Lecture #3

Microcontroller Instruction Set

18-348 Embedded System Engineering

Philip Koopman

Wednesday, 20-Jan-2015

Electrical & Computer ENGINEERING

Carnegie Mellon

April 2013: Traffic Light Heaven in L.A.

Los Angeles syncs up all 4,500 of its traffic lights

Los Angeles is the first city in the world to synchronize all of its traffic lights, hoping to unclog its massive roadway congestion.

It has taken 30 years and $400 million, but Los Angeles has finally synchronized its traffic lights in an effort to reduce traffic congestion, becoming the first city in the world to do so.

Mayor Antonio R. Villaraigosa said with the 4,500 lights now in sync, commuters will save 2.8 minutes driving five miles in Los Angeles, The New York Times reported. Villaraigosa also said that the average speed would rise more than two miles per hour on city streets and that carbon emissions would be greatly reduced as drivers spend less time starting and stopping. According to CBS News, less idling will mean a 1-ton reduction in carbon emissions every year.

Wk #	Week of:	Mon (Sec E)	Tue (Sec A)	Wed (Sec B)	Thu (Sec C)	Fri (Sec D)	Lab Report Due Wednesday	Prelab Due Friday	Fri. Recitation Discusses Labs
1	11-Jan 2016	No Lab	No Lab	Open Lab	Open Lab	Open Lab	None	1	1, 2
2	18-Jan	MLK Day	1	1	1	1	None	2	2, 3
3	25-Jan	1	2	2	2	2	1	3	3, 4
4	1-Feb	2	3	3	3	3	2	4	4, 5
5	8-Feb	3	4	4	4	4	3	5	5, 6
6	15-Feb	4	5	5	5	5	4	6	6, 7
7	22-Feb	5	Open Lab	Open Lab	Open Lab	6	None	None	7, 8
8	29-Feb	6	6	6	6	BREAK	5	7 Due **Thursday**	No Recitation
--	7-Mar	SPRING	BREAK	SPRING	BREAK	BREAK	None	None	No Recitation
9	14-Mar	Open Lab	Open Lab	7	7	7	6	8	8, 9
10	21-Mar	7	7	8	8	8	7	9	9, 10
11	28-Mar	8	8	9	9	9	8	10	10, 11
12	4-Apr	9	9	10	10	10	9	11	11
13	11-Apr	10	10	Open Lab	Carnival	Carnival	None	None	No Recitation
14	18-Apr	Open Lab	Open Lab	Open Lab	Open Lab	Open Lab	10	None	Optional/In-Lab
15	25-Apr	Open Lab	Open Lab	Open Lab	Open Lab	Open Lab	None	None	Optional/In-Lab
16	2-May Finals	TBD	TBD	TBD	TBD	TBD	11 Due (**Thursday**)	None	No Recitation

(*See blackboard for Lab 11 prelab, demo & writeup information)

Where Are We Now?

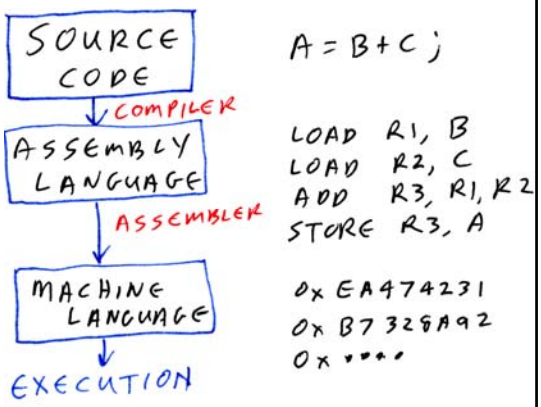
- **Where we've been:**
  - Embedded Hardware
- **Where we're going today:**
  - Instruction set & Assembly Language
- **Where we're going next:**
  - More assembly language
  - Engineering process
  - Embedded C
  - Coding tricks, bit hacking, extended-precision math

Preview

- **Programmer-visible architecture**
  - Registers
  - Addressing modes
- **Branching**
  - Types of branches
  - How condition codes are set
- **Assembly/Disassembly**
  - Review of how instructions are encoded
- **Timing**
  - How long does an instruction take to execute? (simple version)

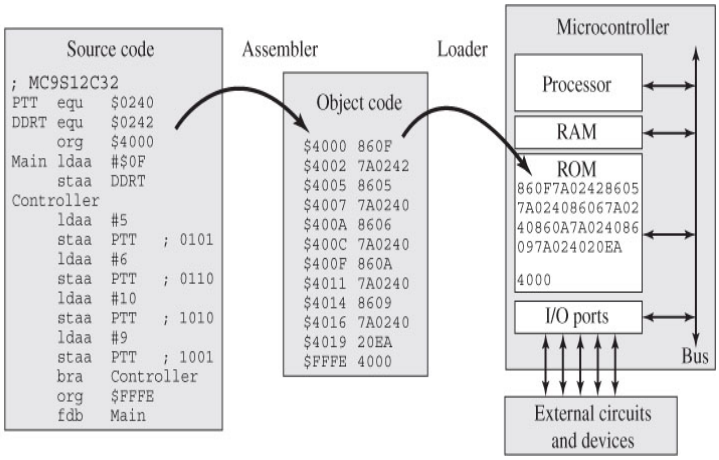
Where Does Assembly Language Fit?

