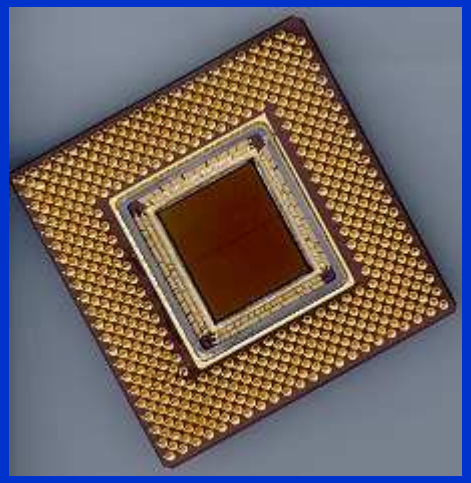


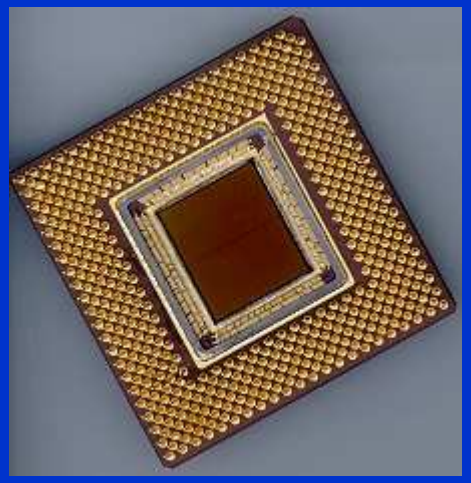
Microcontroller-Based Digital System Design

Module 1

Microcontroller Programming Techniques

Module 1

- **Learning Outcome: "An ability to program a microcontroller to perform various tasks"**
  A. **Microcontroller Architecture and Programming Model**
  B. **Microcontroller Instruction Set Overview**
  C. **Assembly Language Programming Techniques – Control Structures**
  D. **Assembly Language Programming Techniques – Control Structure Applications**
  E. **Assembly Language Programming Techniques – Table Lookup**
  F. **Assembly Language Programming Techniques – Parameter Passing**
  G. **Assembly Language Programming Techniques – Macros and Structured Programming**



Microcontroller-Based Digital System Design

Module 1-A
Microcontroller Architecture and Programming Model

Reading Assignment: *Meyer* Chp 3, pp. 1-29

Learning Objectives:

- **<u>list</u> differences in "world views" regarding the role of microprocessors**
- **<u>define</u> characteristics that distinguish microprocessors**
- **<u>describe</u> the Freescale 68HC(S)12 architecture and programming model**
- **<u>identify</u> different types of memory and <u>describe</u> how each is used**
- **<u>identify</u> instruction addressing modes and syntax**
- **<u>describe</u> the key characteristics of a microprocessor programming model**

Outline

- **Characteristics that distinguish microprocessors**
- **Taxonomy of microprocessors from an application viewpoint**
- **Challenges in selecting an education-appropriate microprocessor**
- **Basic architecture of the Freescale 68HC(S)12**
- **Instruction formats and data types of the 68HC(S)12**

Introduction

- **Two basic "world views" regarding the role of microprocessors are applicable**
  - ***general-purpose view:*** **a microprocessor is an integral part of a machine that runs "shrink-wrapped" software (or on which user-programmed applications can be developed and run) –** ***user programmable***
  - ***embedded view:*** **a microprocessor is a basic digital system building block that can be used to build intelligent products –** ***non-user-programmable***

Introduction

- **Why this distinction is important:**
  - **different *architectural/organizational* characteristics of microprocessors can make them more/less suited for a given application**
  - **the "goodness" or "badness" of a particular microprocessor can only be evaluated in the context of the *intended application***

Characteristics That Distinguish μPs

- **General-purpose applications generally require processors that have the following characteristics:**
  - **support *time-sharing* operating systems**
  - **support *virtual memory*, with multi-level cache and dynamic RAM**
  - **support for *DMA-driven* I/O**
  - **large register sets**
  - **integrated floating point hardware**
  - **primary view of interrupts is that they are "*irritations*" (called "exceptions")**

Characteristics That Distinguish μPs

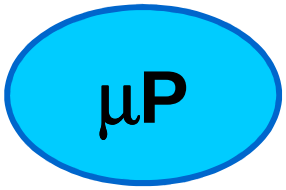
- **Embedded applications generally require processors that have the following characteristics:**
  - ***flexible interrupt structure*** **– interrupts are a "way of life" in event-driven systems**
  - ***fast context switch*** **(generally implies need for small register set)**
  - ***mixture of digital and analog I/O*** **(to facilitate a variety of interfaces with external devices)**
  - ***amenability of assembly-level patching*** **for time-critical code segments**

Taxonomy of Microprocessors

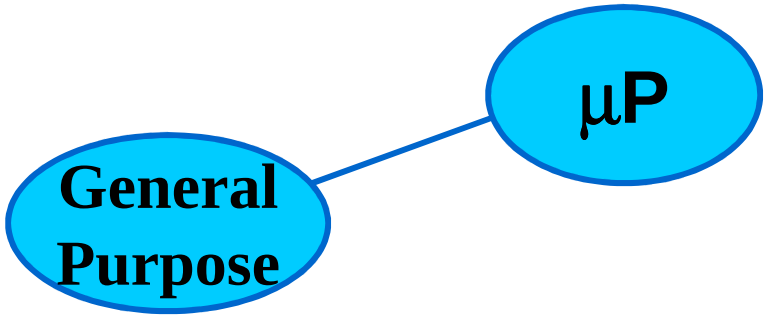
- **Common acronyms**
  - **CISC: complex instruction set computer**
  - **RISC: reduced instruction set computer**

    -or- **reduced instruction set cycles**
  - **DSP: digital signal processor**
- **Bit-width (ALU size) of current devices ranges from 4 to 64**

