Introduction to Embedded Systems

CS/ECE 6780/5780

Al Davis

Today's topics:

- some logistics nagging
- assembly language programming

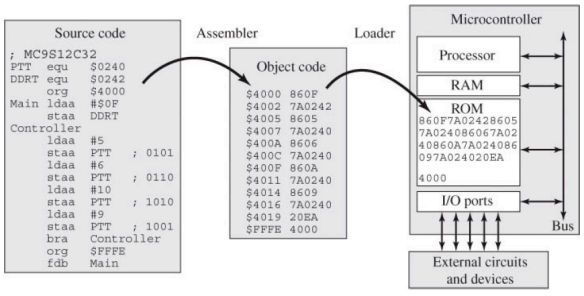
Some Nagging

- **Apparently necessary**
  - several students do not yet have lab partners or lab sections identified
    - » lab section is your choice but you have to let me know via email
    - » note this is due TODAY
  - registered but not on email list
    - » Min, Najar, Sreedharan, Tateoka
  - team identified but no lab section
    - » Bailey & McDougall, Tomer & Oneida
  - no team or lab section
    - » Behera, Fallatah, Jones, Lezin, Maheshwari, Martin, Min
    - » Morley, Najar, Rolfe, Sreedharan, Tateoka, Wiser, Worley
- **If your name appears above - see me after class**
  - note I may have some book-keeping errors
    - » but we have to get this sorted out
    - » lab section assignments will appear on the web today
      - If you spot an error email teach-cs5780 ASAP

Why Assembly?

- **Taken ECE/CS 4400?**
  - then you know
- **Typically you'll write your lab codes in C**
  - If everything works
    - » then no need for assembly/object code knowledge
  - If it doesn't work
    - » examine assembly to see what the compiler did to your code
    - » and some things are just easier in assembly
      - direct control of machine resources
      - you can embed assembly code in your C code files
  - works in simulator but not on the processor
    - » processor only knows bits → object code
- **Hence**
  - you'll need to be familiar with assembly and object code representations
    - » you'll probably do this in JIT mode

Assembly Language Development Process



6812 Assembly Language

- Details are in references on the web page
  - too boring to enumerate in class
  - so we'll cover the highlights
- Addresses and symbols
  - assembler usually is a 2-pass process
    - » 1: build the symbol table:values map
      - explicit value
        - `PTT equ $2040`
      - implicit value: label a particular location in the code
    - » 2: create object file based on symbol table
  - phasing error results if symbol value differs between passes
- Result is a listing file
  - errors are listed if they occur
    - » undefined symbol, illegal opcode, branch distance too far, etc.
    - » symbol table values
    - » hex object code

Syntax

```
Label   Operation   Operand   Comment
PORTA   equ         $0000     ; Assembly time constant
Inp     ldaa        PORTA     ; Read data from PORTA
```


Syntax

- Basic model

```
Label   Operation   Operand   Comment
PORTA   equ         $0000     ; Assembly time constant
Inp     ldaa        PORTA     ; Read data from PORTA
```

- First character is important
  - white space → no label
  - * or ; → comment line
  - character, ".", or "_" → label
    - » case sensitive
      - composed of digits, characters, ".", "$", "_"
      - advice: develop a label convention and stick to it
- Labels (optional : which is ignored)
  - define only once except when defined by `set`

  value is an address of next instruction

  unless defined by `equ or set`

More Syntax

Operations must be proceeded by at least one white space character, and they are case-insensitive (`nop`, `NOP`, `NoP`).

Operations can be an opcode or assembler directive (*pseudo-op*).

Operand must be proceeded by white space.

Operands must not contain any white space unless the following comment begins with a semicolon.

Operands are composed of symbols or expressions.

Operand Types

Operand	Format	Example
no operand	INH	`clra`
#<expression>	IMM	`ldaa #4`
<expression>	DIR,EXT,REL	`ldaa 4`
<expression>,idx	indexed (IND)	`ldaa 4,x`
<expr>,#<expr>	bit set or clear	`bset 4,#$01`
<expr>,#<expr>,< *expr* >	bit test & branch	`brset 4,#$01,foo`
<expr>,idx,#<expr>,< *expr* >	bit test & branch	`brset 4,x,#$01,foo`
<expression>,idx+	IND, post incr	`ldaa 4,x+`
<expression>,idx-	IND, post decr	`ldaa 4,x-`
<expression>,+idx	IND, pre incr	`ldaa 4,+x`
<expression>,-idx	IND, pre decr	`ldaa 4,-x`
acc,idx	accum offset IND	`ldaa A,x`
[<expression>,idx]	IND indirect	`ldaa [4,x]`
[D,idx]	RegD IND indirect	`ldaa [D,x]`

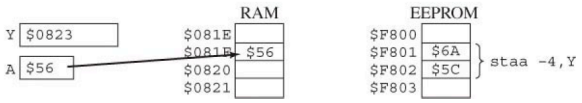
Indexed Addressing Mode

Uses a fixed signed offset with a 16-bit register: X, Y, SP, or PC.

