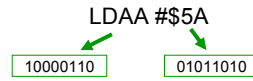


Programming

- A. **Assembly and Other Programming Lang.**
- B. **Source Code, Object Code, and the Assembler**
- C. **C Language for Microcontrollers**
- D. **Fetch/Execute Operations of CPU**
- E. **The Instruction Set and Addressing Modes**
- F. **68HC11 Instruction Set**
- G. **Microcontroller Arithmetic and the CCR**
- H. **Program Flow Control Using Looping & Branching**

Assembly and Other Programming Languages

- **Machine language:** binary encoding of instructions that are executed by a CPU
 - each CPU has its own machine language
 - Ex: %10000110; %01011010
- **Assembly language:** machine instructions are represented into a mnemonic form
 - mnemonic form is then converted into actual processor instructions and associated data
 - Ex:



Assembly and Other Programming Languages

- Disadvantages of AL
 - require knowledge of the processor architecture and instruction set
 - many instructions are required to achieve small tasks
 - source programs tend to be large and difficult to follow
 - programs are machine dependent => requiring complete rewriting if the hardware is changed

Assembly and Other Programming Languages

- High level languages
 - human like languages
 - Ex: Basic, Pascal, FORTRAN, Ada, Cobol, C, Java
 - C: most common high-level language for microcontroller development
- Advantages
 - portable
 - the source program is translated into machine code for each type of CPU
 - What is different is the translator not the program

Programming

- A. Assembly and Other Programming Lang.
- B. Source Code, Object Code, and the Assembler
- C. C Language for Microcontrollers
- D. Fetch/Execute Operations of CPU
- E. The Instruction Set and Addressing Modes
- F. 68HC11 Instruction Set
- G. Microcontroller Arithmetic and the CCR
- H. Program Flow Control Using Looping & Branching

Source Code, Object Code, and the Assembler

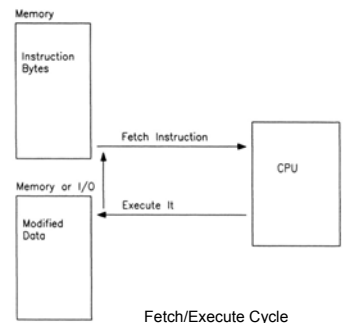
- Machine language
- Assembly language
- Examples
- Manual assembly
- The simulator

Source Code, Object Code, and the Assembler

- Source code
 - Represents original program before it is translated
 - Stored as a file
 - Assembly program applications are more efficient than one written in a high level languages
- Efficiency
 - refers to code size, execution size, energy consumption

Machine Language

- Machine language can directly control the microcontroller's resources
- Ex: LDAA #\$5A
- Code must be stored in memory
 - \$E000: \$86
 - \$E001: \$5A

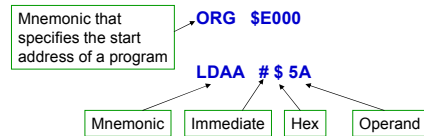


Machine Language

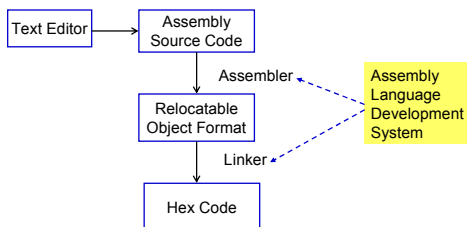
- Instruction: **opcode; operand**
- Opcode specifies the type of operation
 - 68HC11 uses 1-byte and 2-bytes opcodes
 - 2-byte opcodes are composed of a prebyte followed by the real opcode
- Operand tells the CPU what data to operate on
 - An instruction may have 0, 1, 2 or 3 operand bytes

Assembly Language

- Assembly language programs use mnemonics and are typed using a text editor
- Machine code must be stored in memory



Assembly Language



Assembly Language

- Why do we need a linker?
 - one writes the source code in smaller sections
 - the linker helps to develop large applications
- Line assemblers
 - translate source code directly into machine code
- Disassembler
 - translating program that reverses the machine code into the assembly source code

Examples



PROGRAM:
a. read input
b. subtract an offset
c. store the result

* Assembly language program
ORG \$E000

```
LDA $1031 or LDA COOLANT_TEMP
SUBA #$20 or SUBA #CT_OFFSET
STAA $D004 or STAA STORE_TEMP
```

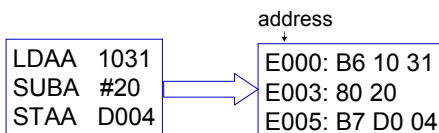
Examples

The labels are assigned using assembler directives:

```
COOLANT_TEMP EQU $1031
CT_OFFSET EQU $20
STORE_TEMP EQU $D004
```

Manual Assembly

Using the manual of the instruction set summary we can convert source code into hex code



By convention machine code is always hex

The Simulator

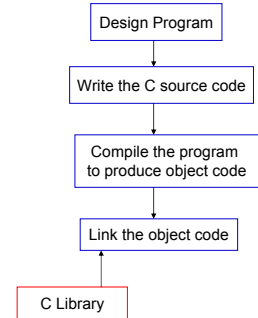
- A microcontroller simulator is a software tool that permits users to simulate the operation of a microcontroller
- The book contains the demo version of the THRSim 11

Programming

- A. Assembly and Other Programming Lang.
- B. Source Code, Object Code, and the Assembler
- C. **C Language for Microcontrollers**
- D. Fetch/Execute Operations of CPU
- E. The Instruction Set and Addressing Modes
- F. 68HC11 Instruction Set
- G. Microcontroller Arithmetic and the CCR
- H. Program Flow Control Using Looping & Branching

C Language for Microcontrollers

- High level languages
 - compiled languages
 - interpreted
- Why C is popular?
 - combines the best of both, the high-level language and the assembly language
 - has features to allow direct control of I/O which is very important for microcontroller applications