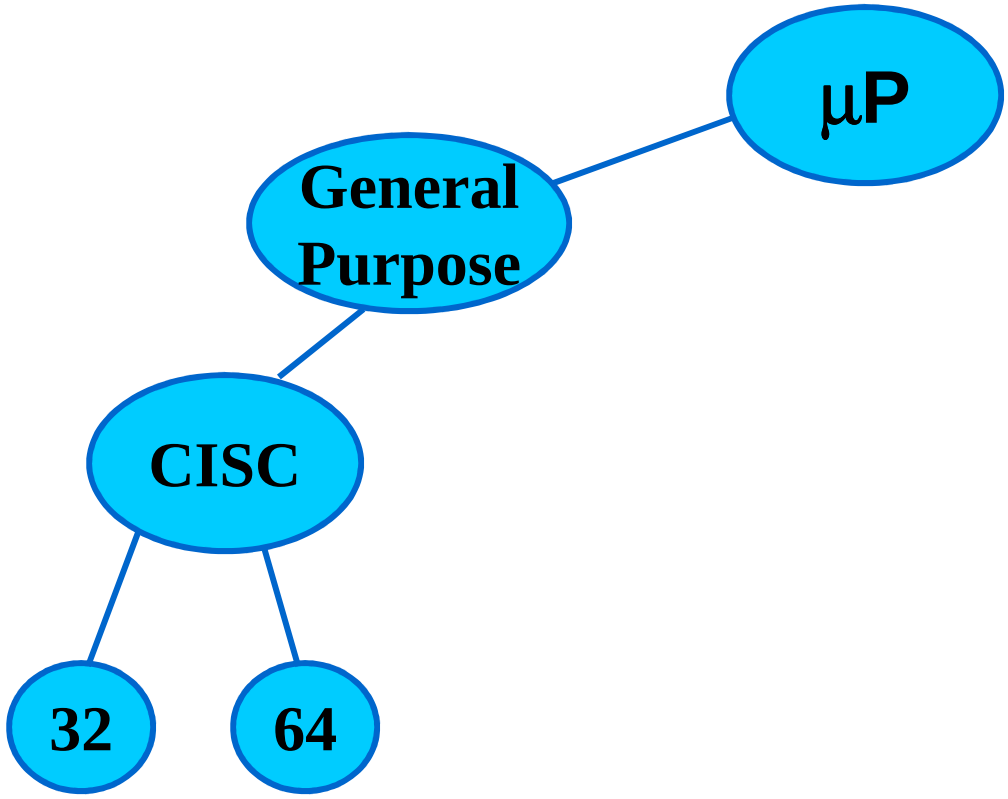
Taxonomy of Microprocessors



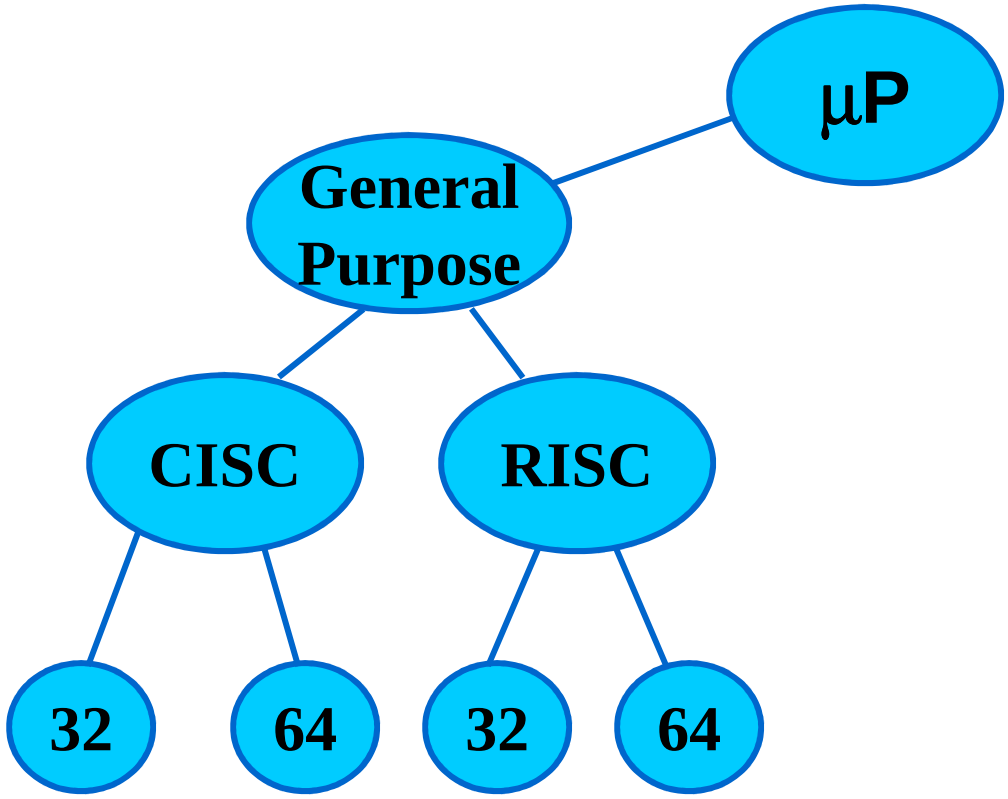
Taxonomy of Microprocessors



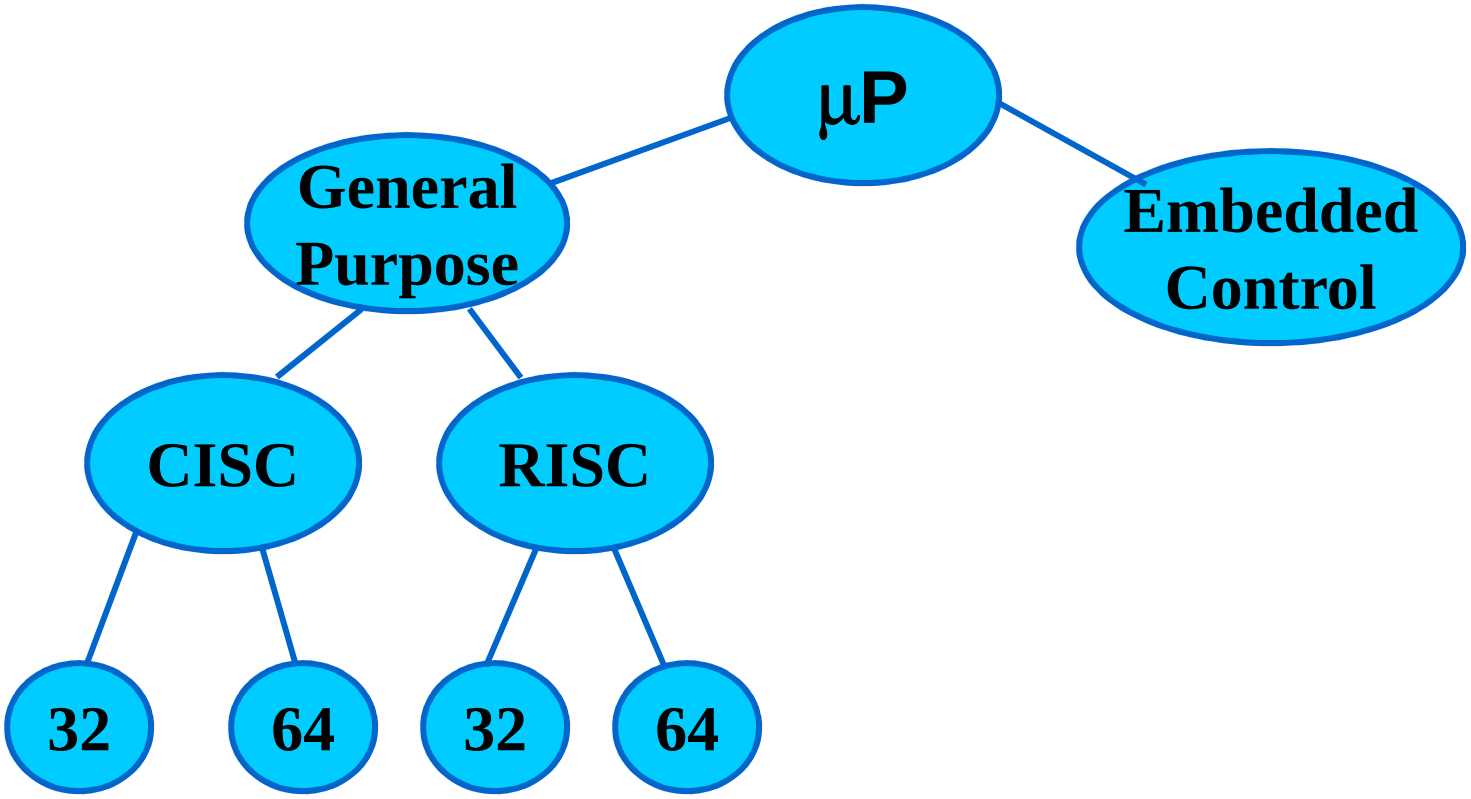
Taxonomy of Microprocessors



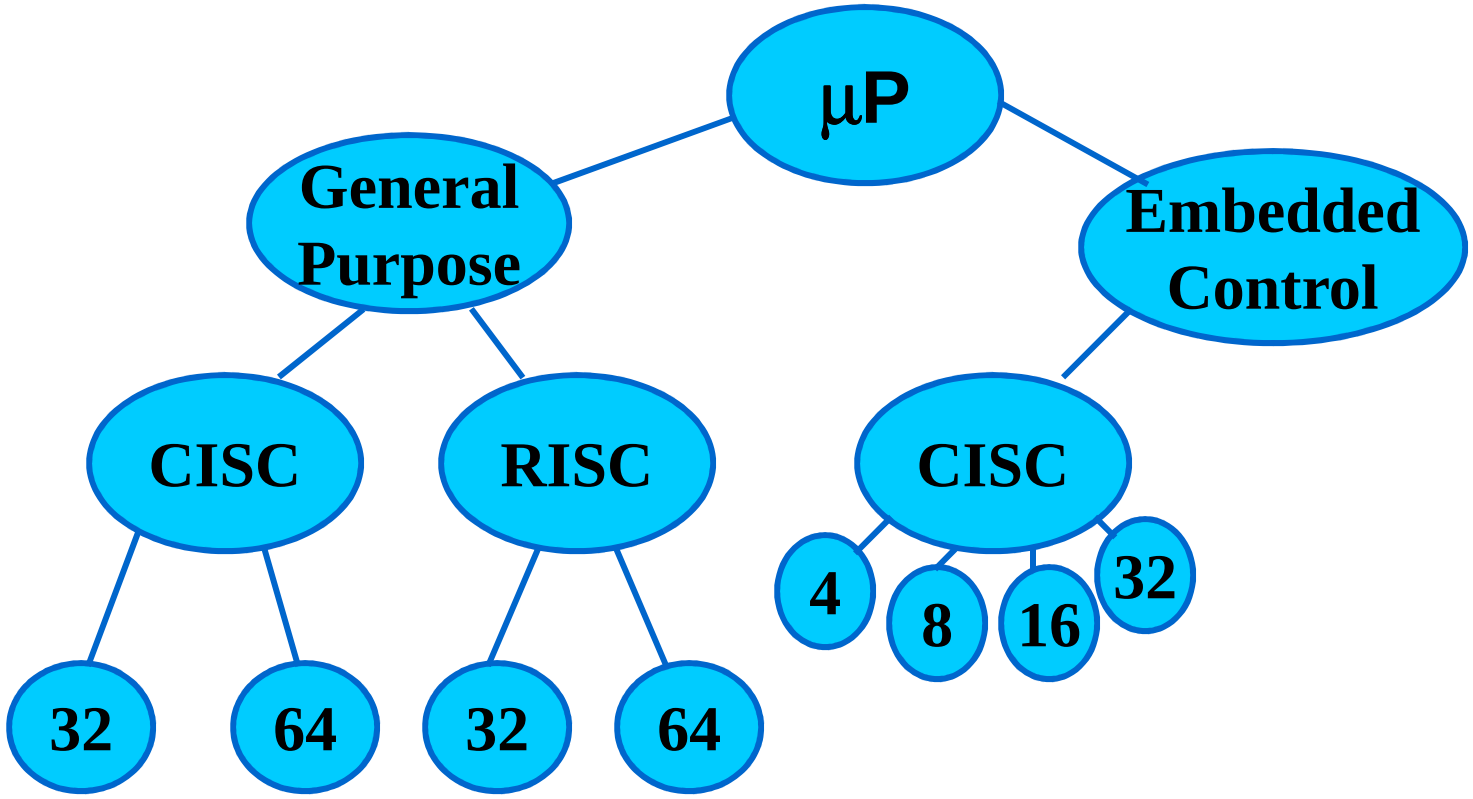
Taxonomy of Microprocessors



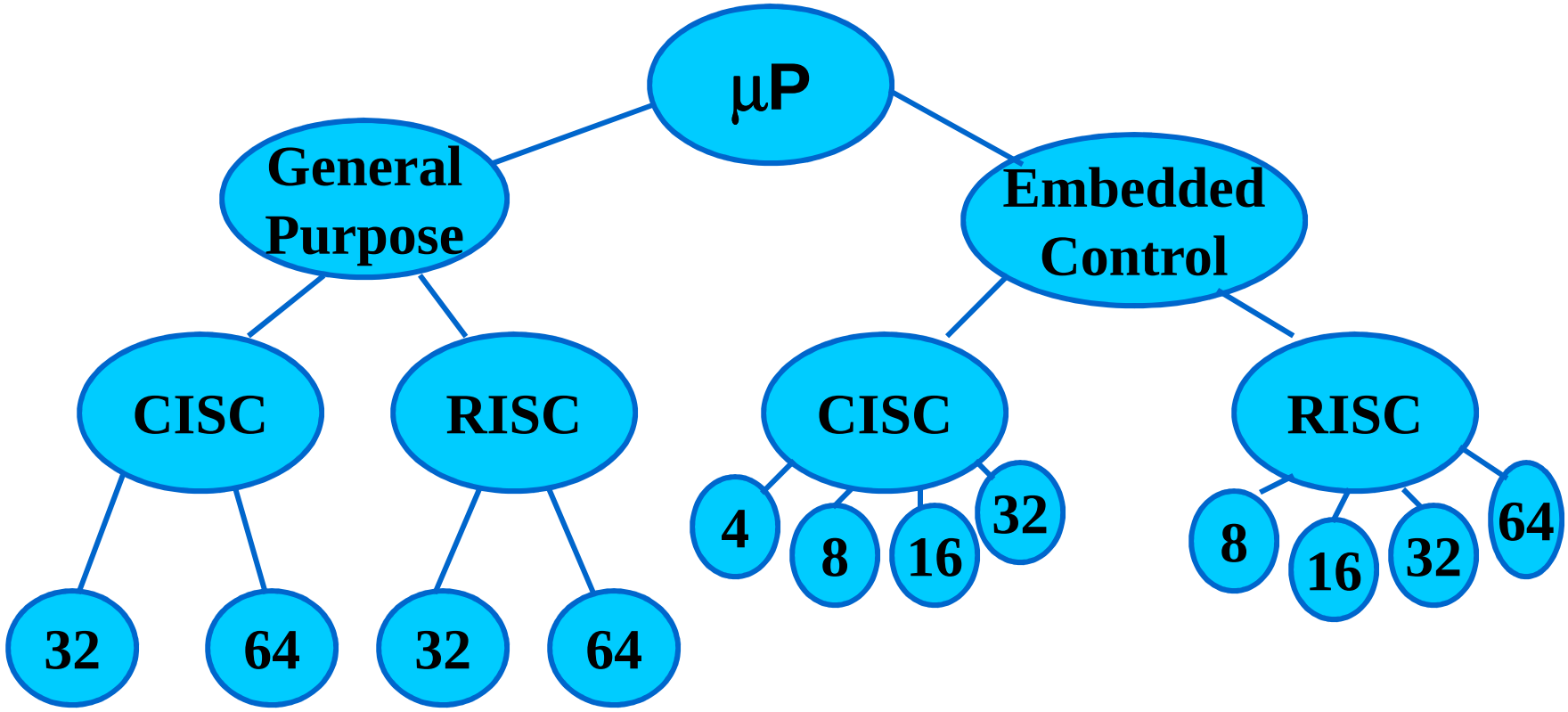
Taxonomy of Microprocessors



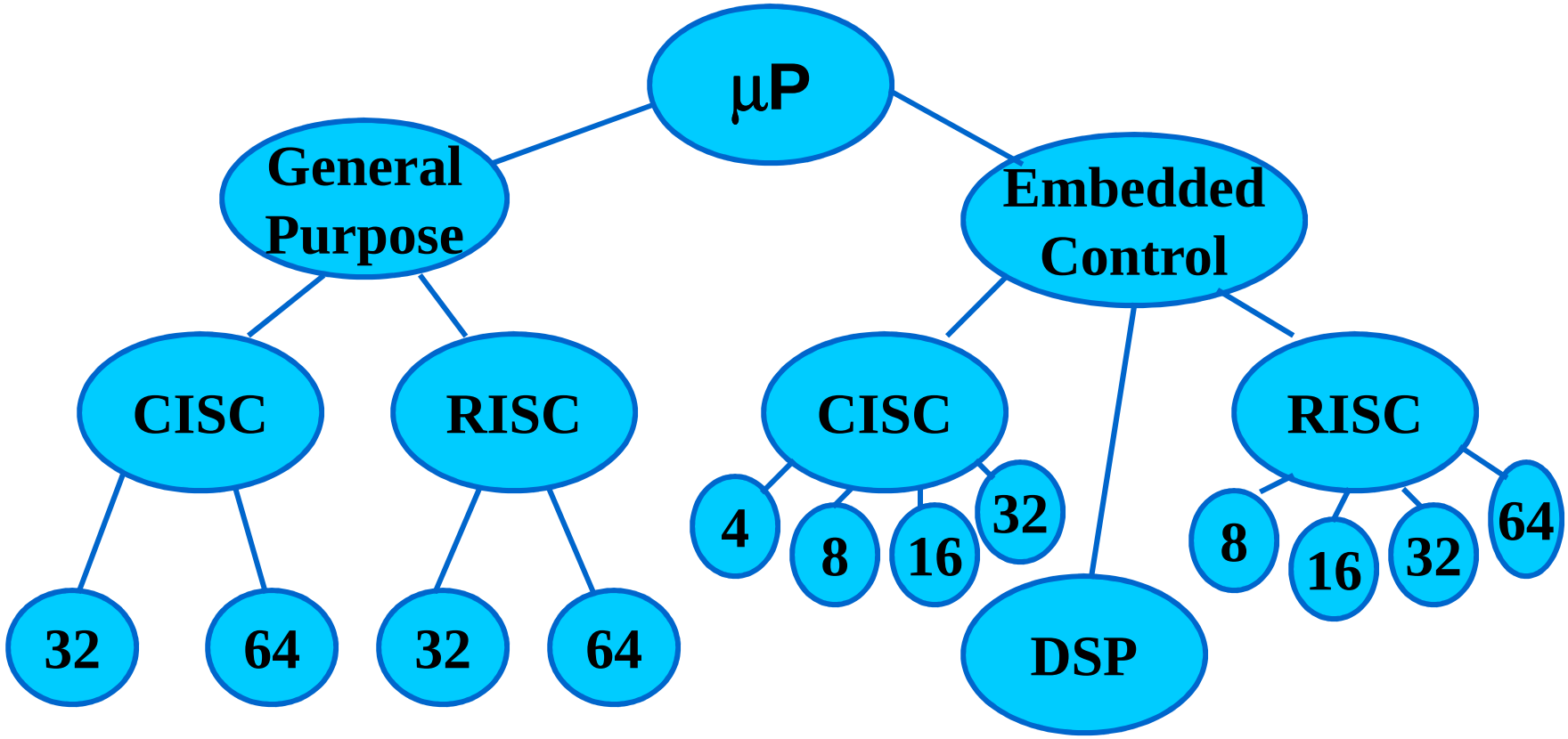
Taxonomy of Microprocessors



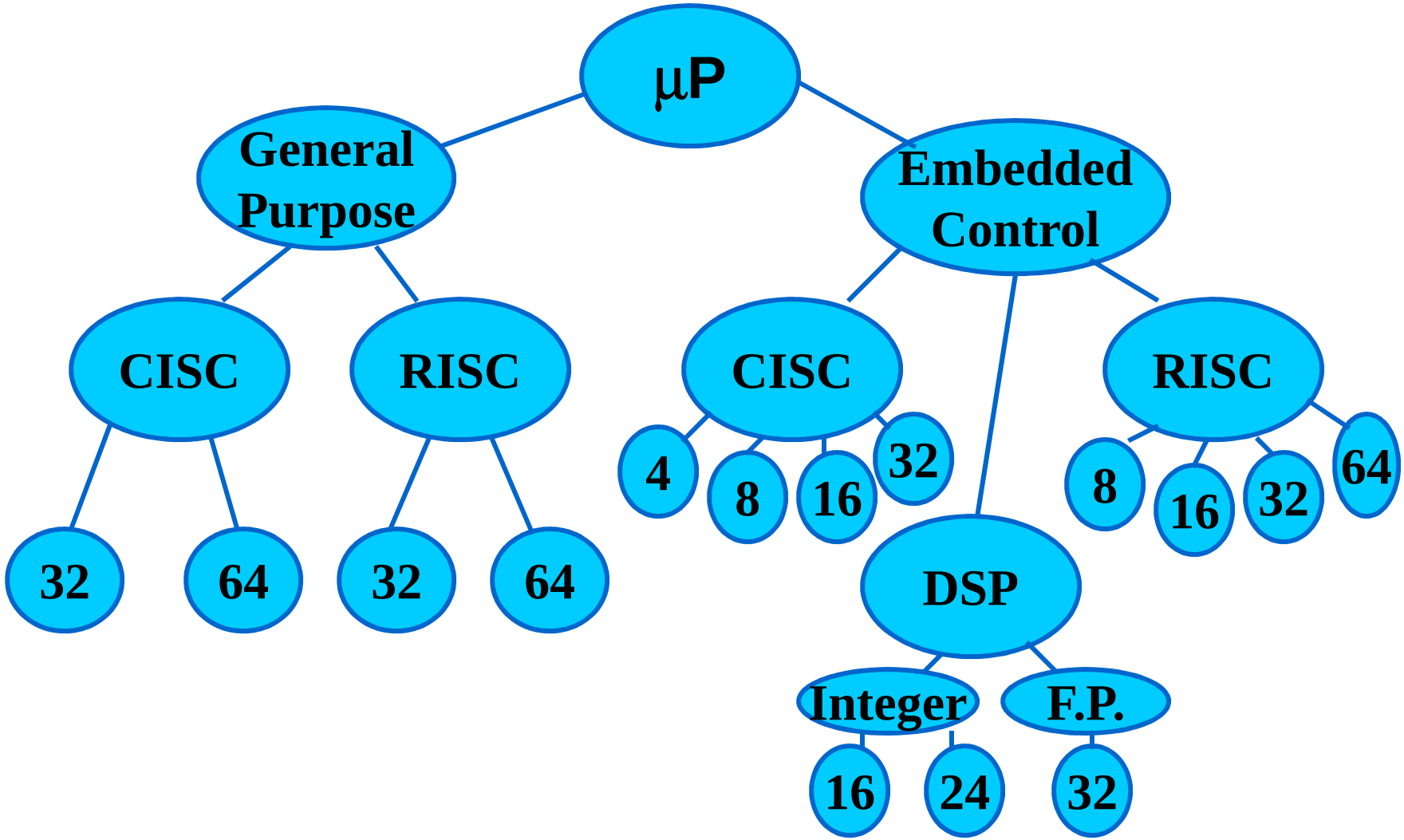
Taxonomy of Microprocessors



Taxonomy of Microprocessors



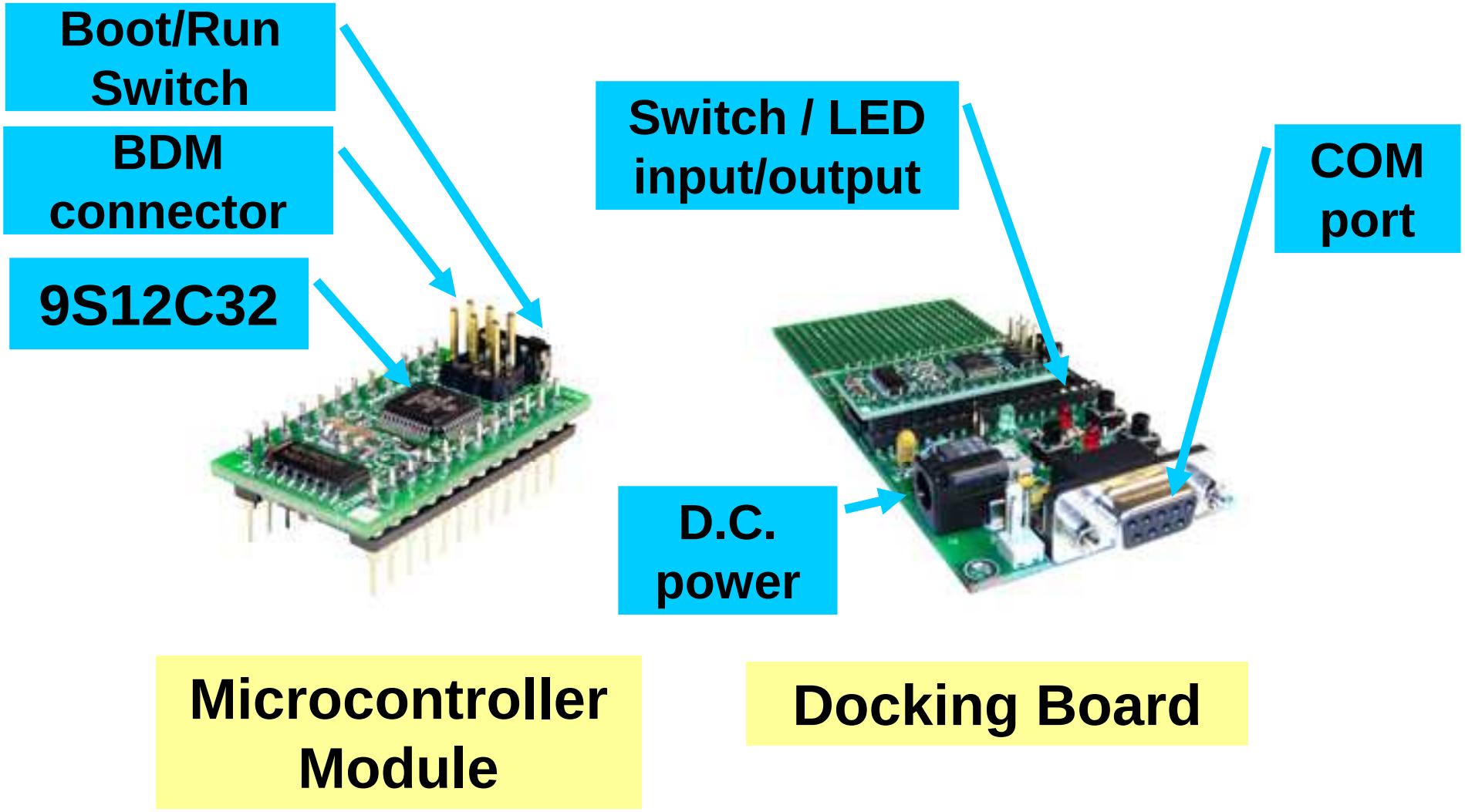
Taxonomy of Microprocessors



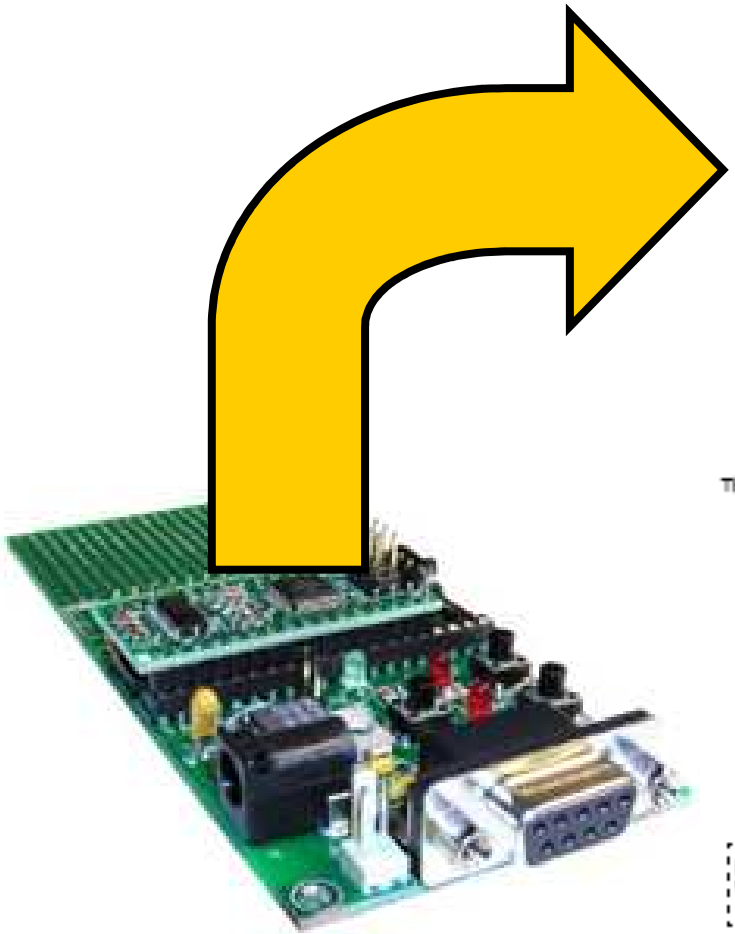
Choosing an Education-Appropriate μP

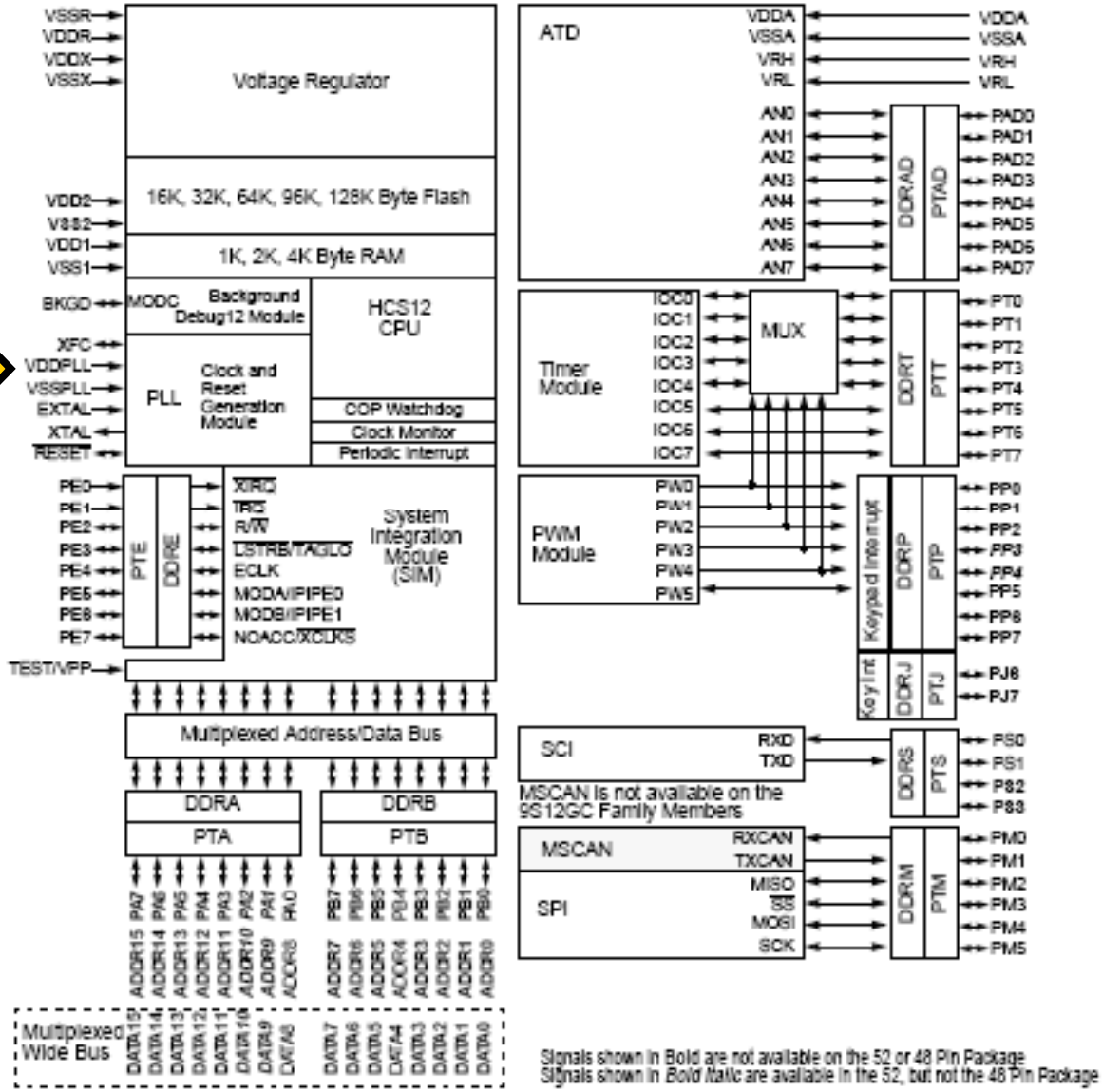
- **Goals:**
  - **introduce basic concepts of computer architecture and machine instruction sets**
  - **provide hands-on experience with a "real" device**
  - **expose students to the "embedded world"**
- **Many devices currently available can be used to achieve these goals**
- **We will use the Freescale 68HC(S)12 as our "architecture of choice" and focus on one variant, the 9S12C32**

Freescale 9S12C32 Development Kit



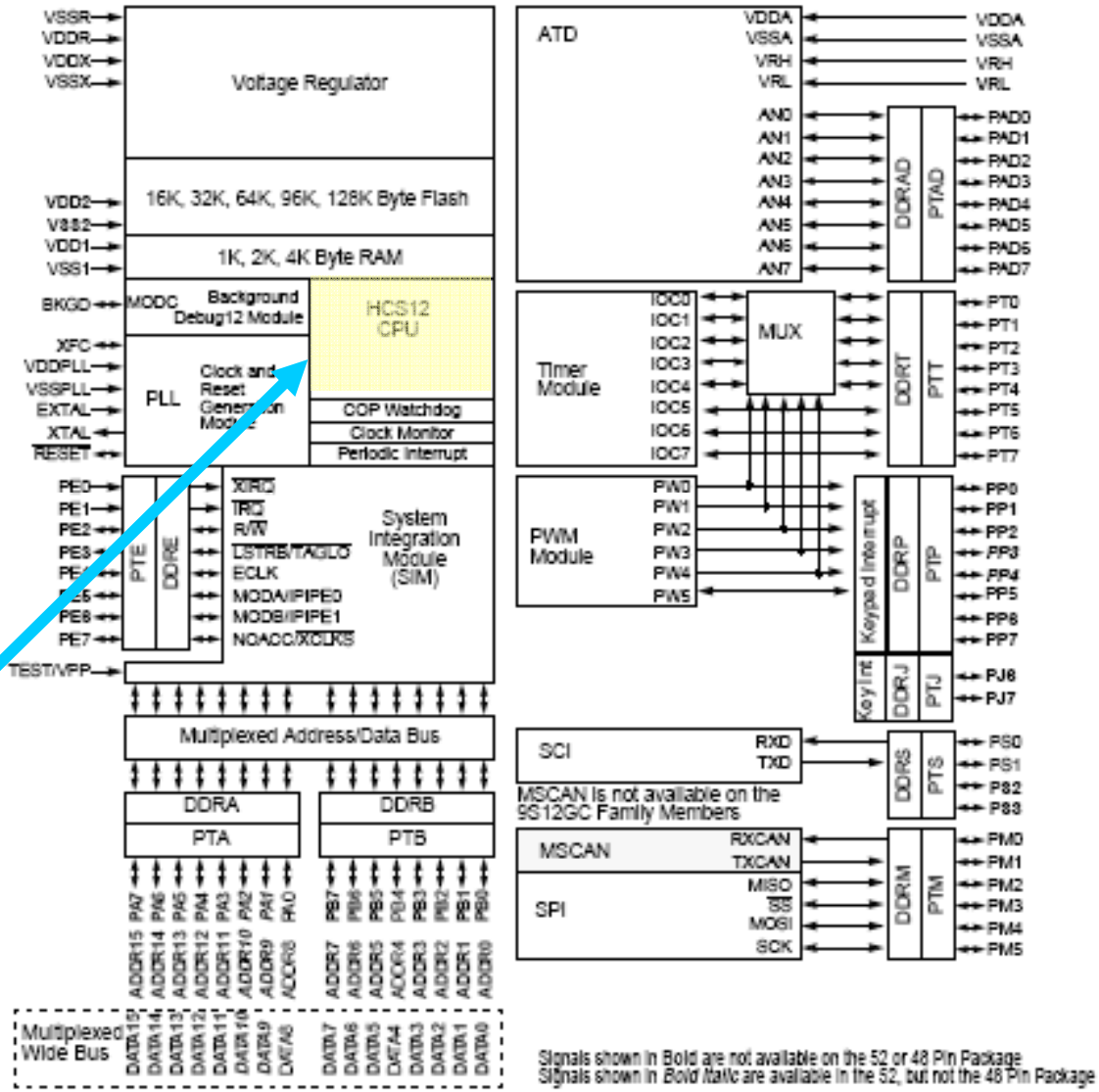
Overview of 9S12C32





Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

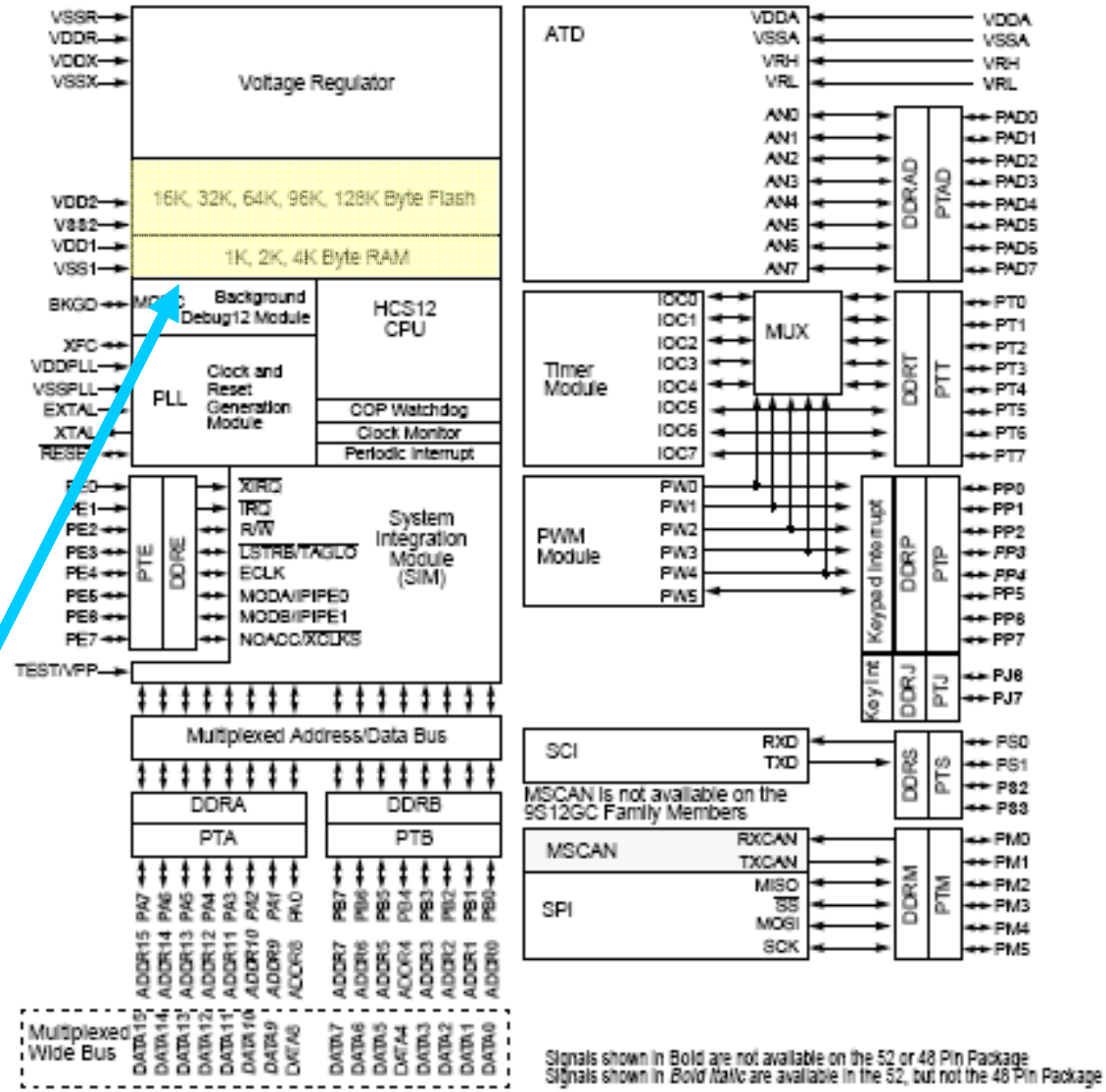


Several things to note…..

The HCS12 CPU has the same architecture and programming model as the HC12

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

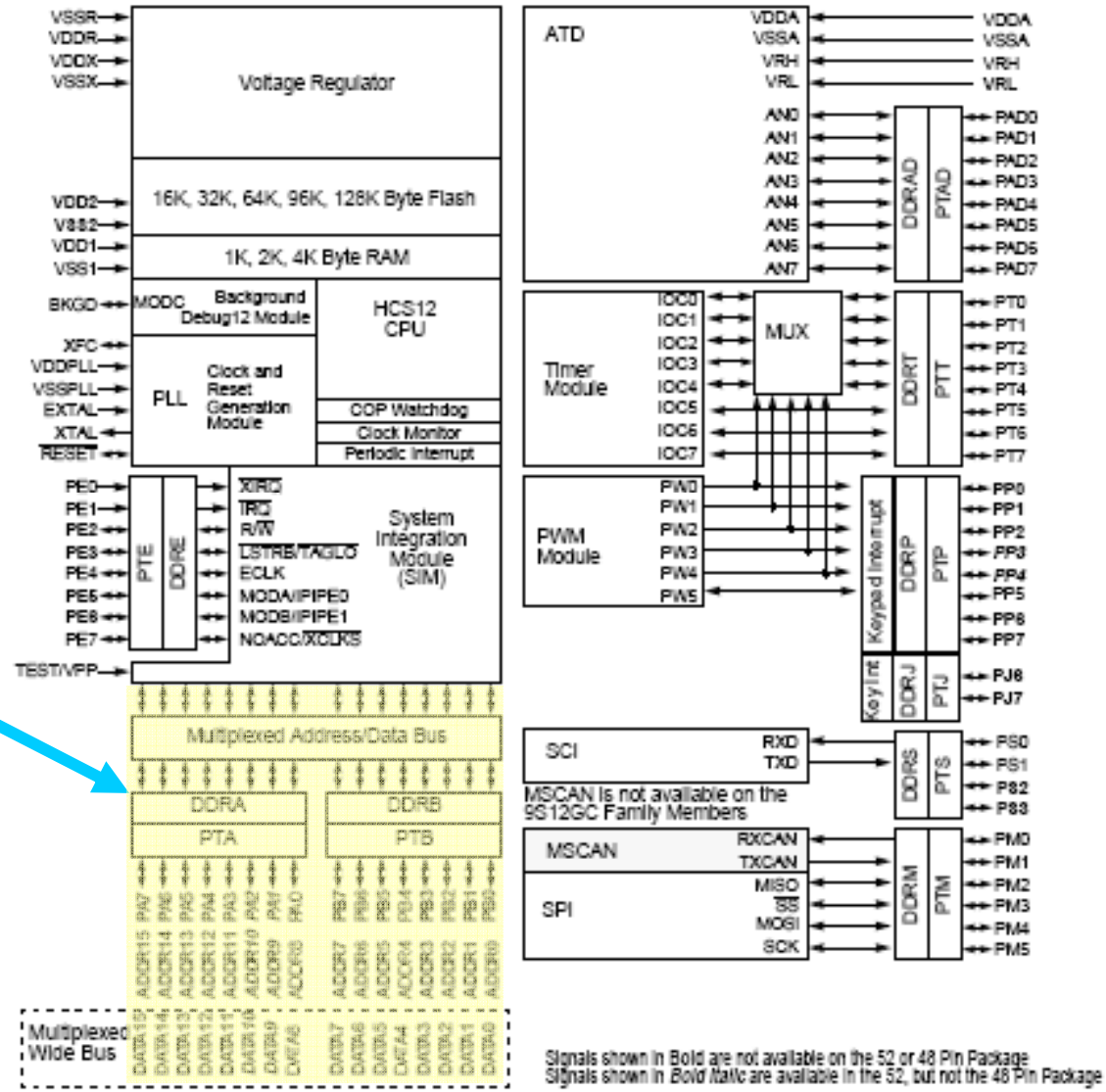


Several things to note…..

The 9S12C32 module has 2K of SRAM and 32K of Flash (no EEROM)

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

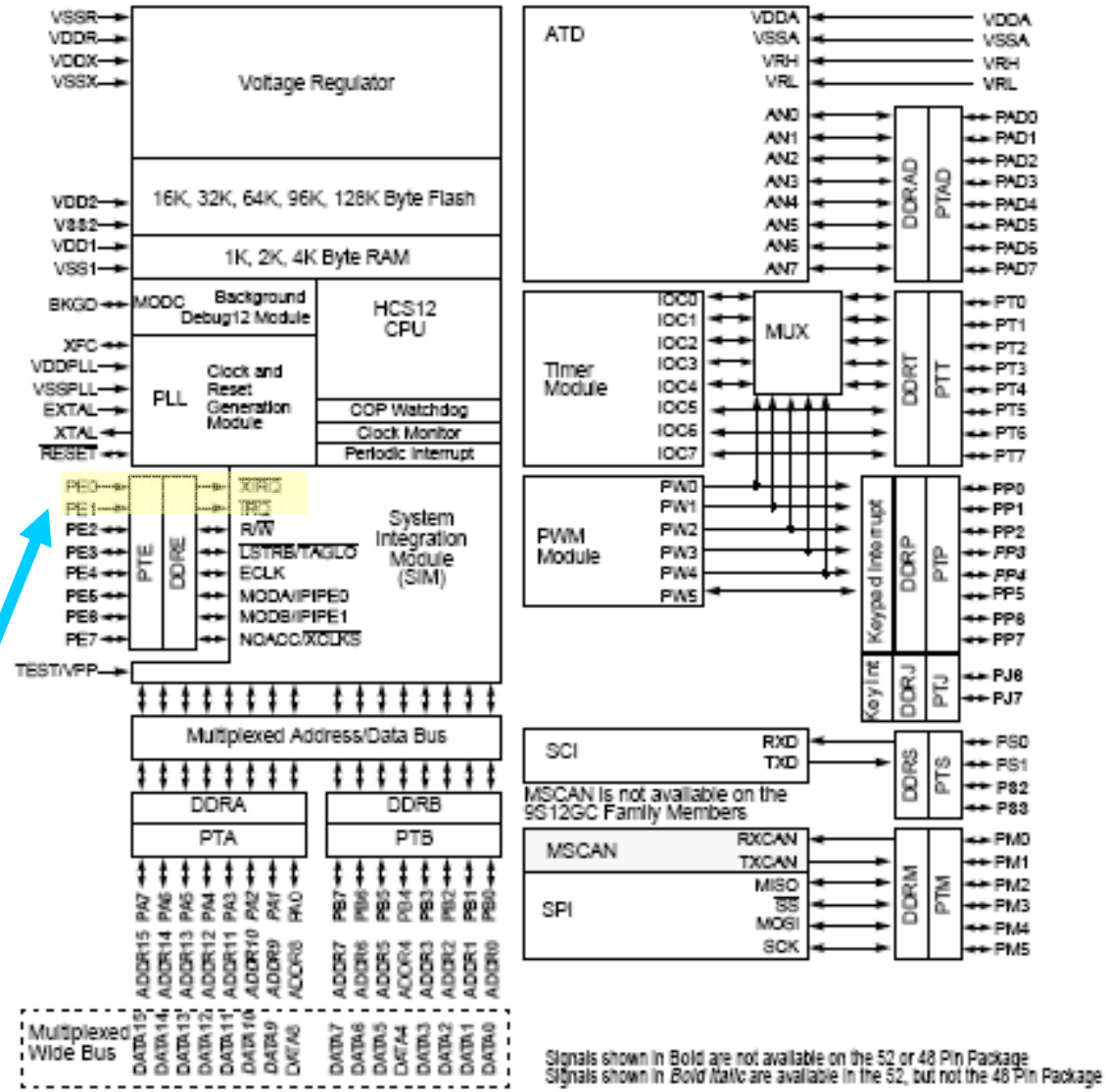


Several things to note…..

The 48-pin version of the chip on this module does not have Ports A & B padded out

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

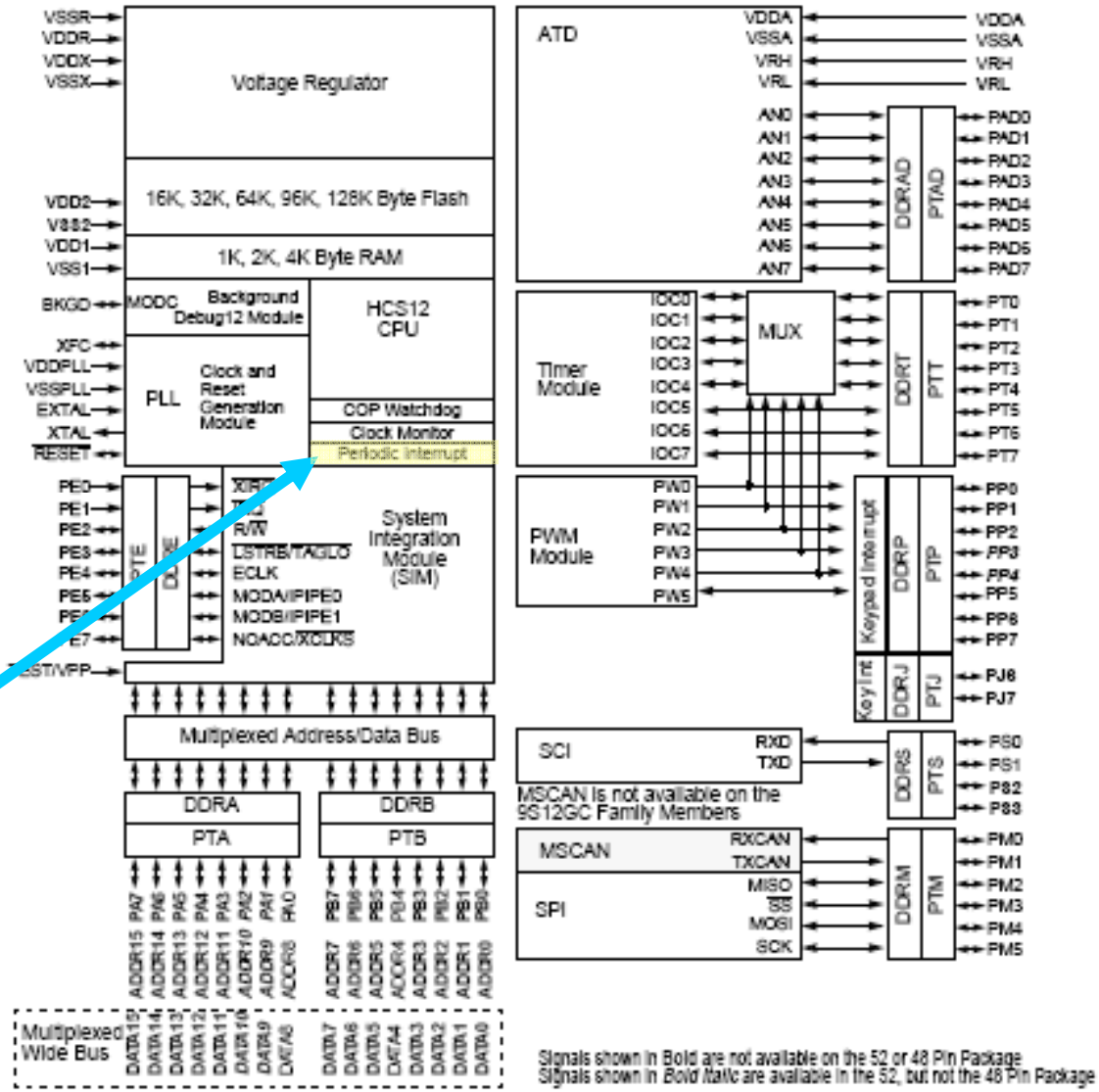


Several things to note…..

External interrupt pins are on Port E

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

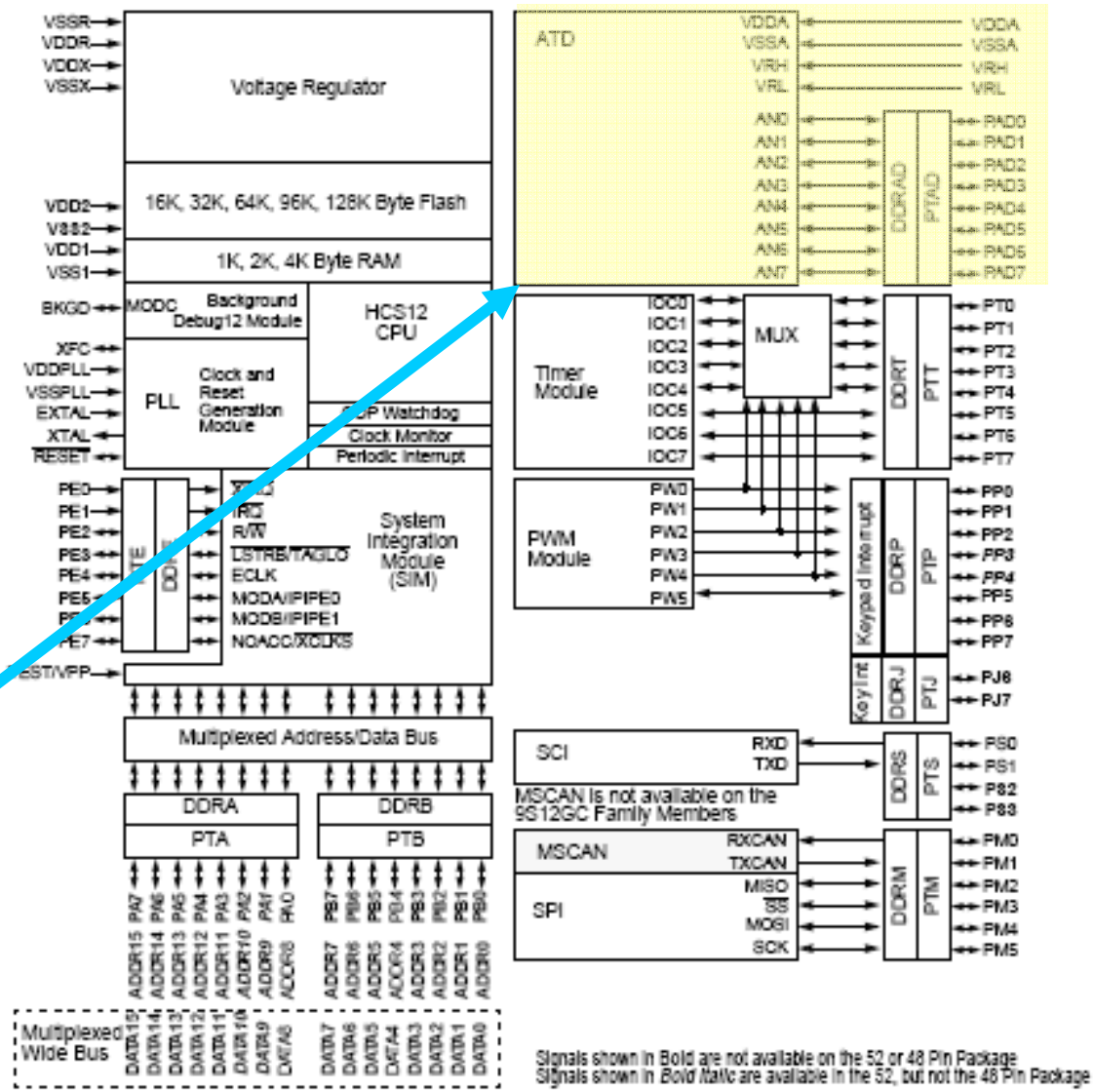


Several things to note…..

Real-time interrupt (RTI) module

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

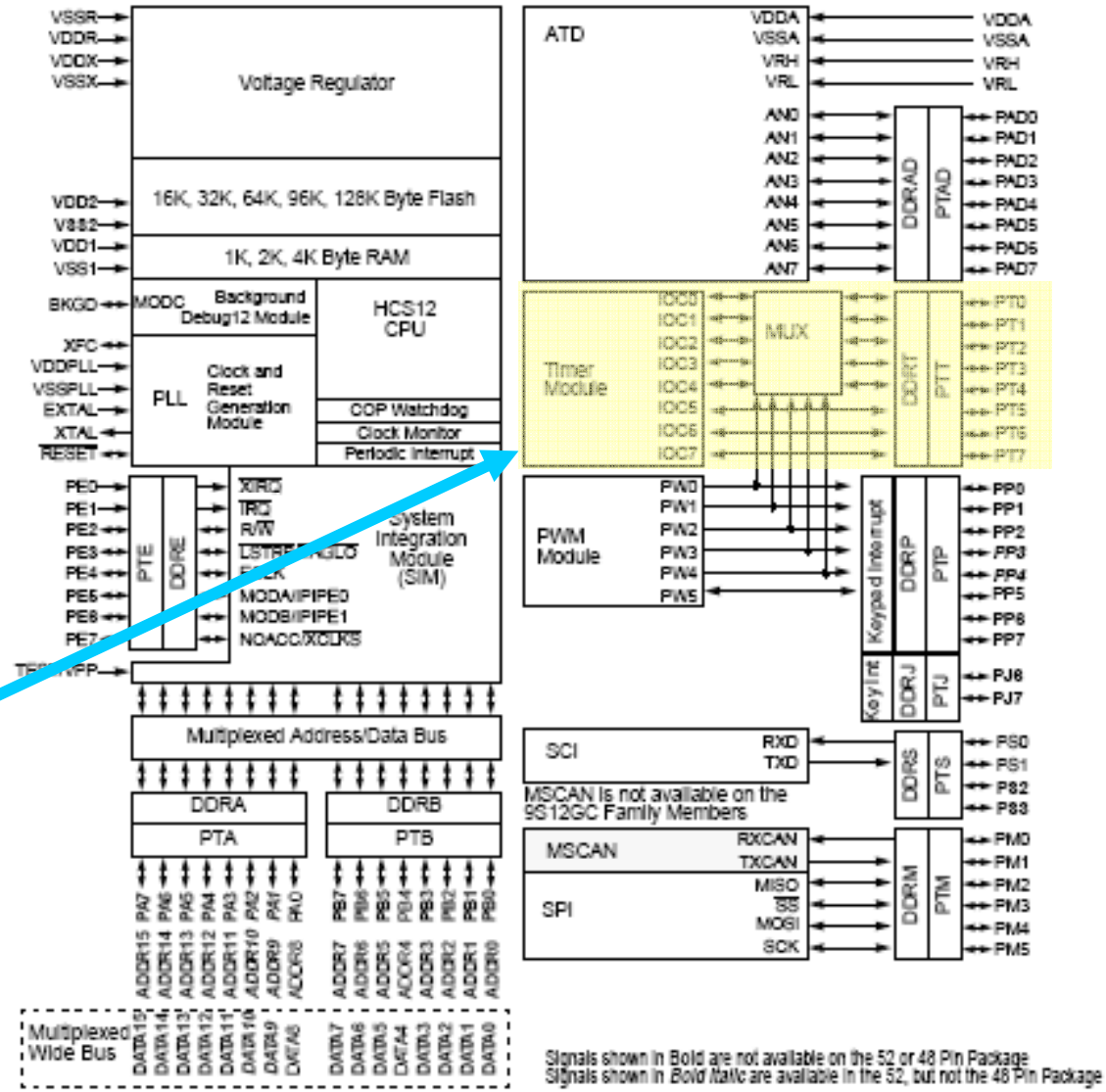


Several things to note…..