Offset can be 5-bits, 9-bits, or 16-bits.

Example (5-bit):

```
Obj code   Op     Operand   Comment
$6A5C      staa   -4,Y      ;[Y-4] = RegA
```



Building the Object Code

`staa -4,Y` → `$6A5C`

First byte is `$6A` - Op code (pg. 254)

Second byte is formatted as `%rr0nnnnn` (pg. 33).

`rr` is `%01` for register Y.

`nnnnn` is `%11100` for `-4`.

`%0101 1100` → `$5C`.

Information is found in the CPU12 Reference Manual (CPU12RM.pdf).

Auto Pre/Post Dec/Inc Indexed Mode

Can be used with the X, Y, and SP registers, but not PC.

The register used is incremented/decremented by the offset value (1 to 8) either before (pre) or after (post) the memory access.

In these examples assume that `RegY=2345`:

```
Op     Operand   Comment
staa   1,Y+      ;Store RegA at 2345, then RegY=2346
staa   4,Y-      ;Store RegA at 2345, then RegY=2341
staa   4,+Y      ;RegY=2349, then store RegA at 2349
staa   1,-Y      ;RegY=2344, then store RegA at 2344
```

Why not the PC register?

Building the Object Code

`staa 1,X+` → $6A30

- First byte is $6A - Op code (pg. 254)
- Second byte is formatted as %rr1pnnnn (pg. 33).
- rr is %00 for register X.
- nnnn is %0000 for 1.
- p is %1 for post.
- %0011 0000 → $30.

Information is found in the CPU12 Reference Manual (CPU12RM.pdf).

Accumulator Offset Indexed

Uses two registers, offset is in A, B, or D, while index is in X, Y, SP, or PC.

Examples:

```
Op     Operand   Comment
ldab   #4
ldy    #2345
staa   B,Y       ;Store value in RegA at 2349
```

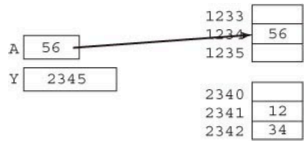

Indexed Indirect Mode

Adds 16-bit offset to 16-bit register (X,Y,SP, or PC) to compute address in which to fetch another address.

This second address is used by the load or store.

Examples:

```
Op     Operand   Comment
ldy    #$2345
staa   [-4,Y]    ;Fetch 16-bit address from $2341,
                 ;store $56 at $1234
```



Accumulator D Offset Indexed Indirect

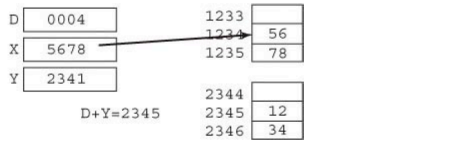
Offset is in D and index is in another 16-bit register.

Computed address is used to fetch another address from memory.

Load or store uses the second address.

Examples:

```
Op     Operand   Comment
ldd    #4
ldy    #$2341
stx    [D,Y]     ;Store value in RegX at $1234
```



Load Effective Address

Used with IND addressing modes.

Calculate the effective address and store it in the specified register: X, Y, or SP.

CC bits are not affected.

Example:

```
leas -4,SP ;SP -= 4 → $1B9C
```

$1B is `leas` op code (first byte).
Second byte is %rr0nnnnn.
%10 is the `rr` code for SP.
%1 1100 is -4.

LEAS – load effective address into SP

Load and Store Instructions

- **Register ↔ Memory moves**
  - **load instructions: ldaa, ldab, ldd, lds, ldx, ldy**
    - » **modes are IMM, DIR, EXT, IND**
  - **store instructions: staa, stab, std, sts, stx, sty**
    - » **modes are DIR, EXT, IND**
  - **CC N & Z bits updated based on moved value**
  - **Examples**

Op	Operand	Comment
ldaa	#$FF	IMM
staa	$25	DIR
ldab	$0025	EXT
std	$05,X	IND
ldd	$C025	EXT

M2M Move Instructions

- **Move a value from one memory location to another**
  - **does not affect the CC register bits**

Move an 8-bit constant into memory:

```
movb  #w,addr   [addr]=w
```

Move an 8-bit value memory to memory:

```
movb  addr1,addr2   [addr2]=[addr1]
```

Move a 16-bit constant into memory:

```
movw  #W,addr   {addr}=W
```

Move a 16-bit value memory to memory:

```
movw  addr1,addr2   {addr2}={addr1}
```


Clear/Set Instructions

Used to initialize memory (`clr`), accumulators (`clra`,`clrb`), or bits in the CC (`clc`, `cli`, `clv`).