- **Source code**
  - High level language (C; Java)
  - Variables and equations
  - One-to-many mapping with assembly language
- **Assembly language**
  - Different for each CPU architecture
  - Registers and operations
  - Usually one-to-one mapping to machine language
- **Machine language**
  - Hex/binary bits
  - Hardware interprets to execute program



Assembler To ROM Process

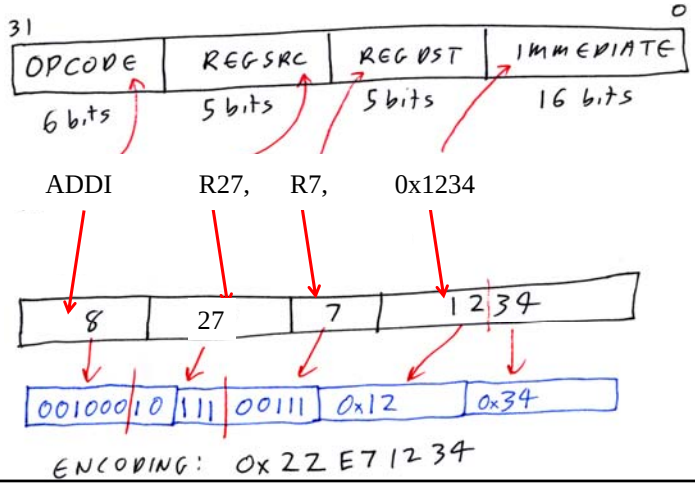
Figure 2.1
Assembly language development process.



[Valvano]

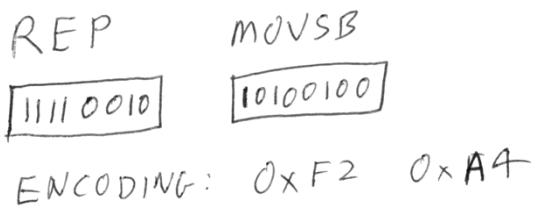
RISC Instruction Set Overview

- **Typically simple encoding format to make hardware simpler/faster**
- **Classical Example: MIPS R2000**
  - R7 <= R27 + 0x1234



CISC Instruction Set Overview

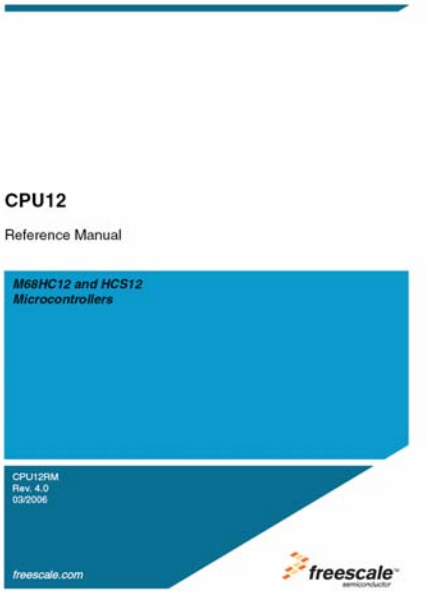
- **Complex encoding for small programs**
- **Classical Example: VAX; Intel 8088**
  - REP MOVSB (8088 String move)
    - Up to 64K bytes moved; source in SI reg; dest in DI reg; count in CX



Accumulator-Based Microcontrollers

- **Usually one or two "main" registers – "accumulators"**
  - Historically called register "A" or "Acc" or registers "A" and "B"
  - This is where the Pentium architecture gets "AX, BX, CX, DX" from
- **Usually one or more "index" registers for addressing modes**
  - Historically called register "X" or registers "X" and "Y"
  - In the Pentium architecture these correspond to SI and DI registers
- **A typical "H = J + K" operation is usually accomplished via:**
  - Load "J" into accumulator
  - Add "K" to "J", putting result into accumulator
  - Store "H" into memory
  - Reuse the accumulator for the next operation (no large register file)
- **Usually microcontrollers are resource-poor**
  - E.g., No cache memory for most 16-bit micros!

CPU12 Resource – Long Version



DECA — Decrement A — DECA

Operation:

(A) – $01 ⇒ A

Description:

Subtract one from the content of accumulator A.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Details:

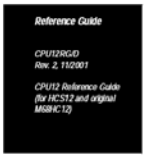
S	X	H	I	N	Z	V	C
=	=	=	=	Δ	Δ	Δ	=

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
V: Set if there was a two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (A) was $80 before the operation.

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
DECA	INH	43	O	O

11

CPU12 Resource – Summary Version



MOTOROLA intelligence everywhere | digital dna

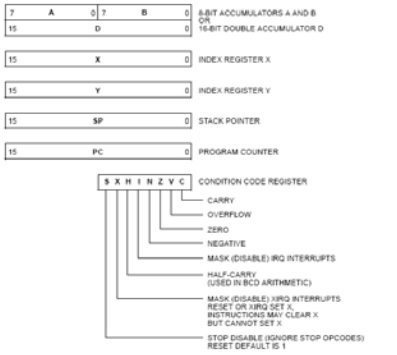
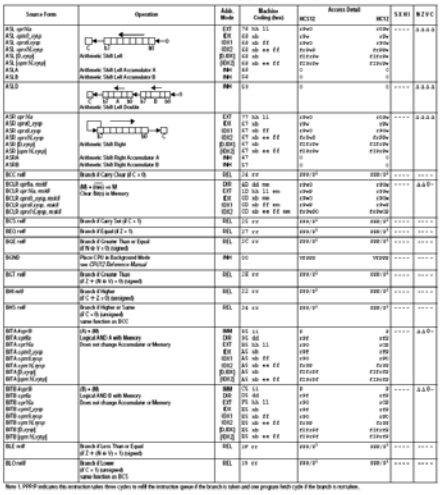