Analog-to-digital (ATD) converter module – inputs are on Port PAD

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

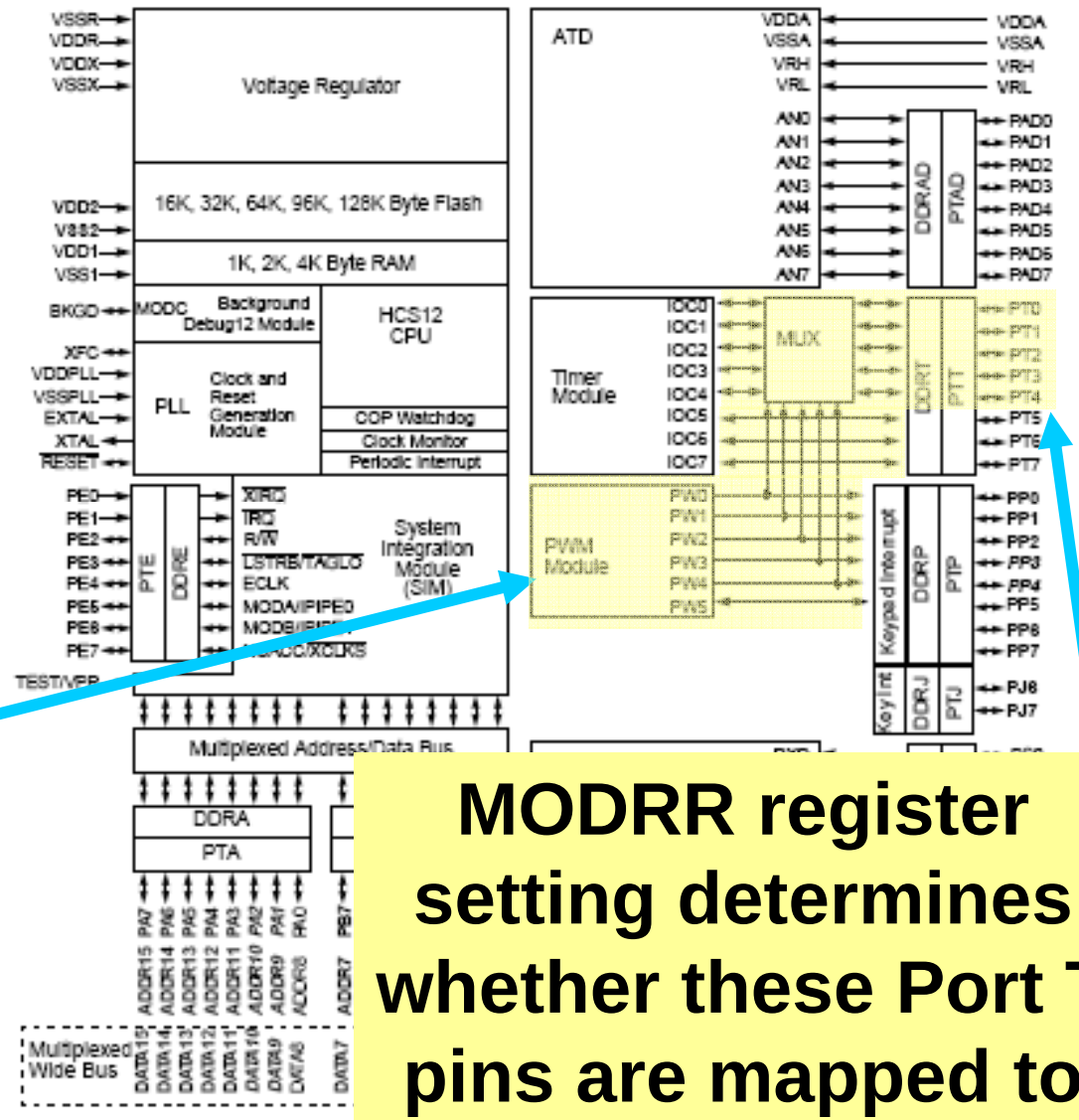


Several things to note…..

Timer (TIM) module – I/O on Port T

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram



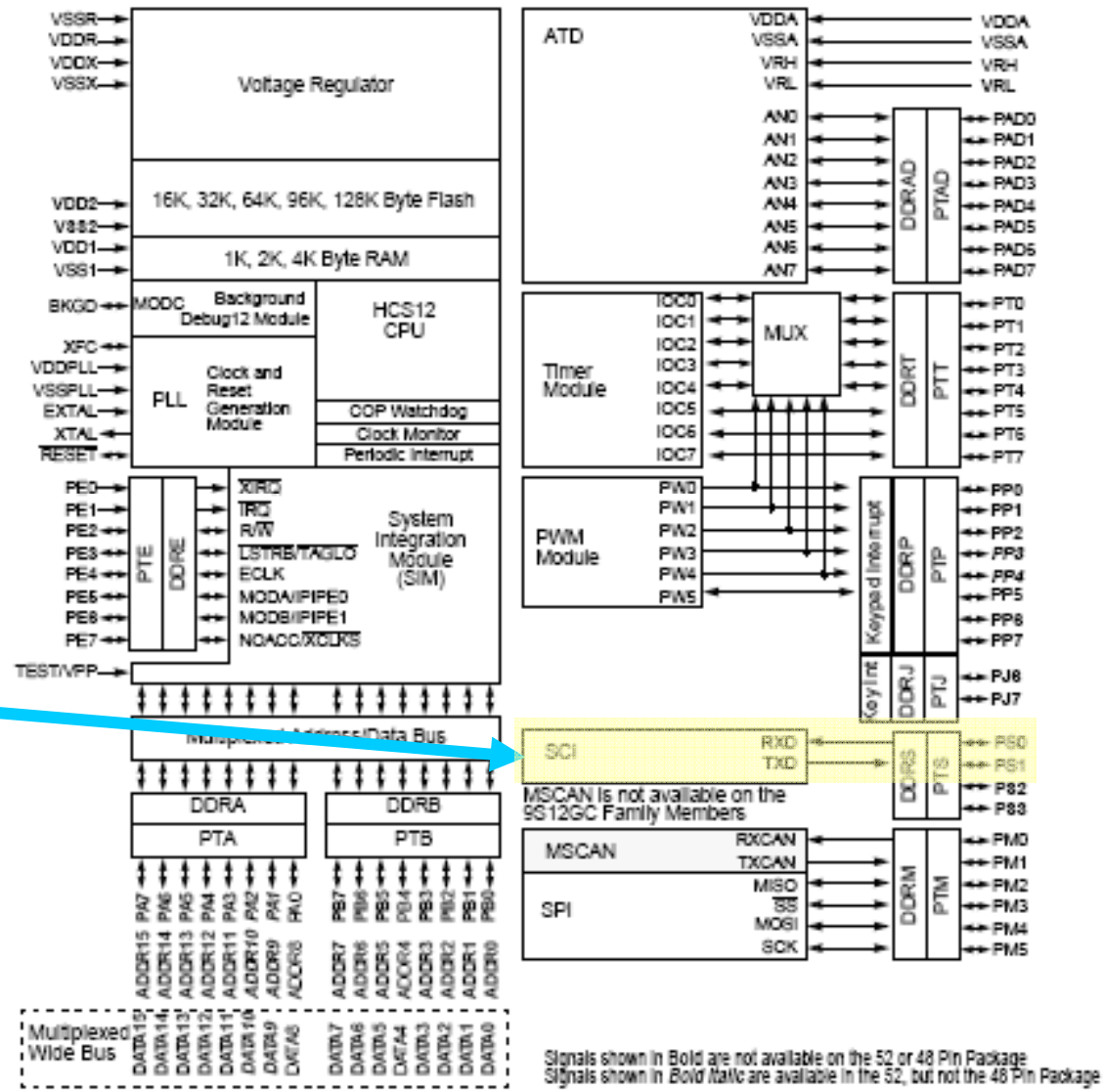
Several things to note…..

Pulse width modulator (PWM) – here, I/O shared with TIM module on Port T

MODRR register setting determines whether these Port T pins are mapped to the TIM or PWM

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

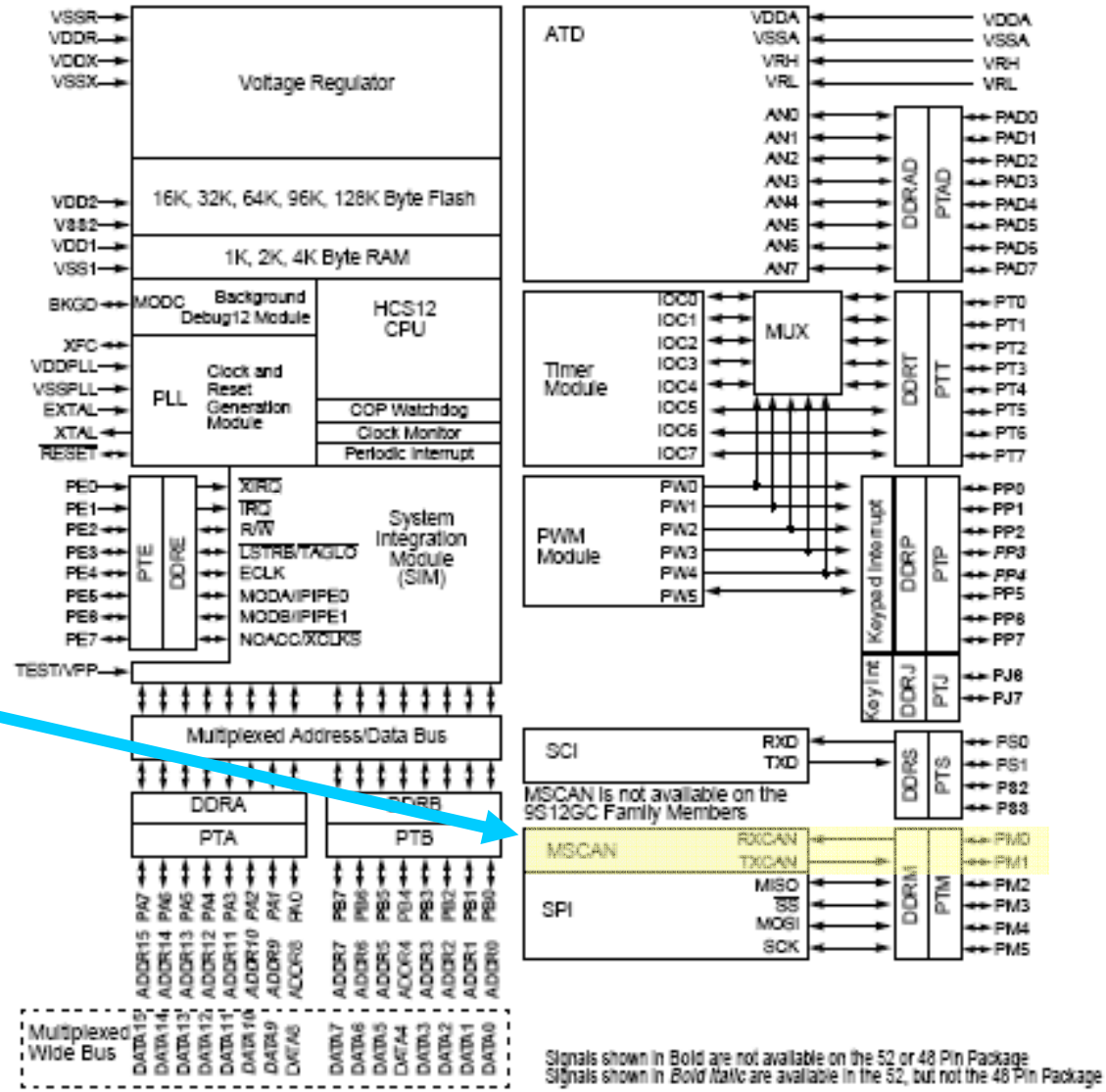


Several things to note…..

Asynchronous serial communications interface (SCI) on Port S

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram

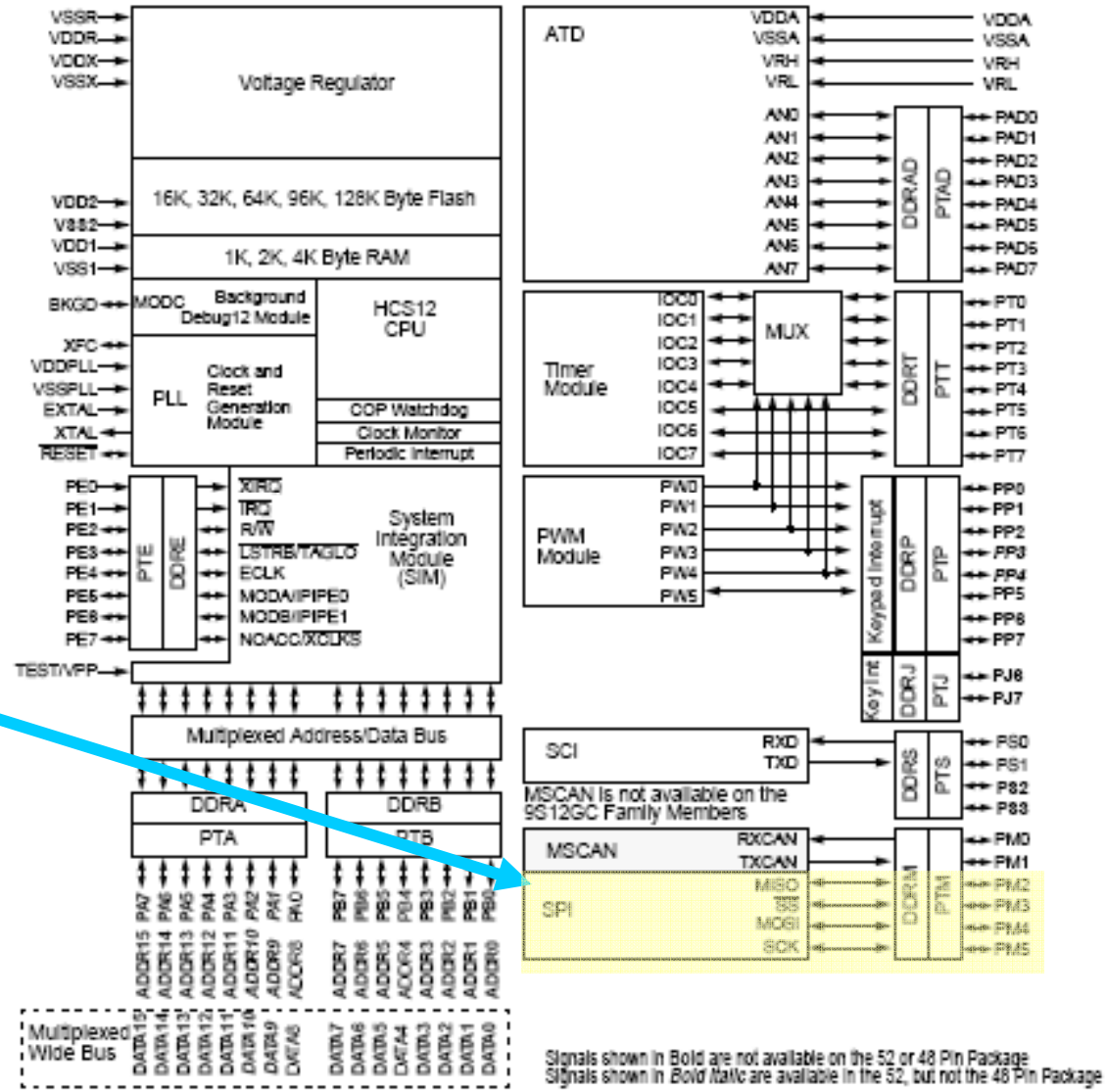


Several things to note…..

Controller area network (MSCAN) on Port M

Overview of 9S12C32

Figure 1-1 MC9S12C-Family Block Diagram



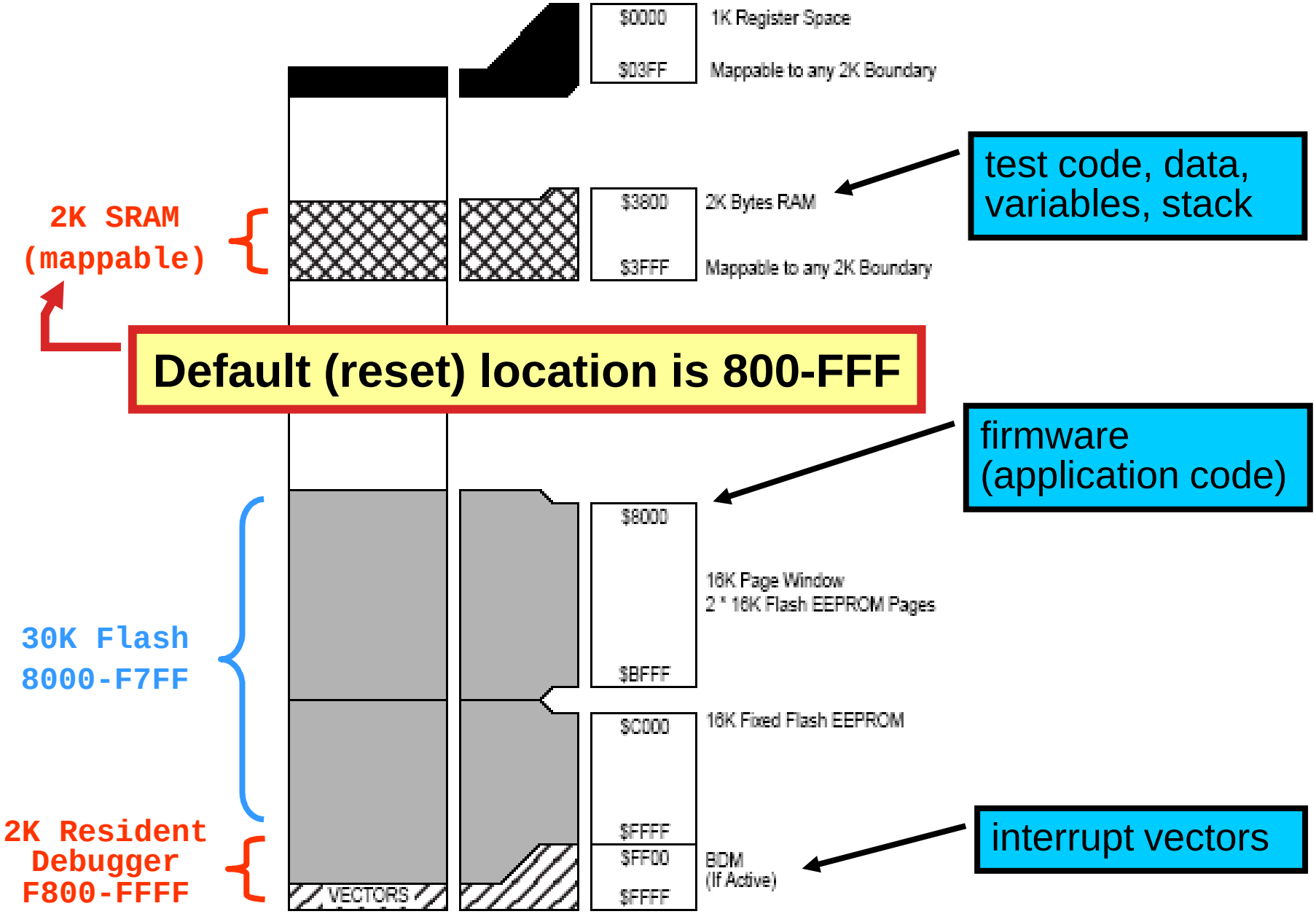
Several things to note…..

Synchronous peripheral interface (SPI) on Port M

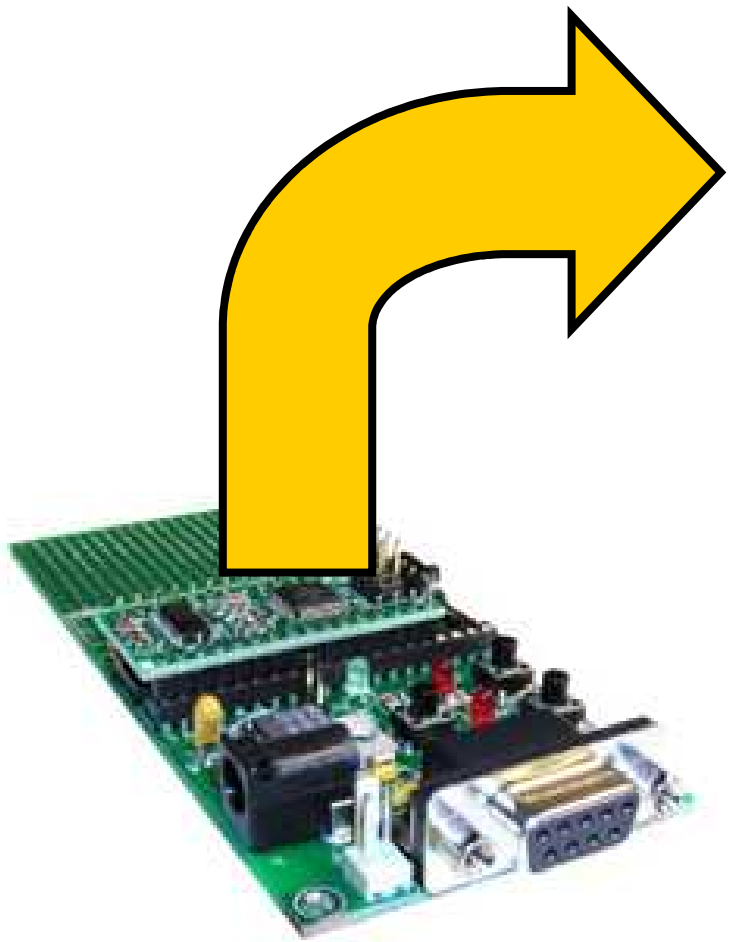
Memory Usages

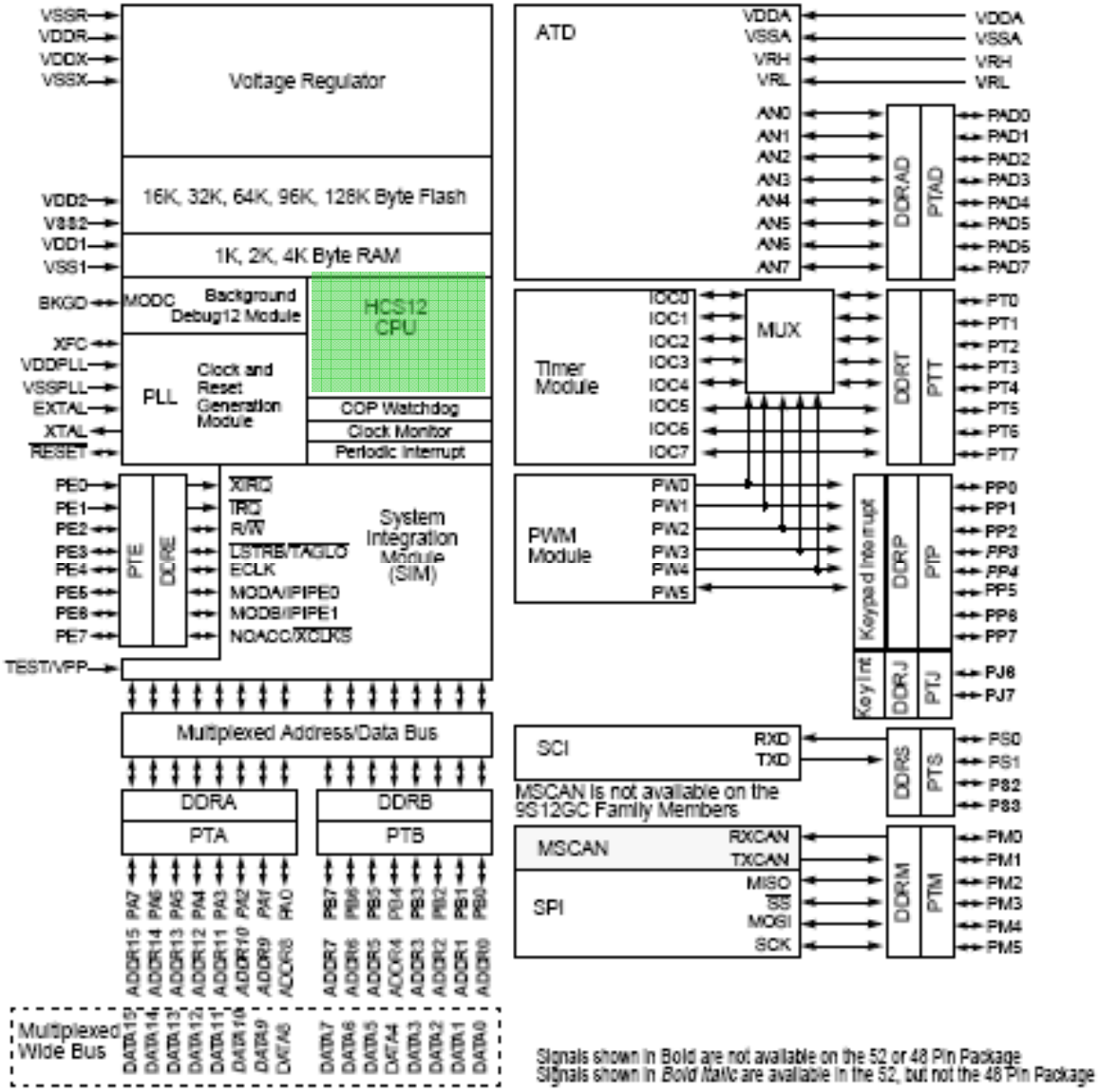
- **SRAM**
  - **Variables**
  - **Stack**
  - **Buffers**
  - **Test code**
- **Flash**
  - **“Turn-key” application code**
  - **Fixed message strings**
  - **Static data**
  - **Vectors (resets and interrupts)**

9S12C32 Memory Map

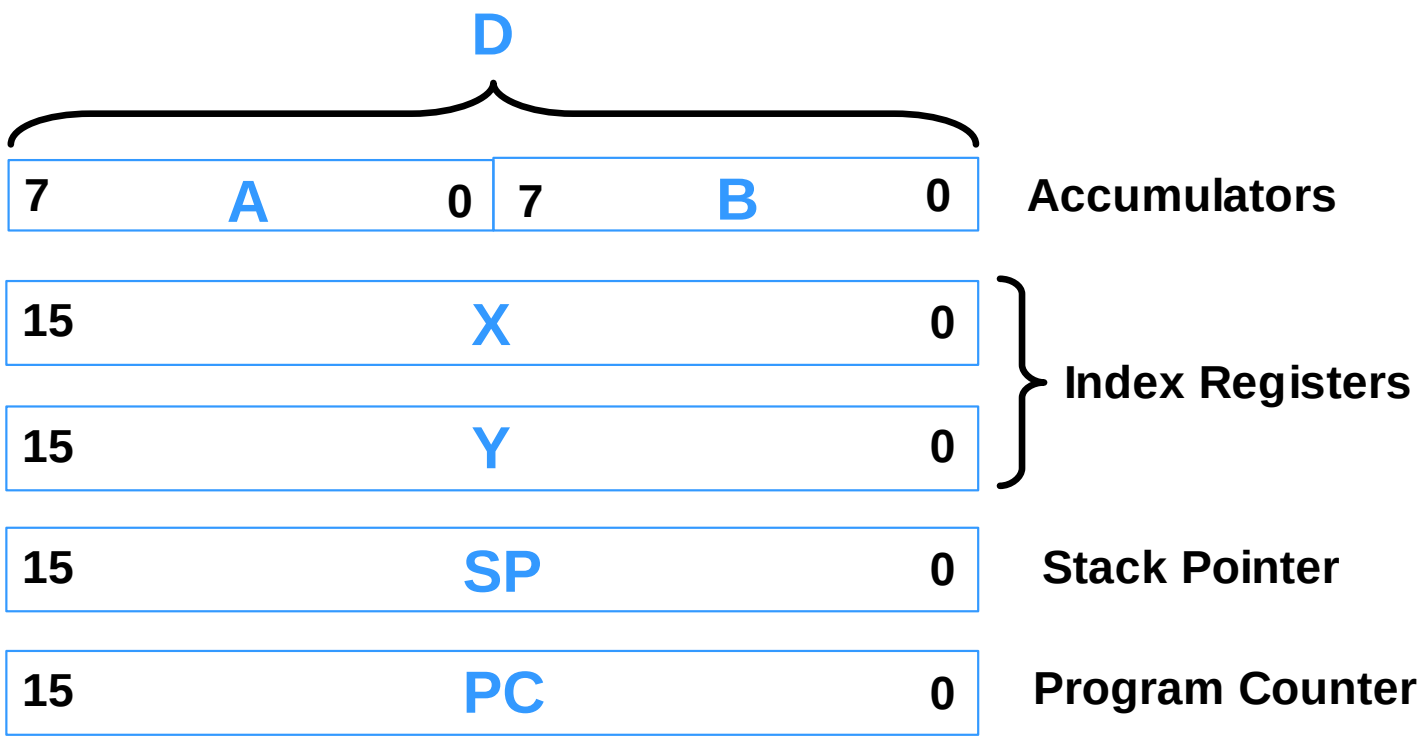


Overview of 68HC(S)12 Architecture





Freescale 68HC(S)12 Programming Model



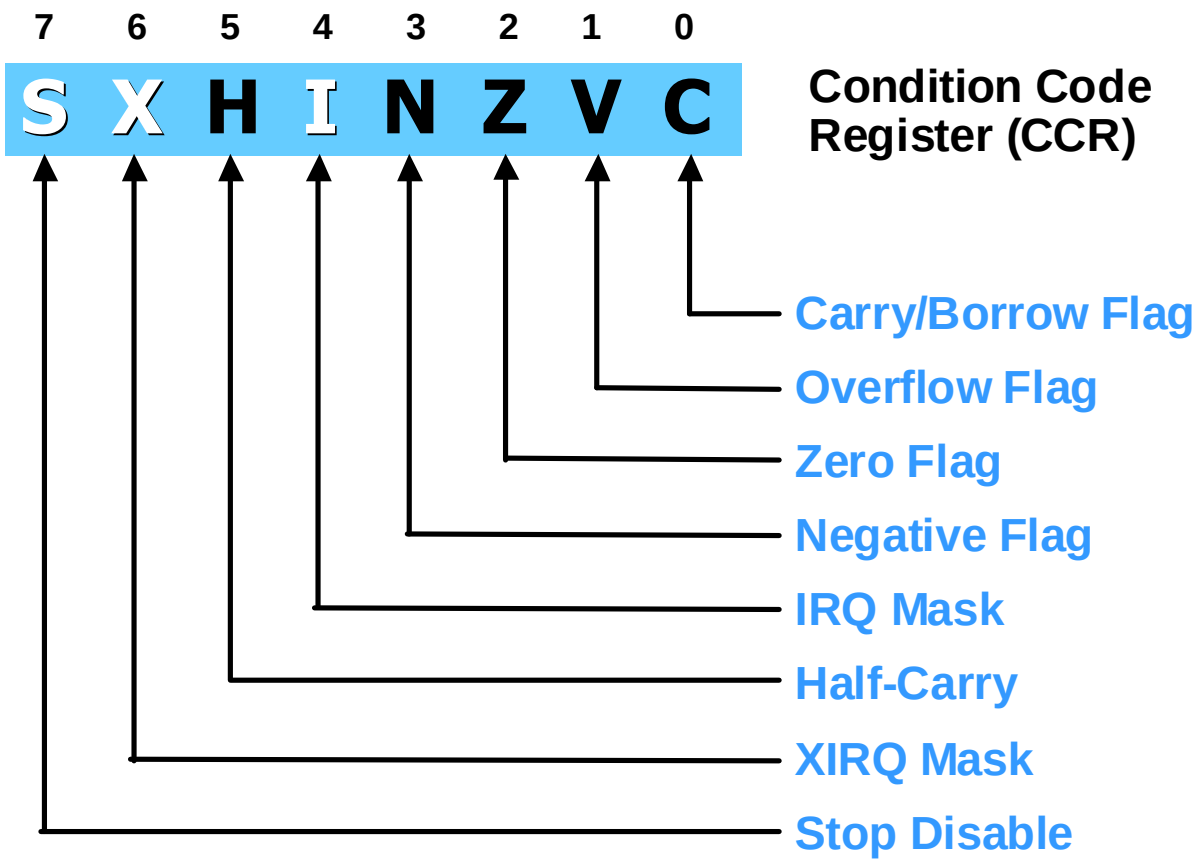
Register Usage and Functions

- **Accumulators** "**A**" and "**B**" (8-bit)
  - arithmetic calculations
  - logical manipulation of data
  - can be concatenated together to form a 16-bit accumulator (referred to as "D")
- **Program Counter** "**PC**" (16-bit)
  - points to next instruction to be executed
- **Stack Pointer** "**SP**" (16-bit)
  - points to top stack item
  - used for subroutine linkage, interrupts
  - also have PSH/PUL instructions

Register Usage and Functions

- **Index Registers** **"X" and "Y" (16-bit)**
  - **used as pointers to operands (typically within some type of data structure or string)**
  - **may be modified by addition of a constant or a register (accumulator) offset**
  - **auto increment/decrement supported**

Condition Code Register



ALU Condition Codes

- **"C" – "carry/borrow" flag (carry out of the sign position for addition, *complement* of carry out of sign position for *subtraction*)**
- **"V" – "overflow" flag (set if two's complement overflow has occurred)**
- **"Z" – "zero" flag (set if result of computation is zero)**
- **"N" – "negative" flag (most significant bit (sign) of computation)**
- **"H" – "half carry" flag (carry out of the lower 4-bits (nibble), only valid after ADD)**

Machine Control Condition Codes

- "I" – "IRQ interrupt mask"
  - "0" – IRQ is not masked (enabled)
  - "1" – IRQ is masked (disabled)
- "X" – "XIRQ interrupt mask"
  - "0" – XIRQ is not masked (enabled)
  - "1" – XIRQ is masked (disabled)
- "S" – "STOP instruction disable"
  - "0" – STOP instruction is enabled
  - "1" – STOP instruction is disabled

Instruction Formats and Data Types

- **Instruction length** varies from **one to six** bytes
- **Opcodes** may be **one or two** bytes
- A "**postbyte**" may follow an opcode to provide additional information about the type of addressing mode used
- An **offset** (one or two bytes) may follow a postbyte
- **Data types** supported include: bit, byte (8-bit), word (16-bit), double word (32-bit), packed BCD, and unsigned fractions

Addressing Modes

- **Definition: The CPU uses an *addressing mode* to determine the *effective address* of where an operand is stored in memory**
- **Commonly used addressing modes**
  - **"immediate" (data immediately follows opcode, i.e., is part of the instruction)**
  - **"extended / absolute" (absolute address of where operand is stored in memory)**
  - **"relative" (desired location is calculated relative to the current value in the PC)**
  - **"indexed" (an index register is used to point to the operand – many variations with offset)**
  - **"indirect" (the operand pointer is in memory)**

Illustrative Instructions

LDAA ***addr*** ***"load accumulator A with the contents of memory location addr"***

STAA ***addr*** ***"store the contents of accumulator A at memory location addr"***

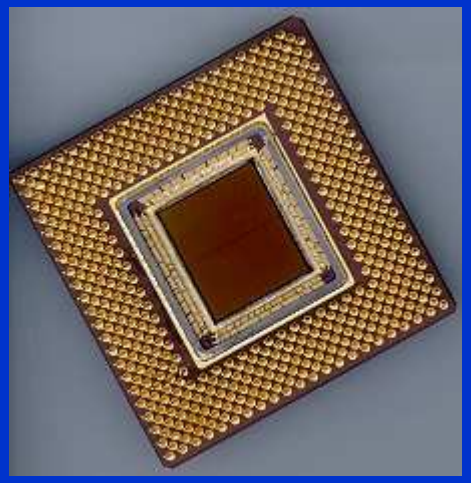
addr **represents the** ***effective address***

Illustrative Instructions

ADDA ***addr*** ***"add the contents of memory location addr to accumulator A"***

SUBA ***addr*** ***"subtract the contents of memory location addr from accumulator A"***

In each case, the result is stored in **A**



Microcontroller-Based Digital System Design

Module 1-B
Microcontroller Instruction Set Overview

Reading Assignment: *Meyer* Chp 3, pp. 30-82

Outline:

- **Introduction**
- **Notation**
- **Addressing Modes**
- **Instruction Groups**

Who are these people? (…and, what are they doing??)