`clr` addressing modes are: EXT, IND.

`clra`, `clrb`, `clc`, `cli`, `clv` are INH.

Examples:

Op	Operand	Comment
clra		INH
clr	$0025	EXT

The carry (C), interrupt mask (I), and overflow (V) bits in the CC can also be set (`sec`, `sei`, `sev`).

Exchange and Transfer Instructions

- **Transfer (all INH)**
  - **tab: A→ B (also tba)**
  - **tap: A→ CC (also tpa)**
  - **tsx, txs, tsy, tys, etc. see manual for full set**
- **Exchange (also INH)**
  - **double move**
    - **xgdx, xgdy**

Add and Subtract

Registers: aba, abx, aby, sba (all INH).

With carry to memory: adca, adcb, sbca, sbcb.

w/o carry to memory: adda, addb, addd, suba, subb, subd.

Addressing modes are: IMM, DIR, EXT, IND.

Examples: 16-bit addition using only A

Op	Operand	Comment
ldaa	$25	load least sig byte
adda	$35	add data at $35 to A
staa	$45	store least sig byte
ldaa	$24	load most sig byte
adca	$34	add data at $34 to A
staa	$44	store most sig byte

Compare

Perform a subtraction to update the CC, but do not alter data register values.

Typically used just before a branch instruction.

Compare registers: cba (INH).

Compare to memory: cmpa, cmpb, cpd, cpx, cpy.

Addressing modes: IMM, DIR, EXT, IND

Example: comparing with a known set point

Obj code	Op	Operand	Comment
$8650	ldaa	#$50	load set point into A
$B11031	cmpa	$1031	compare A to memory

If Z flag is 1 then the contents of $1031 equals $50.

Misc. Arithmetic Instructions

- **Dec/Inc, Negate, Test**

dec, deca, decb, des, dex, dey - decrement

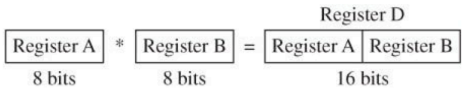
inc, inca, incb, ins, inx, iny - increment

neg, nega, negb - two's complement.

tst, tsta, tstb - subtracts 0 from memory or register and sets *Z* and *N* flags in the CC.

Multiply

Multiplies two unsigned 8-bit values in A and B to produce a 16-bit unsigned product stored in D (i.e., $A \times B \rightarrow D$).



Example:

Op	Operand	Comment
`ldaa`	`#$FF (255)`	IMM
`ldab`	`#$14 (20)`	IMM
`mul`		INH

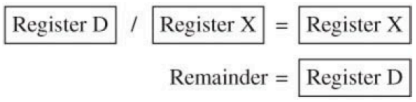
At the end, accumulator D contains $13EC (5100).

$FF * $FF = $FE01

Integer Divide

Use D register for the dividend and X register for the divisor. Resultant placed in X register and remainder in D register.

`idiv` performs integer division.

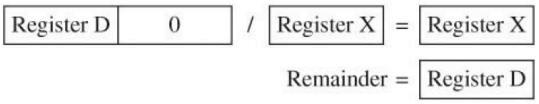


Example:

Obj code	Op	Operand	Comment
$CCFFFF	`ldd`	#$FFFF (65535)	After `idiv` executes
$CE2710	`ldx`	#$2710 (10000)	X contains $0006
$02	`idiv`		D contains $159F (5535)

Fractional Divide

`fdiv` performs fractional division resulting in binary weighted fraction between 0 and 0.999998 (i.e., $(65536 * D)/X \rightarrow X$).



Numerator must be less than denominator or overflow occurs.

Next 16-bits of the fraction can be obtained by reloading the denominator and doing `fdiv` again.

fdiv Example

Op	Operand
`ldd`	#1
`ldx`	#3
`fdiv`	

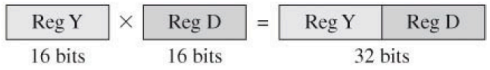
Result: X: $5555 and D: $0001.