C Language for Microcontrollers

- Most C compilers for microcontrollers follow the early standard defined by Kernigham and Ritchie in 1978
- Normally the assembly language created by the C compiler is less efficient however, using C the development of large applications is easier

Programming

- A. Assembly and Other Programming Lang.
- B. Source Code, Object Code, and the Assembler
- C. **C Language for Microcontrollers**
- D. **Fetch/Execute Operations of CPU**
- E. The Instruction Set and Addressing Modes
- F. 68HC11 Instruction Set
- G. Microcontroller Arithmetic and the CCR
- H. Program Flow Control Using Looping & Branching

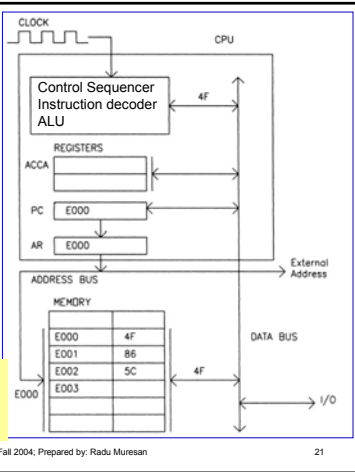
Fetch/Execute Operation of CPU

Clear Accumulator A
Load Accumulator A

CLRA
LDAA #5C

E000: 4F
E001: 86 5C

CPU operations
Fetching CLRA
AR: address reg.

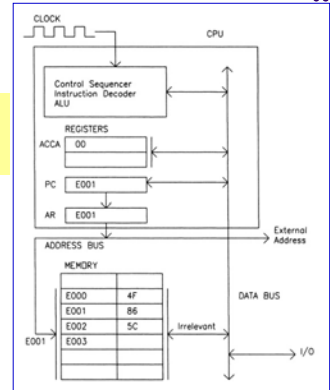


ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

21

Fetch/Execute Operation of CPU

CPU operation
Executing the first instruction CLRA

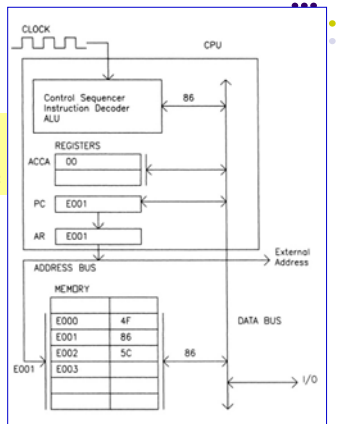


ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

22

Fetch/Execute Operation of CPU

CPU Operation
Fetching the second instruction opcode LDAA#



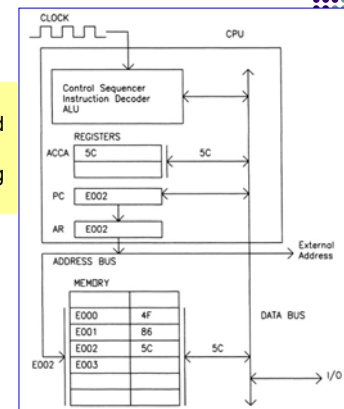
ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

23

Fetch/Execute Operation of CPU

CPU operation
Fetching the second instruction operand (\$5C) and executing the instruction

Note: PC increments to point to the next byte to be fetched



ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

24

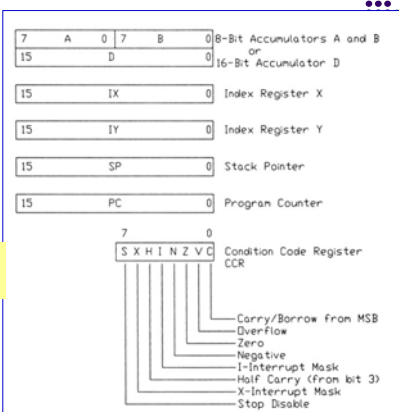
Programming

- A. Assembly and Other Programming Lang.
- B. Source Code, Object Code, and the Assembler
- C. C Language for Microcontrollers
- D. Fetch/Execute Operations of CPU
- E. **The Instruction Set and Addressing Modes**
- F. 68HC11 Instruction Set
- G. Microcontroller Arithmetic and the CCR
- H. Program Flow Control Using Looping & Branching

The Instruction Set and Addressing Modes

- Instruction set references
- Types of instructions
- Addressing modes
- The prebyte
- Inherent addressing mode
- Listing and execution conventions
- Stopping a program
- Immediate addressing mode
- Direct and extended addressing modes
- Indexed addressing mode
- Memory dump convention

Instruction Set References



Programming model of the 68HC11

Instruction Set References

Source Format	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal)	Hex. Code	Condition Codes
				Opcode Operand(s)	16 8 4 2 1	16 X H I N Z V C
ABA	Add Accumulators	$A + B \rightarrow A$	INH	1B	1 2	- - - - - - - - -
ABX	Add B to X	$IX + 00:B \rightarrow IX$	INH	3A	1 3	- - - - - - - - - -
ABY	Add B to Y	$IY + 00:B \rightarrow IY$	INH	18 3A	2 4	- - - - - - - - - -
ADCA (opri)	Add with Carry to A	$A + M + C \rightarrow A$	A IMM A DIR A EXT A IND,X A IND,Y	89 99 dd 89 hh A9 ff 18 A9 ff	2 2 2 3 3 4 2 4 3 5	- - - - - - - - - -
ADCB (opri)	Add with Carry to B	$B + M + C \rightarrow B$	B IMM B DIR B EXT B IND,X B IND,Y	C9 D9 dd F9 hh E9 ff 18 E9 ff	2 2 2 3 3 4 2 4 3 5	- - - - - - - - - -

The instruction set summary can give enough information for using the assembly language

Instruction Set References

Instruction set summary. Operand Notations

dd = 8-bit direct address (\$0000-\$00FF)
(High byte assumed to be \$00)
ff = 8-bit positive offset \$00 (0) to \$FF (256)
(Is added to the index)
hh = high order byte of 16-bit extended address
ii = one byte of immediate data
jj = high order byte of 16-bit immediate data
kk = low order byte of 16-bit immediate data
ll = low order byte of 16-bit extended address
mm = 8-bit bit mask (Set bits to be affected)
rr = signed relative offset \$80(-128) to \$7F(+128)
(Offset relative to the address following the
machine code offset byte)