Learning Objectives

- **<u>list</u> microprocessor instruction groups and <u>classify</u> machine instructions accordingly**
- **<u>determine</u> instruction encoding formats and execution cycle counts**
- **<u>determine</u> the effective address of an operand based on the addressing mode used**
- **<u>describe</u> the operation of the stack and <u>identify</u> the instructions that manipulate it**
- **<u>analyze</u> (trace) the execution of assembly code programs**
- **<u>determine</u> how the condition code register is affected by various arithmetic group instructions**

Introduction

- **The instruction set of any computer can best be understood by dividing into groups of related instructions:**
  - ***data transfer***
  - ***arithmetic***
  - ***logical***
  - ***transfer of control (branch/jump)***
  - ***machine control***
  - ***"special"***

Notation - 1

Notation	How Used	Examples
prefix of $ or *suffix* of h or H	denotes a *hexadecimal* (base 16) number	$1234 = 1234h = 1234H = 1234_{16}
prefix of ! *or suffix* of t or T	denotes a *decimal* (base 10) number	!1234 = 1234t = 1234T = 1234_{10}
prefix of % *or suffix* of b or B	denotes a *binary* (base 2) number	%10101010 = 10101010b = 10101010B = 10101010_2
( )	denotes the *contents of* a register or memory location	(A) (0800h)
;	denotes the beginning of a *comment*	LDAA 0800h ; (A) = (0800h)
:	indicates the *concatenation* of two quantities	16-bit result in (A):(B) ≡ (D) 32-bit result in (D):(X)
addr	shorthand for the *effective address* in memory at which an operand is stored	LDAA *addr* ; (A) = (addr)
rb	shorthand for a *byte-length register*, e.g., A or B	STA*rb* 0800h ; (0800h) = (*rb*)
rw, rwh, rwl	shorthand for a *word-length register*, e.g., X, Y, D, SP, where *rwh* denotes the *high byte* of that register and *rwl* the *low byte*	LD*rw* 0800h ; (*rw*) = (0800h):(0801h) ; -or- ; (*rwh*) = (0800h) ; (*rwl*) = (0801h)

Notation - 2

Notation	How Used	Examples
#	indicates use of *immediate addressing mode* when used before a constant that appears in an instructions operand field	LDAA #80h ; (A) = 80h LDAA #$12 ; (A) = 12h LDAA #$A5 ; (A) = A5h LDAA #10101010b ; (A) = AAh
,	indicates use of *indexed addressing mode* when placed between two entities in the operand field	LDAA 2,X ; (A) = ((X) + 2) STAA D,Y ; ((D)+(Y)) = (A)
[ ]	indicates use of *indirect addressing mode* when used to bracket the operand field	STAA [2,X] ; (((X)+2):((X)+3)) = (A) LDAA [D,Y] ; (A) = (((D)+(Y)):((D)+(Y)+1))
← →	denotes an *assignment* or "copy" (the arrow points *toward the destination*)	(A) ← (B) *means load the A register with the contents of the B register (the contents of B remains the same)*
↔	denotes the *exchange* (or "swap") of contents	(D) ↔ (X) *means exchange the contents of the D and X registers*
~	shorthand for number of instruction execution cycles	*assuming an 8 MHz bus clock, each cycle is* 125 ns (nanoseconds)
'	indicates a (bit-wise) complement	mask' *means the bit-wise complement of mask*

Addressing Mode Summary - 1

Icon	Abbrev.	Name	Description	Examples
⦿	INH	Inherent/Register	Operand(s) is (are) contained in registers; “inherent” means name of register part of instruction mnemonic	`DAA`
#	IMM	Immediate	Operand data “immediately follows” opcode; pound sign (#) denotes use of immediate data	`LDAA #$FF` `LDAA #1`
☏	DIR/EXT	Direct/Extended	Effective address of operand (“absolute” location in memory) follows opcode; called “direct” if the address can be contained in a single byte, or “extended” if two bytes are required	`LDAA $FF ;direct` `STAA 900h ;extended`

Addressing Mode Summary - 2

Icon	Abbrev.	Name	Description	Examples

☞	IDX
	IDX1
	IDX2
		Indexed with Constant Offset	Effective address is determined by adding a (signed) constant offset (5-bit, 8-bit, or 16-bit) to an index register (which may be X, Y, SP, or PC)	`LDAA 0,X`
				`STAA 1,Y`
				`LDAA 5,SP`
				`STAA 2,PC`

	IDX	Indexed with Accumulator Offset	Effective address is determined by adding an (unsigned) accumulator (A, B, or D) to an index register (X, Y, SP, or PC)	`LDAA B,X`
				`STAA B,Y`
				`LDAA D,X`

	IDX	Indexed with Auto Pre-/Post-Increment or Decrement	Effective address is determined by an index register (X, Y, or SP) that can be modified prior to its use (pre-inc/dec) or following its use (post-inc/dec); the amount of pre/post modification possible ranges from 1 to 8	`STAA 1,-X ;pre-dec`
				`LDAA 1,X+ ;post-inc`
				`STAA 8,+X ;pre-inc`
				`LDAA 8,X- ;post-dec`

Addressing Mode Summary - 3

Icon	Abbrev.	Name	Description	Examples
[☞]	[IDX2]	Indexed-Indirect with Constant Offset	Indexed with constant offset addressing mode is used to access a 16-bit pointer in memory, which is then used as the effective address of the operand; brackets denote use of indirection	`LDAA [4,X]` `STAA [2,Y]`
	[D,IDX]	Indexed-Indirect with Accumulator Offset	Indexed with accumulator (D) offset mode is used to access a 16-bit pointer in memory, which is then used as the effective address of the operand; brackets denote use of indirection	`LDAA [D,Y]` `STAA [D,X]`

Clicker Quiz

1. When an 8-bit accumulator offset indexed addressing mode is used:

A. the 8-bit accumulator offset is *zero-extended* to 16-bits before being added to the named index register

B. the 8-bit accumulator offset is *sign-extended* to 16-bits before being added to the named index register

C. the 16-bit index register is *truncated* to 8-bits before being added to the 8-bit accumulator offset

D. the 8-bit accumulator offset is *shifted left eight positions* before being added to the index register

E. none of the above

2. The name of the addressing mode used by the instruction STAA [2,X+] is:

A. indexed with auto-post-increment by two

B. indexed with auto-pre-increment by two

C. indirect indexed with auto-pre-increment by two

D. indirect indexed with auto-post-increment by two

E. none of the above

Data Transfer Group

- **The "theme" that links members of this group together is *transfer of data***
  - ***load***
  - ***store***
  - ***exchange***
  - ***move (transfer)***
  - ***stack manipulation***

Load and Store Registers

Description	Mnemonic	Operation	CC	Examples	Mode	~
Load Register	LDA*rb* *addr* *rb* = A, B *addr* = # ☎ ☞ [☞]	(*rb*) ← (*addr*)	N ← ↕ Z ← ↕ V ← 0	LDAA #1	#	1
				LDAA $FF	☎	3
				LDAB $900	☎	3
				LDAA 1,X	☞	3
				LDAA B,Y	☞	3
				LDAB 2,Y+	☞	3
				LDAA [0,Y]	[☞]	6
				LDAA [D,X]	[☞]	6
	LD*rw* *addr* *rw* = D, X, Y, S *addr* = # ☎ ☞ [☞]	(*rw*) ← (*addr*)	N ← ↕ Z ← ↕ V ← 0	LDD #1	#	2
				LDS #$A00	#	2
				LDX $900	☎	3
				LDY A,X	☞	3
				LDX [D,Y]	[☞]	6
Store Register	STA*rb* *addr* *rb* = A, B *addr* = ☎ ☞ [☞]	(*addr*) ← (*rb*)	N ← ↕ Z ← ↕ V ← 0	STAA $FF	☎	2
				STAB $900	☎	3
				STAA 1,X	☞	2
				STAA B,Y	☞	2
				STAB 2,Y+	☞	2
				STAA [0,Y]	[☞]	5
				STAA [D,X]	[☞]	5
	ST*rw* *addr* *rw* = D, X, Y, S *addr* = ☎ ☞ [☞]	(*addr*) ← (*rw*)	N ← ↕ Z ← ↕ V ← 0	STD $900	☎	3
				STX 2,Y	☞	2
				STY A,X	☞	2
				STX [2,Y]	[☞]	5
				STS [D,Y]	[☞]	5

Short Cut for Conversion Among Powers of 2

- **<u>Method</u>: Size Log_2R Groupings**
  - **when converting a number from base "A" to base "B", where A and B are *powers of 2* (e.g., 2, 4, 8, and 16), a "short cut" can be used**
  - **an *n-digit binary number* can be written for each *base A* digit in the original number, where $n = \log_2 A$**
  - **starting at the *least significant position*, the converted binary digits can be *regrouped* into *m-digit* binary numbers, where $m = \log_2 B$**

Short Cut for Conversion Among Powers of 2

- **<u>Exercise</u>: Convert $(110101)_2$ to base 16 (hex)**

- **<u>Exercise</u>: Convert $(A3F)_{16}$ to base 2 (binary)**

Short Cut for Conversion Among Powers of 2

- **<u>Exercise</u>: Convert $(110101)_2$ to base 16 (hex)**

0011 0101

$(3 \quad 5)_{16}$

- **<u>Exercise</u>: Convert $(A3F)_{16}$ to base 2 (binary)**

Short Cut for Conversion Among Powers of 2

- **<u>Exercise</u>: Convert $(110101)_2$ to base 16 (hex)**

0011 0101

$(3 \quad 5)_{16}$

- **<u>Exercise</u>: Convert $(A3F)_{16}$ to base 2 (binary)**

$(1010 \quad 0011 \quad 1111)_2$

Load Effective Address

Description	Mnemonic	Operation	CC	Examples	Mode	~
Load Effective Address	LEA*rw* *addr* *rw* = X, Y, S *addr* = ☞	(*rw*) ← *addr*	–	`LEAX 2,Y`	☞	2
				`LEAY B,X`	☞	2
				`LEAX D,SP`	☞	2
				`LEAS 1,X+`	☞	2
				`LEAY 2,-X`	☞	2
				`LEAS 200,SP`	☞	2
				`LEAX 1000,SP`	☞	2

The LEA instruction provides a convenient means for incrementing or decrementing an index register an arbitrary amount (as such, it could also be construed as an "arithmetic group" instruction)

Exchange

"rwh" is the high byte of a word-length register

Description	Mnemonic	Operation	CC	Examples	Mode	~
Exchange Register Contents	EXG *rb1,rb2* *rb* = A, B, CCR	(*rb1*) ↔ (*rb2*)	–	EXG A,B	⦿	1
				EXG A,CCR	⦿	1
	EXG *rw1,rw2* *rw* = D, X, Y, S	(*rw1*) ↔ (*rw2*)	–	EXG D,X	⦿	1
				EXG X,Y	⦿	1
	EXG *rb,rw* *rb* = A, B, CCR *rw* = D, X, Y, S	$00 → (*rwh*) (*rb*) ↔ (*rwl*)	–	EXG A,X	⦿	1
				EXG B,Y	⦿	1
				EXG CCR,D	⦿	1
	EXG *rw,rb* *rw* = D, X, Y, S *rb* = A, B, CCR	(*rwh*) ← $00 (*rwl*) ↔ (*rb*)	–	EXG X,A	⦿	1
				EXG Y,B	⦿	1
				EXG D,CCR	⦿	1

"rwl" is the low byte of a word-length register

Mismatched exchanges (byte ↔ word) are "legal" but *not very useful*

Transfer (Move) Register

"rwl" is the low byte of a word-length register

Description	Mnemonic	Operation	CC	Examples	Mode	~
Transfer (Move) Register	TFR *rb1,rb2* *rb* = A, B, CCR	(*rb1*) → (*rb2*)	–	TFR A,B	⦿	1
				TFR A,CCR	⦿	1
	TFR *rw1,rw2* *rw* = D, X, Y, S	(*rw1*) → (*rw2*)	–	TFR X,D	⦿	1
				TFR D,Y	⦿	1
	TFR *rw,rb* *rw* = D, X, Y, S *rb* = A, B, CCR	(*rwl*) → (*rb*)	–	TFR X,A	⦿	1
				TFR Y,B	⦿	1
				TFR X,CCR	⦿	1
	TFR *rb,rw* *rb* = A, B, CCR *rw* = D, X, Y, S	(*rb*) → (*rw*) *rwh* padded with sign of *rb*	–	TFR A,X	⦿	1
				TFR B,Y	⦿	1
				TFR CCR,D	⦿	1

The (mismatched) byte → word TFR instruction performs a *sign extension*

Move Memory

Description	Mnemonic	Operation	CC	Examples	Mode	~
Move Memory	MOVB *addr1,addr2* *addr1* = # ☎ ☞ *addr2* = ☎ ☞	(*addr1*) → (*addr2*)	–	`MOVB #$FF,$900`	# → ☎	4
				`MOVB #2,0,X`	# → ☞	4
				`MOVB $900,$901`	☎ → ☎	6
				`MOVB $900,1,X`	☎ → ☞	5
				`MOVB 1,X-,$900`	☞ → ☎	5
				`MOVB 1,X+,2,Y+`	☞ → ☞	5
	MOVW *addr1,addr2* *addr1* = # ☎ ☞ *addr2* = ☎ ☞	(*addr1*) → (*addr2*) (*addr1*+1) → (*addr2*+1)	–	`MOVW #$FFFF,$900`	# → ☎	5
				`MOVW #1,0,X`	# → ☞	4
				`MOVW $900,$902`	☎ → ☎	6
				`MOVW $900,2,X`	☎ → ☞	5
				`MOVW 2,X-,$900`	☞ → ☎	5
				`MOVW 2,X+,4,Y+`	☞ → ☞	5

Note the six addressing mode permutations (source → destination) possible

Stack Manipulation

Description	Mnemonic	Operation	CC	Examples	Mode	~
Push register onto stack	PSH*rb* *rb* = A, B, C	(SP) ← (SP) – 1 ((SP)) ← (*rb*)	–	PSHA	⦿	2
				PSHB	⦿	2
				PSHC	⦿	2
	PSH*rw* *rw* = D, X, Y	(SP) ← (SP) – 1 ((SP)) ← (*rwl*) (SP) ← (SP) – 1 ((SP)) ← (*rwh*)	–	PSHD	⦿	2
				PSHX	⦿	2
				PSHY	⦿	2
Pull (pop) register from stack	PUL*rb* *rb* = A, B, C	(*rb*) ← ((SP)) (SP) ← (SP) + 1	*	PULA	⦿	3
				PULB	⦿	3
				PULC	⦿	3
	PUL*rw* *rw* = D, X, Y	(*rwh*) ← ((SP)) (SP) ← (SP) + 1 (*rwl*) ← ((SP)) (SP) ← (SP) + 1	–	PULD	⦿	3
				PULX	⦿	3
				PULY	⦿	3

* PULC affects *all* the condition code bits, with the *exception* of X, which cannot be set by a software instruction once it is cleared.

Clicker Quiz

1. The name of the addressing mode used by the instruction `EXG A,B` is:

A. immediate

B. inherent/register

C. direct

D. extended

E. none of the above

2. If a 16-bit item is pushed onto the HC(S)12 stack, the SP register points to:

A. the low byte of the top stack item

B. the high byte of the top stack item

C. the next available stack location

D. the next instruction to execute

E. none of the above

3. If (D)=$AABB, the result in **(D)** after executing the instruction **`TFR D,A`** will be:

A. $AAAA

B. $BBBB

C. $AABB

D. $FFAA

E. none of the above

4. If (D)=$AABB, the result in (D) after executing the instruction TFR A,D will be:

A. $AAAA

B. $BBBB

C. $AABB

D. $FFAA

E. none of the above

5. If `N=+16`, the instruction `LDAA N,Y` will occupy the following number of bytes:

A. 1

B. 2

C. 3

D. 4

E. none of the above

6. If `N=-16`, the instruction `LDAA N,Y` will occupy the following number of bytes:

A. 1

B. 2

C. 3

D. 4

E. none of the above

7. Given that at least four bytes have been pushed onto the HC(S)12 stack, execution of the instruction LEAS 4,SP causes:

A. four additional bytes to be allocated on the stack
B. the top four bytes of the stack to be de-allocated
C. the bottom four bytes of the stack to be de-allocated
D. the stack origin to be moved four locations
E. none of the above

8. If (X)=$8000, execution of the instruction LEAX 1,X+ causes X to be loaded with the value:

A. $7FFF

B. $8000

C. $8001

D. $8002

E. none of the above

9. Execution of the instruction LEAY 1,X+ causes:

A. nothing to happen

B. (X) ← (X)+1

C. (Y) ← (X)+1

D. <u>both</u> B and C

E. none of the above

10. Execution of the instruction LEAY 1,+X causes:

A. nothing to happen

B. (X) $\leftarrow$ (X)+1

C. (Y) $\leftarrow$ (X)+1

D. <u>both</u> B and C

E. none of the above

Arithmetic Group

- **The "theme" that links members of this group together is *arithmetic***
  - ***add***
  - ***subtract***
  - ***complement***
  - ***compare/test***
  - ***increment/decrement***
  - ***multiply***
  - ***divide***
  - ***min/max***

Add/Subtract

Description	Mnemonic	Operation	CC	Examples	Mode	~

Add contents of memory location to register	ADD*rb* *addr*
	rb = A, B
	addr = # ☎ ☞ [☞]
		(*rb*) ← (*rb*) + (*addr*)	N ← ↕
			Z ← ↕
			V ← ↕
			C ← ↕
			H ← ↕
				ADDA #1
				ADDB $900
				ADDA 1,X
				ADDB A,X
				ADDA [2,Y]
					#
					☎
					☞
					☞
					[☞]
						1
						3
						3
						3
						6

	ADC*rb* *addr* *rb* = A, B *addr* = # ☎ ☞ [☞]	(*rb*) ← (*rb*) + (*addr*) + (C)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕ H ← ↕	ADCA #1 ADCB $900 ADCA 1,X ADCB A,X ADCA [2,Y]	# ☎ ☞ ☞ [☞]	1 3 3 3 6

	ADDD *addr*
	addr = # ☎ ☞ [☞]
		(D) ← (D) + (*addr*):(*addr*+1)	N ← ↕
			Z ← ↕
			V ← ↕
			C ← ↕
				ADDD #1
				ADDD $900
				ADDD 1,X
				ADDD [2,Y]
					#
					☎
					☞
					[☞]
						2
						3
						3
						6

Subtract contents of memory location from register	SUB*rb* *addr*
	rb = A, B
	addr = # ☎ ☞ [☞]
		(*rb*) ← (*rb*) – (*addr*)	N ← ↕
			Z ← ↕
			V ← ↕
			C ← ↕
				SUBA #1
				SUBB $900
				SUBA 1,X
				SUBB A,X
				SUBA [2,Y]
					#
					☎
					☞
					☞
					[☞]
						1
						3
						3
						3
						6

	SBC*rb* *addr* *rb* = A, B *addr* = # ☎ ☞ [☞]	(*rb*) ← (*rb*) – (*addr*) – (C)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	SBCA #1 SBCB $900 SBCA 1,X SBCB A,X SBCA [2,Y]	# ☎ ☞ ☞ [☞]	1 3 3 3 6

	SUBD *addr*
	addr = # ☎ ☞ [☞]
		(D) ← (D) – (*addr*):(*addr*+1)	N ← ↕
			Z ← ↕
			V ← ↕
			C ← ↕
				SUBD #1
				SUBD $900
				SUBD 1,X
				SUBD [2,Y]
					#
					☎
					☞
					[☞]
						2
						3
						3
						6

Overflow Detection

- **<u>Summarization</u>: Overflow occurs if two positive numbers are added and a negative result is obtained, or if two negative numbers are added and a positive result is obtained (or, if numbers of *like sign* are added and a result with the *opposite sign* is obtained)**
- **Overflow *cannot* occur when adding numbers of *opposite sign***
- **<u>Another way to detect overflow</u>: If the *carry in* to the sign position is *different* than the *carry out* of the sign position, then overflow has occurred**

Other Conditions of Interest

- **In addition to overflow, other conditions of interest following an arithmetic operation include the following:**
  - **ZERO –** ***the result of the computation was 00…0***
  - **NEGATIVE –** ***the result of the computation was a negative number***
  - **CARRY/BORROW –** ***the computation produced a carry out of the sign position after an addition, or produced a borrow out of the sign position after a subtraction (the complement of the carry out)***
- **These conditions are sometimes referred to as "*condition codes*" or "*flags*"**

Register-to-Register Add

Description	Mnemonic	Operation	CC	Examples	Mode	~
Add registers	ABA	(A) ← (A) + (B)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕ H ← ↕	ABA	⦿	2
	AB*rw* *rw* = X, Y	(*rw*) ← $00:(B) + (*rw*)	–	ABX	⦿	2
				ABY	⦿	2

ABX and ABY are "legacy" instructions that are translated (by the assembler program) into LEAX B,X and LEAY B,Y (respectively)

Decimal Adjust

Description	Mnemonic	Operation	CC	Examples	Mode	~
Decimal Adjust A	DAA	decimal adjust the result of ADD, ADC, or ABA	N ← ↕ Z ← ↕ V ← ? C ← ↕	DAA	⦿	3

Decimal adjust only works correctly after *byte adds* – it does NOT perform a *conversion*, but rather a *correction* ("adjust")

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---         ----------

            ----------
DAA
            ----------
```

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---          -----------
115

             -----------

DAA

             -----------
```

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---         ----------
115          1010   1111
             ----------
DAA
             ----------
```

result of ADD

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---         -----------
115          1010   1111
                   +0110
            -----------
DAA
            -----------
```

result of ADD

since L.N. > 9, add 6 to adjust

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---         ----------
115          1010   1111
                   +0110
            ----------
DAA          1011   0101

            ----------
```

result of ADD

since L.N. > 9, add 6 to adjust

ADD of BCD Operands Followed by DAA

```
 47          0100  0111
+68         +0110  1000
---         -----------
115          1010  1111
                  +0110
            -----------
DAA          1011  0101
            +0110
            -----------
```

result of ADD

since L.N. > 9, add 6 to adjust

since U.N. > 9, add 6 to adjust

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---         -----------
115          1010   1111
                   +0110
            -----------
DAA          1011   0101
            +0110
            -----------
           1 0001   0101
```

result of ADD

since L.N. > 9, add 6 to adjust

since U.N. > 9, add 6 to adjust

CF is hundred's position

ADD of BCD Operands Followed by DAA

```
 47          0100   0111
+68         +0110   1000
---         -----------
115          1010   1111
                   +0110
            -----------
DAA          1011   0101
            +0110
            -----------
           1 0001   0101
```

result of ADD

since L.N. > 9, add 6 to adjust

since U.N. > 9, add 6 to adjust

CF is hundred's position

ten's one's

Complement

Description	Mnemonic	Operation	CC	Examples	Mode	~
Ones' complement	COM*rb* *rb* = A, B	(*rb*) ← $FF – (*rb*)	N ← ↕ Z ← ↕ V ← 0 C ← 1	COMA	⦿	1
	COM *addr* *addr* = ☎ ☞ [☞]	(*addr*) ← $FF – (*addr*)	N ← ↕ Z ← ↕ V ← 0 C ← 1	COM $900	☎	4
				COM 1,X	☞	3
				COM B,X	☞	3
				COM [D,Y]	[☞]	6
Two's complement	NEG*rb* *rb* = A, B	(*rb*) ← $00 – (*rb*)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	NEGB	⦿	1
	NEG *addr* *addr* = ☎ ☞ [☞]	(*addr*) ← $00 – (*addr*)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	NEG $900	☎	4
				NEG 1,X	☞	3
				NEG B,X	☞	3
				NEG [D,Y]	[☞]	6

Since COM performs a "bit-wise" complement, it can also be viewed as a member of the "logical group"

Compare/Test

Description	Mnemonic	Operation	CC	Examples	Mode	~
Compare Accumulators	CBA	set CCR based on (A) – (B)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	CBA	⦿	2
Compare Register with Memory	CMP*rb* *addr* *rb* = A, B *addr* = # ☎ ☞ [☞]	set CCR based on (*rb*) – (*addr*)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	CMPA #2	#	1
				CMPB $900	☎	3
				CMPA 2,X	☞	3
				CMPB [2,Y]	[☞]	6
	CP*rw* *addr* *rw* = D, X, Y, S *addr* = # ☎ ☞ [☞]	set CCR based on (*rw*) – (*addr*):(*addr*+1)	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	CPD #2	#	2
				CPX $900	☎	3
				CPY 2,X	☞	3
				CPS [2,Y]	[☞]	6
Test for Zero	TST*rb* *rb* = A, B	set CCR based on (*rb*) – $00	N ← ↕ Z ← ↕ V ← 0 C ← 0	TSTA	⦿	1
				TSTB	⦿	1
	TST *addr* *addr* = ☎ ☞ [☞]	set CCR based on (*addr*) – $00	N ← ↕ Z ← ↕ V ← 0 C ← 0	TST $900	#	3
				TST 1,X	☞	3
				TST [2,Y]	[☞]	6

Note that CMP sets the condition code bits based on subtracting the operand from the named register

Increment/Decrement

Description	Mnemonic	Operation	CC	Examples	Mode	~
Increment	INC*r* *r* = A, B	(*r*) ← (*r*) + 1	N ← ↕ Z ← ↕ V ← ↕	INCA	⦿	1
	IN*rw* *rw* = X, Y, S	(*rw*) ← (*rw*) + 1	Z ← ↕	INX INY	⦿	1
			–	INS	⦿	1
	INC *addr* *addr* = ☎ ☞ [☞]	(*addr*) ← (*addr*) + 1	N ← ↕ Z ← ↕ V ← ↕	INC $900	☎	4
				INC 1,X	☞	3
				INC B,X	☞	3
				INC [D,Y]	[☞]	6
Decrement	DEC*r* *r* = A, B	(*r*) ← (*r*) – 1	N ← ↕ Z ← ↕ V ← ↕	DECB	⦿	1
	DE*rw* *rw* = X, Y, S	(*rw*) ← (*rw*) – 1	Z ← ↕	DEX DEY	⦿	1
			–	DES	⦿	1
	DEC *addr* *addr* = ☎ ☞ [☞]	(*addr*) ← (*addr*) – 1	N ← ↕ Z ← ↕ V ← ↕	DEC $900	☎	4
				DEC 1,X	☞	3
				DEC B,X	☞	3
				DEC [D,Y]	[☞]	6

ADDA #1 ≠ INCA (INC and DEC do not affect the "C" condition code bit "on purpose" – Why?)

Multiply

Description	Mnemonic	Operation	CC	Examples	Mode	~
8x8 unsigned integer multiply	MUL	(D) ← (A) x (B)	C ← ↕	`MUL`	⦿	3
16x16 unsigned integer multiply	EMUL	(Y):(D) ← (D) x (Y)	N ← ↕ Z ← ↕ C ← ↕	`EMUL`	⦿	3
16x16 signed integer multiply	EMULS	(Y):(D) ← (D) x (Y)	N ← ↕ Z ← ↕ C ← ↕	`EMULS`	⦿	3

Description	Mnemonic	Operation	CC	Examples	~
16x16 integer multiply and accumulate	EMACS *addr* *addr* = special	(*addr*):(*addr*+1):(*addr*+2):(*addr*+3) ← (*addr*):(*addr*+1):(*addr*+2):(*addr*+3) + (((X)) x ((Y)))	N ← ↕ V ← ↕ Z ← ↕ C ← ↕	`EMACS $900`	13

The EMACS instruction can be used for performing signal processing algorithms (e.g., digital filtering)

Divide

Description	Mnemonic	Operation	CC	Examples	Mode	~
16÷16 unsigned integer divide	IDIV	(X) ← (D) ÷ (X) (D) ← remainder	V ← 0 Z ← ↕ C ← ↕	`IDIV`	⦿	12
16÷16 signed integer divide	IDIVS	(X) ← (D) ÷ (X) (D) ← remainder	N ← ↕ V ← ↕ Z ← ↕ C ← ↕	`IDIVS`	⦿	12
32÷16 unsigned integer divide	EDIV	(Y) ← (Y):(D) ÷ (X) (D) ← remainder	N ← ↕ V ← ↕ Z ← ↕ C ← ↕	`EDIV`	⦿	11
32÷16 signed integer divide	EDIVS	(Y) ← (Y):(D) ÷ (X) (D) ← remainder	N ← ↕ V ← ↕ Z ← ↕ C ← ↕	`EDIVS`	⦿	12
32÷16 unsigned fraction divide	FDIV	(X) ← (D) ÷ (X) (D) ← remainder	V ← ↕ Z ← ↕ C ← ↕	`FDIV`	⦿	12

The FDIV instruction assumes the operands are *unsigned binary fractions* $0.2^{-1}\,2^{-2}\,2^{-3}\,2^{-4}\ldots$

binary point

Min/Max - 1

Description	Mnemonic	Operation	CC	Examples	Mode	~
Unsigned 8-bit Minimum	MINA *addr* *addr* = ☞ [☞]	(A) ← min {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MINA 0,X`	☞	4
				`MINA 2,X+`	☞	4
				`MINA 1000t,Y`	☞	5
				`MINA [D,X]`	[☞]	7
				`MINA [2,Y]`	[☞]	7
	MINM *addr* *addr* = ☞ [☞]	(*addr*) ← min {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MINM 0,X`	☞	4
				`MINM 2,X+`	☞	4
				`MINM 1000t,Y`	☞	5
				`MINM [D,X]`	[☞]	7
				`MINM [2,Y]`	[☞]	7
Unsigned 8-bit Maximum	MAXA *addr* *addr* = ☞ [☞]	(A) ← max {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MAXA 0,X`	☞	4
				`MAXA 2,X+`	☞	4
				`MAXA 1000t,Y`	☞	5
				`MAXA [D,X]`	[☞]	7
				`MAXA [2,Y]`	[☞]	7
	MAXM *addr* *addr* = ☞ [☞]	(*addr*) ← max {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MAXM 0,X`	☞	4
				`MAXM 2,X+`	☞	4
				`MAXM 1000t,Y`	☞	5
				`MAXM [D,X]`	[☞]	7
				`MAXM [2,Y]`	[☞]	7

MIN/MAX instructions are not typically included in most microcontroller instruction sets, and therefore could also be included in the "special" group

Min/Max - 2

Description	Mnemonic	Operation	CC	Examples	Mode	~
Unsigned 16-bit Minimum	EMIND *addr* *addr* = ☞ [☞]	(D) ← min {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`EMIND 0,X`	☞	4
				`EMIND 2,X+`	☞	4
				`EMIND 1000t,Y`	☞	5
				`EMIND [D,X]`	[☞]	7
				`EMIND [2,Y]`	[☞]	7
	EMINM *addr* *addr* = ☞ [☞]	(*addr*):(*addr*+1) ← min {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`EMINM 0,X`	☞	4
				`EMINM 2,X+`	☞	4
				`EMINM 1000t,Y`	☞	5
				`EMINM [D,X]`	[☞]	7
				`EMINM [2,Y]`	[☞]	7
Unsigned 16-bit Maximum	EMAXD *addr* *addr* = ☞ [☞]	(D) ← max {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`EMAXD 0,X`	☞	4
				`EMAXD 2,X+`	☞	4
				`EMAXD 1000t,Y`	☞	5
				`EMAXD [D,X]`	[☞]	7
				`EMAXD [2,Y]`	[☞]	7
	EMAXM *addr* *addr* = ☞ [☞]	(*addr*):(*addr*+1) ← max {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`EMAXM 0,X`	☞	4
				`EMAXM 2,X+`	☞	4
				`EMAXM 1000t,Y`	☞	5
				`EMAXM [D,X]`	[☞]	7
				`EMAXM [2,Y]`	[☞]	7

Clicker Quiz

1. The name of the addressing mode used by the instruction DAA is:

A. immediate

B. inherent/register

C. direct

D. extended

E. none of the above

2. When multiplying *unsigned words* using the EMUL instruction, the carry flag (C):

A. can be used to implement extended precision multiplication

B. can be used to round the lower 16-bits of the result

C. can be used to round the upper 16-bits of the result

D. has no use or “social significance”

E. none of the above

3. If **`(D)=$8000`** and **`(X)=$FFFF`**, the value in **`(X)`** after executing **`IDIVS`** is:

A. $0000

B. $7FFF

C. $8000

D. $FFFF

E. none of the above

4. If (D)=$8000 and (X)=$0000, the value in (X) after executing IDIVS is:

A. $0000

B. $7FFF

C. $8000

D. $FFFF

E. none of the above

5. If (D)=$8000 and (X)=$0000, the condition code bit that is set (to indicate a “divide by zero” has been attempted) after executing IDIVS is:

A. C (carry/borrow)
B. N (negative)
C. V (overflow)
D. Z (zero)
E. none of the above

6. The only overflow case (causing V to be set) that can occur when executing IDIVS is:

A. (D) = $7FFF, (X) = $8000

B. (D) = $8000, (X) = $7FFF

C. (D) = $7FFF, (X) = $FFFF

D. (D) = $8000, (X) = $FFFF

E. none of the above

7. The result produced by **`FDIV`** when dividing $2000 by $8000 is:

A. $0000

B. $4000

C. $8000

D. $FFFF

E. none of the above

Logical Group

- **The "theme" that links members of this group together is *logical manipulation and testing* of data**
  - ***boolean***
  - ***clear/set/complement***
  - ***bit test***
  - ***shift and rotate***

