data Test Instructions

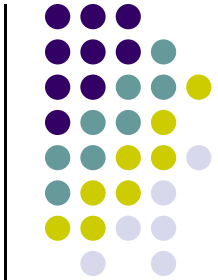
Function	Opcode	Symbolic Operation	Addressing Modes						Condition Codes			
			IMM	DIR	EXT	IDX	IDR	INH	N	Z	V	C
Test Bits in A	BITA	A AND (M)	x	x	x	x	x		↕	↕	0	—
Test Bits in B	BITB	B AND (M)	x	x	x	x	x		↕	↕	0	—
Compare A to B	CBA	A − B						x	↕	↕	↕	↕
Compare A to Memory	CMPA	A − (M)	x	x	x	x	x		↕	↕	↕	↕
Compare B to Memory	CMPB	B − (M)	x	x	x	x	x		↕	↕	↕	↕
Compare D to Memory (16-bit)	CPD	D − (M : M + 1)	x	x	x	x	x		↕	↕	↕	↕
Compare X to Memory (16-bit)	CPX	X − (M : M + 1)	x	x	x	x	x		↕	↕	↕	↕
Compare Y to Memory (16-bit)	CPY	Y − (M : M + 1)	x	x	x	x	x		↕	↕	↕	↕
Compare S to Memory (16-bit)	CPS	S − (M : M + 1)	x	x	x	x	x		↕	↕	↕	↕
Test Memory for Zero or Negative	TST	(M) − 0			x	x	x		↕	↕	0	0
Test A for Zero or Negative	TSTA	A − 0						x	↕	↕	0	0
Test B for Zero or Negative	TSTB	B − 0						x	↕	↕	0	0

Branch Instructions



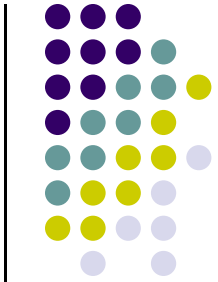
- BRA – Branch Always
  - Unconditional Branch (i.e. GoTo)
- BEQ - Branch if equal
  - Z bit = 1
- BNE – Branch if not equal
  - Z bit = 0
- BCC – Branch if Carry Clear
  - C bit = 0
- BCS – Branch if Carry Set
  - C bit = 1

Branch Instructions



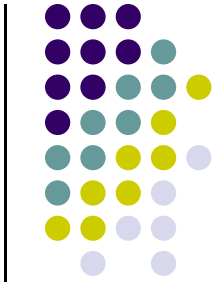
- BMI - Branch if Minus
  - Branch if N = 1
- BPL – Branch if Positive
  - Branch if N = 0
- BVS – Branch if Overflow Set
  - Branch if V=1
- BVC – Branch if Overflow Clear
  - Branch if V=0

Branch Instructions Unsigned



- BHI - Branch if High
  - Unsigned, Branch if C OR Z = 0
- BHS – Branch if Higher or Same
  - Unsigned, Branch if C=0
- BLO – Branch if Low
  - Unsigned, Branch if C = 1
- BLS – Branch if Lower or Same
  - Unsigned, Branch if C OR Z = 1

Branch Instructions Signed



- BGE - Branch if Greater Than or Equal
  - Signed, Branch if N XOR V = 0
- BGT – Branch if Greater Than
  - Signed, Branch if Z OR (N XOR V) = 0
- BLE – Branch if Less Than or Equal
  - Signed, Branch if Z OR (N XOR V) = 1
- BLT – Branch if Less Than
  - Signed, Branch if (N XOR V) = 1

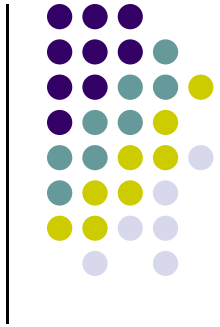
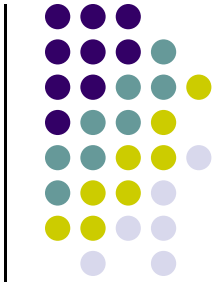


TABLE 7-26 Conditional Branch Instructions for Signed and Unsigned Data

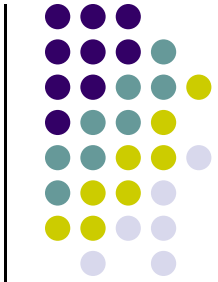
Signed Data Tests		Unsigned Data Tests		Universal Tests	
BMI	Minus				
BPL	Plus				
BVS	Two's-complement overflow	BCS	Carry set = unsigned overflow	BCS	Carry set
BVC	No two's-complement overflow	BCC	Carry clear = no overflow	BCC	Carry clear
BLT	Less than	BLO	Lower than		
BGE	Greater than or equal	BHS	Higher or the same		
BLE	Less than or equal	BLS	Lower or the same		
BGT	Greater than	BHI	Higher than		
BEQ	Equal	BEQ	Equal	BEQ	Equal
BNE	Not equal	BNE	Not equal	BNE	Not Equal