Types of Instructions

- Data handling
- Arithmetic
- Logic
- Data test
- Jump and branch
- Conditional code

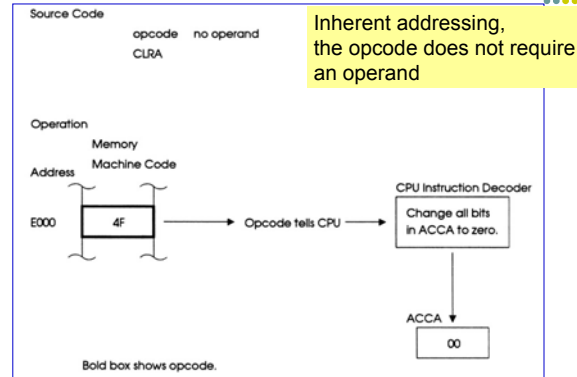
Addressing Modes

- Inherent
- Immediate
- Extended
- Direct
- Indexed
- Relative

Prebyte

- Opcodes based on the earlier 6801 microcontroller have a single byte
- New instructions + any instruction dealing with index register Y have 2-byte opcodes
- A few hex numbers were reserved for the first opcode to specify that the following byte is also part of the opcode

Inherent Addressing Mode



ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

33

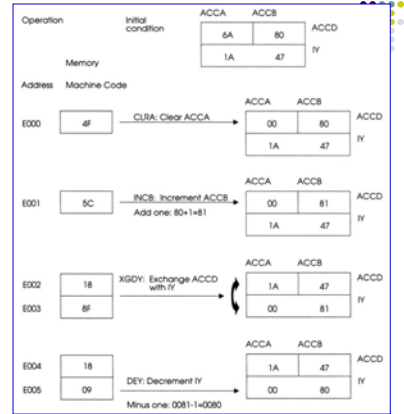
Inherent Addressing Mode

Operations
Initial condition
0 → A

B+1 → B

Y → D, D → Y

Y-1 → Y



ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

34

Listing and Executing Conventions

```
ORG $E000 ;define start address
E000 4F CLRA ;clears ACCA
E001 5C INCB ;increment ACCB
E002 18 8F XGDY ;swap ACCD with IY
E004 18 09 DEY ;decrement IY
```

Listing

PC	ACCA	ACCB	IY	Operation
E000	6A	80	1A47	Initial cond.
E001	00	80	1A47	0 → A
E002	00	81	1A47	B+1 → B
E004	1A	47	0081	Y → D, D → Y
E006	1A	47	0080	Y-1 → Y

Program trace

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

35

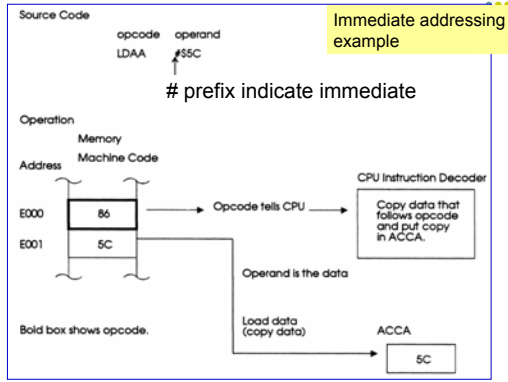
Stopping a Program

- STOP: stop processing
 - Opcode: CF
 - stops internal clocks, puts the processor in the standby mode
 - recovery from STOP may be accomplished by RESET[^] or an interrupt
 - there are potential problems with the STOP instruction (M68HC11)
 - insert a NOP before STOP
- BRA *
 - BRA (rel)
 - Branch always
 - Opcode: 20
 - Operation: PC ← (PC) + \$002 + rr
 - Operands: rr
 - Note: rr represents signed relative offset
 - The assembler obtains the relative address, **Rel**, from the absolute address and the current value of the location counter

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

36

Immediate Addressing Mode



ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

37

Immediate Addressing Mode

Machine Code	Label	Operation	Operand	Comments
	CAT	EQU	7	CAT SAME AS 7
		ORG	\$1000	SET LOCATION COUNTER
	REGS	EQU	*	ADDR(REGS) IS \$1000
86	16	LDAA	#22	DECIMAL 22 → ACCA (\$16)
C8	34	EORB	#\$34	XOR (\$34,ACCB) → ACCB
81	24	CMPA	##%100100	RIGHT ALIGNED BINARY
86	07	LDAA	#CAT	7 → ACCA
CC	12	LDD	#\$1234	
CC	00	LDD	#7	7 → ACCA:ACCB
86	12	LDAA	##i: 22	OCTAL
86	41	LDAA	##'A	ASCII
CE	10	LDX	##REGS	ADDR(REGS) → X

Program example containing instructions that use immediate addressing

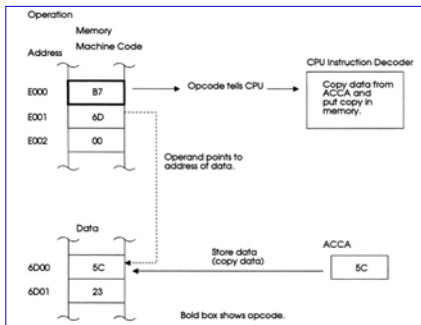
ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

38

Extended Addressing Mode

STAA \$D00 ; two-byte operand for extended mode

no # prefix



39

Extended Addressing Mode

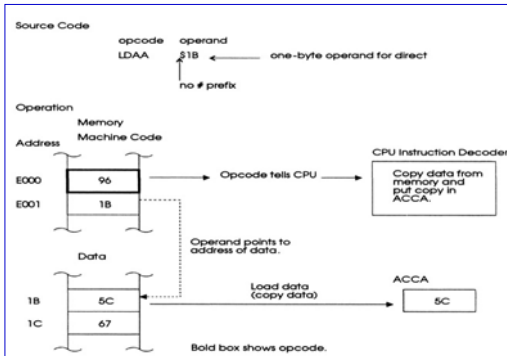
- In the extended addressing mode, the effective address (EA) of the instruction appears explicitly in the two bytes following the opcode
- The length of the most instructions using extended mode is 3 bytes