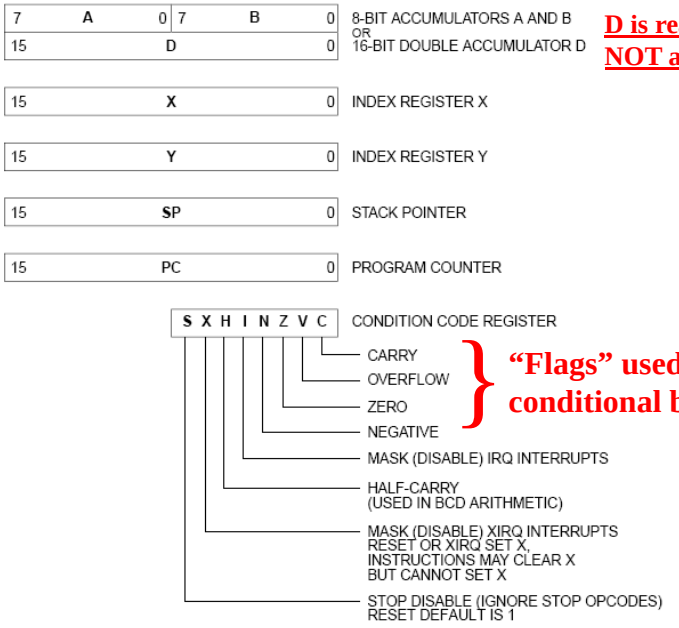
Figure 1. Programming Model

Instruction Set Summary (Sheet 2 of 14)



12

"CPU12" Programming Model – (MC9S12C128)



D is really just A:B
NOT a separate register!

"Flags" used for conditional branches

[Motorola01]

The CPU12 Reference Guide

- **Summarizes assembly language programming info**
  - Lots of info there …. This lecture is an intro to that material

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12	Access Detail HC12	S X H I	N Z V C
ABA	(A) + (B) ⇒ A Add Accumulators A and B	INH	18 06	OO	OO	- - Δ -	Δ Δ Δ Δ
ABX	(B) + (X) ⇒ X *Translates to* LEAX B,X	IDX	1A E5	Pf	PP[1]	- - - -	- - - -
ABY	(B) + (Y) ⇒ Y *Translates to* LEAY B,Y	IDX	19 ED	Pf	PP[1]	- - - -	- - - -
ADCA #*opr8i* ADCA *opr8a* ADCA *opr16a* ADCA *oprx0_xysp* ADCA *oprx9,xysp* ADCA *oprx16,xysp* ADCA [D,*xysp*] ADCA [*oprx16,xysp*]	(A) + (M) + C ⇒ A Add with Carry to A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	89 ii 99 dd B9 hh ll A9 xb A9 xb ff A9 xb ee ff A9 xb A9 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrfP	P rfP rOP rfP rPO frPP flPrfP fIPrfP	- - Δ -	Δ Δ Δ Δ
ADCB #*opr8i* ADCB *opr8a* ADCB *opr16a*	(B) + (M) + C ⇒ B Add with Carry to B	IMM DIR EXT	C9 ii D9 dd F9 hh ll	P rPf rPO	P rfP rOP	- - Δ -	Δ Δ Δ Δ

[Motorola01]

ALU Operations – Addition as an Example

- **"Inherent" address modes:**
  - ABA (B) + (A) => A **A**dd accumulator **B** to **A**
    - Encoding: 18 06
  - ABX (B) + (X) => X **A**dd accumulator **B** to **X**
    - Encoding: 1A E5

- **Immediate Operand:**
  - ADDD #value (D) + jj:kk => D **Add** to **D**
    - Add constant value to D (example: D <= D + 1234)
    - Encoding: C3 jj kk
    - Example: ADDD #$534 Adds hex 534 (0x534) to D reg

- **"Extended" operand – location in memory at 16-bit address:**
  - ADDD address (D) + [HH:LL] => D **Add** to **D**
    - Fetch a memory location and add to D
    - Encoding: F3 HH LL
    - Example: ADDD $5910 Adds 16-bit value at $5910 to D
  - NOTE: "[xyz]" notation means "Fetch from address xyz"

Address Modes

Address Modes

IMM	— Immediate
IDX	— Indexed (no extension bytes) includes: 5-bit constant offset Pre/post increment/decrement by 1 . . . 8 Accumulator A, B, or D offset
IDX1	— 9-bit signed offset (1 extension byte)
IDX2	— 16-bit signed offset (2 extension bytes)
[D, IDX]	— Indexed indirect (accumulator D offset)
[IDX2]	— Indexed indirect (16-bit offset)
INH	— Inherent (no operands in object code)
REL	— 2's complement relative offset (branches)
DIR	– Direct (8-bit memory address with zero high bits)
EXT	– Extended (16-bit memory address)

[Motorola01]

Instruction Description Notation

abc — A or B or CCR

abcdxys — A or B or CCR or D or X or Y or SP. Some assemblers also allow T2 or T3.

abd — A or B or D

abdxys — A or B or D or X or Y or SP

dxys — D or X or Y or SP

msk8 — 8-bit mask, some assemblers require # symbol before value

opr8i — 8-bit immediate value

opr16i — 16-bit immediate value

opr8a — 8-bit address used with direct address mode

opr16a — 16-bit address value

oprx0_xysp — Indexed addressing postbyte code:

oprx3,–xys	Predecrement X or Y or SP by 1 . . . 8
oprx3,+xys	Preincrement X or Y or SP by 1 . . . 8
oprx3,xys–	Postdecrement X or Y or SP by 1 . . . 8
oprx3,xys+	Postincrement X or Y or SP by 1 . . . 8
oprx5,xysp	5-bit constant offset from X or Y or SP or PC
abd,xysp	Accumulator A or B or D offset from X or Y or SP or PC

oprx3 — Any positive integer 1 . . . 8 for pre/post increment/decrement

oprx5 — Any value in the range –16 . . . +15

oprx9 — Any value in the range –256 . . . +255

oprx16 — Any value in the range –32,768 . . . 65,535

page — 8-bit value for PPAGE, some assemblers require # symbol before this value

rel8 — Label of branch destination within –256 to +255 locations

rel9 — Label of branch destination within –512 to +511 locations

rel16 — Any label within 64K memory space

trapnum — Any 8-bit value in the range $30-$39 or $40-$FF

xys — X or Y or SP

xysp — X or Y or SP or PC

[Motorola01]

Notation for Encoding of Instruction Bytes

Machine Coding

`dd` — 8-bit direct address $0000 to $00FF. (High byte assumed to be $00).