Branch Examples



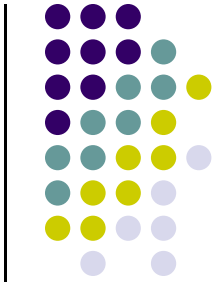
- Example 1
  - LDAA #$F2 (-14, signed) (242 unsigned)
  - LDAB #$F0 (-16 signed) (240 unsigned)
  - CBA (Compute A-B)
    - Result is $F2-$F0 = $02 (no borrow)
      - N bit = 0 (result is not negative)
      - Z bit = 0 (result is not zero)
      - V bit = 0 (2's comp overflow did not occur)
      - C bit = 0 (For a subtraction, **C is set if we have a borrow)**
  - Note: A >= B for both signed and unsigned numbers.
  - The following instructions will branch
    - BNE,BCC,BPL,BVC,BHI,BHS,BGE,BGT

Branch Example



- Example 2
  - LDAA #$F2 (-14, signed) (242 unsigned)
  - LDAB #$F2 (-14, signed) (242 unsigned)
  - CBA (Compute A-B)
    - Result is $F2-$F2 = $00
      - N bit = 0
      - Z bit = 1 (result is zero)
      - V bit = 0
      - C bit = 0 (no borrow)
  - Note: A = B for both signed and unsigned numbers.
  - The following instructions will branch
    - BEQ,BCC,BPL,BVC,BHS,BLS,BGE,BLE

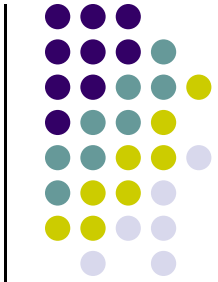
Branch Example



- Example 3
  - LDAA #$F2 (-14, signed) (242 unsigned)
  - LDAB #$FF (-1, signed) (255 unsigned)
  - CBA (Compute A-B)
    - Result is $F2-$FF = $F3 (with a borrow)
      - N bit = 1
      - Z bit = 0
      - V bit = 0
      - C bit = 1 (we have a borrow)

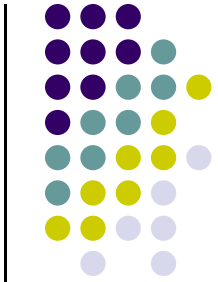
- Note: A <= B for both signed and unsigned numbers.
- The following instructions will branch
  - BNE,BCS,BMI,BVC,BLO,BLS,BLE,BLT

Branch Example



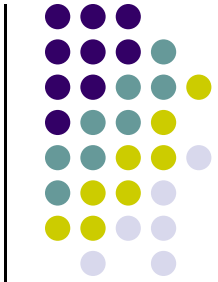
- Example 4
  - LDAA #$F2 (-14, signed) (242 unsigned)
  - LDAB #$02 (2, signed) (2 unsigned)
  - CBA (Compute A-B)
    - Result is $F2-$02 =$F0
      - N bit = 1 (result is negative: signed, but positive unsigned)
      - Z bit = 0
      - V bit = 0
      - C bit = 0 (no borrow)
  - Note: A <= B for signed and A>=B unsigned numbers.
  - The following instructions will branch
    - BNE,BCC,BMI,BVC,BHI,BHS,BLE,BLT

Branch Example



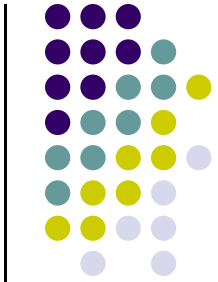
- Example 5
  - LDAA #$02 (2, signed) (2 unsigned)
  - LDAB #$F2 (-14, signed) (242 unsigned)
  - CBA (Compute A-B)
    - Result is $02-$F2 =$10 (with borrow)
      - N bit = 0 (result is positive for signed numbers)
      - Z bit = 0
      - V bit = 0
      - C bit = 1 (borrow)
  - Note: A >= B for signed and A<=B unsigned numbers.
  - The following instructions will branch
    - BNE,BCS,BPL, BVC, BLO,BLS,BGE,BGT,