Boolean Operations (AND, OR, XOR)

Description	Mnemonic	Operation	CC	Examples	Mode	~
AND	AND*rb* *addr* *rb* = A, B *addr* = # ☏ ☞ [☞]	(*rb*) ← (*rb*) ∩ (*addr*)	N ← ↕ Z ← ↕ V ← 0	ANDA #1	#	1
				ANDA $FF	☏	3
				ANDB 900h	☏	3
				ANDA 1,X	☞	3
				ANDA B,Y	☞	3
				ANDB 2,Y+	☞	3
				ANDA [0,Y]	[☞]	6
				ANDA [D,X]	[☞]	6
ANDCC	ANDCC *addr* *addr* = #	(CCR) ← (CCR) ∩ *data*	all	ANDCC #$FE	#	1
OR	OR*rb* *addr* *rb* = A, B *addr* = # ☏ ☞ [☞]	(*rb*) ← (*rb*) ∪ (*addr*)	N ← ↕ Z ← ↕ V ← 0	ORA #1	#	1
				ORA $FF	☏	3
				ORB 900h	☏	3
				ORA 1,X	☞	3
				ORA B,Y	☞	3
				ORB 2,Y+	☞	3
				ORA [0,Y]	[☞]	6
				ORA [D,X]	[☞]	6
ORCC	ORCC *addr* *addr* = #	(CCR) ← (CCR) ∪ *data*	all	ORCC #1	#	1
XOR	EOR*rb* *addr* *rb* = A, B *addr* = # ☏ ☞ [☞]	(*rb*) ← (*rb*) ⊕ (*addr*)	N ← ↕ Z ← ↕ V ← 0	EORA #1	#	1
				EORA $FF	☏	3
				EORB 900h	☏	3
				EORA 1,X	☞	3
				EORA B,Y	☞	3
				EORB 2,Y+	☞	3
				EORA [0,Y]	[☞]	6
				EORA [D,X]	[☞]	6

ANDCC can be used to clear CCR bits

ORCC can be used to set CCR bits

Condition Code Set/Clear

Description	Mnemonic	Operation	CC	Examples	Mode	~
Clear C bit of CCR	CLC	(C) ← 0	(C) ← 0	`CLC`	⦿	1
Set C bit of CCR	SEC	(C) ← 1	(C) ← 1	`SEC`	⦿	1
Clear V bit of CCR	CLV	(V) ← 0	(V) ← 0	`CLV`	⦿	1
Set V bit of CCR	SEV	(V) ← 1	(V) ← 1	`SEV`	⦿	1
Clear I bit of CCR	CLI	(I) ← 0	(I) ← 0	`CLI`	⦿	1
Set I bit of CCR	SEI	(I) ← 1	(I) ← 1	`SEI`	⦿	1

These are all "legacy" instructions – ANDCC and ORCC provide a more general way of setting/clearing individual condition code bits (or groups of bits)

Byte Clear and Complement

Description	Mnemonic	Operation	CC	Examples	Mode	~
Clear	CLR*rb* *rb* = A, B	(*rb*) ← $00	N ← 0 Z ← 1 V ← 0 C ← 0	CLRA	⦿	1
	CLR *addr* *addr* = ☎ ☞ [☞]	(*addr*) ← $00	N ← 0 Z ← 1 V ← 0 C ← 0	CLR $900	☎	3
				CLR 1,X	☞	2
				CLR B,X	☞	2
				CLR [D,Y]	[☞]	5
Complement	COM*rb* *rb* = A, B	(*rb*) ← $FF – (*rb*)	N ← ↕ Z ← ↕ V ← 0 C ← 1	COMA	⦿	1
	COM *addr* *addr* = ☎ ☞ [☞]	(*addr*) ← $FF – (*addr*)	N ← ↕ Z ← ↕ V ← 0 C ← 1	COM $900	☎	4
				COM 1,X	☞	3
				COM B,X	☞	3
				COM [D,Y]	[☞]	6

Recall that COM was also considered a member of the arithmetic group

Bit Clear/Set and Test

Description	Mnemonic	Operation	CC	Examples	Mode	~
Bit clear	BCLR *addr,mask*	(*addr*) ← (*addr*) ∩ mask8′	N ← ↕ Z ← ↕ V ← 0	`BCLR $50,$FE`	☏	4
				`BCLR $900,$FE`	☏	4
	addr = ☏ ☞			`BCLR 1,X,$01`	☞	4
				`BCLR 2,X+,$F0`	☞	4
				`BCLR 1000t,Y,$02`	☞	6
Bit set	BSET *addr,mask*	(*addr*) ← (*addr*) ∪ mask8	N ← ↕ Z ← ↕ V ← 0	`BSET $50,$FE`	☏	4
				`BSET $900,$FE`	☏	4
	addr = ☏ ☞			`BSET 1,X,$01`	☞	4
				`BSET 2,X+,$F0`	☞	4
				`BSET 1000t,Y,$02`	☞	6

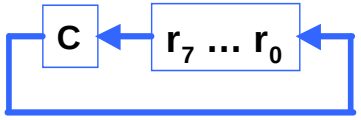
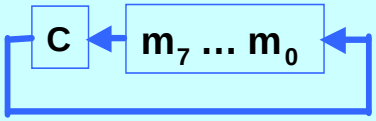
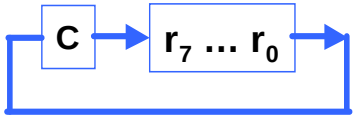
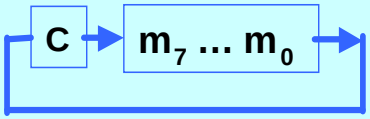
Bit Clear/Set and Test

Description	Mnemonic	Operation	CC	Examples	Mode	~
Bit clear	BCLR *addr,mask* *addr* = ☏ ☞	(*addr*) ← (*addr*) ∩ mask8′	N ← ↕ Z ← ↕ V ← 0	`BCLR $50,$FE`	☏	4
				`BCLR $900,$FE`	☏	4
				`BCLR 1,X,$01`	☞	4
				`BCLR 2,X+,$F0`	☞	4
				`BCLR 1000t,Y,$02`	☞	6
Bit set	BSET *addr,mask* *addr* = ☏ ☞	(*addr*) ← (*addr*) ∪ mask8	N ← ↕ Z ← ↕ V ← 0	`BSET $50,$FE`	☏	4
				`BSET $900,$FE`	☏	4
				`BSET 1,X,$01`	☞	4
				`BSET 2,X+,$F0`	☞	4
				`BSET 1000t,Y,$02`	☞	6

Description	Mnemonic	Operation	CC	Examples	Mode	~
Bit test	BIT*rb* *addr* *rb* = A, B *addr* = # ☏ ☞ [☞]	set CCR based on (*rb*) ∩ (*addr*)	N ← ↕ Z ← ↕ V ← 0	`BITA #1`	#	1
				`BITA $FF`	☏	3
				`BITB 900h`	☏	3
				`BITA 1,X`	☞	3
				`BITA B,Y`	☞	3
				`BITB 2,Y+`	☞	3
				`BITA [0,Y]`	[☞]	6
				`BITA [D,X]`	[☞]	6

These instructions will prove to be very useful for setting/clearing and testing bits of control/status registers

Rotate

Description	Mnemonic	Operation	CC	Examples	Mode	~
Rotate left through carry	ROL*rb* *rb* = A, B		N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ROLA	⦿	1
	ROL *addr* *addr* = ☎ ☞ [☞]		N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ROL $900 ROL 1,X ROL B,X ROL [D,Y]	☎ ☞ ☞ [☞]	4 3 3 6
Rotate right through carry	ROR*rb* *rb* = A, B		N ← ↕ Z ← ↕ V ← ↕ C ← ↕	RORA	⦿	1
	ROR *addr* *addr* = ☎ ☞ [☞]		N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ROR $900 ROR 1,X ROR B,X ROR [D,Y]	☎ ☞ ☞ [☞]	4 3 3 6

Note that these are 9-bit rotate operations, where the "C" bit is appended as the *most significant position*

Arithmetic Shift

Description	Mnemonic	Operation	CC	Examples	Mode	~
Arithmetic shift left	ASL*rb* *rb* = A, B	C ← r_7 … r_0 ← 0	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ASLA	⦿	1
	ASL*rw* *rw* = D	C ← a_7 … a_0 ← b_7 … b_0 ← 0		ASLD	⦿	1
	ASL *addr* *addr* = ☎ ☞ [☞]	C ← m_7 … m_0 ← 0	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ASL $900	☎	4
				ASL 1,X	☞	3
				ASL B,X	☞	3
				ASL [D,Y]	[☞]	6
Arithmetic shift right	ASR*rb* *rb* = A, B	r_7 … r_0 → C	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ASRA	⦿	1
	ASR *addr* *addr* = ☎ ☞ [☞]	m_7 … m_0 → C	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	ASR $900	☎	4
				ASR 1,X	☞	3
				ASR B,X	☞	3
				ASR [D,Y]	[☞]	6

Arithmetic shifts are *sign-preserving* – when shifting left, the sign is preserved in the "C" bit; when shifting right, the sign bit is replicated

Logical Shift

Description	Mnemonic	Operation	CC	Examples	Mode	~
Logical shift left	LSL*rb* *rb* = A, B	C ← r_7 … r_0 ← 0	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	LSLA	⦿	1
	LSL*rw* *rw* = D	C ← a_7 … a_0 ← b_7 … b_0 ← 0		LSLD	⦿	1
	LSL *addr* *addr* = ☎ ☞ [☞]	C ← m_7 … m_0 ← 0	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	LSL $900	☎	4
				LSL 1,X	☞	3
				LSL B,X	☞	3
				LSL [D,Y]	[☞]	6
Logical shift right	LSR*rb* *rb* = A, B	0 → r_7 … r_0 → C	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	LSRA	⦿	1
	LSR*rw* *rw* = D	0 → a_7 … a_0 → b_7 … b_0 → C		LSRD	⦿	1
	LSR *addr* *addr* = ☎ ☞ [☞]	0 → m_7 … m_0 → C	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	LSR $900	☎	4
				LSR 1,X	☞	3
				LSR B,X	☞	3
				LSR [D,Y]	[☞]	6

Logical shifts are "zero-fill" shifts – note that ASL and LSL are *equivalent* (they generate the *same opcode*)

Note the 16-bit variants of LSL/ASL and LSR

Transfer-of-Control Group

- **The "theme" that links members of this group together is *transfer-of-control* from one location of a program to another**
  - ***unconditional jumps and branches***
  - ***subroutine linkage***
  - ***conditional branches***
  - ***compound test and branch***

Unconditional Jump

Description	Mnemonic	Operation	CC	Examples	Mode	~
Jump	JMP *addr* *addr* = ☏ ☞ [☞]	(PC) ← *addr*	–	JMP $900	☏	3
				JMP 0,X	☞	3
				JMP 100t,Y	☞	3
				JMP 1000t,S	☞	4
				JMP [D,Y]	[☞]	6
				JMP [1000t,S]	[☞]	6

Indexed jumps can be used to implement "computed go-to" transfers

Indirect jumps can be used to implement "vector" tables

Unconditional Branch

Description	Mnemonic	Operation	CC	Examples	Mode	~
(Short) Branch	BRA *rel8*	(PC) ← (PC) + *rel8**	–	BRA *label*		2
Long Branch	LBRA *rel16*	(PC) ← (PC) + *rel16**	–	LBRA *label*		4

*Calculation of the two's complement relative offset must take into account the byte-length of the branch instruction. The "short" branch (BRA) instruction occupies two bytes while the "long" branch (LBRA) instruction occupies four bytes. Because the program counter is automatically incremented as a by-product of the instruction fetch, the offset calculation must compensate for this.

```
0800                        1          org     800h
                            2
0800 [01] 20FE              3  short   bra     short
                            4
0802 [04] 1820FFFC          5  long    lbra    long
                            6
0806                        7          end
                            8

Symbol Table

LONG                0802
SHORT               0800
```

-2

-4

2-byte instruction

4-byte instruction

Subroutine Linkage

Description	Mnemonic	Operation	CC	Examples	Mode	~
Jump to Subroutine	JSR *addr* *addr* = ☎ ☞ [☞]	(SP) ← (SP) – 2 ((SP)) ← (PC*h*) ((SP)+1) ← (PC*l*) (PC) ← *addr*	–	`JSR $20`	☎	4
				`JSR $900`	☎	4
				`JSR 0,X`	☞	4
				`JSR 100t,Y`	☞	4
				`JSR 1000t,S`	☞	5
				`JSR [D,Y]`	[☞]	7
				`JSR [1000t,S]`	[☞]	7
Branch to Subroutine	BSR *rel8**	(SP) ← (SP) – 2 ((SP)) ← (PC*h*) ((SP)+1) ← (PC*l*) (PC) ← (PC) + *rel8**	–	`BSR label`	🚹	4
Return from Subroutine	RTS	(PC*h*) ← ((SP)) (PC*l*) ← ((SP)+1) (SP) ← (SP) + 2	–	`RTS`	⊙	4

*Calculation of the two's complement relative offset must take into account the byte-length of the BSR instruction, which is two bytes.

The indirect version of JSR can be used to implement a subroutine jump table

Conditional Branches – Simple

Description	Mnemonic	Operation*	CC	Examples	Mode	~**
Branch if carry clear C = 0	BCC *rel8*	(PC) ← (PC) + *rel8*	–	`BCC label`		3/1
	LBCC *rel16*	(PC) ← (PC) + *rel16*	–	`LBCC label`		4/3
Branch if carry set C = 1	BCS *rel8*	(PC) ← (PC) + *rel8*	–	`BCS label`		3/1
	LBCS *rel16*	(PC) ← (PC) + *rel16*	–	`LBCS label`		4/3
Branch if not equal Z = 0	BNE *rel8*	(PC) ← (PC) + *rel8*	–	`BNE label`		3/1
	LBNE *rel16*	(PC) ← (PC) + *rel16*	–	`LBNE label`		4/3
Branch if equal Z = 1	BEQ *rel8*	(PC) ← (PC) + *rel8*	–	`BEQ label`		3/1
	LBEQ *rel16*	(PC) ← (PC) + *rel16*	–	`LBEQ label`		4/3

*Operation performed if branch is *taken*. If branch is *not* taken, the instruction effectively becomes a "no operation" (NOP). Calculation of the two's complement relative offset must take into account the byte-length of the branch instruction itself (2 for short, 4 for long).

**The first number indicates the number of cycles consumed if the branch is *taken*; the second number indicates the number of cycles consumed if the branch is *not taken*.

Conditional Branches – Simple

Description	Mnemonic	Operation*	CC	Examples	Mode	~**
Branch if positive N = 0	BPL *rel8*	(PC) ← (PC) + *rel8*	–	`BPL label`		3/1
	LBPL *rel16*	(PC) ← (PC) + *rel16*	–	`LBPL label`		4/3
Branch if negative N = 1	BMI *rel8*	(PC) ← (PC) + *rel8*	–	`BMI label`		3/1
	LBMI *rel16*	(PC) ← (PC) + *rel16*	–	`LBMI label`		4/3
Branch if overflow clear V = 0	BVC *rel8*	(PC) ← (PC) + *rel8*	–	`BVC label`		3/1
	LBVC *rel16*	(PC) ← (PC) + *rel16*	–	`LBVC label`		4/3
Branch if overflow set V = 1	BVS *rel8*	(PC) ← (PC) + *rel8*	–	`BVS label`		3/1
	LBVS *rel16*	(PC) ← (PC) + *rel16*	–	`LBVS label`		4/3
Branch never (No-op)	BRN *rel8*	–	–	`BRN label`		1
	LBRN *rel16*	–	–	`LBRN label`		3

*Operation performed if branch is *taken*. If branch is *not* taken, the instruction effectively becomes a "no operation" (NOP). Calculation of the two's complement relative offset must take into account the byte-length of the branch instruction itself (2 for short, 4 for long).

**The first number indicates the number of cycles consumed if the branch is *taken*; the second number indicates the number of cycles consumed if the branch is *not taken*.

Conditional Branches – Signed

Description	Mnemonic	Operation*	CC	Examples	Mode	~**
Branch if greater than $Z + (N \oplus V) = 0$	BGT *rel8*	(PC) ← (PC) + *rel8*	–	`BGT` *`label`*		3/1
	LBGT *rel16*	(PC) ← (PC) + *rel16*	–	`LBGT` *`label`*		4/3
Branch if less than or equal to $Z + (N \oplus V) = 1$	BLE *rel8*	(PC) ← (PC) + *rel8*	–	`BLT` *`label`*		3/1
	LBLE *rel16*	(PC) ← (PC) + *rel16*	–	`LBLT` *`label`*		4/3
Branch if greater than or equal $N \oplus V = 0$	BGE *rel8*	(PC) ← (PC) + *rel8*	–	`BGE` *`label`*		3/1
	LBGE *rel16*	(PC) ← (PC) + *rel16*	–	`LBGE` *`label`*		4/3
Branch if less than $N \oplus V = 1$	BLT *rel8*	(PC) ← (PC) + *rel8*	–	`BLT` *`label`*		3/1
	LBLT *rel16*	(PC) ← (PC) + *rel16*	–	`LBLT` *`label`*		4/3

*Operation performed if branch is *taken*. If branch is *not* taken, the instruction effectively becomes a "no operation" (NOP). Calculation of the two's complement relative offset must take into account the byte-length of the branch instruction itself (2 for short, 4 for long).

**The first number indicates the number of cycles consumed if the branch is *taken*; the second number indicates the number of cycles consumed if the branch is *not taken*.

Derivation of Signed Conditionals

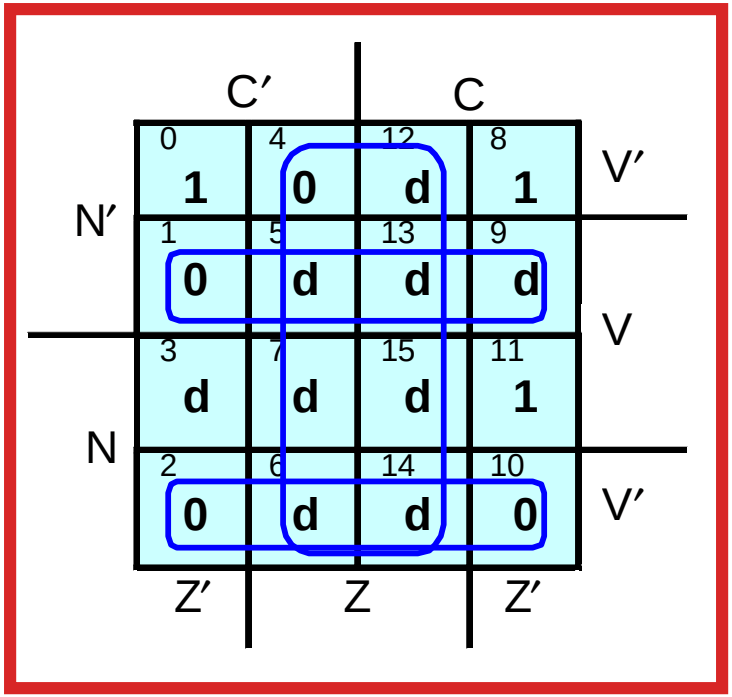
R_1	R_0	(R)	M_1	M_0	(M)	?	C	Z	N	V
0	0	0	0	0	0	(R) = (M)	0	1	0	0
0	0	0	0	1	+1	(R) < (M)	1	0	1	0
0	0	0	1	0	-2	(R) > (M)	1	0	1	1
0	0	0	1	1	-1	(R) > (M)	1	0	0	0
0	1	+1	0	0	0	(R) > (M)	0	0	0	0
0	1	+1	0	1	+1	(R) = (M)	0	1	0	0
0	1	+1	1	0	-2	(R) > (M)	1	0	1	1
0	1	+1	1	1	-1	(R) > (M)	1	0	1	1
1	0	-2	0	0	0	(R) < (M)	0	0	1	0
1	0	-2	0	1	+1	(R) < (M)	0	0	0	1
1	0	-2	1	0	-2	(R) = (M)	0	1	0	0
1	0	-2	1	1	-1	(R) < (M)	1	0	1	0
1	1	-1	0	0	0	(R) < (M)	0	0	1	0
1	1	-1	0	1	+1	(R) < (M)	0	0	1	0
1	1	-1	1	0	-2	(R) > (M)	0	0	0	0
1	1	-1	1	1	-1	(R) = (M)	0	1	0	0

$Z = 1$
(R) = (M)

$Z + (N \oplus V) = 0$
(R) > (M)

$N \oplus V = 1$
(R) < (M)

Derivation of Signed Conditionals



BLE condition
= Z + (N ⊕ V)

BGT condition
= (Z + (N ⊕ V))´

Derivation of Signed Conditionals

	C′		C		
N′	0: **0**	4: **0**	12: **d**	8: **0**	V′
	1: **1**	5: **d**	13: **d**	9: **d**	V
N	3: **d**	7: **d**	15: **d**	11: **0**	
	2: **1**	6: **d**	14: **d**	10: **1**	V′
	Z′	Z		Z′	

BLT condition

= N´•V + N•V´

= N ⊕ V

BGE condition

= (N ⊕ V)´

Conditional Branches – Unsigned

Description	Mnemonic	Operation*	CC	Examples	Mode	~**
Branch if higher than C + Z = 0	BHI *rel8*	(PC) ← (PC) + *rel8*	–	`BHI label`		3/1
	LBHI *rel16*	(PC) ← (PC) + *rel16*	–	`LBHI label`		4/3
Branch if lower than or same C + Z = 1	BLS *rel8*	(PC) ← (PC) + *rel8*	–	`BLS label`		3/1
	LBLS *rel16*	(PC) ← (PC) + *rel16*	–	`LBLS label`		4/3
Branch if higher than or same C = 0	BHS *rel8*	(PC) ← (PC) + *rel8*	–	`BHS label`		3/1
	LBHS *rel16*	(PC) ← (PC) + *rel16*	–	`LBHS label`		4/3
Branch if lower than C = 1	BLO *rel8*	(PC) ← (PC) + *rel8*	–	`BLO label`		3/1
	LBLO *rel16*	(PC) ← (PC) + *rel16*	–	`LBLO label`		4/3

*Operation performed if branch is *taken*. If branch is *not* taken, the instruction effectively becomes a "no operation" (NOP). Calculation of the two's complement relative offset must take into account the byte-length of the branch instruction itself (2 for short, 4 for long).

**The first number indicates the number of cycles consumed if the branch is *taken*; the second number indicates the number of cycles consumed if the branch is *not taken*.

Derivation of Unsigned Conditionals

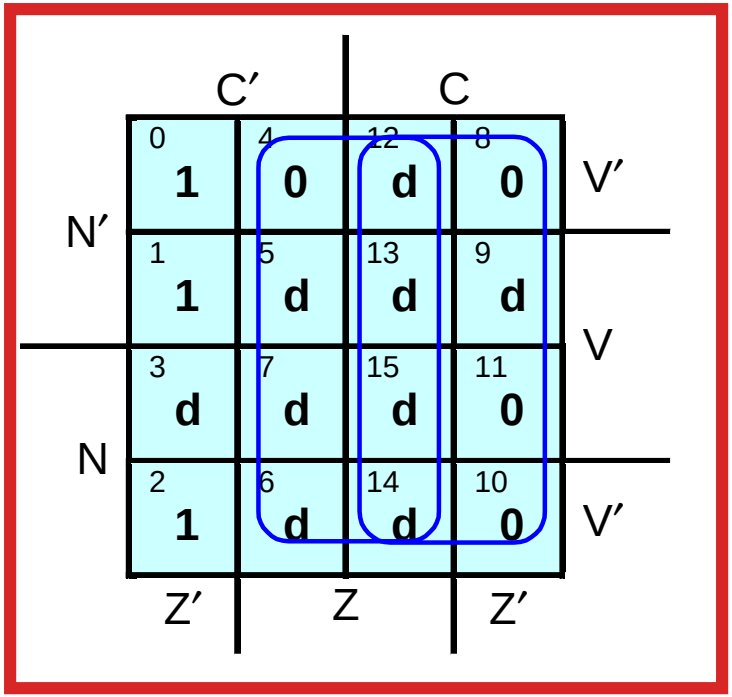
R_1	R_0	(R)	M_1	M_0	(M)	?	C	Z	N	V
0	0	0	0	0	0	(R) = (M)	0	1	0	0
0	0	0	0	1	+1	(R) < (M)	1	0	1	0
0	0	0	1	0	+2	(R) < (M)	1	0	1	1
0	0	0	1	1	+3	(R) < (M)	1	0	0	0
0	1	+1	0	0	0	(R) > (M)	0	0	0	0
0	1	+1	0	1	+1	(R) = (M)	0	1	0	0
0	1	+1	1	0	+2	(R) < (M)	1	0	1	1
0	1	+1	1	1	+3	(R) < (M)	1	0	1	1
1	0	+2	0	0	0	(R) > (M)	0	0	1	0
1	0	+2	0	1	+1	(R) > (M)	0	0	0	1
1	0	+2	1	0	+2	(R) = (M)	0	1	0	0
1	0	+2	1	1	+3	(R) < (M)	1	0	1	0
1	1	+3	0	0	0	(R) > (M)	0	0	1	0
1	1	+3	0	1	+1	(R) > (M)	0	0	1	0
1	1	+3	1	0	+2	(R) > (M)	0	0	0	0
1	1	+3	1	1	+3	(R) = (M)	0	1	0	0

Z = 1
(R) = (M)

C + Z = 0
(R) > (M)

C = 1
(R) < (M)

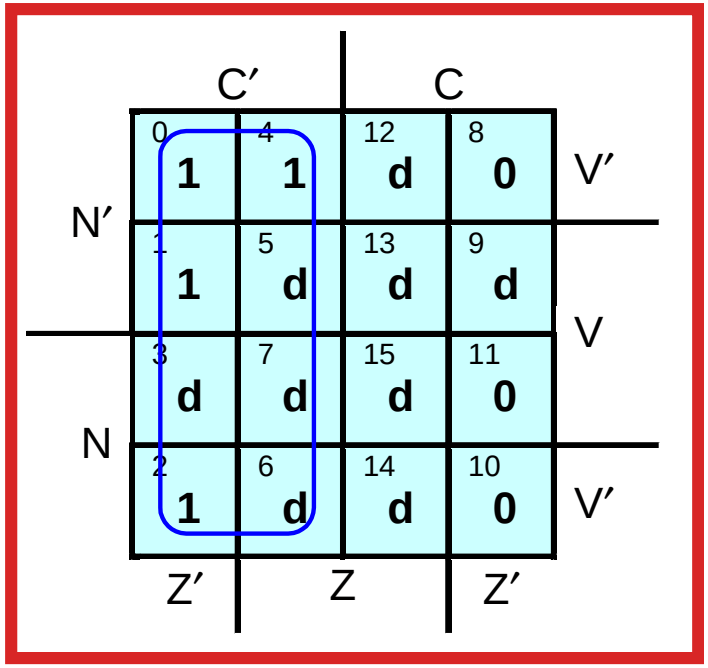
Derivation of Unsigned Conditionals



BLS condition
= C + Z

BHI condition
= (C + Z)´

Derivation of Unsigned Conditionals



BLO condition
= C

BHS condition
= C´

Signed vs. Unsigned Conditionals

- **Example: Difference between BGT and BHI**

```
; signed conditional
LDAA    #$01    ;interpret as +1
CMPA    #$FF    ;interpret as -1
BGT     label   ;branch taken

; unsigned conditional
LDAA    #$01    ;interpret as 1
CMPA    #$FF    ;interpret as 255₁₀
BHI     label   ;branch not taken
```

Bit Test and Branch

Description	Mnemonic	Operation	CC	Examples	M	~
Branch if bits clear	BRCLR *addr*,*mask8*,*rel8* *addr* = ☎ ☞	IF (*addr*) ∩ *mask8* = 0 THEN (PC) ← (PC) + *rel8*	–	`BRCLR $50,01,label`	☎	4
				`BRCLR $900,01,label`	☎	5
				`BRCLR 0,X,$FF,label`	☞	4
				`BRCLR 10t,X,01,label`	☞	4
				`BRCLR 100t,Y,02,label`	☞	6
				`BRCLR 1000t,S,03,label`	☞	8
Branch if bits set	BRSET *addr*,*mask8*,*rel8* *addr* = ☎ ☞	IF (*addr*)′ ∩ *mask8* = 0 THEN (PC) ← (PC) + *rel8*	–	`BRSET $50,01,label`	☎	4
				`BRSET $900,01,label`	☎	5
				`BRSET 0,X,$FF,label`	☞	4
				`BRSET 10t,X,01,label`	☞	4
				`BRSET 100t,Y,02,label`	☞	6
				`BRSET 1000t,S,03,label`	☞	8

These instructions are very useful for testing status registers (of on-chip peripheral devices) and branching based on the state of a single bit or a set of bits

Register Test and Branch

Description	Mnemonic	Operation	CC	Examples	Mode	~
Test Register and Branch if Zero	TBEQ *r*,*rel9* *r* = A,B,D,X,Y,S	IF (*r*) = 0 THEN (PC) ← (PC) + *rel9*	–	`TBEQ A,label`		3
				`TBEQ Y,label`		3
Test Register and Branch if Not Zero	TBNE r,*rel9* *r* = A,B,D,X,Y,S	IF (*r*) ≠ 0 THEN (PC) ← (PC) + *rel9*	–	`TBNE X,label`		3
				`TBNE SP,label`		3

"Compound" test and branch instructions (such as these and the ones on the next slide) can greatly simplify management and control of loop structures

Note that, for these instructions, the relative offset is extended to 9-bits

INC/DEC Register, Test and Branch

Description	Mnemonic	Operation	CC	Examples	Mode	~
INC Register and Branch if Zero	IBEQ *r*,*rel9* *r* = A,B,D,X,Y,SP	(*r*) ← (*r*) + 1 IF (*r*) = 0 THEN (PC) ← (PC) + *rel9*	–	`IBEQ A,label`		3
				`IBEQ Y,label`		3
INC Register and Branch if Not Zero	IBNE r,*rel9* *r* = A,B,D,X,Y,SP	(*r*) ← (*r*) + 1 IF (*r*) ≠ 0 THEN (PC) ← (PC) + *rel9*	–	`IBNE X,label`		3
				`IBNE SP,label`		3
DEC Register and Branch if Zero	DBEQ *r*,*rel9* *r* = A,B,D,X,Y,SP	(*r*) ← (*r*) – 1 IF (*r*) = 0 THEN (PC) ← (PC) + *rel9*	–	`DBEQ A,label`		3
				`DBEQ Y,label`		3
DEC Register and Branch if Not Zero	DBNE r,*rel9* *r* = A,B,D,X,Y,SP	(*r*) ← (*r*) – 1 IF (*r*) ≠ 0 THEN (PC) ← (PC) + *rel9*	–	`DBNE X,label`		3
				`DBNE SP,label`		3

Note that, for these instructions, the relative offset is extended to 9-bits

Clicker Quiz

```
        org     $8000
        jmp     iloop
iloop   lbra    iloop
```

1. The address to which the JMP instruction transfers control is:

   A. $8000

   B. $8001

   C. $8002

   D. $8003

   E. none of the above

```
        org     $8000
        lbra    iloop
iloop   jmp     iloop
```

2. The offset assembled for the LBRA instruction is:

A. $0000

B. $0001

C. $0002

D. $0003

E. none of the above

Machine Control Group

- **The "theme" that links members of this group together is *machine control* in response to interrupts and exceptions**
- **Definition: An *interrupt* is an unexpected (asynchronous), hardware-induced subroutine call**
- **Example: A sensor firing or a button being pressed could be used to generate an interrupt**
- **Definition: An *exception* is an unexpected run-time anomaly**
- **Example: Fetching an unimplemented instruction opcode would cause an exception ("trap")**

Machine Control - 1

Description	Mnemonic	Operation	CC	Examples	M	~
Return from Interrupt	RTI	(CCR) ← ((SP)), (SP) ← (SP) + 1, (D) ← ((SP)), (SP) ← (SP) + 2, (X) ← ((SP)), (SP) ← (SP) + 2, (Y) ← ((SP)), (SP) ← (SP) + 2, (PC) ← ((SP)), (SP) ← (SP) + 2	all[1]	`RTI`	⦿	8/10[2]
Unimplemented Opcode Trap	TRAP	(SP) ← (SP) – 2, ((SP)) ← (PC), (SP) ← (SP) – 2, ((SP)) ← (Y), (SP) ← (SP) – 2, ((SP)) ← (X), (SP) ← (SP) – 2, ((SP)) ← (D), (SP) ← (SP) – 1, ((SP)) ← (CCR), **I** bit of CCR ← 1, **(PC) ← (Trap Vector)**	–	$18 *tn*[3]	⦿	11
Software Interrupt	SWI	(SP) ← (SP) – 2, ((SP)) ← (PC), (SP) ← (SP) – 2, ((SP)) ← (Y), (SP) ← (SP) – 2, ((SP)) ← (X), (SP) ← (SP) – 2, ((SP)) ← (D), (SP) ← (SP) – 1, ((SP)) ← (CCR), **I** bit of CCR ← 1, **(PC) ← (SWI Vector)**	–	`SWI`	⦿	9

[1] RTI affects *all* the condition code bits, with the *exception* of X, which cannot be set by a software instruction once it is cleared.

[2] Normal execution requires 8 cycles. If another interrupt is pending when the RTI is executed, 10 cycles are consumed.

[3] Unimplemented 2-byte opcodes are those where the first opcode byte is $18 and the second opcode byte ranges from $30 to $39 or $40 to $FF.