`ee` — High-order byte of a 16-bit constant offset for indexed addressing.

`eb` — Exchange/Transfer post-byte. See **Table 3** on page 23.

`ff` — Low-order eight bits of a 9-bit signed constant offset for indexed addressing, or low-order byte of a 16-bit constant offset for indexed addressing.

`hh` — High-order byte of a 16-bit extended address.

`ii` — 8-bit immediate data value.

`jj` — High-order byte of a 16-bit immediate data value.

`kk` — Low-order byte of a 16-bit immediate data value.

`lb` — Loop primitive (DBNE) post-byte. See **Table 4** on page 24.

`ll` — Low-order byte of a 16-bit extended address.

`mm` — 8-bit immediate mask value for bit manipulation instructions. Set bits indicate bits to be affected.

`pg` — Program page (bank) number used in CALL instruction.

`qq` — High-order byte of a 16-bit relative offset for long branches.

`tn` — Trap number $30–$39 or $40–$FF.

`rr` — Signed relative offset $80 (–128) to $7F (+127). Offset relative to the byte following the relative offset byte, or low-order byte of a 16-bit relative offset for long branches.

`xb` — Indexed addressing post-byte. See **Table 1** on page 21 and **Table 2** on page 22.

ALU Operations – Addition Example Revisited

- **"Inherent" address modes:**
  - ABA (B) + (A) => A **A**dd accumulator **B** to **A**
    - Encoding: 18 06
  - ABX (B) + (X) => X **A**dd accumulator **B** to **X**
    - Encoding: 1A E5

- **Immediate Operand:**
  - ADDD #opr16i (D) + jj:kk => D **Add** to **D**
    - Add constant value to D (example: D <= D + 1234)
    - Encoding: C3 jj kk
    - Example: ADDD #$534 Adds hex 534 (0x534) to D reg

- **"Extended" operand – location in memory at 16-bit address:**
  - ADDD opr16a (D) + [HH:LL] => D **Add** to **D**
    - Fetch a memory location and add to D
    - Encoding: F3 HH LL
    - Example: ADDD $5910 Adds 16-bit value at $5910 to D

ALU Operations – Addition – 2

- **Immediate Operand:**
  - ADDD #opr16i (D) + jj:kk => D **Add** to **D**
    - Add constant value to D (example: D <= D + 1234)
    - Encoding: C3 jj kk
    - Example: ADDD #$534 Adds hex 534 (0x534) to D reg
  - What C code would result in this instruction?

```
register int16 T;   // assume that X is kept in machine register D
T = T + 0x534;
```

- **"Extended" operand – location in memory at 16-bit address:**
  - ADDD opr16a (D) + [HH:LL] => D **Add** to **D**
    - Fetch a memory location and add to D
    - Encoding: F3 HH LL
    - Example: ADDD $5910 Adds 16-bit value at $5910 to D
  - What C code would result in this instruction?

```
static int16 B;     // B is a variable that happens to be at address $5910
T = T + B;
```

ALU Operations – Addition – 2

- **"Direct" operand – location in memory at 8-bit address:**
  - ADDD oprx8a (D) + [00:LL] => D **Add** to **D**
    - Fetch a memory location and add to D; address is 0..FF ("page zero" of memory)
    - Encoding: D3 LL
    - Example: ADDD $0038
  - Special optimized mode for smaller code size and faster execution
    - Especially for earlier 8-bit processors, but still can be useful
    - Gives you 256 bytes of memory halfway between "memory" and "register" in terms of ease & speed of access
    - Assembler knows to use this mode automatically based on address being $00xx
  - Result – programs often optimized to store variables in first 256 bytes of RAM
    - If you have very limited RAM, this is worth doing to save time & space!
    - But it also promotes use of shared RAM for variables, which is bug prone
  - What C code would result in this instruction?

```
static int16 B;      // B is a variable that happens to be at address $0038
T = T + B;
```


ALU Operations – Addition – 3

- **"Indexed" operand – memory indexed; pre/post increment/decrement**
  - ADDD oprx,xysp (D) + [EE:FF+XYSP] => D
    - Add oprx to X, Y, SP or PC; use address to fetch from memory; add value into D
    - Encoding: E3 xb // E3 xb ff // E3 xb ee ff
      (Signed offset value; encoding varies – 5 bits, 9 bits; 16 bits)
    - Example: ADDD $FFF0, X add value at (X-$16_{10}$) to D
      Encoding: E3 10 (5 bit signed constant … "$10")
      (see Table 1 of CPU12 reference guide for xb byte encoding)
  - Special optimized mode for smaller code size and faster execution
    - "xb" can do many tricks, including support for post/pre-increment/decrement to access arrays
  - What C code would result in this instruction?

```
static int16 B[100];
register int16 *p = &B[50];   // assume "p" is stored in register X
T = T + *(p-8);      // adds B[42] to T
```

Indexed Examples

Figure 2.2
Example of the 6811 indexed addressing mode.

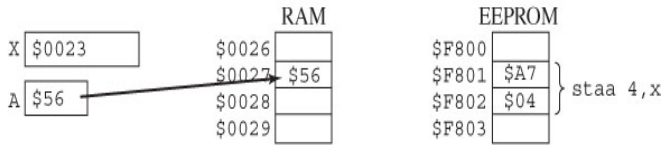


Figure 2.3
Example of the 6812 indexed addressing mode.

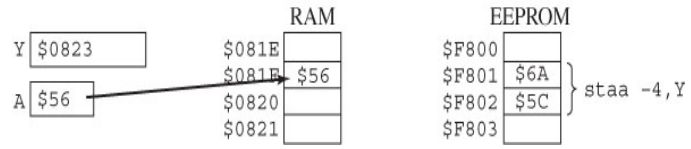


Figure 2.4
Another example of the 6812 indexed addressing mode.