Branch Example



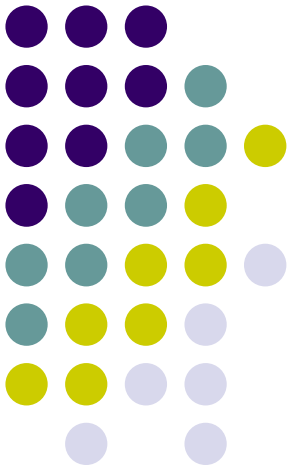
- Example 6
  - LDAA #$80 (-128 signed, +128 unsigned)
  - LDAB #$7F (+127 signed, +127 unsigned)
  - CBA (Compute A-B)
    - Result is $80-$7F = $01 (no borrow)
      - N bit = 0 (result is negative unsigned, positive signed)
      - Z bit = 0
      - V bit = 1 (Two's comp overflow)
      - C bit = 0 (not carry out)
- Two's comp overflow occurs when the borrow from MSB is not equal to the borrow out of the MSB. In this case, borrow out =1 and borrow in = 1
  - The following instructions will branch
    - BNE,BCC,BPL, BVS, BLE,BLT,BHI,BHS

Overflow bit (V bit) check

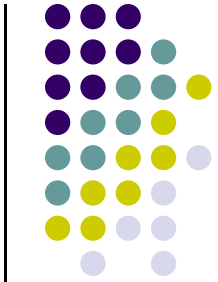


- You can also look at the result of the subtraction to determine of an overflow has occurred. If the result is greater than range, you have overflow.
- For example, given 8-bit
  - Signed range is -128 to +127
  - So, if the result of the subtraction is outside of the range, you have overflow.

TPS Quizzes

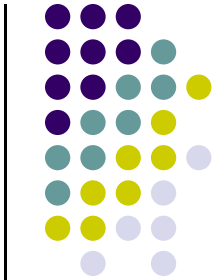


Quiz #1



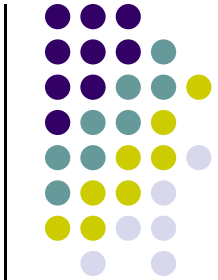
- Let A=$50 and ($1000) = $E0
- Given Code Fragment:
  - CMPA $1000
- Indicate whether each of the following branches will be taken
- BPL,BMI, BCC, BCS, BVC, BVS, BGE, BLE, BGT, BLT, BEQ, BNE, BHS, BLS, BHI, BLO

Solution



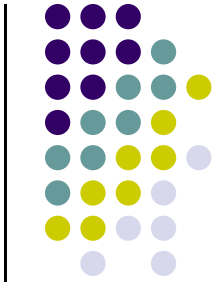
- A=$50, B=$E0
  - A is 80 unsigned, and 80 signed.
  - B is 224 unsigned and -32 signed
  - So (A<B) unsigned, (A>B) signed
- (A-B) = $50 - $E0 = $70 (with borrow)
  - N = 0
  - Z = 0
  - V = 0 (A-B)=(80-(-32))=112 (in range)
  - C = 1 (borrow)

Branch executed in RED



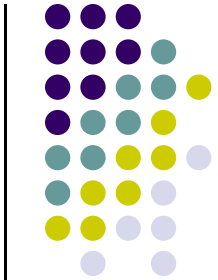
- N=0, Z=0, V=0, C=1
- BPL (Z=0),BMI, BCC, BCS (C=1), BVC (V=0), BVS, BGE (N XOR V=0), BLE, BGT (N XOR V=0), BLT, BEQ, BNE (Z=0), BHS, BLS (C=1), BHI, BLO (C=1)

Quiz #2



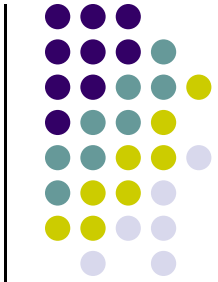
- Let A=$88 and ($1000) = $3F
- Given Code Fragment:
  - CMPA $1000
- Indicate whether each of the following branches will be taken
- BPL,BMI, BCC, BCS, BVC, BVS, BGE, BLE, BGT, BLT, BEQ, BNE, BHS, BLS, BHI, BLO