Machine Control - 2

Description	Mnemonic	Operation	CC	Examples	M	~
Enter Background Debug Mode	BGND	Like a software interrupt, but no registers are stacked – routines in the BDM ROM control operation	–	BGND	⦿	5
Wait for Interrupt	WAI	(SP) ← (SP) – 2, ((SP)) ← (PC), (SP) ← (SP) – 2, ((SP)) ← (Y), (SP) ← (SP) – 2, ((SP)) ← (X), (SP) ← (SP) – 2, ((SP)) ← (D), (SP) ← (SP) – 1, ((SP)) ← (CCR), **Stop CPU Clocks**	–	WAI	⦿	8/5[4]
Stop Processing	STOP	(SP) ← (SP) – 2, ((SP)) ← (PC), (SP) ← (SP) – 2, ((SP)) ← (Y), (SP) ← (SP) – 2, ((SP)) ← (X), (SP) ← (SP) – 2, ((SP)) ← (D), (SP) ← (SP) – 1, ((SP)) ← (CCR), **Stop All Clocks**	–	STOP	⦿	9/5[4]
No-operation	NOP	–	–	NOP	⦿	1

[4] The cycles listed correspond to entering and exiting WAI or STOP

Special Group

- **The "theme" that links members of this group together is *special instructions* not normally found on "generic" microcontrollers**
  - ***min/max***
  - ***multiply and accumulate***
  - ***table lookup and interpolate***
  - ***fuzzy logic***

Min/Max - 1

Description	Mnemonic	Operation	CC	Examples	Mode	~
Unsigned 8-bit Minimum	MINA *addr* *addr* = ☞ [☞]	(A) ← min {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MINA 0,X`	☞	4
				`MINA 2,X+`	☞	4
				`MINA 1000t,Y`	☞	5
				`MINA [D,X]`	[☞]	7
				`MINA [2,Y]`	[☞]	7
	MINM *addr* *addr* = ☞ [☞]	(*addr*) ← min {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MINM 0,X`	☞	4
				`MINM 2,X+`	☞	4
				`MINM 1000t,Y`	☞	5
				`MINM [D,X]`	[☞]	7
				`MINM [2,Y]`	[☞]	7
Unsigned 8-bit Maximum	MAXA *addr* *addr* = ☞ [☞]	(A) ← max {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MAXA 0,X`	☞	4
				`MAXA 2,X+`	☞	4
				`MAXA 1000t,Y`	☞	5
				`MAXA [D,X]`	[☞]	7
				`MAXA [2,Y]`	[☞]	7
	MAXM *addr* *addr* = ☞ [☞]	(*addr*) ← max {(A), (*addr*)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	`MAXM 0,X`	☞	4
				`MAXM 2,X+`	☞	4
				`MAXM 1000t,Y`	☞	5
				`MAXM [D,X]`	[☞]	7
				`MAXM [2,Y]`	[☞]	7

The min/max instructions were also included in the arithmetic group

Min/Max - 2

Description	Mnemonic	Operation	CC	Examples	Mode	~
Unsigned 16-bit Minimum	EMIND *addr* *addr* = ☞ [☞]	(D) ← min {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	EMIND 0,X	☞	4
				EMIND 2,X+	☞	4
				EMIND 1000t,Y	☞	5
				EMIND [D,X]	[☞]	7
				EMIND [2,Y]	[☞]	7
	EMINM *addr* *addr* = ☞ [☞]	(*addr*):(*addr*+1) ← min {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	EMINM 0,X	☞	4
				EMINM 2,X+	☞	4
				EMINM 1000t,Y	☞	5
				EMINM [D,X]	[☞]	7
				EMINM [2,Y]	[☞]	7
Unsigned 16-bit Maximum	EMAXD *addr* *addr* = ☞ [☞]	(D) ← max {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	EMAXD 0,X	☞	4
				EMAXD 2,X+	☞	4
				EMAXD 1000t,Y	☞	5
				EMAXD [D,X]	[☞]	7
				EMAXD [2,Y]	[☞]	7
	EMAXM *addr* *addr* = ☞ [☞]	(*addr*):(*addr*+1) ← max {(D), (*addr*):(*addr*+1)}	N ← ↕ Z ← ↕ V ← ↕ C ← ↕	EMAXM 0,X	☞	4
				EMAXM 2,X+	☞	4
				EMAXM 1000t,Y	☞	5
				EMAXM [D,X]	[☞]	7
				EMAXM [2,Y]	[☞]	7

Multiply and Accumulate

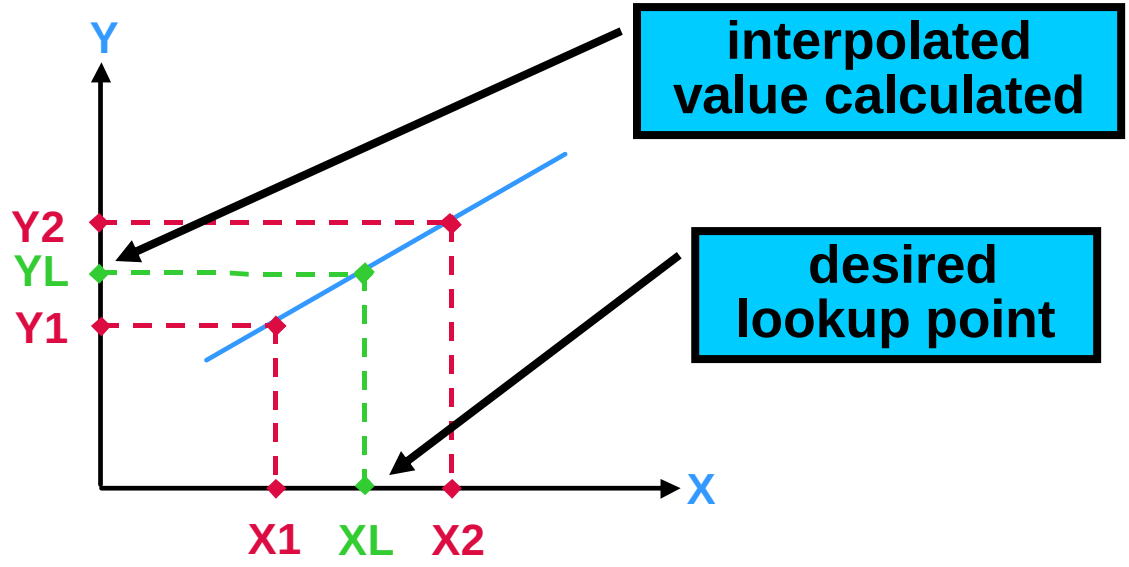
Description	Mnemonic	Operation	CC	Examples	~
16x16 integer multiply and accumulate	EMACS *addr* *addr* = special	(*addr*):(*addr*+1):(*addr*+2):(*addr*+3) ← (*addr*):(*addr*+1):(*addr*+2):(*addr*+3) + (((X)) x ((Y)))	N ← ⇕ V ← ⇕ Z ← ⇕ C ← ⇕	`EMACS $900`	13

The "multiply and accumulate" operation is a staple of most digital signal processing (DSP) algorithms. The 9S12C32 (running at 24 MHz) is "fast enough" to perform DSP functions on audio signals (of approx. 10 KHz bandwidth)

Table Lookup and Interpolate

Description	Mnemonic	Operation	CC	Examples	Mode	~
Table Lookup and Interpolate	TBL *addr* *addr* = ☞*	(A) ← (addr) + { (B) X {(addr+1) – (addr) } }	N ← ↕ Z ← ↕ C ← ?	`TBL 0,X`	☞	8
				`TBL 2,X+`	☞	8
				`TBL 2,Y-`	☞	8
				`TBL -16t,PC`	☞	8
				`TBL 15t,SP`	☞	8
	ETBL *addr* *addr* = ☞*	(D) ← (addr):(addr+1) + { (B) X { (addr+2):(addr+3) – (addr):(addr+1) } }	N ← ↕ Z ← ↕ C ← ?	`ETBL 0,X`	☞	10
				`ETBL 2,X+`	☞	10
				`ETBL 2,Y-`	☞	10
				`ETBL -16t,PC`	☞	10
				`ETBL 15t,SP`	☞	10

*Only indexed modes with “short” constant offsets (requiring no extension bytes) can be used.



Fuzzy Logic

Description	Mnemonic	Operation	CC	Examples	~
Determine Grade of Membership	MEM	((Y)) ← grade of membership (Y) ← (Y) + 1 (X) ← (X) + 4	N ← ? Z ← ? V ← ? C ← ? H ← ?	MEM	5
Fuzzy Logic Rule Evaluation	REV	MIN – MAX rule evaluation	N ← ? Z ← ? V ← 1 C ← ? H ← ?	REV	*
Fuzzy Logic Rule Evaluation (Weighted)	REVW	MIN – MAX rule evaluation with optional rule weighting; C bit in CCR selects weighted (1) or unweighted (0) rule evaluation	N ← ? Z ← ? V ← 1 C ← ? H ← ?	REVW	*
Weighted Average	WAV	Performs weighted average calculations on values stored in memory	N ← ? Z ← 1 V ← ? C ← ? H ← ?	WAV	*

*Number of cycles varies based on number of elements in rule list.

These instructions take *so many cycles* that provisions have been added to allow them to be interrupted *in progress*

Fun Things to Think About

- **The 68HC(S)12 has about 200 different instructions (of which there are about 1000 variants) – how many instructions are *really* needed to make a viable computer?**

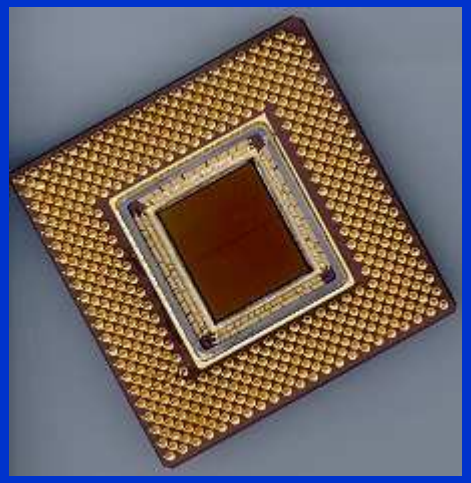
  PIC microcontrollers have as few as 33

- **The 68HC(S)12 has a fairly simple programming model (in terms of user accessible registers) – can a viable computer be built with *even fewer* registers?**

  The 68HC05 has just A and X registers

- **What capabilities would a *much larger* register set afford?**

  Less "memory traffic" (load/store idea)



Microcontroller-Based Digital System Design

Module 1-C

Assembly Language Programming Techniques: Control Structures

Outline

- **Introduction**
  - **Pseudo Ops**
  - **Expressions**
  - **Constants**
  - **Strings**

- **Basic control structures**
  - **IF- ELSE IF**
  - **SWITCH (CASE)**
  - **FOR**
  - **WHILE**
  - **DO WHILE**
  - **Assembly-style FOR and DO loops**

Learning Objectives

- **list commonly used pseudo-ops and describe their purpose**
- **describe how expressions can be utilized in assembly language code**
- **list the types of constants that can be utilized and describe how they are specified**
- **describe the utility of labels**
- **describe how if-then-else and case control structures can be realized in assembly code**
- **distinguish among and effectively utilize for, do, repeat, and while loop control structures**
- **compile C language code segments into assembly language**
- **compare "hand compiled" C code segments with machine-compiled code**

Introduction

- **The basic idea of a *control structure* is to allow controlled execution of a certain block of code**
- **The control structures described here will be relatively generic (i.e., not tied to any specific high-level language)**
- **The basic concepts covered here are applicable to assembly-level programming in general**
- **Note that each *assembler program* will have its own *conventions* and "peculiarities" (e.g, comments delineated by semicolons, etc.)**

Pseudo Ops

- **A *pseudo op* is an *assembler directive*, i.e., it tells the assembler to "do something" but does *not* generate any *executable code* in and of itself**
- **Each *assembler program* has its own set of pseudo ops that it recognizes**

Pseudo Ops

- **The pseudo ops recognized by the assembler program include the following:**
  - **RMB: reserve memory byte – allocates memory (only) for a variable or array**
  - **EQU: equate a symbol (label) with a numeric value (mainly used to improve the readability of assembly code)**
  - **FCB (or DB): form constant byte (or define byte) – assign an initial value to an 8-bit (byte) variable or array**
  - **FDB (or DW): form double byte (or define word – assign an initial value to a 16-bit (word) variable or array**

Pseudo Ops

- **The pseudo ops recognized by assembler program include the following:**
  - **FCC: form constant character – used with quoted strings**
  - **ORG: originate code – sets the assembler program's internal location counter to the address specified (if this statement is missing, 0000h is assumed)**
  - **INCLUDE: include the specified assembly source file at the current location**
  - **END: indicates the end of the assembly source file**

Expressions

- **Expressions can also be included in the *operation field* of certain instructions**
- **It is important to remember, however, that these expressions are *evaluated* at *assembly time* (<u>not</u> during *program execution*)**

- **<u>Examples</u>:**

```
ldaa   #2*value/37
bne    exit+1
staa   mem+2
```

Constants

- **Hexadecimal values can be specified using either a *prefix* of "$" (CW) or a *suffix* of "h" (other assemblers - note that all hexadecimal constants specified using a suffix of "h" *must* have a *leading digit* in the range of 0-9)**
- **<u>Examples</u>: $A7 is the same as 0A7h**
  $3A is the same as 3Ah
- **Decimal values can be specified using a suffix of "t" (CW default base – no suffix)**
- **Binary values can be specified using a suffix of "b" (CW – prefix of %)**
- **<u>Example</u>: 01100111b is the same as 67h**

Strings

- **Quoted strings are converted to equivalent ASCII characters by the assembler program**

- **<u>Examples</u>:**

```
        cmpa   #'J'
string  fcc    "Hello world"
```

Labels

- **Labels are symbols used to represent memory locations in which program or data are stored**
- **When used in the operand field of a transfer-of-control instruction, the assembler converts these symbols to absolute locations (for "jump" instructions) or signed relative offsets (for "branch" instructions)**
- **<u>Examples</u>:**

```
        org     $800
        jmp     iloop     ; address is $803
iloop   bra     iloop     ; offset is $FE
```

IF-ELSE IF

C Syntax:

if (condition 1)

<statement 1>;

else if (condition 2)

<statement 2>;

else if (condition 3)

<statement 3>;

Example:

```
if(i==1)
   pay=100;
else if(i==2)
   pay=200;
else if(i==3)
   pay=300;
else if(i==4)
   pay=400;
```

Hand Compilation:

```
i       rmb     1;
pay     rmb     2;

        ldaa    i
if1
        cmpa    #1
        bne     if2
        movw    #100,pay
        bra     if_end
if2
        cmpa    #2
        bne     if3
        movw    #200,pay
        bra     if_end
if3
        cmpa    #3
        bne     if4
        movw    #300,pay
        bra     if_end
if4
        cmpa    #4
        bne     if_end
        movw    #400,pay

if_end
```

SWITCH (CASE)

C Syntax:

```
switch (variable) {
  case value 1:
      <statement 1>;
      break;
  case value 2:
      <statement 2>;
      break;
  case value 3:
     <statement 3>;
     break;
  default:
     <statement default>;
}
```

Example:

```
switch(i) {
    case 1:
        pay=100;
        break;
    case 2:
        pay=200;
        break;
    case 3:
        pay=300;
        break;
    case 4:
        pay=400;
        break;
    default:
        pay=0;
}
```

Hand Compilation:

```
i          rmb      1;
pay        rmb      2;

case_start
           ldaa     i
case1
           cmpa     #1
           bne      case2
           movw     #100,pay
           bra      case_exit
case2
           cmpa     #2
           bne      case3
           movw     #200,pay
           bra      case_exit
case3
           cmpa     #3
           bne      case4
           movw     #300,pay
           bra      case_exit
case4
           cmpa     #4
           bne      default
           movw     #400,pay
           bra      case_exit
default
           movw     #0,pay
case_exit
```

FOR

C Syntax:

for (variable initialization; condition; variable update) {

<statements executed while condition is true>;

}

Example:

```
unsigned int I;

for(I=1;I<=10;I++) {
<statements>;
}
```

16-bit unsigned control variable

Hand Compilation:

```
I       rmb   2
for_start
        movw  #1,I
for_loop
        ldd   I
        cpd   #10
        bhi   for_exit
        <statements>
        ldd   I
        addd  #1
        std   I
        bra   for_loop
for_exit
        <statements>
```

WHILE

C Syntax:

```
while (condition) {
<statements executed while condition is true>;
}
```

Example:

```
unsigned int I;

I=1;
while(I<=10) {
<statements>;
I++;
}
```

16-bit unsigned control variable

Hand Compilation:

```
I       rmb  2
while_start
        movw  #1,I
while_loop
        ldd   I
        cpd   #10
        bhi   while_exit
        <statements>
        ldd   I
        addd  #1
        std   I
        bra   while_loop
while_exit
        <statements>
```

DO WHILE

C Syntax:

do {

<statements executed while condition is true>;

} while (condition)

Example:

```
unsigned int I;

I=1;
do {
<statements>;
I++;
} while(I<=10)
```

16-bit unsigned control variable

Hand Compilation:

```
I       rmb  2
dowhile_start
        movw  #1,I
dowhile_loop
        <statements>
        ldd   I
        addd  #1
        std   I
        cpd   #10
        bls   dowhile_loop
dowhile_exit
        <statements>
```

"Assembly Style" FOR (0 – 255)

Key: Use *registers* as loop counters

```
ITER EQU   ___ ; unsigned byte

for_start
      ldab #ITER+1
for_loop
      dbeq b,for_exit

      <statements>

      bra  for_loop
for_exit
      <statements>
```

Basic idea: Count (B) *down*, and use the ZERO condition as a *completion indicator*

calculation of ITER+1 is done at *assembly time*

make use of "compound" decrement and branch instruction

"Assembly Style" FOR (0 – 65,535)

Key: Use *registers* as loop counters

```
ITER EQU   ___ ; unsigned word

for_start
      ldx  #ITER+1
for_loop
      dbeq x,for_exit

      <statements>

      bra  for_loop
for_exit
      <statements>
```

Basic idea: Count (X) *down*, and use the ZERO condition as a *completion indicator*

calculation of ITER+1 is done at *assembly time*

make use of "compound" decrement and branch instruction

Note: In "assembly style" FOR constructs, the *loop overhead* is reduced to 3 instructions

“Assembly Style” DO (1 – 256)

```
ITER EQU    ___ ; unsigned byte

do_start
      ldab #ITER
do_loop

      <statements>

      dbne b,do_loop
do_exit
      <statements>
```

code block is executed *at least* once

completion check at *bottom* of loop

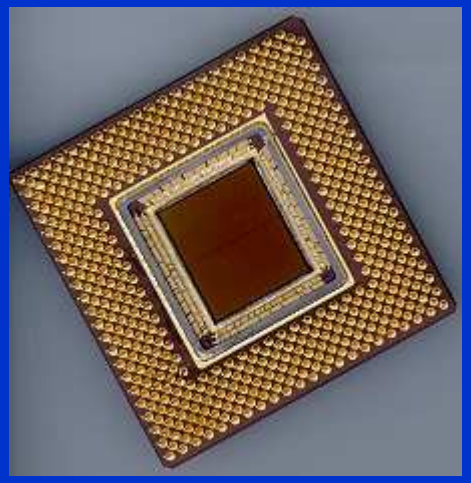
“Assembly Style” DO (1 – 65,536)

```
ITER EQU    ___ ; unsigned byte

do_start
      ldx   #ITER
do_loop

      <statements>

      dbne x,do_loop
do_exit
      <statements>
```

code block is executed *at least* once

completion check at *bottom* of loop

Note: In “assembly style” DO constructs, the *loop overhead* is reduced to 2 instructions



Microcontroller-Based Digital System Design

Module 1-D

Assembly Language Programming Techniques: Control Structure Applications

Outline

- **“Straight-line” programming example**
- **Control structure programming examples**
  - **Software delay**
  - **Extended-precision binary add**
  - **Extended-precision decimal subtract**

Learning Objective

- **<u>calculate</u> cycle counts of various loop structures and <u>compare</u> their effectiveness**

Basic Programming Example:

Write a complete 68HC(S)12 assembly program that converts a 4-digit BCD number, stored at locations $900 and $901, into its 16-bit binary equivalent, stored at locations $902 and $903.

Illustration: If locations $900 and $901 contain 20_{BCD} and 48_{BCD}, respectively (which represent 2048_{10}), the program should produce the 16-bit result 0800_{16} in locations $902 and $903.

Solution - 1

```
; 4-digit packed BCD number is in (900h):(901h)
; 16-bit converted binary number is in (902h):(903h)
;
; Method:
;
; Binary = ((((Thousands Dig) X 10) + (Hundreds Dig)) X 10
;           + (Tens Dig)) X 10 + (Ones Dig)
;
        org     800h
;
; process thousands digits
;
convw   ldaa    $900    ; get most significant two digits
                        ;   of BCD number
        lsra            ; move thousands position to lower nibble
        lsra
        lsra
        lsra
        ldab    #10t
        mul             ; (D) = (Thousands Dig) X 10
        std     $902    ; use result location as an accumulator
```

Solution - 2

```
;
; process hundreds digit
;
        ldab    $900    ; reload most significant two digits
        andb    #$0F    ; mask off upper nibble
        clra            ; zero extend
        addd    $902    ; add hundreds digit to running total
        ldy     #10t
        emul            ; multiply running total by 10
        std     $902    ; save in result accumulator
;
; process tens digit
;
        ldab    $901    ; get least significant two digits
                        ;   of BCD number
        lsrb            ; move tens position to lower nibble
        lsrb
        lsrb
        lsrb
        clra            ; zero extend
        addd    $902    ; add tens digit to running total
        ldy     #10t
        emul            ; multiply running total by 10
        std     $902
```

Solution - 3

```
;
; process ones digit
;
        ldab    $901    ; reload least significant two digits
        andb    #$0F    ; mask off upper nibble
        clra            ; zero extend
        addd    $902    ; add to running total
        std     $902    ; store final result
        stop            ; DONE
;
; test data declaration
;
        org     900h
bcd     fdb     $2048
bin     fdb     $0000
        end
```

Solution - 3

```
;
; process ones digit
;
        ldab    $901    ; reload least significant two digits
        andb    #$0F    ; mask off upper nibble
        clra            ; zero extend
        addd    $902    ; add to running total
        std     $902    ; store final result
        stop            ; DONE
;
; test data declaration
;
        org     900h
bcd     fdb     $2048
bin     fdb     $0000
        end
```

Question: How many cycles are required to perform a conversion? **60**

Software Delay

- **Software delay routines can be used in applications requiring *non-precision* time delays (intervals)**
- **Example application: *key de-bouncing***
- **<u>Question</u>: Why are the delays provided by software routines (potentially) non-precision?**

  - parameter-dependent overhead

  - interrupt processing

- **<u>Question</u>: For applications that require precision delays, what approach should be used instead?**

  - timer module

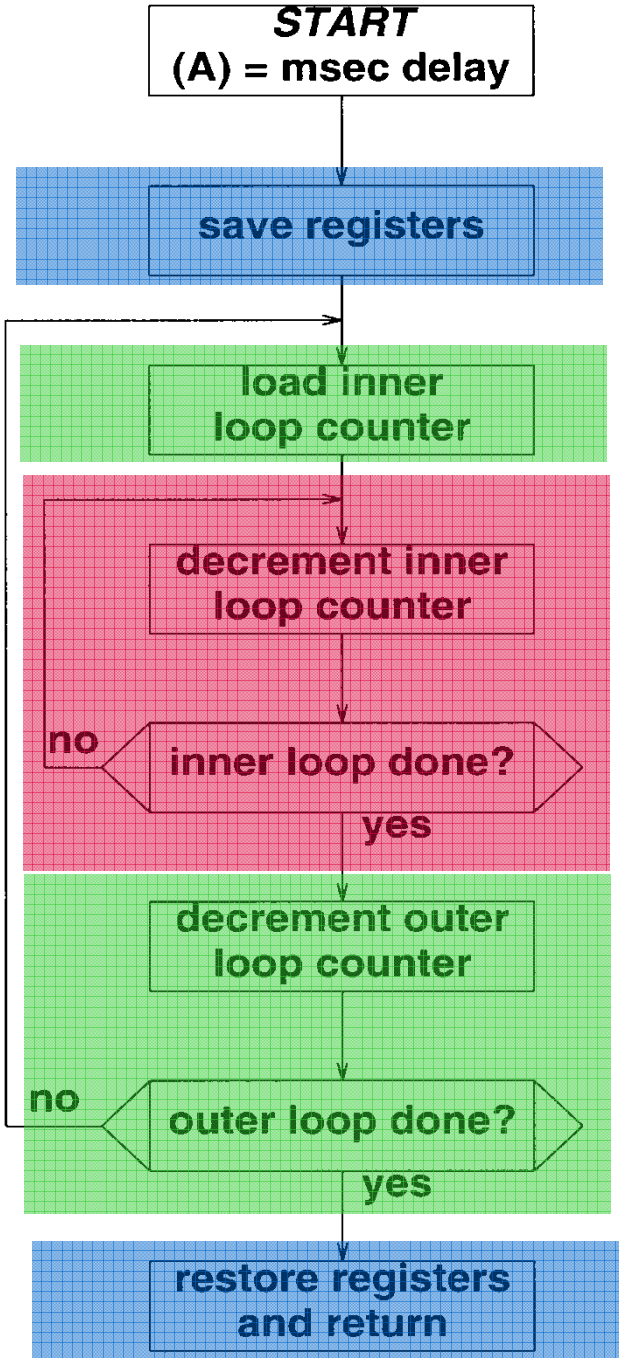
Software Delay

- **Method:**
  - **utilize *doubly-nested* loop**
  - **construct *inner loop* to provide one millisecond (1 ms) of delay**
  - **construct *outer loop* to control the number of times the inner loop is entered**
  - **total execution time will be *approximately* equal to the number of times the inner loop is executed**
- **Entry/Exit conditions:**
  - **at entry, (A) = number of milliseconds to delay**
  - **at exit, all registers should be *unchanged***

delay

loopo

loopi




```
pshx          [2]
psha          [2]
pshc          [2]
ldx   #_______ [2]

dbne  x,loopi  [3]

dbne  a,loopo  [3]

pulc          [3]
pula          [3]
pulx          [3]
rts           [5]
```

Software Delay

Note: Need to know *clock speed* of CPU to do this – here, assume it is 8 MHz, i.e., each *cycle* is 125 ns

- **Execution time analysis:**
  - **formula for total cycles consumed**

$$\text{Total Cycles} = (A) * [\,(X)*3 + 5] + 20$$

  - **solve for (A) = 1 (i.e., 1 ms delay)**

$$8000 = (1) * [\,(X)*3 + 5] + 20$$

Here, X ~ 2658_{10}

  - **solve for (A) = 100 (i.e., 100 ms delay)**

$$800{,}000 = (100) * [\,(X)*3 + 5] + 20$$

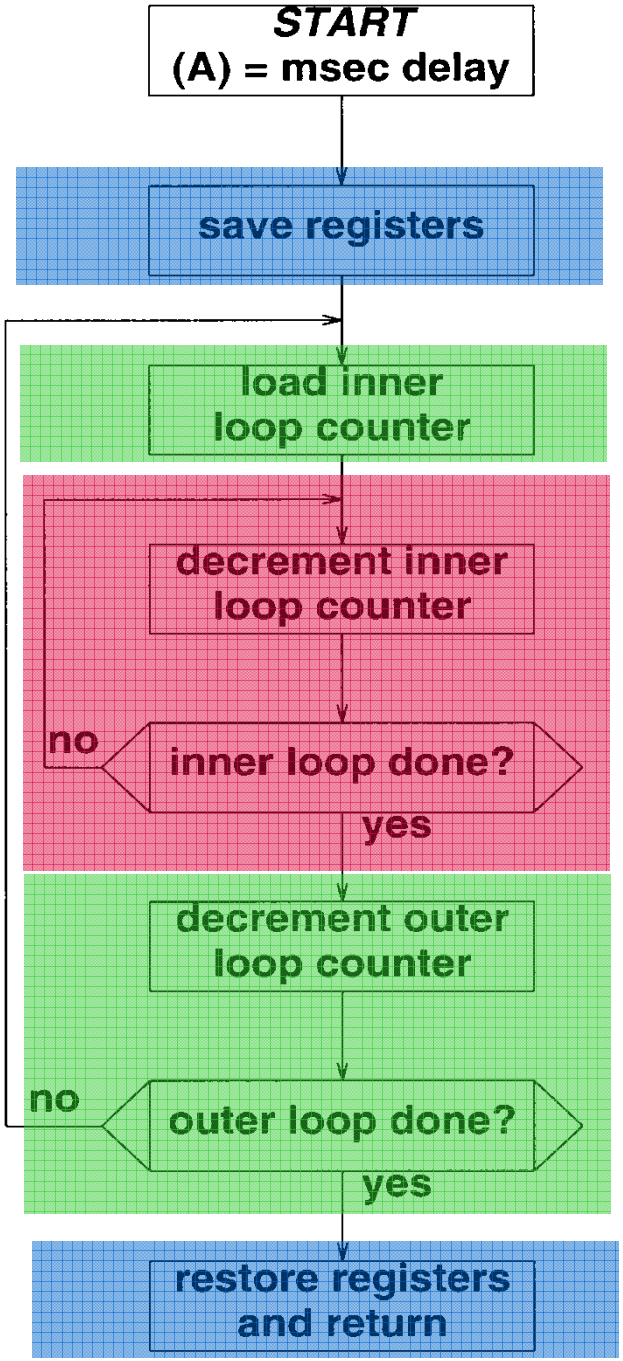
Here, X ~ 2664_{10}

So, what value should be used??

delay

loopo

loopi




```
pshx          [2]
psha          [2]
pshc          [2]
ldx   #2661t  [2]

dbne  x,loopi [3]

dbne  a,loopo [3]

pulc          [3]
pula          [3]
pulx          [3]
rts           [5]
```

Clicker Quiz

```
N         equ      ?

          org      $800
macme
          pshb                ; (2)
          pshx                ; (2)
          pshy                ; (2)
          ldx      #XA        ; (2)
          ldy      #YA        ; (2)
          movw     #0,ACM     ; (5)
          movw     #0,ACM+2   ; (5)
          ldab     #N         ; (1)
loop
          emacs    ACM        ; (13)
          leax     2,x        ; (2)
          leay     2,y        ; (2)
          dbne     b,loop     ; (3)

          puly                ; (3)
          pulx                ; (3)
          pulb                ; (3)
          rts                 ; (5)

ACM       rmb      4
XA        fdb      ?,?,?
YA        fdb      ?,?,?
```

1. **If N=0, the total number of cycles consumed by "macme" is:**
   - **A. 55**
   - **B. 235**
   - **C. 5135**
   - **D. 5155**
   - **E. none of these**

```
N         equ      ?

          org      $800
macme
          pshb                ; (2)
          pshx                ; (2)
          pshy                ; (2)
          ldx      #XA        ; (2)
          ldy      #YA        ; (2)
          movw     #0,ACM     ; (5)
          movw     #0,ACM+2   ; (5)
          ldab     #N         ; (1)
loop
          emacs    ACM        ; (13)
          leax     2,x        ; (2)
          leay     2,y        ; (2)
          dbne     b,loop     ; (3)

          puly                ; (3)
          pulx                ; (3)
          pulb                ; (3)
          rts                 ; (5)

ACM       rmb      4
XA        fdb      ?,?,?
YA        fdb      ?,?,?
```

2. **If N=1, the total number of cycles consumed by "macme" is:**

   A. **55**

   B. **235**

   C. **5135**

   D. **5155**

   E. **none of these**

```
N         equ      ?

          org      $800
macme
          pshb                ; (2)
          pshx                ; (2)
          pshy                ; (2)
          ldx      #XA        ; (2)
          ldy      #YA        ; (2)
          movw     #0,ACM     ; (5)
          movw     #0,ACM+2   ; (5)
          ldab     #N         ; (1)
loop
          emacs    ACM        ; (13)
          leax     2,x        ; (2)
          leay     2,y        ; (2)
          dbne     b,loop     ; (3)

          puly                ; (3)
          pulx                ; (3)
          pulb                ; (3)
          rts                 ; (5)

ACM       rmb      4
XA        fdb      ?,?,?
YA        fdb      ?,?,?
```

3. **If N=10, the total number of cycles consumed by "macme" is:**
   - **A. 55**
   - **B. 235**
   - **C. 5135**
   - **D. 5155**
   - **E. none of these**

```
N         equ      ?

          org      $800
macme
          pshb                ; (2)
          pshx                ; (2)
          pshy                ; (2)
          ldx      #XA        ; (2)
          ldy      #YA        ; (2)
          movw     #0,ACM     ; (5)
          movw     #0,ACM+2   ; (5)
          ldab     #N         ; (1)
loop
          emacs    ACM        ; (13)
          leax     2,x        ; (2)
          leay     2,y        ; (2)
          dbne     b,loop     ; (3)

          puly                ; (3)
          pulx                ; (3)
          pulb                ; (3)
          rts                 ; (5)

ACM       rmb      4
XA        fdb      ?,?,?
YA        fdb      ?,?,?
```

4. **If N=255, the total number of cycles consumed by "macme" is:**
   - **A. 55**
   - **B. 235**
   - **C. 5135**
   - **D. 5155**
   - **E. none of these**

Extended Precision Binary Add

- **Extended (sometimes called "infinite") precision arithmetic routines are based on using the *carry flag* (CF) to *propagate* a carry (or borrow) forward, starting with the least significant byte and ending with the most significant byte**
- **For addition, the operands are called the "augend" (the number on *top*) and the "addend" (the number on the *bottom*)**
- **Index registers are typically used as *pointers* to the operand and result arrays**
- ***Auto-increment/decrement* addressing can be used as a convenient means of *"bumping"* the pointers after each iteration**

Extended Precision Binary Add

Assume the following declarations:

```
augend   FCB    lsb, · · · ,msb
addend   FCB    lsb, · · · ,msb
nbytes   EQU    ?            ; number of bytes, each operand
```

Method:

(augend) ← (augend) + (addend)

(augend) ← (X)
+ (addend) ← (Y)

(result) ← (X)

Note: Here, the result *overwrites* the augend

Extended Precision Binary Add

```
addn    ldab    #nbytes   ; B is loop counter
        ldx     #augend   ; X and Y are pointers
        ldy     #addend   ;   to operand arrays
        clc               ; CF ← 0 initially

loop    ldaa    0,x       ; access augend byte

        adca    1,y+      ; add addend, bump pointer

        staa    1,x+      ; store result in augend byte,
                          ;   bump pointer

        dbne    b,loop    ; continue until complete

        rts
```