Assuming decimal point is to the left of the MSB $5555 = 0.333328247...

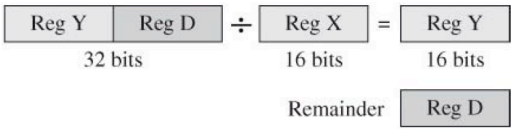
Another `fdiv` refines the value to: 0.33333333325572

Extended Precision Arithmetic

`emul` and `emuls` perform 16 × 16 unsigned and signed multiplication.



`ediv` and `edivs` perform 32 × 16 unsigned and signed division.



MAC

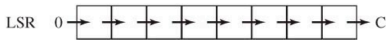
`emacs` performs a 16 × 16 signed multiply followed by a 32-bit signed addition.

Uses indexed addressing to access two 16-bit inputs and extended addressing to access the 32-bit sum.

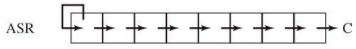
$$< U > \; = \; < U > + \{X\} * \{Y\}$$

Shifts

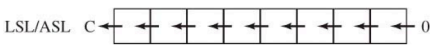
Logical shift right (`lsr`, `lsra`, `lsrb`, `lsrd`) shifts 0's into MSB.



Arithmetic shift right (`asr`, `asra`, `asrb`) retains MSB value.



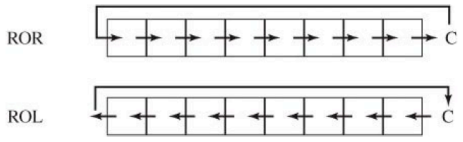
Logical shift left (`lsl`, `lsla`, `lslb`, `lsld`) and arithmetic shift left (`asl`, `asla`, `aslb`, `asld`) are equivalent (same op code).



Rotate

Rotate right (`ror`, `rora`, `rorb`) put carry bit into the MSB.

Rotate left (`rol`, `rola`, `rolb`) put carry bit into the LSB.



Bitwise Logical Operations

AND - `anda`, `andb` (IMM, DIR, EXT, IND).

Inclusive OR - `oraa`, `orab` (IMM, DIR, EXT, IND).

Exclusive OR - `eora`, `eorb` (IMM, DIR, EXT, IND).

1's complement - `com`, `coma`, `comb`.

Example: masking unwanted bits

Obj code	Op	Operand	Comment
$8634	ldaa	#$34	%00110100
$840F	anda	#$0F	%00001111
			Result in A is %00000100

Bit Test, Set, & Clear

`bita` and `bitb` instructions perform an AND operation and update *N* and *Z* flags of the CC w/o altering the operand.

`bclr` and `bset` instructions are used to clear or set bit(s) in a given memory location.

```
bclr addr, mm
bset addr, mm
```

where `addr` is a memory location specified using DIR, EXT, or IND addressing mode and `mm` is a mask byte.

Stack Instructions

Stack pointer (RegSP) defines the top of the stack.

Should be loaded with a RAM memory address early in any assembly language program.

Push and pull instructions put data onto and take data off the stack.

`psha`, `pshb`, `pshx`, `pshy`, `pula`, `pulb`, `pulx`, `puly` (all INH).

REMEMBER: stack grows down (lower address value)

Subroutine Linkage (Manual)

Op	Comment
pshy	INH
pshx	INH
pshb	INH
psha	INH
tpa	INH
psha	INH
	body of subroutine
pula	INH
tap	INH
pula	INH
pulb	INH
pulx	INH
puly	INH

Callee saves state to stack
Restores state on return

Subroutine Call and Return

`bsr` - branch to subroutine using REL addressing.

`jsr` - jumps to subroutine using DIR, EXT, or IND addressing.

On either `bsr` or `jsr`, PC is automatically pushed onto the stack (least significant byte first).

`rts` - return from subroutine, PC automatically pulled off the stack and jumps to that location.

bsr offset is 8-bit signed value

Jump, Branch, Branch Always

`jmp` and `bra` instructions are unconditional.

`bra` uses relative addressing (REL) so it can only be used to jump -128 or 127 instructions.

`jmp` can use EXT and IND addressing so it can be used to jump anywhere in the 64K address space.

`bra $` stops progress of CPU, but it continues to execute this instruction.

Single Condition Branches

`bcc` - branch if carry clear (i.e., $C = 0$).

`bcs` - branch if carry set (i.e., $C = 1$).

`bne` - branch if not equal to zero (i.e., $Z = 0$).

`beq` - branch if equal to zero (i.e., $Z = 1$).

`bpl` - branch if positive or zero (i.e., $N = 0$).