Solution



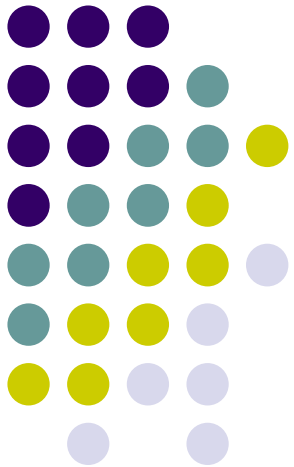
- A=$88, B=$3F
  - A is 136 unsigned, and -120 signed.
  - B is 63 unsigned and 63 signed
  - So (A>=B) unsigned, (A<=B) signed
- (A-B) = $88 - $3F = $49 (with no borrow)
  - N = 0
  - Z = 0
  - V = 1 (A-B)=(-120- 63)=-183 (out of range)
  - C = 0 (no borrow)

Branch executed in RED

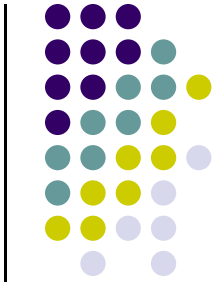


- N=0, Z=0, V=1, C=0
- BPL (Z=0),BMI, BCC (C=0), BCS (C=1), BVS (V=1), BVC, BGE, BLE(N XOR V=1), BGT, BLT(N XOR V=0), BEQ, BNE (Z=0), BHS (C=0), BLS (C=1), BHI (C=0), BLO (C=1)

Loop Primitive Instructions

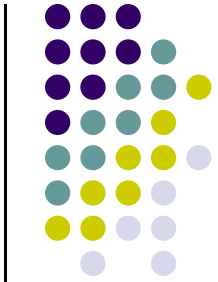


Loop Primitive Instructions



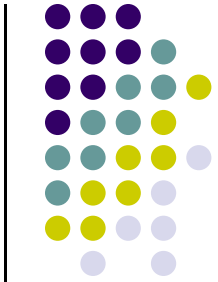
- Mnemonic Operation
  - DBEQ: Decrement Register and Branch if Zero
  - DBNE: Decrement Register and Branch if not Zero
  - IBEQ: Increment Register and Branch if Zero
  - IBNE: Increment Register and Branch if not Zero
  - TBEQ: Test Register and Branch if Zero
  - TBNE: Test Register and Branch if not Zero
- Addressing modes: REL
- Condition codes: Not Changed

Test Instructions - TST



- Test if Memory = 0
- Mnemonic Operation
  - TST: Test memory = 0: (m) - 0
  - TSTA: Test A = 0 : A - 0
  - TSTB: Test B = 0: B - 0
- Addressing modes: INH **or** Ext, X, Y
- Condition codes: N,Z,V=0,C=0

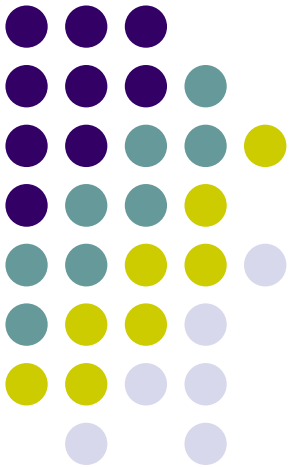
Test Bits Instructions - BIT



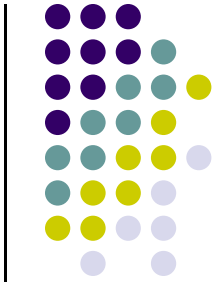
- Mnemonic Operation
  - BITA: AND Register A with memory: A AND (MEM)
  - BITB: AND Register B with memory: B AND (MEM)
- Addressing modes: IMM,DIR, Ext, X, Y
- Condition codes: N,Z,C=V=0
- Note: Memory does NOT change, Only CCR

Subroutine Instructions

JMP/JSR/BSR/RTS



JMP Instructions



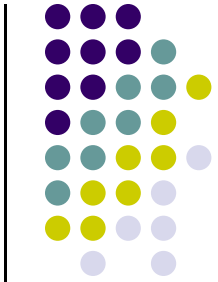
- Jump to Address
  - Similar to BRA but uses 16 bit absolute address
- Mnemonic Operation
  - JMP Address
- Addressing modes: EXT, X, Y
  - Condition codes: none
  - PC ← Address

JMP Example

```
Program EQU $E000
Stack   EQU $00FF
        ORG Program
E000:   8E 00 FF   Top:    LDS #Stack
E003:   7E E0 03   Done:   JMP Done
********** Compare with
E000:   8E 00 FF   Top:    LDS #Stack
E003:   20 FE      Done:   BRA Done
```

Why use JMP instead of BRA?

Why use JMP instead of BRA?



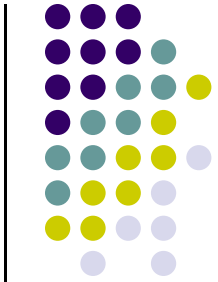
We can only Branch up to -128 to +127 bytes from the current address. If we need to Branch to an address outside of this range, we'll need to use JMP instruction.