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

40

Direct Addressing Mode

LDAA \$1B ; one-byte operand that is an address



Direct Addressing Mode

- The least significant byte of the effective address (EA) of the instruction appears in the byte following the opcode
- The high-order byte of the EA is assumed to be \$00
- Direct page is defined by the EA limits: \$0000-\$00FF
- Advantages
 - the execution time is reduced by one cycle
 - In the M68HC11 MCU the software can configure the memory map so that internal RAM, and/or internal registers or external memory space can occupy these addresses

ENGG4640/3640, Fall 2004; Prepared by: Radu Muresan

42

Direct and Extended Addressing Modes

Machine Code	Label	Operation	Operand	Comments
B3 00 12	CAT	SUBD EQU	CAT \$12	FWD REF TO CAT DEFINE CAT = \$12
93 12		SUBD	CAT	BKWD REF TO CAT
7F 00 12		CLR	CAT	EXTENDED ONLY

First reference to CAT is a forward reference and the assembler selected the extended mode

Second reference, the assembler knows the symbol value ⇒ direct mode

Last reference, extended mode since CLR has no direct mode

Some assemblers allow the direct or extended addressing modes to be forced by using < or > before the operand

ENGG4640/3640; Fall 2004; Prepared by: Radu Muresan

43

Direct and Extended Addressing Modes

- Accessing input/output ports

- Each bit of an I/O port corresponds to an external pin of the I/O port
- Two of 68HC11 ports are port B (output) and port C (input)
- The addresses of their corresponding registers are \$1004 and \$1003

LDAA #A5 ;load data to be outputted
STAA \$1004 ;send it to port B
LDAA \$1003 ;read data from port C

ENGG4640/3640; Fall 2004; Prepared by: Radu Muresan

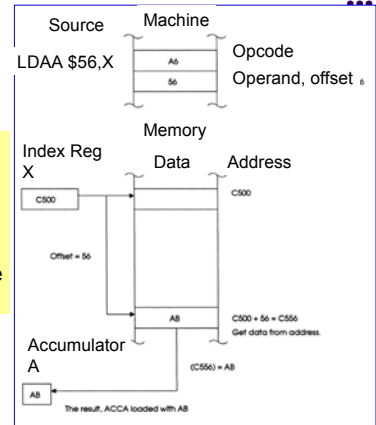
44

Indexed Addressing Mode

- In the index addressing mode, either index register X or Y is used in calculating the effective address (EA)
- EA is variable and depends on the content of index registers X or Y and a fixed, 8-bit, unsigned offset contained in the instruction
- Dynamic single-byte offsets are facilitated by the use of the add accumulator B to index register X (ABX) instr.
- More complex address calculations can be obtained by the use of instructions XGDX and XGDY

Indexed Addressing Mode

Access on-chip registers:
initialize the index register to the starting address of the register block and use an 8-bit offset



Index Addressing Mode

```
ORG $E000 ;start address
```

```
LDAA $00,X ;load with indexed mode
ADDA $01,X ;add with indexed mode
STAA $20,Y ;store with indexed mode
ABY ;an inherent mode
* ;instruction to modify IY
INY ;another one which
* ;increments IY
STAA $30,Y ;again, store using
* ;indexed mode
BRA * ;stop program
```

OPERATIONS

```
(X+0) → A
A+(X+1) → A
A → (Y+20)
B+Y → Y
Y+1 → Y
A → (Y+30)
```

Index Addressing Mode For look-up table applications

```
ORG $E000
;address $30 contains the data from the sensor
LDAB $30 ;get stored differential
;pressure signal
LDX #$B600 ;point to square root table
ABX ;look up its square root
LDAA $00,X ;and load it to find flow rate
```

$$F = K\sqrt{h}$$

Microcontroller reads a signal from a sensor and stores it at address \$30
Block of memory starting at \$B600 contains the square root of all single-byte numbers

Programming

- A. Assembly and Other Programming Lang.
- B. Source Code, Object Code, and the Assembler
- C. C Language for Microcontrollers
- D. Fetch/Execute Operations of CPU
- E. The Instruction Set and Addressing Modes
- F. **68HC11 Instruction Set**
- G. Microcontroller Arithmetic and the CCR
- H. Program Flow Control Using Looping & Branching

M68HC12 Versus M68HC11 Similarities

- The programmer's model and interrupt stacking order for the CPU12 are identical to that of M68HC11
- All source code for the M68HC11 is accepted by CPU12 with no modifications
- Most M68HC11 instructions even assemble to the same object code on the CPU12

M68HC12 Versus M68HC11 Improvements

- Most obvious improvement is that the CPU12 is a 16-bit processor with an ALU that is 20 bits wide for some operations
- All data busses in the M68HC12 are 16-bits
- External bus interface is normally 16-bits although a narrow 8-bit option can be selected
- There is an instruction queue (similar to a pipeline) that caches program information so that at least 2 more bytes of object code are visible to CPU at the start of execution

M68HC12 Versus M68HC11 Improvements

- Due to the queue feature, many instructions can execute in one cycle with no delay for fetching additional program information
- The indexed addressing mode has been enhanced
 - in M68HC11 Y-relative indexed instructions had an extra prebyte. As a result the instructions have an extra byte and an extra clock cycle
 - In CPU12, all indexed instructions have an opcode (a postbyte, and 0, 1, 2 extension bytes)
- Also there are new instruction and basically the old set was enhanced

M68HC12

- CPU12 programming reference:
 - Todd D. Morton, "Embedded Microcontrollers," Prentice Hall, 2001 (Chapters 4 & 5).