`bmi` - branch if negative (i.e., $N = 1$).

`bvc` - branch if overflow clear (i.e., $V = 0$).

`bvs` - branch if overflow set (i.e., $V = 1$).

`brn` - branch never hard to say what this is good for

Example: Equality Tests

```
C Code             Assembly Code
=================================================
if (G2==G1) {          ldaa G2
  isEqual();           cmpa G1
}                      bne  next    ;skip if not equal
                       jsr  isEqual ;G2==G1
                 next
=================================================
if (G2!=G1) {          ldaa G2
  isNotEqual();        cmpa G1
}                      beq  next       ;skip if equal
                       jsr  isNotEqual ;G2!=G1
                 next
```

Unsigned Number Branches

These branches usually follow `cba`, `cmp(A,B,D)`, `cp(X,Y)`, `sba`, `sub(A,B,D)` instructions.

`bhi` - branch if higher '$>$' (i.e., $C + Z = 0$).

`bhs` - branch if higher or same '$\geq$' (i.e., $C = 0$).

`blo` - branch if lower '$<$' (i.e., $C = 1$).

`bls` - branch if lower or same '$\leq$' (i.e., $C + Z = 1$).

Signed Number Branches

These branches usually follow `cba`, `cmp(A,B,D)`, `cp(X,Y)`, `sba`, `sub(A,B,D)` instructions.

`bgt` - branch if greater '$>$' (i.e., $Z + (N \oplus V) = 0$).

`bge` - branch if greater or equal '$\geq$' (i.e., $N \oplus V = 0$).

`blt` - branch if less '$<$' (i.e., $N \oplus V = 1$).

`ble` - branch if less or equal '$\leq$' (i.e., $Z + (N \oplus V) = 1$).

Example: Unsigned Tests

```
C Code             Assembly Code
=====================================================
unsigned int G1;       ldaa G2
unsigned int G2;       cmpa G1
if (G2 > G1) {         bls  next       ;skip if G2<=G1
  isGreater();         jsr  isGreater  ;G2>G1
}                 next
=====================================================
unsigned int G1;       ldaa G2
unsigned int G2;       cmpa G1
if (G2 > G1) {         blo  next         ;skip if G2<G1
  isGreaterEq();       jsr  isGreaterEq  ;G2>=G1
}                 next
```


Miscellaneous But Useful

`nop` — no operation, creates a 2-cycle delay.

`swi` — trigger a software interrupt.

`rti` — called at the end of an *interrupt service routine* to restore the CPU registers.

`wai` — puts CPU into standby mode waiting for an interrupt; CPU clock is stopped but other MCU clocks can continue to run.

`stop` — stop all clocks to save power; start on $\overline{RESET}$, $\overline{XIRQ}$, or unmasked $\overline{IRQ}$; RAM, I/O space, and registers are preserved.

Assembler Pseudo-Ops

Set the location to put the following object code (`org`, `.org`):

```
org <expression>
```

Equate symbol to a value (`equ`, `=`):

```
<label> equ <expression>
```

Redefinable equate symbol to a value (`set`):

```
<label> set <expression>
```

Reserve multiple bytes (`rmb`, `ds`, `ds.b`, `.blkb`):

```
<label> rmb <expression>
```

Reserve multiple words (`ds.w`, `.blkw`):

```
<label> ds.w <expression>
```

Reserve multiple 32-bit words (`ds.l`, `.blkl`):

```
<label> ds.l <expression>
```


More Pseudo-Ops

Form constant byte (`fcb`, `dc.b`, `db`, `.byte`):

```
<label> fcb <expression>
```

Form double byte (`fdb`, `dc.w`, `dw`, `.word`):

```
<label> fdb <expression>
```

Define 32-bit constant (`dc.l`, `dl`, `.long`):

```
<label> fqb <expression>
```

Form constant character string (`fcc`):

```
hello fcc ''Hello World'',0
```


Concluding Remarks

- **Boring and still incomplete**
  - **hopefully you have background to read understand the ISA & assembler**
    - » **read the reference documentation for the whole scoop**
- **Addressing modes are the key to reading and writing assembly**
  - **you'll tend to read it more than write → debug**
  - **write usually only happens when you need low-level HW control**
  - **condition codes and subsequent branches are important**
    - » **ignore and bugs appear**
- **Extensive math support**
  - **for operations wider than 8 bits**
- **Assembly coding is hard**
  - **easy to make basic and serious mistakes**
    - » **save & restore state, mis-matched stack frames, CC screw-ups**
    - » **Pandora's box in a way**