* EXAMPLE. Assume FAR_LABEL is +$1000 bytes away from this address

```
TSTA
BEQ  FAR_LABEL
LDAA #NUMA    ;  This is the A<>0 code
```

- This code will give us an error (out of range)
- However, we'll see that there is a LBEQ instruction that we could use.

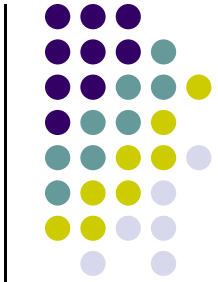
Why use JMP instead of BRA?



* We should use a JMP instead

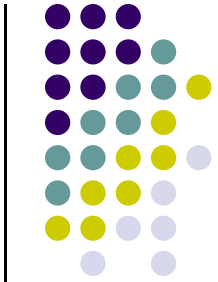
```
        TSTA
        BNE L1          ; Use a BNE because the label
                        ;  is local.
        JMP  FAR_LABEL
L1:    LDAA #NUMA       ; This is the A<>0 code
```

JSR Instructions



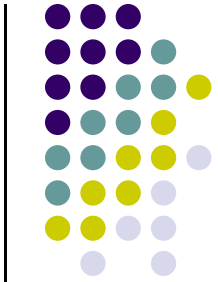
- Jump to Subroutine
  - Similar to JMP **except** we have the ability to return to the calling program. Same as HLL.
- Mnemonic Operation
  - JSR Address
- Addressing modes: DIR, EXT, X, Y
  - Condition codes: none
    - PSH PC ; Program Counter pushed onto stack
    - PC ← Address

JSR Instruction



- JSR
  - Pushes Program Counter onto Stack
    - Note: PC is pointing to the next instruction
  - Loads PC with Addressing of Subroutine
    - Starts executing program beginning at EA
  - How do we return from the subroutine?

RTS - Instruction

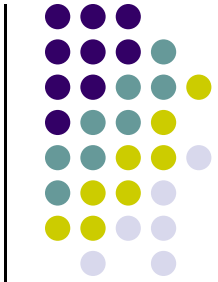


- RTS – Return from Subroutine
  - PULL PC from Stack
    - Recall this contains the EA of the instruction following the original JSR/BSR
  - PC ← Value pulled from stack
    - Execute program
  - Note: You must POP anything you have PUSHed onto the stack or the RTS will start executing the wrong program

Subroutine Example

```
*************************
*   Subroutine Name:  BCD2ASC
*   Input parameter list: A
*   Output parameter list:D
*   Registers changed list: A,B
*************************
* This routine converts an Packed BCD
* into an ASCII character.
************************************************
Lower   EQU  $0F
Upper   EQU   $F0
ASCII   EQU   $30
Bcd2asc: PSHA              ;  Save A value
          PULB             ;  Pull into B register
          ANDB #LOWER      ;  Mask lower nybble
          ORAB #ASCII      ; Convert to ASCII
          ANDA #UPPER      ;  Mask upper nybble
          ORAA #ASCII      ; Convert to ASCII
          RTS              ; Result is stored in D register
```

Using Subroutine

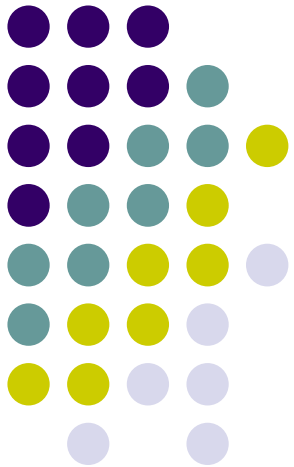


- …….. ; code fragment

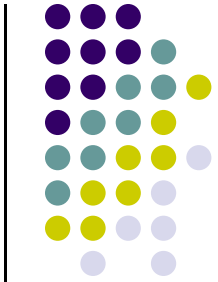
- JSR BCD2ASC ; Call subroutine to do conversion
- E200: STDD Output……. ; Do something with the result
- ……..
  ; More code
  Done: BRA Done ; End of main program

Assume SP=$0500
After JSR, SP=$04FE
Stack memory looks like:
$04FE: E2 00
If we PSH anything on the stack, we must PUL it off before the RTS instruction
RTS instruction will place PC←$E200