68HC11 Instruction Set

- Accumulator and memory instructions
- Stack and index register instructions
- Condition code register instructions
- Program control instructions

Accumulator and Memory Instructions

- Most of these instructions use two operands
 - one operand is either an accumulator or and index register
 - whereas, the second operand is usually obtained from memory
- Subgroups
 - loads, stores and transfers
 - arithmetic operations
 - multiply and divide
 - logical operations
 - data testing and bit manipulation
 - shifts and rotates

Loads, Stores, and Transfers

Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Clear Mem Byte	CLR			*	*	*	
Clear Accum A	CLRA						*
Clear Accum B	CLRB						*
Load Accum A	LDAA	*	*	*	*	*	
Load Accum B	LDAB	*	*	*	*	*	
Load D Accum	LDD	*	*	*	*	*	
Pull A from Stack	PULA						*
Pull B from Stack	PULB						*
Push A onto Stack	PSHA						*
Push B onto Stack	PSHB						*

Loads, Stores, and Transfers

Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Store Acc A	STAA	*	*	*	*	*	
Store Acc B	STAB	*	*	*	*	*	
Store D Acc	STD	*	*	*	*	*	
Transfer A to B	TAB						*
Transfer A to CCR	TAP						*
Transfer B to A	TBA						*
Transfer CCR to A	TPA						*
Exchange D with X	XGDX						*
Exchange D with Y	EGDY						*

Arithmetic Operations

Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Add Acc	ABA						*
Add Acc B to X	ABX						*
Add Acc B to Y	ABY						*
Add with Carry to A	ADCA	*	*	*	*	*	
Add with Carry to B	ADCB	*	*	*	*	*	
Add Memory to A	ADDA	*	*	*	*	*	
Add Memory to B	ADDB	*	*	*	*	*	
Add Mem to D (16b)	ADDD	*	*	*	*	*	
Compare A to B	CBA						*
Compare A to Mem	CMPA	*	*	*	*	*	
Compare B to Mem	CMPB	*	*	*	*	*	
Compare D to Mem	CPD	*	*	*	*	*	
Decimal Adjust A	DAA						*
Decrement Mem Byte	DEC			*	*	*	
Decrement Acc A	DECA						*

Arithmetic Operations

Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Decrement Acc B	DECB						*
Increment Mem Byte	INC			*	*	*	
Increment Acc A	INCA						*
Increment Acc B	INCB						*
Twos Compl Mem Byte	NEG			*	*	*	
Twos Compl Acc A	NEGA						*
Twos Compl Acc B	NEGB						*
Subtract w Carry from A	SBCA	*	*	*	*	*	
Subtract w Carry from B	SBCB	*	*	*	*	*	
Subtract Mem from A	SUBA	*	*	*	*	*	
Subtract Mem from B	SUBB	*	*	*	*	*	
Subtract Mem from D	SUBD	*	*	*	*	*	
Test for Zero or Minus	TST			*	*	*	
Test for Zero or Minus A	TSTA						*
Test for Zero or Minus B	TSTB						*

Arithmetic Operations

- Example
- Add the following numbers:

$$\begin{array}{r} \$00CC+ \\ \$3276 \\ \hline \$3342 \end{array}$$
- 68HC11 can add 2 bytes or 2 double bytes
- The following is the single byte addition program

Memory data:

0000: 00
0001: CC
0002: 32
0003: 76
0004: xx
0005: xx

LDAA \$01 ; load CC to A
ADDA \$03 ; add A to 76
STAA \$05 ; store result
LDAA \$00 ; load 00 to A
ADCA \$02 ; add w carry A to 32
STAA \$04 ; store the result

Multiply and Divide

Function	Mnemonic	INH
Multiply (AxB=>D)	MUL	X
Fractional Divide (D/X => X; r=>D)	FDIV	X
Integer Divide (D/X => X; r=>D)	IDIV	X

Logical Operations

Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
AND A with Memory	ANDA	*	*	*	*	*	
AND B with Memory	ANDB	*	*	*	*	*	
Bit(s) Test A with Mem	BITA	*	*	*	*	*	
Bit(s) Test B with Mem	BITB	*	*	*	*	*	
Ones Comp Mem Byte	COM			*	*	*	
Ones Complement A	COMA						*
Ones Complement B	COMB						*
OR A with Mem (excl)	EORA	*	*	*	*	*	
OR B with Mem (excl)	EORB	*	*	*	*	*	
OR A with Mem (incl)	ORAA	*	*	*	*	*	
OR B with Mem (incl)	ORAB	*	*	*	*	*	

Logic Operations

- Perform Boolean operations AND, OR, and EXCLUSIVE OR for each bit in a byte

0000: xx xx ...	LDD \$06 ; loads 9D to A and 9C to B
0006: 9D	ANDA #\$A5 ;ands A with A5
0007: 9C	ORAB #\$A5 ; ors B with A5
0008: A5	EORB \$08 ; xors B with mem(08)
0009: xx xx ...	

Data Testing and Bit Manipulation

Function	Mnemonic	IMM	DIR	EXT	IX	IY
Bit(s) Test A with Mem	BITA	*	*	*	*	*
Bit(s) Test B with Mem	BITB	*	*	*	*	*
Clear Bit(s) in Mem	BCLR		*		*	*
Set Bit(s) in Mem	BSET		*		*	*
Branch if Bit(s) Clear	BRCLR		*		*	*
Branch if Bit(s) Set	BRSET		*		*	*

Shifts and Rotates



Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Arithm Shift Left Mem	ASL			*	*	*	
Arithm Shift Left A	ASLA						*
Arithm Shift Left B	ASLB						*
Arithm Shift Left Double	ASLD						*
Arithm Shift Right Mem	ASR			*	*	*	
Arithm Shift Right A	ASRA						*
Arithm Shift Right B	ASRB						*
(Logical Shift Left Mem)	(LSL)			*	*	*	
(Logical Shift Left A)	(LSLA)						*
(Logical Shift Left B)	(LSLB)						*
(Logical Shift Left Dou)	(LSLD)						*