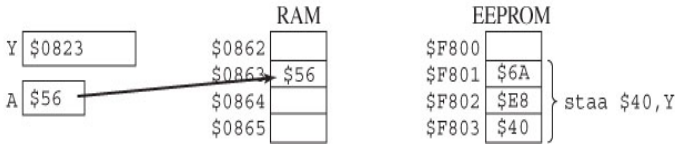
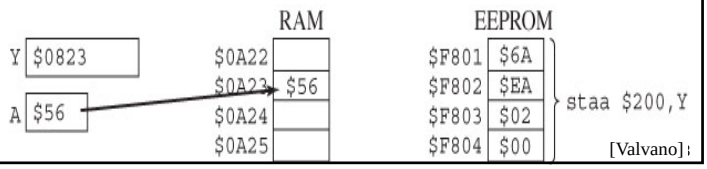


Figure 2.5
A third example of the 6812 indexed addressing mode.



[Valvano]

ALU Operations – Addition – 4

- **"Indexed Indirect" operand – use memory value as address, with offset**
  - ADDD [oprx16,xysp] (D) + **[** [EE:FF+XYSP] **]** => D
    - Add oprx to X, Y, SP or PC; use address to fetch from memory; use the value fetched from memory to fetch from a different memory location; add value into D
    - Encoding: E3 xb ee ff
      Example: ADDD [$8, X] add value at [(X+8)] to D
      Encoding: E3 E3 00 08 16-bit constant offset
      (see Table 1 of CPU12 reference guide for xb byte encoding)

  - What C code would result in this instruction?

```
static int16 vart;
register int16 *p;
static int16 *B[100];   // B is a variable that happens to be at address $38

B[4] = &vart;
p = &B[0];   // assume "p" is stored in register X
T = T + *(*(p+4));      // adds vart to T
```

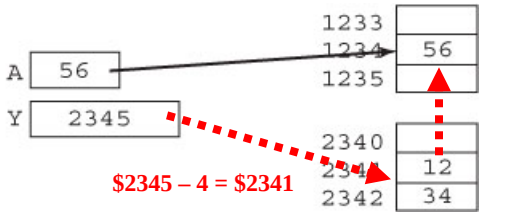
Indexed Indirect Example

LDAA #$56

LDY #$2345

STAA [-4,Y] ; Fetch 16-bit address from $2341, store A at $1234

Figure 2.6
Example of the 6812 indexed-indirect addressing mode.



[Valvano]

Had Enough Yet?

- **Really, all these modes get used in real programs**
  - You've already seen very similar stuff in 18-240, but that's more RISC-like
  - We expect you to be able to tell us what a short, simple program we've written does if it uses any of the modes described during lecture
  - There are even trickier modes – seldom used but nice to have
  - See Valvano Section 2.2 for more discussion

Other Math & Load/Store Instructions

- **Math**
  - ADD – integer addition (2's complement)
  - SBD – integer subtraction (2's complement)
  - CMP – compare (do a subtraction to set flags but don't store result)
- **Logic**
  - AND – logical bit-wise and
  - ORA – logical bit-wise or
  - EOR – bit-wise exclusive or (xor)
  - ASL, ASR – arithmetic shift left and right (shift right sign-extends)
  - LSR – logical shift right
- **Data movement**
  - LDA, LDX, … – load from memory to a register
  - STA, STX, … – store from register to memory
  - MOV – memory to memory movement
- **Bit operations and other instructions**
  - Later…

Control Flow Instructions

- **Used to go somewhere other than the next sequential instruction**
  - Unconditional branch – always changes flow ("goto instruction x")
  - Conditional branch – change flow sometimes, depending on some condition
- **Addressing modes**
  - REL: Relative to PC – "go forward or backward N bytes"
    - Uses an 8-bit offset rr for the branch target
    - Most branches are short, so only need a few bits for the offset
    - Works the same even if segment of code is moved in memory
  - EXT: Extended hh:ll – "go to 16-bit address hh:ll"
    - Takes more bits to specify
    - No limit on how far away the branch can be

Relative Addressing

- **Relative address computed as:**
  - Address of next in-line instruction *after* the branch instruction
    - Because the PC already points to the next in-line instruction at execution time
  - Plus relative byte rr treated as a *signed* value
    - rr of 0..$7F is a forward relative branch
    - rr of $80..$FF is a backward relative branch

- **Example: BCC cy_clr**
  - Next instruction is at $0009; rr = $03
  - $0009 + $03 = $000C (cy_clr)

- **Example: BRA asm_loop**
  - Next instruction is at $000F; rr=$F7
  - $000F + $F7 = $000F + $FFF7 = $000F - $0009 = $0006 (asm_loop)

```
                        asm_main:
000000 180B 01xx                    MOVB    #1,temp_byte
000004 xx
000005 87                           CLRA
                        asm_loop:
000006 52                           INCB
000007 2403                         BCC     cy_clr
000009 43                           DECA
00000A 43                           DECA
00000B 43                           DECA

00000C A7               cy_clr:     NOP
00000D 20F7                         BRA     asm_loop
```


Unconditional Branch

- **JMP instruction – Jump**
  - JMP $1256 -- jump to address $1256
    JMP Target_Name
  - JMP also supports indexed addressing modes – why are they useful?
  - BRA $12 -- jump to $12 past current instruction
    - Relative addressing ("rr") to save a byte and make code relocatable

- **JSR instruction – Jump to Subroutine**
  - JSR $7614 -- jump to address $7614, saving return address
  - JSR Subr_Name
  - Supports DIRect (8 bit offset to page 0) and EXTended, as well as indexed addressing
  - More about how this instruction works in the next lecture