Arithmetic Examples

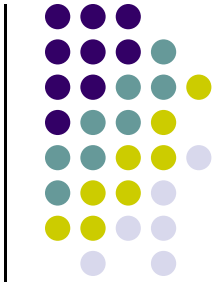


16-bit addition using a 8-bit register



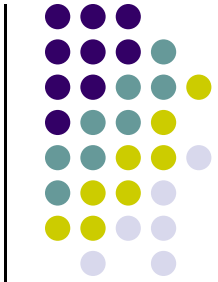
```
;   Calculate Sum = Data1+Data2 using only the 8-bit A Register
Sum   DS.B 2
Data1 DS.B 2
Data2 DS.B 2
            LDS #Stack
            CLI
            LDAA Data1+1  ;   Get low byte of Data1
            ADDA Data2+1  ;   Add low byte of Data2
            STAA Sum+1    ;  Store A in low byte of Sum
            LDAA Data1    ;  Get high byte of Data1
            ADCA Data2    ;   A ← Data1+Data2+C
            STAA Sum      ;  Store A into high byte of Sum
   Spin:    BRA Spin
```

16-bit addition using a 16-bit register



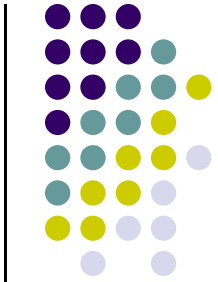
```
;   Calculate Sum = Data1+Data2 using the 16-bit d Register
Sum   DS.B 2
Data1 DS.B 2
Data2 DS.B 2
                LDS #Stack
                CLI
                LDD Data1       ;   Get Data1
                ADDD Data2      ;   Add Data2
                STD Sum         ;  Store Sum
    Spin:       BRA Spin
```

32-bit addition using A and D registers



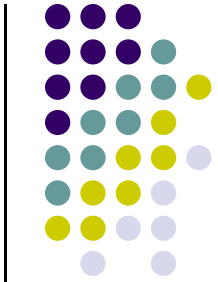
```
;   Calculate 32-bit Sum = Data1+Data2 using the 16-bit D Register
Sum   DS.W 2    ;  Use DS.W to define a 32 bit memory location
Data1 DS.W 2
Data2 DS.W 2
              LDS #Stack
              CLI
              LDD Data1+2      ;   Get low word of Data1
              ADDD Data2+2     ;   Add low word of Data2
              STD Sum+2        ;  Store D in low word of Sum
              LDAA Data1+1      ;  Get low byte of high word of Data1
              ADCA Data2+1      ;   Add low byte of high word of Data2 to carry
              STAA Sum+1        ;  Store low byte of high word of sum
              LDAA Data1        ;   Get high byte of high word of Data1
              ADCA Data2        ;  Add high byte of high word of Data2 to carry
              STAA Sum          ;   Store high byte of high word of Sum
    Spin:     BRA Spin
```

8-bit unsigned multiplication 16-bit result



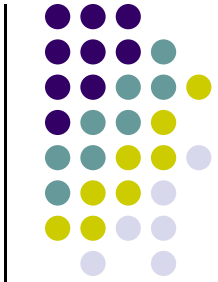
```
;   Calculate Result = Data1*Data2
Result   DS.B 2
Data1 DS.B 1
Data2 DS.B 1
                LDS #Stack
                CLI
                LDAA Data1      ;   Get Data1
                LDAB Data2      ;   Get Data2
                MUL             ;    D ← A*N
                STD Result      ;  Store Sum
    Spin:       BRA Spin
```

8-bit signed multiplication 16-bit result



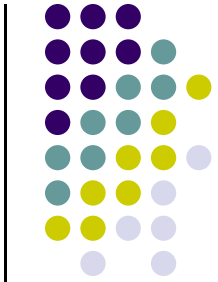
```
;   Calculate Result = Data1*Data2
Result   DS.B 2
Data1 DS.B 1
Data2 DS.B 1
                LDS #Stack
                CLI
                LDAA Data1      ;   Get Data1
                SEX A,Y         ; Sign extend A into Y
                LDAA Data2      ;    Get Data2
                SEX A,D         ; Sign extend A into D
                EMULS           ; Signed Extended Multiply
                STD Result      ;  D has low 16-bits of result
    Spin:       BRA Spin
```