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

69

Shifts and Rotates

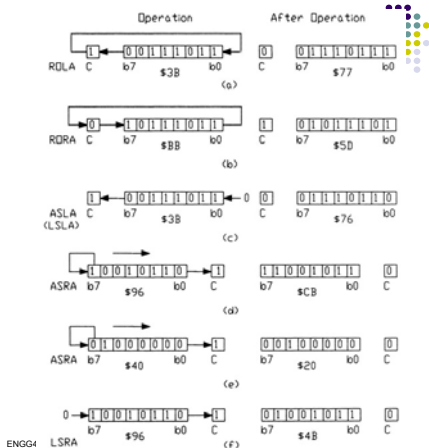


Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Logical Shift Right Mem	LSR			*	*	*	
Logical Shift Right A	LSRA						*
Logical Shift Right B	LSRB						*
Logical Shift Right Dou	LSRD						*
Rotate Left Mem	ROL			*	*	*	
Rotate Left A	ROLA						*
Rotate Left B	ROLB						*
Rotate Right Mem	ROR			*	*	*	
Rotate Right A	RORA						*
Rotate Right B	RORB						*

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

70

Shift and Rotate Instructions



ENGG4



Alarm Application



Problem: A set of 8 alarm sensors are connected to port C. Count how many alarm sensors are activated.

NOTE: The address of port C is \$1003.

```

* Count the number of alarm sensors that are on.
  ORG $E000 ;start address of program

  CLC      ;make sure that bit C is initially off
  CLRB    ;make sure that ACCB is initially zero
  LDAA $1003 ;get alarm inputs from port C
  *      ;and load them into ACCA
  ... continue next slide
    
```

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan

72

Alarm Application



- * Shift bits into ACCA (port C data) eight times and increment ACCB
- * each time a high bit is found (by adding with carry).

```

LSLA          ;C = PC7
ADCB # $00   ;add 1 if C set
LSLA          ;C = PC6
ADCB # $00   ;add 1 if C set
LSLA          ;C = PC5
ADCB # $00   ;add 1 if C set
LSLA          ;C = PC4
ADCB # $00   ;add 1 if C set
    
```

How many alarm sensors are activated?

```

LSLA          ;C = PC3
ADCB # $00   ;add 1 if C set
LSLA          ;C = PC2
ADCB # $00   ;add 1 if C set
LSLA          ;C = PC1
ADCB # $00   ;add 1 if C set
LSLA          ;C = PC0
ADCB # $00   ;add 1 if C set
BRA *        ;stop program
    
```

Stack and Index Register Instructions



Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Add Accum B to X	ABX						*
Add Accum B to Y	ABY						*
Compare X to Mem (16b)	CPX	*	*	*	*	*	
Compare Y to Mem (16b)	CPY	*	*	*	*	*	
Decrement Stack Pointer	DES						
Decrement Index Reg X	DEX						*
Decrement Index Reg Y	DEY						*
Increment Stack Pointer	INS						*
Increment Index Reg X	INX						*
Increment Index Reg Y	INY						*
Load Index Reg X	LDX	*	*	*	*	*	
Load Index Reg Y	LDY	*	*	*	*	*	
Load Stack Pointer	LDS	*	*	*	*	*	

Stack and Index Register Instructions



Function	Mnemonic	IMM	DIR	EXT	IX	IY	INH
Pull X from Stack	PULX						*
Pull Y from Stack	PULY						*
Push X onto Stack	PSHX						*
Push Y onto Stack	PSHY						*
Store Index Reg X	STX	*	*	*	*	*	
Store Index Reg Y	STY	*	*	*	*	*	
Store Stack Pointer	STS	*	*	*	*	*	
Transfer SP to X	TSX						*
Transfer SP to Y	TSY						*
Transfer X to SP	TXS						*
Transfer Y to SP	TYS						*
Exchange D with X	XGDX						*
Exchange D with Y	XGDY						*

Condition Code Register Instructions



Function	Mnemonic	INH
Clear Carry Bit	CLC	*
Clear Interrupt Mask Bit	CLI	*
Clear Overflow Bit	CLV	*
Set Carry Bit	SEC	*
Set Interrupt Mask Bit	SEI	*
Set Overflow Bit	SEV	*
Transfer A to CCR	TAP	*
Transfer CCR to A	TPA	*

Function	Mnemonic	REL	DIR	IX	IY	Comments
Branch if C Clear	BCC	*				C=0?
Branch if C Set	BCS	*				C=1?
Branch if = Zero	BEQ	*				Z=1?
Branch if > or =	BGE	*				Signed >=
Branch if >	BGT	*				Signed >
Branch if Higher	BHI	*				Unsigned >
B if Higher or same	BHS	*				Unsigned >=
Branch if < or =	BLE	*				Signed <=
Branch if Lower	BLO	*				Unsigned <
B if Lower or same	BLS	*				Unsigned <=
Branch if Less Than	BLT	*				Signed <
Branch if Minus	BMI	*				N=1?
Branch if not =	BNE	*				Z=0?
Branch if Plus	BPL	*				N=0?
B if Bit(s) Clear in M	BRCLR		*	*	*	Bit Manipul
Branch Never	BRN	*				3-cycle NOP
B if Bit(s) Set in Mem	BRSET		*	*	*	Bit Manipul
B if Overflow Clear	BVC	*				V=0?
B if Overflow Set	BVS	*				V=1?