Conditional Branch

- **Branch on some condition**
  - Always with RELative (rr 8-bit offset) addressing
    - Look at detailed instruction set description for specifics of exactly what address the offset is added to
  - Condition determines instruction name
  - BCC $08 – branch 8 bytes ahead if carry bit clear
  - BCS Loop – branch to label "Loop" if carry bit set
  - BEQ / BNE – branch based on Z bit ("Equal" after compare instruction)
  - BMI / BPL – branch based on N bit (sign bit)
- **Other complex conditions that can be used after a CMP instruction**
  - BGT – branch if greater than
  - BLE – branch if less than or equal
  - …

Condition Codes

- **Status bits inside CPU that indicate results of operations**
  - C = carry-out bit
  - Z = whether last result was zero
  - N = whether last result was "negative" (highest bit set)
  - V = whether last result resulted in an arithmetic overflow
- **Set by some (but not all instructions)**
  - CMP – subtracts but doesn't store result; sets CC bits for later "BGE, BGT" etc
  - ADD and most arithmetic operations – sets CC bits
  - MOV instructions – generally do **NOT** set CC bits on this CPU
    - But, on a few other CPUs they do – so be careful of this!

C & V flags

- **Carry: did the previous operation result in a carry out bit?**
  - $FFFF + 1 = $0000 + Carry out
  - $7FFF + $8000 = $FFFF + No Carry out
  - Carry-in bit, if set, adds 1 to sum for ADC
    - we'll do multi-precision arithmetic later
  - Carry bit is set if there is an *unsigned* add or subtract overflow
    - Result is on other side of $0000/$FFFF boundary
- **Overflow (V): did the previous operation result in a signed overflow?**
  - $FFFF + 1 = $0000 no signed overflow (-1 + 1 = 0)
  - $7FFF + 1 = $8000 has signed overflow (32767 + 1 ➔ -32768)
  - This is overflow in the normal signed arithmetic sense that you are used to
    - Result is on other side of $8000/$7FFF boundary
- **Note that the idea of "overflow" depends on signed vs. unsigned**
  - Hardware itself is sign agnostic – software has to keep track of data types
  - Carry flag indicates unsigned overflow
  - V flag indicates signed overflow

Look For Annotations Showing CC Bits Set

Instruction Set Summary (Sheet 5 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12	Access Detail HC12	S X H I	N Z V C
DBNE *abdxys, rel9*	(cntr) - 1 ⇒ cntr If (cntr) not = 0, then Branch; else Continue to next instruction Decrement Counter and Branch if ≠ 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	PPP	----	----
DEC *opr16a* DEC *oprx0_xysp* DEC *oprx9,xysp* DEC *oprx16,xysp* DEC [D,*xysp*] DEC [*oprx16,xysp*]	(M) - $01 ⇒ M Decrement Memory Location	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	73 hh ll 63 xb 63 xb ff 63 xb ee ff 63 xb 63 xb ee ff	rPWO rPw rPWO frPWP fIfrPw fIPrPw	rOPw rPw rPOW frPPw fIfrPw fIPrPw	----	ΔΔΔ-
DECA DECB	(A) - $01 ⇒ A Decrement A (B) - $01 ⇒ B Decrement B	INH INH	43 53	O O	O O		
DES	(SP) - $0001 ⇒ SP *Translates to* LEAS -1,SP	IDX	1B 9F	Pf	Pf[1]	----	----
DEX	(X) - $0001 ⇒ X Decrement Index Register X	INH	09	O	O	----	Δ--
DEY	(Y) - $0001 ⇒ Y Decrement Index Register Y	INH	03	O	O	----	Δ--

[Motorola01]

Assembler to Hex

- **Sometimes (less often these days, but sometimes) you have to write your own assembler!**

- **In this course, we want you to do just a little by hand to get a feel**
  - LDAB #254

LDAB #*opr8i*
LDAB *opr8a*
LDAB *opr16a*
LDAB *oprx0_xysp*
LDAB *oprx9,xysp*
LDAB *oprx16,xysp*
LDAB [D,*xysp*]
LDAB [*oprx16,xysp*]
	(M) ⇒ B
	Load Accumulator B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C6 ii
			D6 dd
			F6 hh ll
			E6 xb
			E6 xb ff
			E6 xb ee ff
			E6 xb
			E6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrfP
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						----	Δ Δ 0 -

  - Addressing mode is:_____
  - Opcode is:_____
  - Operand is:_____
  - Full encoding is:_____ _____

[Motorola01] 35

Hex to Assembler (Dis-Assembly)

- **If all you have is an image of a program in memory, what does it do?**
  - Important for debugging
  - Important for reverse engineering (competitive analysis; legacy components)

- **Start with Hex, and figure out what instruction is**
  - AA E2 23 CC

ORAA #*opr8i* ORAA *opr8a* ORAA *opr16a* ORAA *oprx0_xysp* ORAA *oprx9,xysp* ORAA *oprx16,xysp* ORAA [D,*xysp*] ORAA [*oprx16,xysp*]	(A) + (M) ⇒ A Logical OR A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8A ii 9A dd BA hh ll AA xb AA xb ff AA xb ee ff AA xb AA xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrfP	P rfP rOP rfP rPO frPP fIfrfP fIPrfP	----	Δ Δ 0 -

  - ORAA – one of the indexed versions [Motorola01]
  - Need to look up XB value => _____

Table 1. Indexed Addressing Mode Postbyte Encoding (xb)