16-bit unsigned multiplication 32-bit result



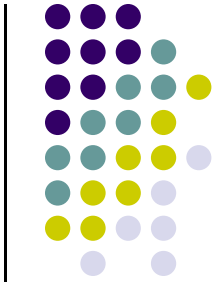
```
;   Calculate Result = Data1*Data2
Result   DS.B 4     ; Use 4 bytes for 32-bit result
Data1 DS.B 2
Data2 DS.B 2
                LDS #Stack
                CLI
                LDD  Data1      ;   Get Data1
                LDY Data2       ;   Get Data2
                EMUL            ;    Y:D ← D*Y
                STD Result+2    ;  Store low word of Result
                STY Result      ;  Store high word of Result
    Spin:       BRA Spin
```

16-bit signed multiplication 32-bit result



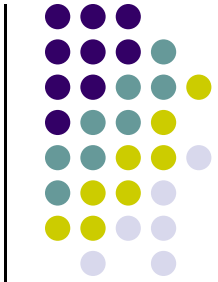
```
;   Calculate Result = Data1*Data2
Result   DS.B 4     ; Use 4 bytes for 32-bit result
Data1 DS.B 2
Data2 DS.B 2
                LDS #Stack
                CLI
                LDD  Data1       ;   Get Data1
                LDY Data2        ;   Get Data2
                EMULS            ;    Y:D ← D*Y
                STD Result+2     ;  Store low word of Result
                STY Result       ;  Store high word of Result
    Spin:       BRA Spin
```

16-bit integer divide (unsigned)



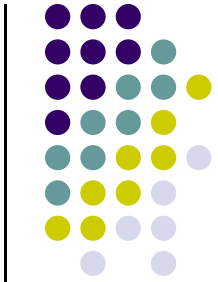
```
;   Calculate Result = Data2/Data1   Integer Divide
Result   DS.B 2
Rem  DS.B 2
Data1 DS.B 2
Data2 DS.B 2
                LDS #Stack
                CLI
                LDD Data2       ;   D ← Data2
                LDX Data1       ;   X ← Data1
                IDIV            ;   X ← Int(D/X)  D← Remainder
                STD Rem         ;  Store remainder
                STX Result        ;  Store result
    Spin:       BRA Spin
```

16-bit integer divide (unsigned) Example



```
;   What are Result and Rem after this code fragment executes
Result   DS.B 2
Rem  DS.B 2
Data1 DC.B $00,$0A
Data2 DC.B  $00,$C2
                LDS #Stack
                CLI
                LDD Data2       ;   D ← Data2
                LDX Data1       ;   X ← Data1
                IDIV            ;   X ← Int(D/X)  D← Remainder
                STD Rem         ;  Store remainder
                STX Result        ;  Store result
    Spin:       BRA Spin
```

16-bit integer divide (unsigned) Solution



D ← $00C2

X ← $000A

IDIV ; X ← Int(D/X) D← Remainder

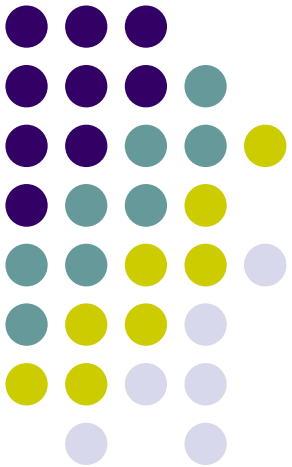
X← Int(D/X) = Int($00C2/$000A) = $0013

D← Rem(D/X) = $04

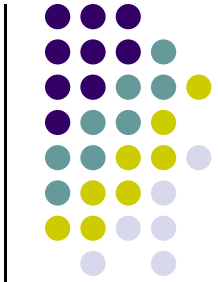
Note: D = 194 and X=10 , so Int(D/X) = 19 = $13

Rem = 194=19*10 = 4

Special Arithmetic Operations



Hexadecimal Number System



- Base 16
- Sixteen Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Example: $EF56_{16}$
- Positional Number System
- $$16^{n-1}\cdots 16^4 16^3 16^2 16^1 16^0$$

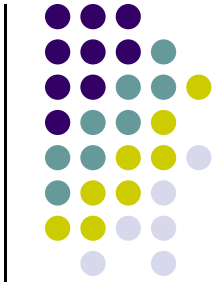
0000	0
0001	1
0010	2
0011	3

0100	4
0101	5
0110	6
0111	7

1000	8
1001	9
1010	A
1011	B

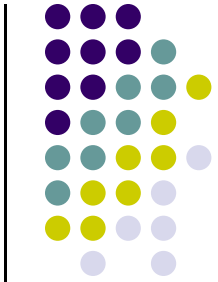
1100	C
1101	D
1110	E
1111	F

Binary Coded Decimals (Packed and UnPACKED BCDs)