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan 77

Jumps; Subroutine Calls and Returns; Interrupt Handling; and Others

Function	Mnemonic	REL	DIR	EXT	IX	IY	INH
Jump	JMP		*	*	*	*	
...							
Branch to Subroutine	BSR	*					
Jump to Subroutine	JSR		*	*	*	*	
Return from Subroutine	RTS						*
...							
Return from Interrupt	RTI						*
Software Interrupt	SWI						*
Wait for Interrupt	WAI						*
...							
No Operation (2-cycle)	NOP						*
Stop Clocks	STOP						*
Test	TEST						*

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan 78

Programming

A.	Assembly and Other Programming Lang.
B.	Source Code, Object Code, and the Assembler
C.	C Language for Microcontrollers
D.	Fetch/Execute Operations of CPU
E.	The Instruction Set and Addressing Modes
F.	68HC11 Instruction Set
G.	Microcontroller Arithmetic and the CCR
H.	Program Flow Control Using Looping & Branching

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan 79

Microcontroller Arithmetic and the CCR

- Two's complement and the sign bit
- Carry, overflow, zero, and half carry
- Binary-coded-decimal (BCD) arithmetic
- Multiplication
- Integer division
- Fractional division
- Floating point numbers

ENGG4640/3640, Fall 2004, Prepared by: Radu Muresan 80

Two's Complement and the Sign Bit

- The MSB of a binary number is also called the signed bit
- Many 68HC11 instructions treat bytes as signed integers
- The double accumulator instructions assume a 16-bit signed number
- A negative result sets the N flag of CCR to 1

Method 1 for 2's complement

Ex. A = %11011000
 2's A = $2^8 - A$

$$\begin{array}{r} \%100000000 - \\ 11011000 \\ \hline 00101000 \end{array}$$

Method 2 for 2's complement

Ex. A = %11011000
 2's(A) = 1's(A) + 1
 1's(A) = %00100111
 1's(A) + 1 = %00101000

Carry, Overflow, Zero, and Half Carry

- 8-bit unsigned addition and subtraction
 - Carry flag (C) indicates a carry or a borrow when adding or subtracting unsigned 8-bit integers
 - Half carry (H) is set if there is a carry from bit 3 to bit 4
 - Zero flag (Z) is set when the result of the operation is zero
 - Comparison: equality when Z = 1

Carry, Overflow, Zero, and Half Carry

- 8-bit signed addition and subtraction
 - Range of operation -128 (\$80) to +127 (\$7F)
 - Overflow (V) is set if the result of adding or subtracting exceeds the range
 - N is the copy of the MSB
 - V is different than C

Overflow occurs in the following situations:

+ve plus +ve equals -ve
 -ve plus -ve equals +ve
 +ve minus -ve equals -ve
 -ve minus +ve equals +ve

Carry, Overflow, Zero, and Half Carry

Add (R = X plus M)

$$\begin{aligned} H &= X_3 \cdot M_3 + M_3 \cdot \overline{R_3} + \overline{R_3} \cdot X_3 \\ N &= R_7 \\ Z &= \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0} \\ V &= X_7 \cdot \overline{M_7} \cdot \overline{R_7} + \overline{X_7} \cdot M_7 \cdot R_7 \\ C &= X_7 \cdot M_7 + M_7 \cdot \overline{R_7} + \overline{R_7} \cdot X_7 \end{aligned}$$

ADCA, ADCB, ADDA, ADDB

Set if carry from bit 3.
 Set if MSB of result is set.
 Set if result is \$00.
 Set if two's-complement overflow results.
 Set if carry from MSB.

Subtract (R = X minus M)

$$\begin{aligned} N &= R_7 \\ Z &= \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0} \\ V &= X_7 \cdot \overline{M_7} \cdot R_7 + \overline{X_7} \cdot M_7 \cdot \overline{R_7} \\ C &= \overline{X_7} \cdot M_7 + M_7 \cdot \overline{R_7} + R_7 \cdot \overline{X_7} \end{aligned}$$

SBCA, SBCB, SUBA, SUBB

Set if MSB of result is set.
 Set if result is \$00.
 Set if two's-complement overflow results.
 Set if $|MI| > |XI|$ hence borrow required.

Definitions

· Boolean AND
 + Boolean OR
 - Boolean NOT
 Xi Bit i (0 to 7) of accumulator (A or B)
 Mi Bit (0 to 7) of memory (operand) byte
 Ri Bit i (0 to 7) of result
 MSB Most significant bit (bit 7 for byte operations)
 | | Absolute value

Flags

H Half carry
 N Negative
 Z Zero
 V Overflow
 C Carry

Binary-Coded-Decimal (BCD) Arithmetic



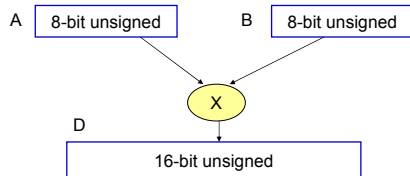
- BCD is a binary number system that uses 4-bit binary representation for the decimal digits
- BCD binary arithmetic must be adjusted for valid BCD results
- Instruction used for decimal adjust is: DAA
- DAA must follow immediately the arithmetic instruction (ABA, ADD, or ADC)

Binary-Coded-Decimal (BCD) Arithmetic



State of C Bit Before DAA (Column 1)	Upper Half-Byte of ACCA (Bits 7-4) (Column 2)	Initial Half-Carry H Bit from CCR (Column 3)	Lower Half-Byte of ACCA (Bits 3-0) (Column 4)	Number Added of ACCA by DAA (Column 5)	State of C Bit After DAA (Column 6)
0	0-9	0	0-9	00	0
0	0-B	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	0-3	1	0-3	66	1

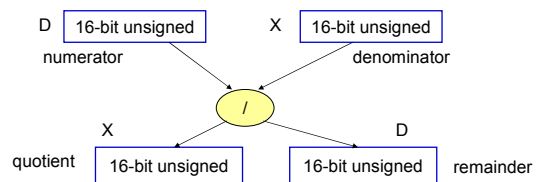
Multiplication: MUL unsigned



The **carry flag** allows rounding the most significant byte of the result through the sequence: MUL, ADCA #0