00 0,X 5b const	10 -16,X 5b const	20 1,+X pre-inc	30 1,X+ post-inc	40 0,Y 5b const	50 -16,Y 5b const	60 1,+Y pre-inc	70 1,Y+ post-inc	80 0,SP 5b const	90 -16,SP 5b const	A0 1,+SP pre-inc	B0 1,SP+ post-inc	C0 0,PC 5b const	D0 -16,PC 5b const	E0 n,X 9b const	F0 n,SP 9b const
01 1,X 5b const	11 -15,X 5b const	21 2,+X pre-inc	31 2,X+ post-inc	41 1,Y 5b const	51 -15,Y 5b const	61 2,+Y pre-inc	71 2,Y+ post-inc	81 1,SP 5b const	91 -15,SP 5b const	A1 2,+SP pre-inc	B1 2,SP+ post-inc	C1 1,PC 5b const	D1 -15,PC 5b const	E1 -n,X 9b const	F1 -n,SP 9b const
02 2,X 5b const	12 -14,X 5b const	22 3,+X pre-inc	32 3,X+ post-inc	42 2,Y 5b const	52 -14,Y 5b const	62 3,+Y pre-inc	72 3,Y+ post-inc	82 2,SP 5b const	92 -14,SP 5b const	A2 3,+SP pre-inc	B2 3,SP+ post-inc	C2 2,PC 5b const	D2 -14,PC 5b const	E2 n,X 16b const	F2 n,SP 16b const
03 3,X 5b const	13 -13,X 5b const	23 4,+X pre-inc	33 4,X+ post-inc	43 3,Y 5b const	53 -13,Y 5b const	63 4,+Y pre-inc	73 4,Y+ post-inc	83 3,SP 5b const	93 -13,SP 5b const	A3 4,+SP pre-inc	B3 4,SP+ post-inc	C3 3,PC 5b const	D3 -13,PC 5b const	E3 [n,X] 16b indr	F3 [n,SP] 16b indr

36

Easier Way To Find Op-Code Information

[Motorola01]

Table 6. CPU12 Opcode Map (Sheet 1 of 2)

00 †5 BGND INH 1	10 1 ANDCC IM 2	20 3 BRA RL 2	30 3 PULX IH 1	40 1 NEGA IH 1	50 1 NEGB IH 1	60 3-6 NEG ID 2-4	70 4 NEG EX 3	80 1 SUBA IM 2	90 3 SUBA DI 2	A0 3-6 SUBA ID 2-4	B0 3 SUBA EX 3	C
01 5 MEM IH 1	11 11 EDIV IH 1	21 1 BRN RL 2	31 3 PULY IH 1	41 1 COMA IH 1	51 1 COMB IH 1	61 3-6 COM ID 2-4	71 4 COM EX 3	81 1 CMPA IM 2	91 3 CMPA DI 2	A1 3-6 CMPA ID 2-4	B1 3 CMPA EX 3	C
02 1 INY IH 1	12 ‡1 MUL IH 1	22 3/1 BHI RL 2	32 3 PULA IH 1	42 1 INCA IH 1	52 1 INCB IH 1	62 3-6 INC ID 2-4	72 4 INC EX 3	82 1 SBCA IM 2	92 3 SBCA DI 2	A2 3-6 SBCA ID 2-4	B2 3 SBCA EX 3	C
03 1 DEY IH 1	13 3 EMUL IH 1	23 3/1 BLS RL 2	33 3 PULB IH 1	43 1 DECA IH 1	53 1 DECB IH 1	63 3-6 DEC ID 2-4	73 4 DEC EX 3	83 2 SUBD IM 3	93 3 SUBD DI 2	A3 3-6 SUBD ID 2-4	B3 3 SUBD EX 3	C
04 3 loop* RL 3	14 1 ORCC IM 2	24 3/1 BCC RL 2	34 2 PSHX IH 1	44 1 LSRA IH 1	54 1 LSRB IH 1	64 3-6 LSR ID 2-4	74 4 LSR EX 3	84 1 ANDA IM 2	94 3 ANDA DI 2	A4 3-6 ANDA ID 2-4	B4 3 ANDA EX 3	C
05 3-6 JMP ID 2-4	15 4-7 JSR ID 2-4	25 3/1 BCS RL 2	35 2 PSHY IH 1	45 1 ROLA IH 1	55 1 ROLB IH 1	65 3-6 ROL ID 2-4	75 4 ROL EX 3	85 1 BITA IM 2	95 3 BITA DI 2	A5 3-6 BITA ID 2-4	B5 3 BITA EX 3	C
06 3 JMP EX 3	16 4 JSR EX 3	26 3/1 BNE RL 2	36 2 PSHA IH 1	46 1 RORA IH 1	56 1 RORB IH 1	66 3-6 ROR ID 2-4	76 4 ROR EX 3	86 1 LDAA IM 2	96 3 LDAA DI 2	A6 3-6 LDAA ID 2-4	B6 3 LDAA EX 3	C
07 4 BSR RL 2	17 4 JSR DI 2	27 3/1 BEQ RL 2	37 2 PSHB IH 1	47 1 ASRA IH 1	57 1 ASRB IH 1	67 3-6 ASR ID 2-4	77 4 ASR EX 3	87 1 CLRA IH 1	97 1 TSTA IH 1	A7 1 NOP IH 1	B7 1 TFR/EXG IH 2	C
08 1 INX IH 1	18 - page 2 - -	28 3/1 BVC RL 2	38 3 PULC IH 1	48 1 ASLA IH 1	58 1 ASLB IH 1	68 3-6 ASL ID 2-4	78 4 ASL EX 3	88 1 EORA IM 2	98 3 EORA DI 2	A8 3-6 EORA ID 2-4	B8 3 EORA EX 3	C
09 1 DEX IH 1	19 2 LEAY ID 2-4	29 3/1 BVS RL 2	39 2 PSHC IH 1	49 1 LSRD IH 1	59 1 ASLD IH 1	69 ‡2-4 CLR ID 2-4	79 3 CLR EX 3	89 1 ADCA IM 2	99 3 ADCA DI 2	A9 3-6 ADCA ID 2-4	B9 3 ADCA EX 3	C
0A ‡7 RTC IH 1	1A 2 LEAX ID 2-4	2A 3/1 BPL RL 2	3A 3 PULD IH 1	4A ‡7 CALL EX 4	5A 2 STAA DI 2	6A ‡2-4 STAA ID 2-4	7A 3 STAA EX 3	8A 1 ORAA IM 2	9A 3 ORAA DI 2	AA 3-6 ORAA ID 2-4	BA 3 ORAA EX 3	C
0B †8 RTI	1B 2 LEAS	2B 3/1 BMI	3B 2 PSHD	4B ‡7-10 CALL	5B 2 STAB	6B ‡2-4 STAB	7B 3 STAB	8B 1 ADDA	9B 3 ADDA	AB 3-6 ADDA	BB 3 ADDA	C
								2	DI 2	ID 2-4	EX 3	
								D	9C 3 CPD	AC 3-6 CPD	BC 3 CPD	C
								3	DI 2	ID 2-4	EX 3	
								2	9D 3	AD 3-6	BD 3	C

Key to Table 6:

Opcode → 00 5 ← Number of HCS12 cycles (‡ indicates HC12 different)
Mnemonic → BGND ←
Address Mode → IH 1 ← Number of bytes

Performance – How Many Clock Cycles?