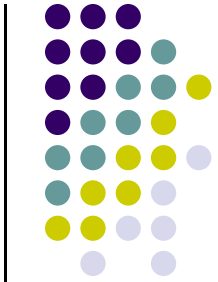
- Use 4-bits 0-9 to represent a decimal number
- Example: 1234_{10}
  - As hex = $4D2
  - As BCD =$0102, $0304
  - As Packed BCD = $1234
- Example: 5137
  - As Hex = $1411
  - As BCD = $0501,$0307
  - As Packed BCD = $5137

Decimal Arithmetic Instruction



- DAA – Decimal Adjust Instruction
  - Used to adjust the result of the addition of two packed BCDs
  - Use immediately after the ADDA instruction. Effects the A register
  - Example:
    - 34 = $22 (hex) or $34 as packed BCD
    - 29 = $1D (hex) or $29 as packed BCD
    - $34+$29 = $5D (not a valid packed BCD)
    - DAA: A← (A) +$06 = $63 (correct packed BCD)

Decimal Arithmetic Instruction



- DAA – Decimal Adjust Instruction
  - Why do we add $06?
  - Note:
    - $01+$09 = $0A + $06 = $10
  - What about $01+$02 = $03?
  - DAA checks CCR to determine if adjust is needed, so DAA: $03+$00 = $03

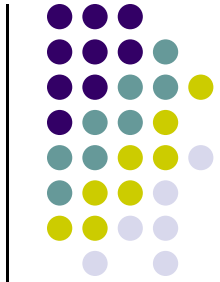
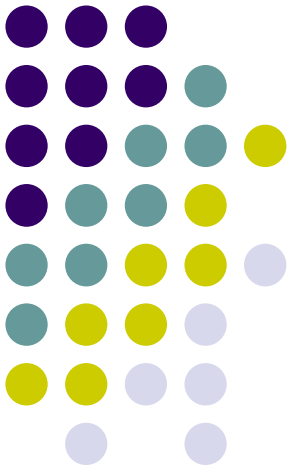


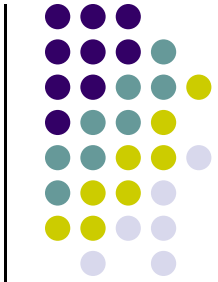
TABLE 7-24 DAA Instruction Correction Values

Initial C Bit	Value of A[7:4]	Initial H Bit	Value of A[3:0]	Correction Value	Corrected C Bit
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-2	1	0-3	66	1

Miscellaneous Instructions

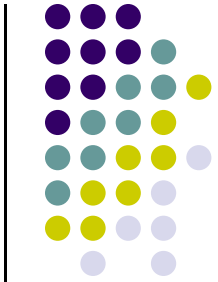


Miscellaneous Instructions



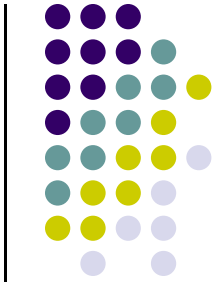
- Long Branch
  - LBXX - Relative address is 16-bits
- Call/RTC
  - Jump to subroutine in extended memory
- SEX – Sign Extend A,B, CCR
- Fuzzy Logic Instructions

NOP Instruction



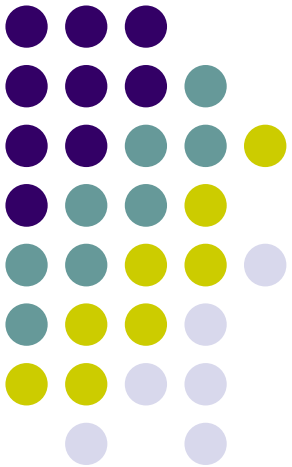
- NOP – No operation
  - Performs “no operation”
  - Can be used for “crude” timing routines
    - 2 Clock Cycles for each instruction
  - Also used for debugging
    - Insert NOPs in place of SWI
      - Software Interrupts

STOP Instruction

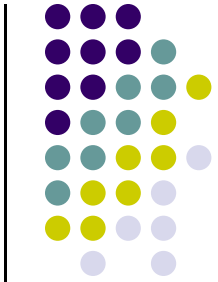


- STOP– STOP operation
  - Puts the CPU “to sleep”
    - Can be “awakened” by using an interrupt
  - Can be disabled by setting the STOP bit

Interrupt Instructions



Interrupt Instructions



- CLI– Clear Interrupt Mask
  - Clears I bit to 0
- SEI – Set Interrupt Mask
  - Sets I bit to 0
- RTI – Return from Interrupt
- SWI – Software Interrupt
- WAI – Wait for Interrupt

End of Section