Condition codes: C set if bit 7 of the result (ACCB) is set; cleared otherwise

Unsigned Integer Division: IDIV



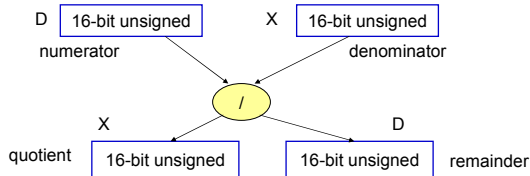
Condition codes

Z: set if result is \$0000; cleared otherwise

V: 0, cleared

C: set if denominator was \$0000; cleared otherwise

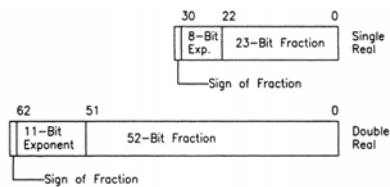
Unsigned Fractional Divide: FDIV



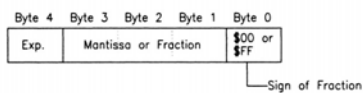
Unsigned Fractional Divide: FDIV

- Assumption: numerator < denominator
 - The radix point is to the left of bit 15 for the quotient
 - In case of overflow the quotient is set to \$FFFF => 0.99998
 - The remainder can be resolved into a binary-weighted fraction
 - A result of \$0001 => 0.000015; and \$FFFF => 0.99998
- **Condition codes**
 - Z: set if result is \$0000; cleared otherwise
 - V: set if denominator was less than or equal to the numerator; cleared otherwise
 - C: set if denominator was \$0000; cleared otherwise

Floating Point Numbers



Some Floating-Point Formats for IEEE 754



Format for the 68HC11 Floating-Point Package

Programming

- Assembly and Other Programming Lang.
- Source Code, Object Code, and the Assembler
- C Language for Microcontrollers
- Fetch/Execute Operations of CPU
- The Instruction Set and Addressing Modes
- 68HC11 Instruction Set
- Microcontroller Arithmetic and the CCR
- Program Flow Control Using Looping & Branching

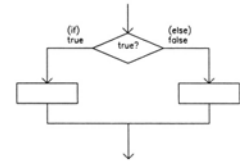
Program Flow Control Using Looping and Branching

- Flow control
- Conditional branches
- Relative addressing
- Secondary memory reference instructions
- Jump instructions
- Relocatable programs

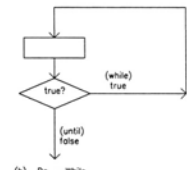
Flow Control

The flow control structures showed in the figures are easy to program using assembly language

In high-level languages, there are variations of these structures.



(a) If - Else



(b) Do - While

Conditional Branches

- * Listing 2.11
- * Demonstrate branch instructions
- * BEQ - Check if Z = 0
- * If (\$00) is zero then skip next two instructions
- * else load (\$01) and store it in (\$00)
- * BRA - It's always true!
- * If true then execute itself again.

```

ORG   $E000   ;start address
LDAA  $00
BEQ   THERE  ;if ACCA == 0 branch to THERE
LDAB  $01    ;($01) -> ($00)
STAB  $00    ;when ACCA not 0
THERE STAA $01 ;($00) -> ($01)
HERE  BRA   HERE ;always branch to HERE
  
```

When the conditional branches are used it is important to ensure that the CCR is not corrupted before ex. the branch

Conditional Branches. If-Else

- * Listing 2.12
- * Demonstrate if-else
- * if carry set then write 'T' to \$00
- * else write 'F' to \$00
- * in this case, we define TRUE to be condition when carry is set and we define FALSE to be when carry is not set

```

ORG   $E000
BCS   TRUE  ;true if C set
FALSE LDAB 'F' ;else false if C clear
      BRA   SKIP ;and skip next
      ;instruction
TRUE  LDAB 'T' ;executed if true (C=1)
SKIP  STAB $00 ;store appropriate result
HERE  BRA   HERE ;stop program
  
```

Conditional Branches. If-Else



- * Demonstrate a do-while (DOWH) loop
- * Move 16 bytes starting at address \$C000 to a block at address \$C500. Use the following strategy.
- * do a transfer of a byte while counter is not zero
- * We also introduce some common entities, counters and pointers. In this case, ACCB is a counter
- * IX is a source (src) pointer and IY is a destination (dst) pointer. When we say "move," it really means "copy"!

Conditional Branches. If-Else



```
*-----*
      ORG   $E000
      LDX  #$C000;initialize src pointer
      LDY  #$C500;initialize dst pointer
      LDAB #$10  ;initialize counter
DOWH  LDAA $00,X ;move byte from src
      STAA $00,Y ;to dst
      INX          ;increment src pointer
      INY          ;increment dst pointer
      DECB        ;decrement count, to
*                ;indicate another move
      BNE  DOWH  ;Do-while count not zero
      HERE BRA  HERE ;stop program when done
```

- * This sample is source code only.
- * Corresponding machine code is not shown.

Relative Addressing



$$PC_{new} = PC_{old} + T + rr$$

$$rr = PC_{new} - PC_{old} - T$$

Note: relative address is signed and you must extend the sign when adding or subtracting to 16-bit numbers

The absolute 16-bit addresses are unsigned

Secondary Memory Reference Instructions



- Many I/O operations rely on whether certain bits are set or cleared
- Branch if Bit(s) set: BRSET
- Branch if Bit(s) clear: BRCLR
- A special byte in the instruction called mask specifies which bits to check

```
LDX  #$1000
BRSET $03,X $07 TRUE_IF_SET
BRCLR $03,X $01 TRUE_IF_CLR
```

Jump Instructions



```
START CBA
      BEQ  SKIP
      JMP  NOT_EQUAL
SKIP  INCA
```

Relocatable Programs



- A relocatable program is one that can start anywhere in memory without changing the machine code
- Branch instructions are relocatable because their address is relative to their instruction itself
- Jump instructions are not relocatable
- Relocatability is not important for program executing in ROM
- Programs executing in RAM must be relocatable since the OS could load them anywhere

Assignments



- Chapter 2. Exercises 1 to 4; and 13 to 22
- Note: Study Chapter 1 to 2 from Spasov book for the quiz