- **This is not so easy to figure out**
  - See pages 73-75 of the CPU 12 reference manual
- **In general, factors affecting speed are:**
  - Does the chip have an 8-bit or 16-bit memory bus? (Ours has a 16-bit bus)
    - 8-bit bus needs one memory cycle per byte
    - 16-bit bus needs one memory cycle per 2 bytes, but odd addresses only get 1 byte
  - How many bytes in the encoded instruction itself?
    - AA E2 23 CC takes 4 bytes of fetching
      - » 2 bus cycles if word aligned
      - » 3 bus cycles if unaligned (but get next instruction byte "for free" on 3rd cycle)
  - How many bytes of data
    - Need to read data and, potentially write it
  - Is there an instruction prefetch queue that can hide some fetch delay?
  - Is it a complicated computation that consumes clock cycles (e.g., division)?
- **Usual lower bound estimate**
  - Count up clock cycles for memory touches and probably it takes that or longer

Simple Timing Example

- **ADCA $1246**
  - EXT format – access detail is "rPO" for HCS12
    - r – 8-bit data read
    - P – 16-bit program word access to fetch next instruction
    - O – either prefetch cycle or free cycle (memory bus idle) based on alignment
  - Total is 3 clock cycles
    - (lower case letters are 8-bits; upper case letters are 16-bit accesses)
    - Simple rule – count letters for best case # of clock cycles

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ADCA #*opr8i*	IMM	89 ii	P	P
ADCA *opr8a*	DIR	99 dd	rPf	rfP
ADCA *opr16a*	EXT	B9 hh ll	rPO	rOP
ADCA *oprx0_xysp*	IDX	A9 xb	rPf	rfP
ADCA *oprx9,xysp*	IDX1	A9 xb ff	rPO	rPO
ADCA *oprx16,xysp*	IDX2	A9 xb ee ff	frPP	frPP
ADCA [D,*xysp*]	[D,IDX]	A9 xb	fIfrPf	fIfrfP
ADCA [*oprx16,xysp*]	[IDX2]	A9 xb ee ff	fIPrPf	fIPrfP

Another Timing Example

- **Recall that "D" is a 16-bit register comprised of A:B**
- **ADDD $1247, X**
  - IDX2 format – access detail is "fRPP" for HCS12
    - f – free cycle (to add address to computation performed, memory bus idle)
    - R – 16-bit data read
    - P – 16-bit program word access to fetch next instruction
    - P – 16-bit program word access to fetch next instruction
  - Total is 4 or 5 clock cycles
    - 4 for minimum; plus 1 if value of X+$1247 is odd (straddles word boundaries)

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ADDD #*opr16i*	IMM	C3 jj kk	PO	OP
ADDD *opr8a*	DIR	D3 dd	RPf	RfP
ADDD *opr16a*	EXT	F3 hh ll	RPO	ROP
ADDD *oprx0_xysp*	IDX	E3 xb	RPf	RfP
ADDD *oprx9,xysp*	IDX1	E3 xb ff	RPO	RPO
ADDD *oprx16,xysp*	IDX2	E3 xb ee ff	fRPP	fRPP
ADDD [*D,xysp*]	[D,IDX]	E3 xb	fIfRPf	fIfRfP
ADDD [*oprx16,xysp*]	[IDX2]	E3 xb ee ff	fIPRPf	fIPRfP

Preview of Labels for Prelab 2

- **Labels are a convenient way to refer to a particular address**
  - Can be used for program addresses as well as data addresses
  - You know it is a label because it starts in column 1 (":" is optional)
- **Assume you are currently assembling to address $4712**
  - (how you do that comes in the next lecture)

```
Mylabela:
          ABA        ; this is at address $4712
Mylabelb:
Mylabelc
          PSHA       ; this is at address $4713
```

  - The following all do EXACTLY the same thing:
    - JMP $4713
    - JMP Mylabelb
    - JMP Mylabelc

Preview of Assembler Psuedo-Ops

- **The following are assembler directives, not HC12 instructions**
  - Labels – refer to an address by name instead of hex number
  - ORG: define the address where data/code starts
  - DS: Define Storage (allocate space in RAM)
  - DC: Define Constant (allocate space in ROM/flash)
  - EQU: Equate (like an equal sign for assembler variables)
- **This is for orientation when looking at code**
  - Specifics in the next lecture

Lecture 3 Lab Skills

- **Write an assembly language program and run it**
- **Manually convert assembly language to hex**
- **Manually convert hex program to assembly language**

Lecture 3 Review

- **CPU12 programmer model**
  - Registers
  - Condition codes
- **Memory Addressing modes**
  - Given an instruction using one of the modes described and some memory contents, what happens?
- **Assembly**
  - Given some assembly language, what is the hex object code?
  - Given some hex object code, what is the assembly language
- **Simple timing**
  - Given an encoded instruction, what is the minimum number of clocks to execute?
    - Be able to count number of letters in the timing column
    - We do not expect you to figure out all the rules for straddling word boundaries etc.
  - Branch cycle counting covered in next lecture