Basic Extended Precision Modifications

- **Modifying the extended precision *binary ADD* routine to perform a *binary SUBTRACT* or *decimal ADD* is fairly straight-forward**
- **Note: For subtraction, the operands are called the "minuend" (the number on the *top*) and the "subtrahend" (the number on the *bottom*)**
- **<u>Question</u>: What modification(s) are necessary for each case?**

Extended Precision Binary Subtract

```
addn    ldab    #nbytes   ; B is loop counter
        ldx     #augend   ; X and Y are pointers
        ldy     #addend   ;   to operand arrays
        clc               ; CF ← 0 initially

loop    ldaa    0,x       ; access augend byte

        sbca    1,y+      ; add addend, bump pointer

        staa    1,x+      ; store result in augend byte,
                          ;   bump pointer

        dbne    b,loop    ; continue until complete

        rts
```

Extended Precision Decimal Add

```
addn    ldab    #nbytes   ; B is loop counter
        ldx     #augend   ; X and Y are pointers
        ldy     #addend   ;   to operand arrays
        clc               ; CF ← 0 initially

loop    ldaa    0,x       ; access augend byte

        adca    1,y+      ; add addend, bump pointer
        daa
        staa    1,x+      ; store result in augend byte,
                          ;   bump pointer

        dbne    b,loop    ; continue until complete

        rts
```

Extended Precision Decimal Subtract

- **Writing an extended precision *decimal subtract* routine, however, is a bit more challenging**
- **<u>Question</u>: Why?**

There is no "decimal adjust after subtraction" instruction

Extended Precision Decimal Subtract

Assume the following declarations:

```
minuend FCB    lsb, · · · ,msb   operands stored in
subhend FCB    lsb, · · · ,msb   packed BCD format
nbytes  EQU    ?                 ; number of bytes, each operand
```

Method:

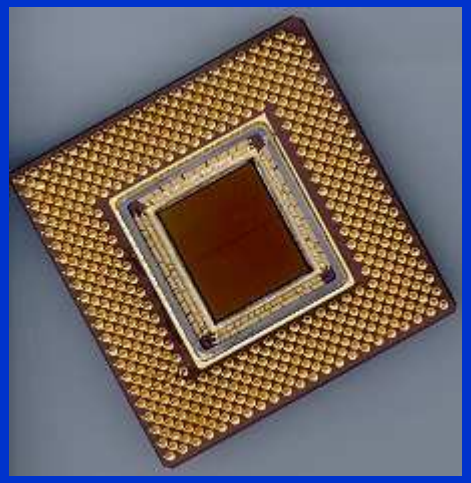
Form the radix complement of the subtrahend *first*, then *add* it to the minuend and perform a *decimal adjust*

$$(\text{result})_{BCD} \leftarrow \underbrace{(\text{minuend})_{BCD} + \underbrace{[\,99_{BCD} - (\text{subhend})_{BCD} + 1]}_{\text{radix (10's) complement of subtrahend}}}_{\text{DAA}}$$

Extended Precision Decimal Subtract

```
subbcd  ldab    #nbytes   ; B is loop counter
        ldx     #minuend; X and Y are pointers
        ldy     #subhend
        sec               ; CF ← 1 initially

loop    ldaa    #99h      ; (A) ← 99 in BCD format
        adca    #0        ; add in CF bit
        suba    1,y+      ; form ten's comp of subtrahend

        adda    0,x       ; add to minuend
        daa               ; decimal adjust
        staa    1,x+      ; store result in minuend array

        dbne    b,loop    ; continue until complete

        rts
```

Add 1 to get radix complement *and* propagate carry



Microcontroller-Based Digital System Design

Module 1-E

Assembly Language Programming Techniques: Table-Lookup

Outline

- **Table lookup techniques and applications**
  - **Linear index**
  - **Non-linear index**
  - **Lookup and interpolate (using TBL)**
  - **Jump tables**
- **Example Application**

Learning Objective

- **<u>identify</u> potential applications of table lookup and <u>effectively utilize</u> them**

Table Lookup

- **Table lookup is a common technique used in a variety of microcontroller-based systems**
- **Applications include the following:**
  - **data conversion (e.g., linear → log)**
  - **waveform generation**
  - **linearization of device characteristics (e.g., thermocouple)**
  - **jump tables for code module execution**

Table Lookup - Linear Index

Example: Simple lookup

```
index     rmb    _____    ; index variable
table     fcb    _____    ; table entry 0
          •
          •
          •
          fcb    _____    ; table entry N-1
```

Code to access table entry:

CASE 1: $0 \le N \le 255$ (unsigned† byte index)

```
          ldab   index    ; B contains 8-bit index
          ldx    #table   ; X points to start of table
          ldaa   b,x      ; (A) ← TABLE[INDEX]
```

†Here the 8-bit offset is "zero extended" before it is added to the 16-bit index register, effectively making the offset unsigned

Table Lookup - Linear Index

CASE 2: $0 \le N \le 65{,}535$ (unsigned‡ word index)

```
ldd   index      ; D contains 16-bit index
ldx   #table     ; X points to start of table
ldaa  d,x        ; (A) ← TABLE[INDEX]
```

‡Since the address bus and offset are both 16-bits, it does not matter whether a 16-bit offset is considered to be a signed or an unsigned value (e.g., FFFFh may be thought of as either +65,535 or as -1)

Table Lookup - Non-linear Index

- **A non-linear index simply means that the index only takes on a *subset* of all possible values, i.e., there are *don't cares* in the table**
- **<u>Example</u>: Packed BCD number used as a table lookup index, say for a calendar program (note that there is no "month 00", nor are there months "0A" through "0F")**
- **<u>Question</u>: In such an application, is it more efficient to use the "raw" BCD value as a table lookup index (thus "wasting" space in the table), or convert the BCD value to binary before using it as a lookup index (thus "saving" space in the table)?**

<u>A "pair-o-docs"</u>: Sometimes, in order to <u>*save*</u> space, you have to <u>*waste*</u> it!

Table Lookup - Non-linear Index

Example:

Table of days in each month, index in packed BCD format

```
month   rmb   1     ; month number, range 01 to 12 in BCD
days    fcb   XX    ; days in month 00h
        fcb   31h   ; days in month 01h
        fcb   28h   ; days in month 02h
        •
        •
        fcb   30h   ; days in month 09h
        fcb   XX    ; days in month 0Ah
        •
        •
        fcb   XX    ; days in month 0Fh
        fcb   31h   ; days in month 10h
        fcb   30h   ; days in month 11h
        fcb   31h   ; days in month 12h
```

wasted space

Table Lookup - Non-linear Index

Code to access table entry:

```
ldab    month  ; B contains index, packed BCD
ldx     #days  ; X points to start of table
ldaa    b,x    ; (A) ← DAYS[MONTH]
```

Note: Same code as “normal” case

Lookup and Interpolate (TBL)

- **The "TBL" instruction can be used to perform a *linear interpolation* on values that fall between a *pair of data entries* stored in memory**
- **<u>Example</u>: Estimation of room temperature based on data read from an analog input channel interfaced to a silicon diode circuit**
- **Operation performed:**

$$(A) \leftarrow (addr) + [\ (B) \times \{\ (addr+1) - (addr)\ \}\ ]$$

where *indexed addressing* mode is used and (B) is interpreted as a *binary fraction**

***of form $0.2^{-1}\,2^{-2}\,2^{-3}\,2^{-4}$ ... (unsigned)**

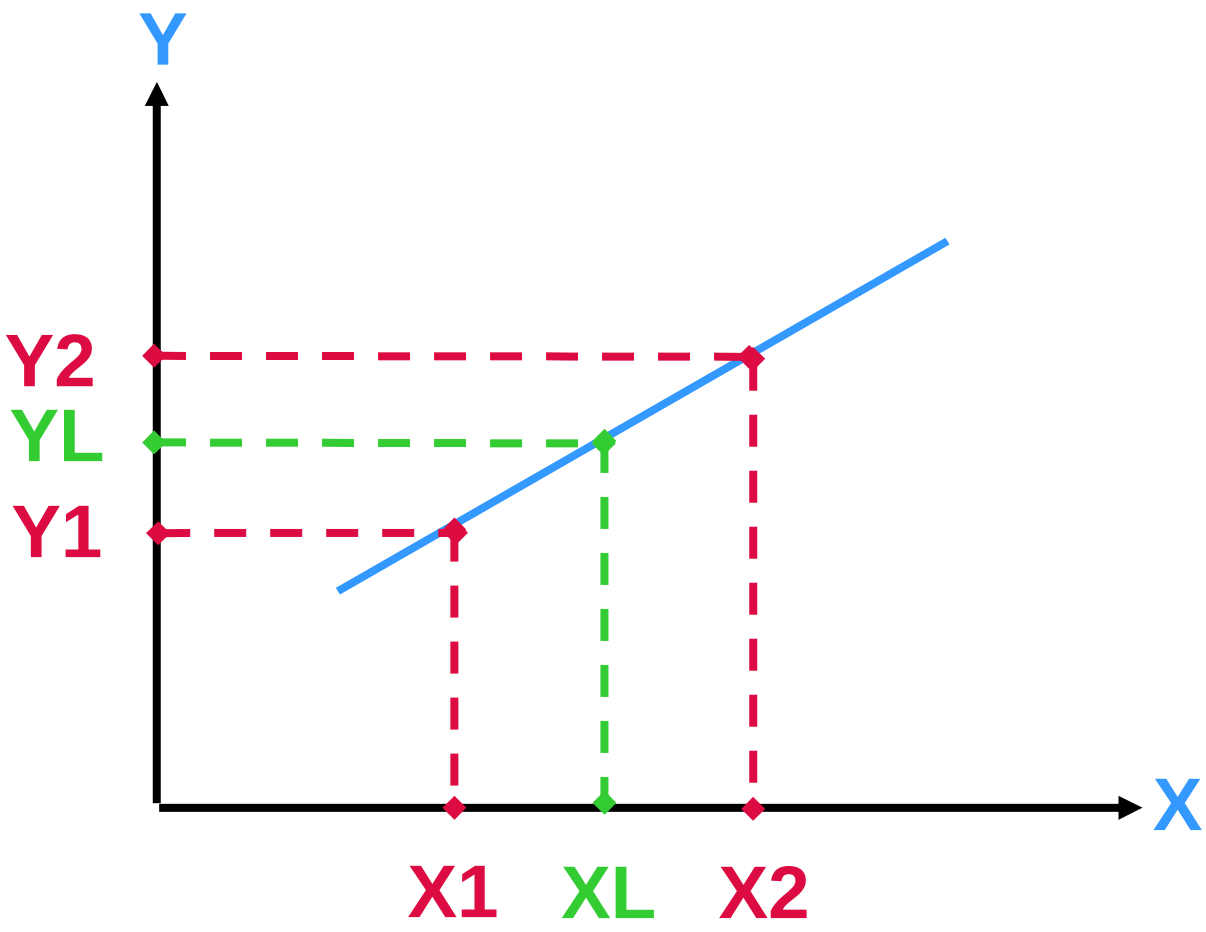
Lookup and Interpolate (TBL)

- **Use of TBL instruction:**
  - **set up *index register* to point to table entry "X1" (the table entry closest to, but less than or equal to, the desired lookup value)**
  - **"X2" is the table entry that follows "X1", and "XL" is the *desired lookup point* (between "X1" and "X2") along the X-axis**
  - **calculate (XL-X1) ÷ (X2-X1) and place the resulting binary fraction in the B register (typically requires use of FDIV instruction)**
  - **execute the TBL instruction; the result placed in the A register will be the *interpolated result* along the Y-axis:**

$$(A) \leftarrow Y1 + [\ (B) \times (Y2 - Y1)\ ]$$

Lookup and Interpolate (TBL)

- **TBL in action...**



Clicker Quiz

```
        org     $800
        ldx     #table
        ldab    #$40
        tbl     0,x
        staa    lookup
        ldab    #$C0
        tbl     1,x
        staa    lookup+1
halt    bra     halt

table   fcb     8
        fcb     16
        fcb     24

lookup  rmb     2
```

1. The (base 10) value that gets stored at location `lookup` is:

   A. 8

   B. 10

   C. 16

   D. 22

   E. none of the above

```
        org     $800
        ldx     #table
        ldab    #$40
        tbl     0,x
        staa    lookup
        ldab    #$C0
        tbl     1,x
        staa    lookup+1
halt    bra     halt

table   fcb     8
        fcb     16
        fcb     24

lookup  rmb     2
```

2. The (base 10) value that gets stored at location `lookup+1` is:

   A. 8

   B. 10

   C. 16

   D. 22

   E. none of the above

Jump Tables

- **A jump table is a special form of lookup table that contains addresses of subroutines**
- **<u>Note</u>: Here, the table entries are "double-byte" (16-bits) in length, since they represent addresses in memory**
- **<u>Applications</u>:**
  - **select ("vector to") a specific interrupt service routine under hardware control**
  - **select a function/subroutine to execute based on an input stream that has been parsed by a command interpreter**

Jump Tables

Example: Subroutine jump table

```
sindex        rmb    1          ; range of sindex is 1 to N (N ≤ 128)
stabl  +0     fdb    subr1
       +2     •
       +4     •
              •
  (N-1)*2     fdb    subrN
```

Calling sequence:

```
            sindex ← (1, N)
            jsr    sselect
  ──────►   {next instruction}
```

puts desired return address on stack

Jump Tables

Code to access table entry and transfer control:

Note: STABL offset is 2 * (SINDEX – 1)

```
sselect     ldab    sindex ; (B) ← subroutine index
            decb           ; remove bias
            aslb           ; multiply by 2
            clra           ; (A) ← 0
                           ; (D) = 2 * (SINDEX – 1)

            ldx     #stabl ; (X) ← starting address of table

            jmp     [d,x]  ; transfer control to selected
                           ;   subroutine
```

note use of indirect addressing mode

Example Application

Write an interactive "stupid quote" generator that prompts the user for a single character identifier (here, "a" through "f") and prints the corresponding quote on the emulated terminal screen. If the character entered is "out of range", an error message should be printed. If the character q is entered, the program should terminate. The user interface should function as follows (user input is in **bold**):

```
Welcome to the Stupid Quote Generator

Where do you want to go today (a-f)? a

Windows 7 will solve all your problems

Where do you want to go today (a-f)? d

We only ship *bug-free* software (i.e., all the bugs are free)

Where do you want to go today (a-f)? g

Message index is out of range

Where do you want to go today (a-f)? 2

Message index is out of range

Where do you want to go today (a-f)? q

Nice talking at you...
```

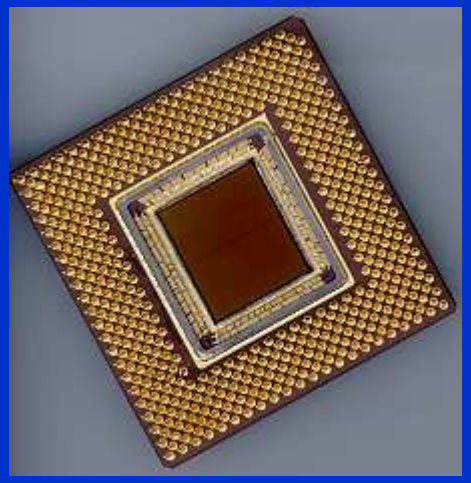
Example Application

This program consists of three modular components:

- a main program that implements the user interface.
- a pmsgx subroutine that prints the string pointed to by X at entry and terminates when an ASCII null character is encountered.
- a lookup subroutine that is passed the message index (in the range of 0 to N-1) in the A register and returns the starting address of the desired message string in the X register.

In addition, two I/O library routines are provided:

- inchar — inputs an ASCII character from the terminal keyboard and returns it in the A register
- outchar — prints the ASCII character passed to it in the A register on the terminal screen



Microcontroller-Based Digital System Design

Module 1-F

Assembly Language Programming Techniques: Parameter Passing

Outline

- **Introduction**
- **Parameter passing techniques**
  - **Call by value (using registers)**
  - **Call by name (global parameter area)**
  - **Following "call" instruction**
  - **Using the stack**
- **Example Application**

Learning Objective

- **<u>distinguish</u> among and <u>effectively utilize</u> different parameter passing techniques**

Introduction

- **There are several basic ways in which parameters can be passed to subroutines**
- **Each technique has its associated advantages and disadvantages, depending on the application**
- **The choice of which parameter passing technique is “best” is highly application dependent**

Method 1: Call by Value (Registers)

- **The "call by value" method of passing parameters uses the *CPU registers* available in the programming model**
- **It can only be used in cases where there is a *small number* (i.e., less than 4) of parameters, such as *conversion* and *device driver* routines**
- **For the 68HC12, the registers available for this purpose are A, B, X, and Y**
- **Example: Packed BCD to binary conversion**
- **Calling sequence:**

  (A) ← packed BCD number
  jsr bcdb

Conditions at Exit:

(A) unchanged
(B) ← converted binary

Method 1: Call by Value (Registers)

Conversion subroutine: **conv binary = (10 x u.n.) + l.n.**

```
bcdb    psha            ; save A register on stack
        anda    #0f0h   ; mask off lower nibble
                        ; => (A) = 16 * (upper nibble)
        lsra
        tfr     a,b     ; (B) = 8 * (upper nibble)
        lsra
        lsra            ; (A) = 2 * (upper nibble)
        psha            ; store temporarily on stack
        addb    1,sp+   ; (B) = (2 + 8) * (upper nibble)
        ldaa    0,sp    ; restore original value passed
        anda    #0fh    ; mask off upper nibble
        psha            ; store temporarily on stack
        addb    1,sp+   ; (B) = 10 * (upper) + (lower)
        pula            ; restore original value of (A)
        rts
```

"BAB" ≡ (B) ← (A) + (B)

Method 2: Call by Name (Global Area)

- **The "call by name" method of passing parameters uses a *CPU index register* as a *pointer* to the beginning of a *global* parameter area**
- **Where used:**
  - **pass a character string**
  - **pass a data structure (e.g., an array)**

Method 2: Call by Name (Global Area)

- **<u>Example</u>: Subroutine that sorts an array in ascending order**
- **Data structure declaration:**

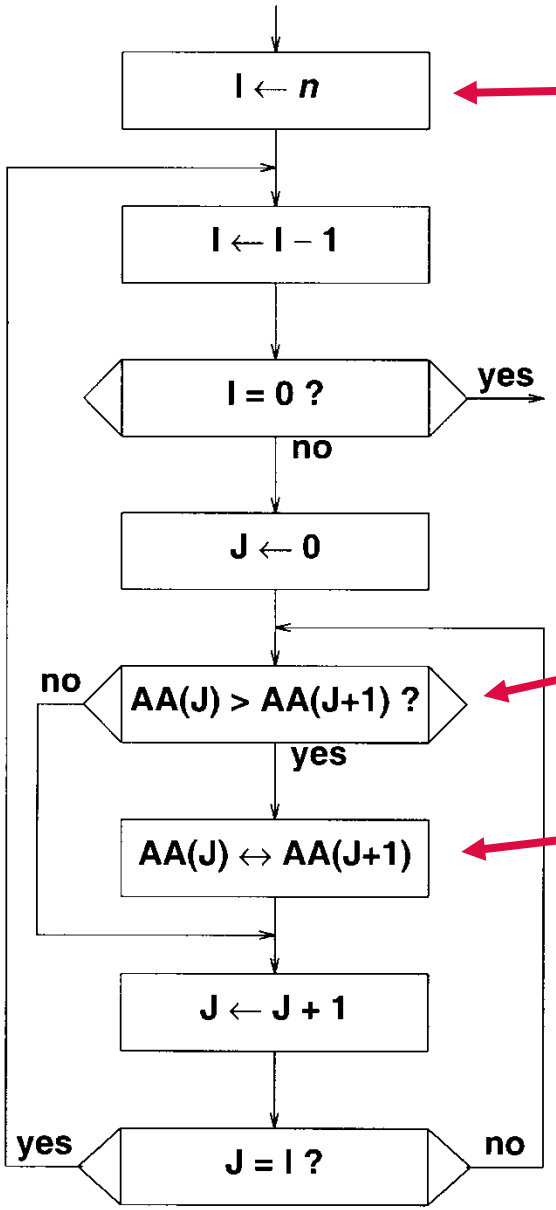
```
AA      fcb     n       ; first parmeter is number of elements
        fcb     ____    ; AA array element 0
        fcb     ____    ; AA array element 1
        .
        .
        .
        fcb     ____    ; AA array element N-1
```

Method 2: Call by Name (Global Area)

- **Calling sequence:**

```
ldx   #AA      ; X register points to
jsr   sortem   ; start of data structure
```

- **Conditions at exit:**
  - **X unchanged (still points to beginning of data structure)**
  - **array sorted in ascending order**
- **Method: "Bubble Sort"**
  - **use pairwise comparison and exchange**
  - **use X register to point to AA[i]**
  - **use Y register to point to AA[i+1]**




```
I        rmb   1
TMP      rmb   1

sortem   pshx
         ldaa  1,x+
         staa  I

         leay  1,x

loopo    dec   I

         beq   exit

         clrb

loopi    ldaa  b,x
         cmpa  b,y
         ble   noxchg

         staa  TMP
         ldaa  b,y
         staa  b,x
         ldaa  TMP
         staa  b,y

noxchg   incb

         cmpb  I
         bne   loopi
         bra   loopo

exit     pulx
         rts
```

initialize X and Y pointers

Note: B ≡ J

Method 3: Following “Call” Instruction

- **The “data following call” method of passing parameters uses space allocated *between* the “call” (here, JSR or BSR) instruction and the next instruction to provide a *private* parameter area**
- **Since the data immediately follows the “call”, the top stack item (“return address”) will be pointing to this data upon entry to the subroutine**
- **Note that the code contained in the subroutine must *correct* the return address on the stack *before* executing an RTS instruction to return to the calling program**

Method 3: Following "Call" Instruction

Example: String printing routine

Calling sequence:

```
jsr     pmsg
fcb     "Print this string"
fcb     0dh         ; ASCII return character
fcb     0ah         ; ASCII line feed character
fcb     0           ; ASCII null character (string termination)
{next instruction}
```

desired return address

Conditions at exit:

return address on stack corrected

return address placed on stack points here

Method 3: Following “Call” Instruction

Subroutine code:

```
pmsg    pulx                ; (X) ← starting address of string

loop    ldaa    1,x+        ; access character of string
        cmpa    #0          ; test for ASCII null character
        beq     exit

        jsr     outchar     ; use routine "outchar" to print character
                            ;   passed in (A)
        bra     loop
exit    pshx                ; place corrected return address
                            ;  back on stack
        rts
```

Method 4: Using the Stack

- **The *system stack* is the technique most commonly used by *high-level language* compilers**
- **While this method has a significant amount of "overhead" associated with it (in terms of the complexity of the calling and exit sequences), it makes possible *two important features* associated with modern high-level languages:**
  - **recursion (the ability of a subroutine to *call itself*)**
  - **reentrancy (the ability of a code module to be *shared* among quasi-simultaneously executing tasks)**

Method 4: Using the Stack

Example: Extended-precision add

Parameter arrays:

```
N          equ    number of bytes

augend     fcb    msb,...,lsb
addend     fcb    msb,...,lsb
sum        rmb    N
```

Note: Here, data is stored in *high order byte first* format (also referred to as "*big endian*")

Method 4: Using the Stack

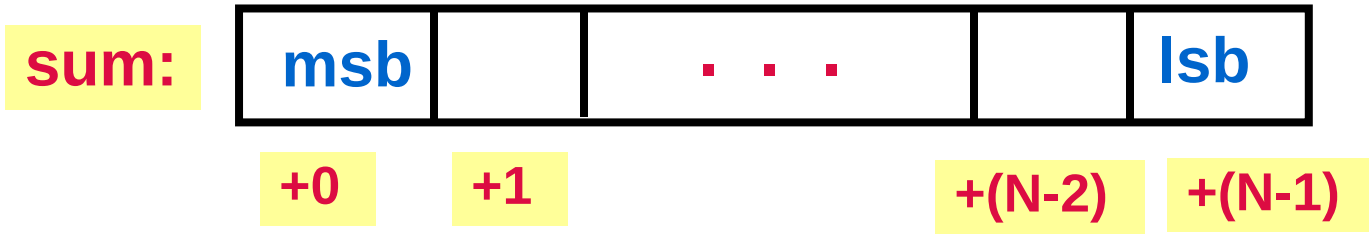
Calling sequence:

```
start     ldx    #augend  ; X points to high order augend byte
          ldy    #addend  ; Y points to high order addend byte
          ldab   #N
psh_op    beq    psh_n
          ldaa   1,y+
          psha            ; push addend byte, high byte first
          ldaa   1,x+
          psha            ; push augend byte, high byte first
          decb
          bra    psh_op
psh_n     ldab   #N
          pshb            ; push byte count on stack
          jsr    epadds   ; call epadds routine (return address
                          ;   now on top of stack)
```

Method 4: Using the Stack

Exit sequence:

```
            ldab    1,sp+       ; top stack item (after return from
            ldx     #sum-1      ;   epadds) is byte count
            leax    b,x         ; X points to low order byte of sum array
pop_lp      tbeq    b,edone     ; exit when byte count reaches zero
            ldaa    2,sp+       ; get result byte from stack (start with
            staa    1,x-        ;   low byte (and skip over addend byte)
            decb
            bra     pop_lp
edone       {next instruction}
```



Method 4: Using the Stack

Extended-precision add subroutine code:

```
epadds    clc                 ; CF ← 0
          leax   2,sp         ; X points to byte count (ret addr on top)
          ldab   1,x+         ; B set to byte count, X now points to
add_lp    beq    epexit       ;   low order byte of augend
          ldaa   0,x          ; load augend byte
          adca   1,x          ; add addend byte plus carry propagated
          staa   2,x+         ; replace augend byte with result byte
          decb                ;   and bump X to next addend byte
          bra    add_lp
epexit    rts
```

Stack State at Entry to “epadds”

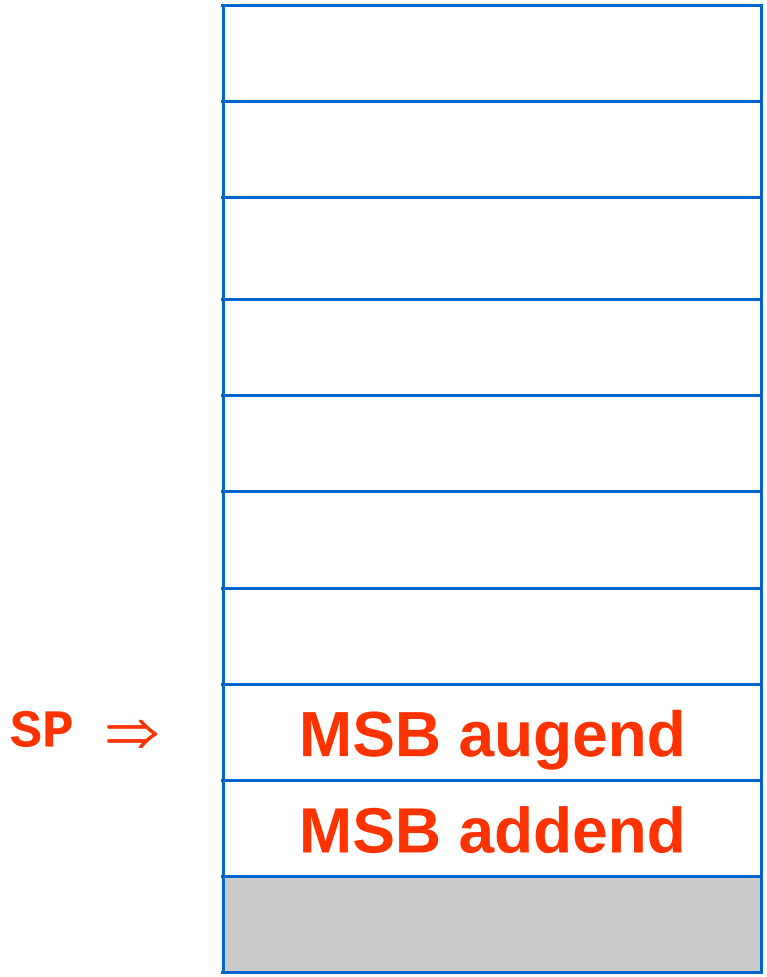
Calling sequence:

```
start     ldx    #augend
          ldy    #addend
          ldab   #N
psh_op    beq    psh_n
          ldaa   1,y+
          psha
          ldaa   1,x+
          psha
          decb
          bra    psh_op
psh_n     ldab   #N
          pshb
          jsr    epadds

augend    fcb    msb,...,lsb
addend    fcb    msb,...,lsb
sum       rmb    N
```

Stack State at Entry to “epadds”

Calling sequence:

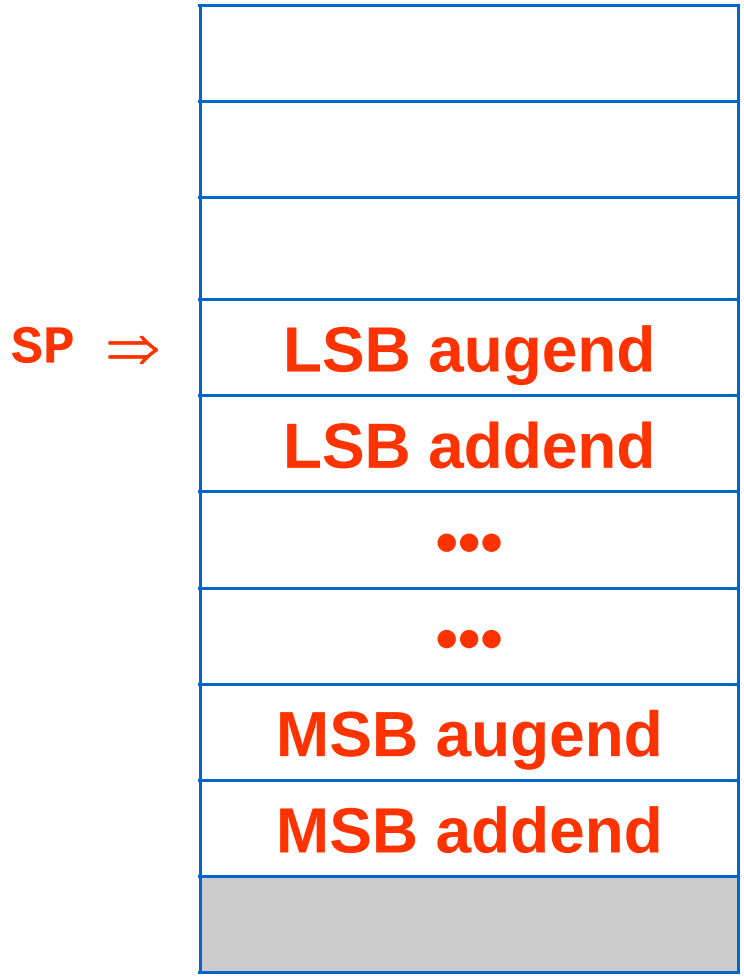
```
start     ldx     #augend
          ldy     #addend
          ldab    #N
psh_op    beq     psh_n
          ldaa    1,y+
          psha
          ldaa    1,x+
          psha
          decb
          bra     psh_op
psh_n     ldab    #N
          pshb
          jsr     epadds

augend    fcb     msb,...,lsb
addend    fcb     msb,...,lsb
sum       rmb     N
```



Stack State at Entry to “epadds”

Calling sequence:

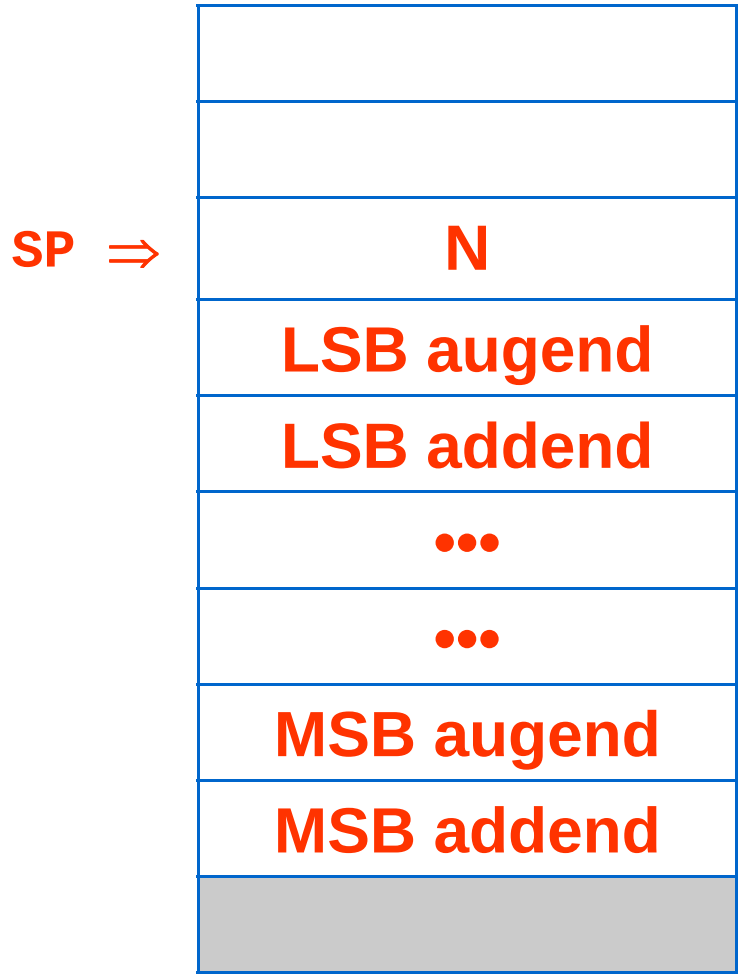
```
start     ldx    #augend
          ldy    #addend
          ldab   #N
psh_op    beq    psh_n
          ldaa   1,y+
          psha
          ldaa   1,x+
          psha
          decb
          bra    psh_op
psh_n     ldab   #N
          pshb
          jsr    epadds

augend    fcb    msb,...,lsb
addend    fcb    msb,...,lsb
sum       rmb    N
```



Stack State at Entry to “epadds”

Calling sequence:

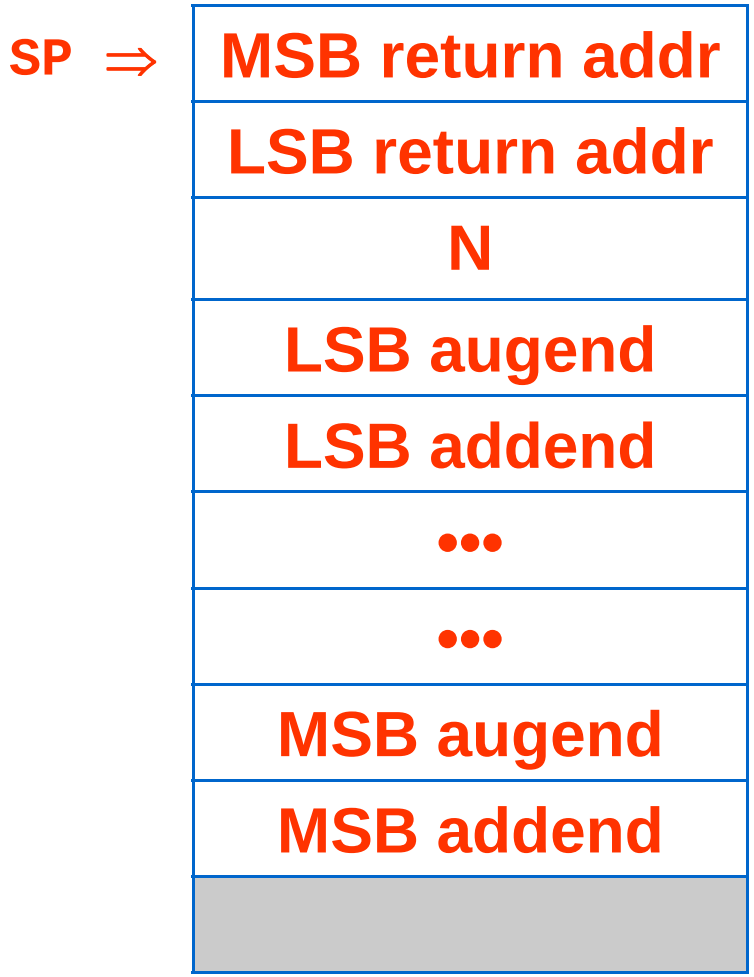
```
start     ldx   #augend
          ldy   #addend
          ldab  #N
psh_op    beq   psh_n
          ldaa  1,y+
          psha
          ldaa  1,x+
          psha
          decb
          bra   psh_op
psh_n     ldab  #N
          pshb
          jsr   epadds

augend    fcb   msb,...,lsb
addend    fcb   msb,...,lsb
sum       rmb   N
```



Stack State at Entry to “epadds”

Calling sequence:

```
start     ldx   #augend
          ldy   #addend
          ldab  #N
psh_op    beq   psh_n
          ldaa  1,y+
          psha
          ldaa  1,x+
          psha
          decb
          bra   psh_op
psh_n     ldab  #N
          pshb
          jsr   epadds

augend    fcb   msb,...,lsb
addend    fcb   msb,...,lsb
sum       rmb   N
```



Stack State as “epadds” Runs

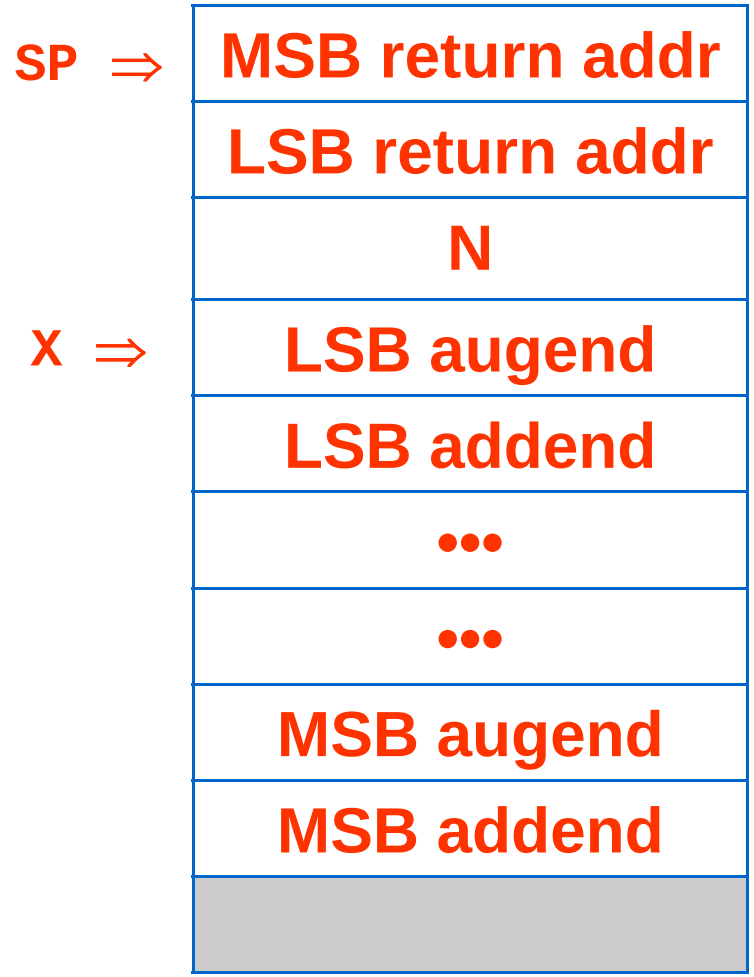
```
epadds    clc
          leax   2,sp
          ldab   1,x+
add_lp    beq    epexit
          ldaa   0,x
          adca   1,x
          staa   2,x+
          decb
          bra    add_lp
epexit    rts
```

SP ⇒	MSB return addr
	LSB return addr
X ⇒	N
X* ⇒	LSB augend
	LSB addend
	•••
	•••
	MSB augend
	MSB addend

*after post-increment by 1

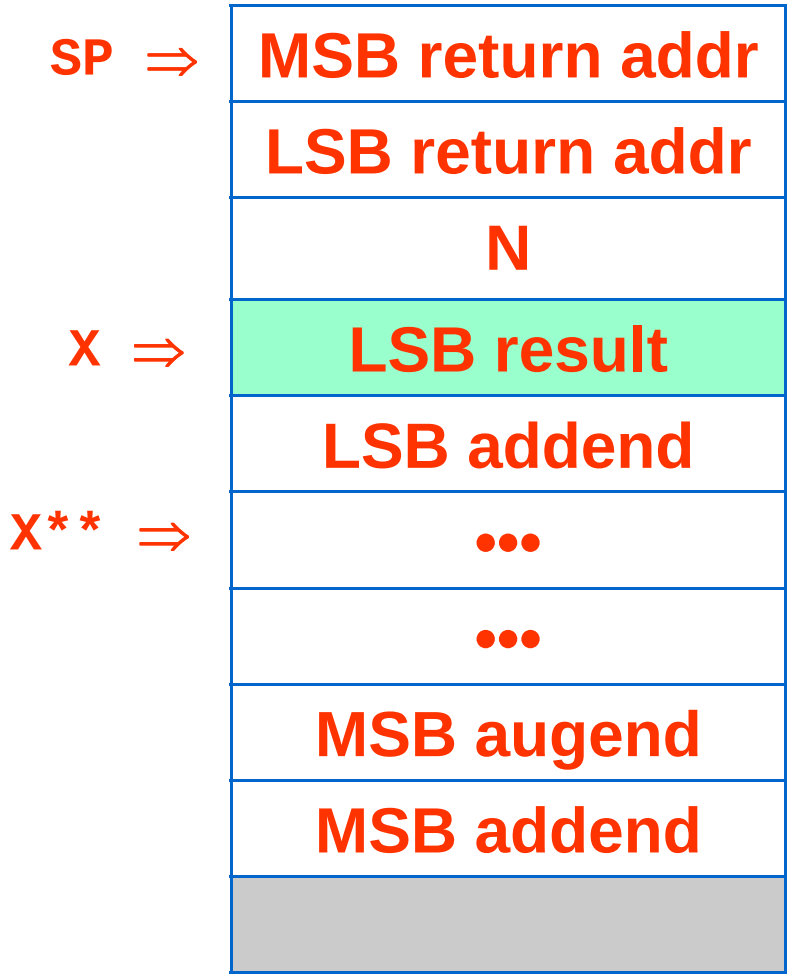
Stack State as “epadds” Runs

```
epadds      clc
            leax    2,sp
            ldab    1,x+
add_lp      beq     epexit
            ldaa    0,x
            adca    1,x
            staa    2,x+
            decb
            bra     add_lp
epexit      rts
```



Stack State as “epadds” Runs

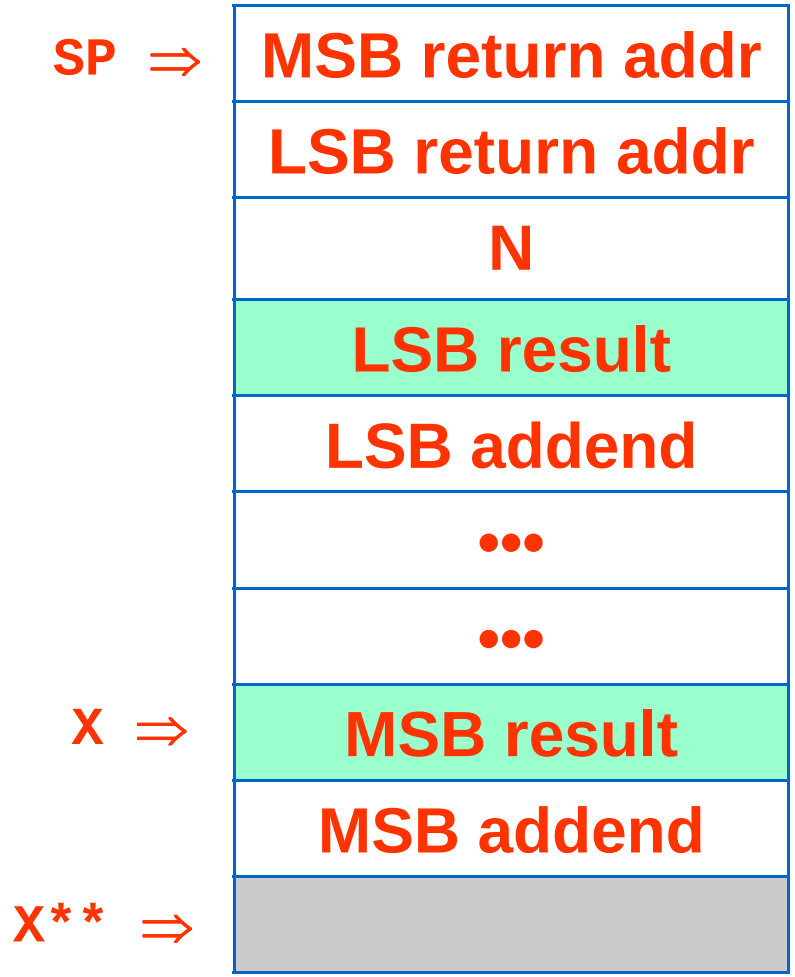
```
epadds      clc
            leax    2,sp
            ldab    1,x+
add_lp      beq     epexit
            ldaa    0,x
            adca    1,x
            staa    2,x+
            decb
            bra     add_lp
epexit      rts
```



**after post-increment by 2

Stack State Just Before “epadds” Returns

```
epadds      clc
            leax    2,sp
            ldab    1,x+
add_lp      beq     epexit
            ldaa    0,x
            adca    1,x
            staa    2,x+
            decb
            bra     add_lp
epexit      rts
```

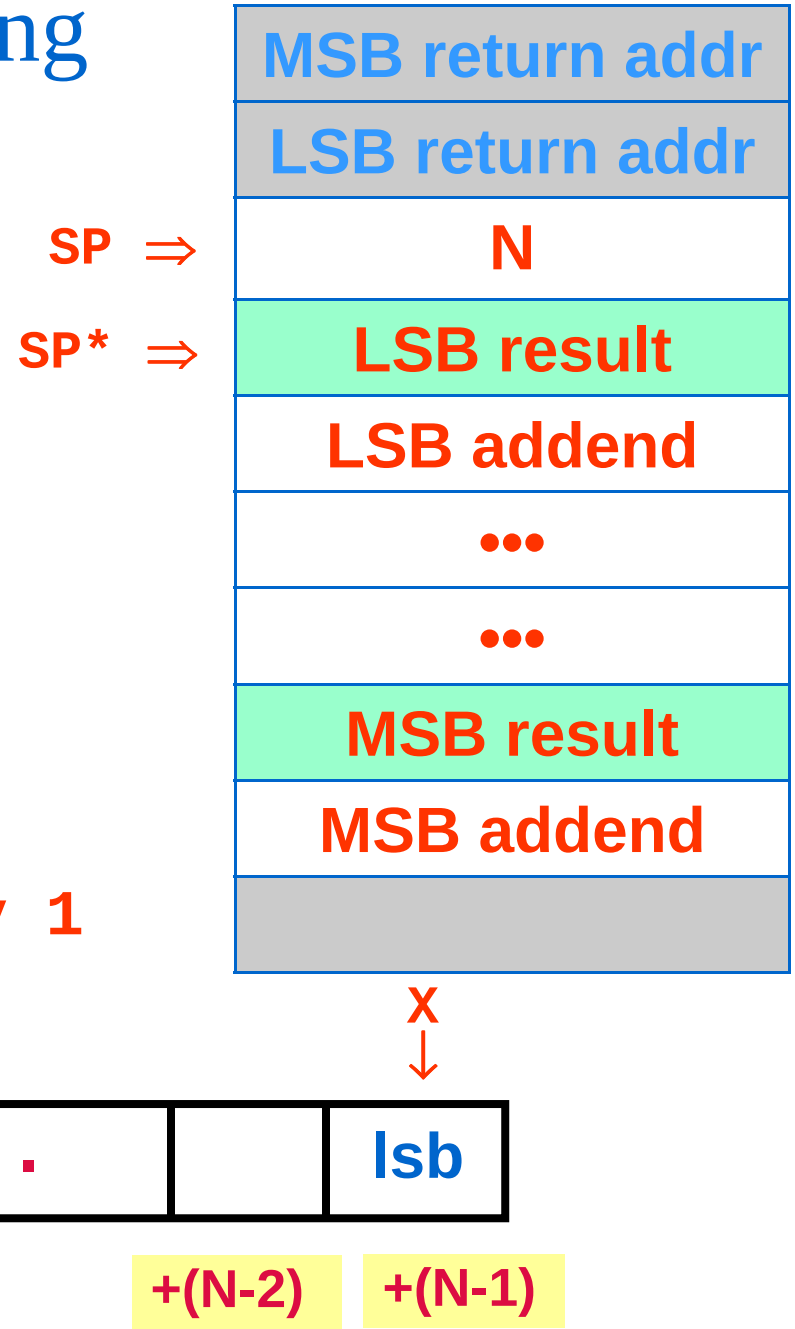


**after post-increment by 2

Exit Sequence Following Return From "epadds"

```
            ldab    1,sp+
            ldx     #sum-1
            leax    b,x
pop_lp      tbeq    b,edone
            ldaa    2,sp+
            staa    1,x-
            decb
            bra     pop_lp
edone       {next instruction
```

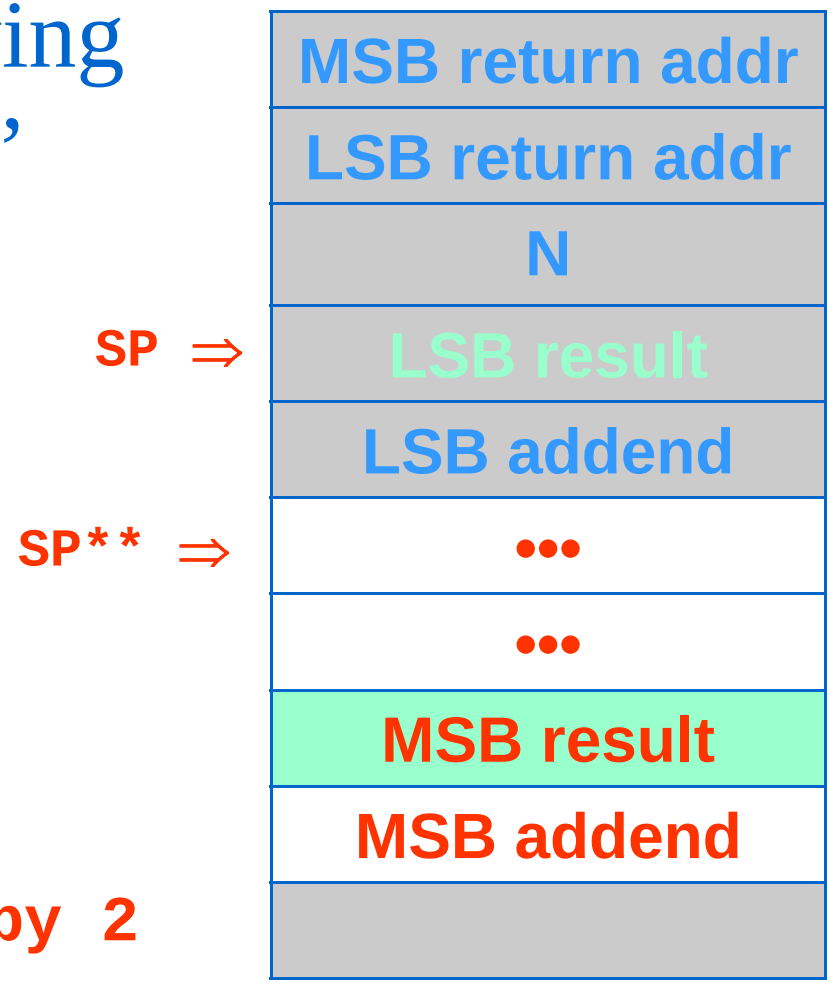
*after post-increment by 1

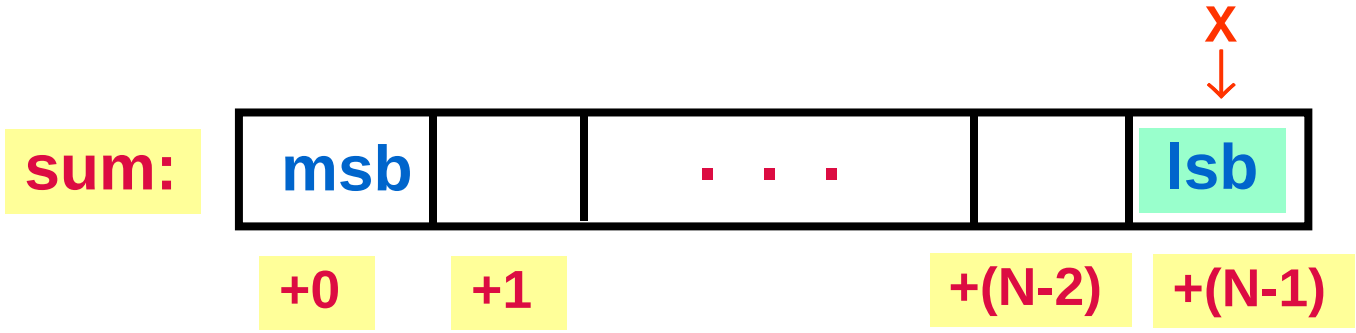


Exit Sequence Following Return From "epadds"

```
            ldab    1,sp+
            ldx     #sum-1
            leax    b,x
pop_lp      tbeq    b,edone
            ldaa    2,sp+
            staa    1,x-
            decb
            bra     pop_lp
edone       {next instruction
```

**after post-increment by 2

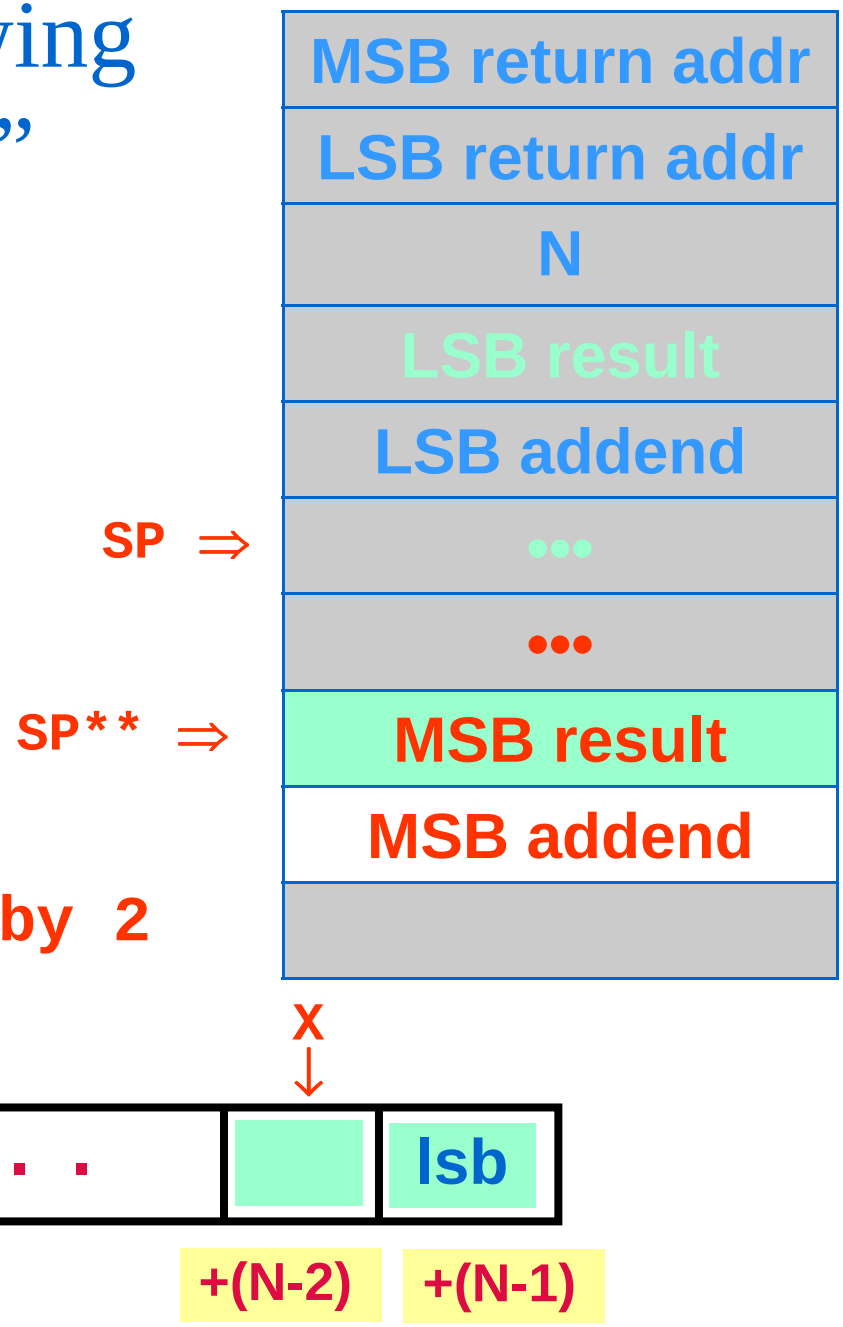




Exit Sequence Following Return From "epadds"

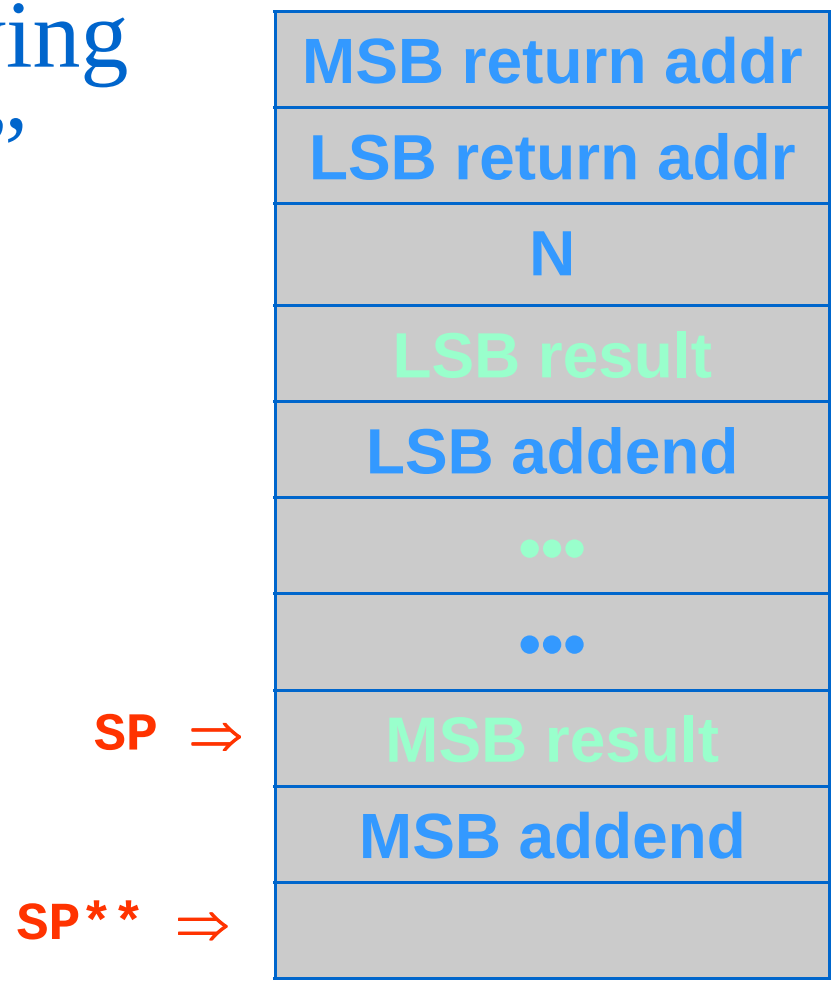
```
            ldab    1,sp+
            ldx     #sum-1
            leax    b,x
pop_lp      tbeq    b,edone
            ldaa    2,sp+
            staa    1,x-
            decb
            bra     pop_lp
edone       {next instruction
```

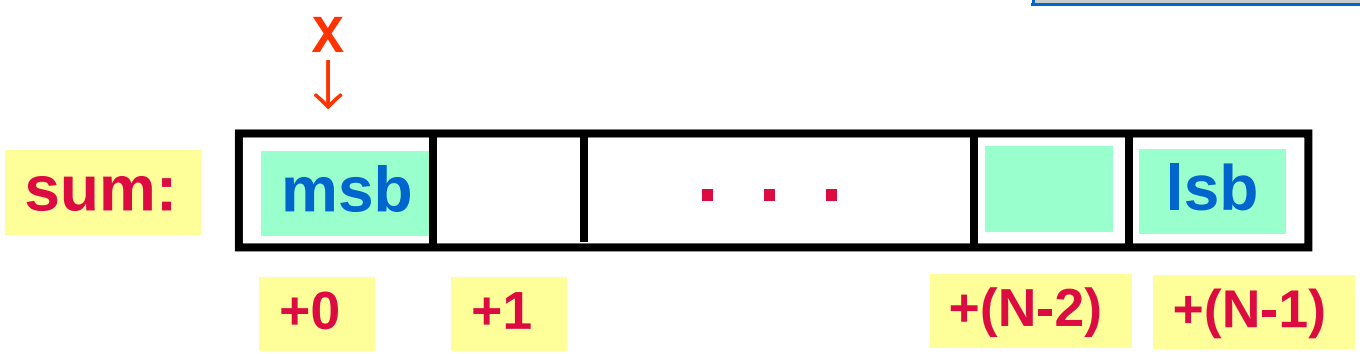
**after post-increment by 2



Exit Sequence Following Return From “epadds”

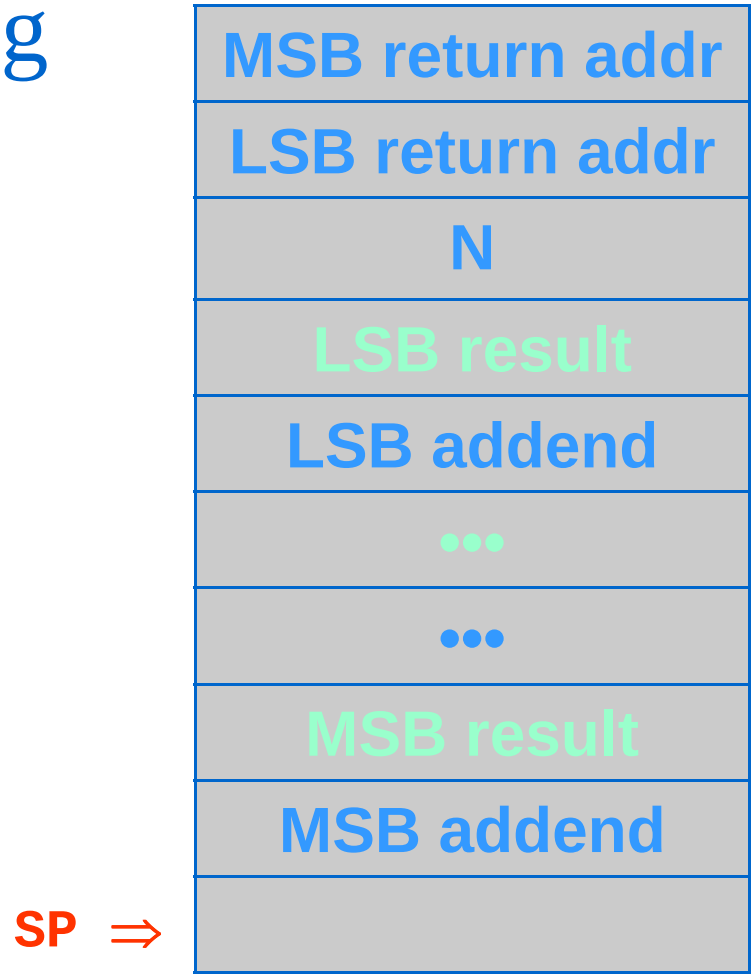
```
            ldab    1,sp+
            ldx     #sum-1
            leax    b,x
pop_lp      tbeq    b,edone
            ldaa    2,sp+
            staa    1,x-
            decb
            bra     pop_lp
edone       {next instruction
```

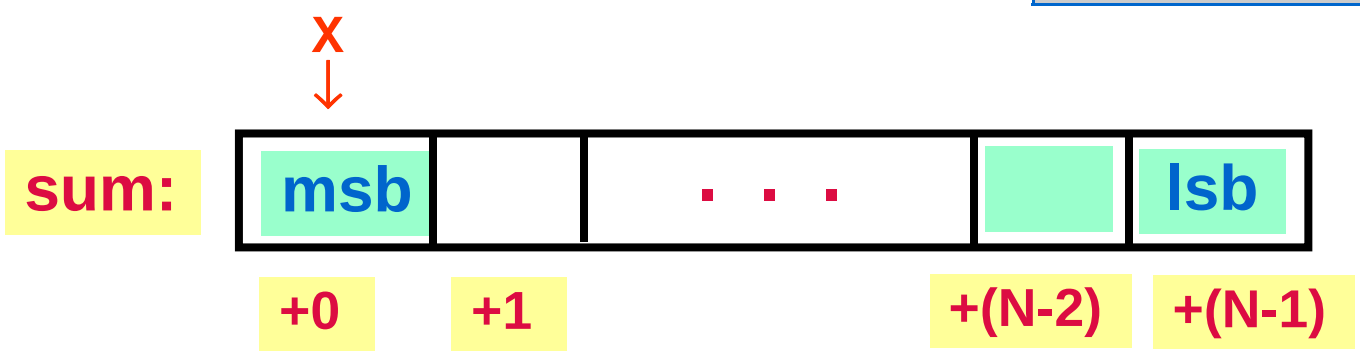




Exit Sequence Following Return From “epadds”

```
            ldab    1,sp+
            ldx     #sum-1
            leax    b,x
pop_lp      tbeq    b,edone
            ldaa    2,sp+
            staa    1,x-
            decb
            bra     pop_lp
edone       {next instruction
```





Example Application

Write a program that prompts a user for a five-digit access code (or "PIN"), checks it against a valid combination stored in memory, prompts the user to enter the code a second time if an error is made, and denies access if the PIN is entered incorrectly the second time. For security, the digits of the PIN should be echoed to the screen as "*" characters. Three possible scenarios are outlined below:

```
1   Welcome to the MegaMoney ATM!
    Enter PIN: *****                 ← valid code – first try
    Access granted

2   Enter PIN: *****                 ← invalid code – first try
    Error in PIN – try again…

    Enter PIN: *****                 ← valid code – second try
    Access granted

3   Enter PIN: *****                 ← invalid code – first try
    Error in PIN – try again…

    Enter PIN: *****                 ← invalid code – second try
    Sorry – access denied
    Your money is our money until you can figure out how to withdraw it!!
```

STEP 1: Write a subroutine **checkp** that compares the 5-digit PIN entered by the user against a 5-digit PIN stored in memory. At entry to **checkp**, the PIN entered by the user is pushed onto the stack (in the order that it was entered), as five separate bytes each containing a single BCD digit (i.e., it is in "unpacked" format). Also at entry to **checkp**, the X register points to the valid 5-digit code stored in memory; it, too, is stored in "unpacked" format, in five consecutive bytes. At exit, **checkp** should return with the carry flag set (C=1) if the two combinations match, or with the carry flag clear (C=0) if the two combinations do not match. Also at exit, the combination tested should be removed from the stack and the X register should be restored to its original value (no other registers need to saved/restored).

```
checkp
;
; Subroutine checkp compares the 5-digit code stored in memory
; (pointed to by X) with the 5-digit code passed on the stack
;
; If the two codes match, checkp returns with C=1; else, C=0
; The 5-byte code passed to this routine is de-allocated before exit
; The X register retains its original value
;
        puly            ; save return address in Y
        ldab    #NDIGS  ; B used as loop counter and
        decb            ;   as pointer offset
; Comparison loop
checkl
        pula            ; get digit of combination entered
        cmpa    b,x     ; compare with valid code
        bne     checkb  ; if mismatch, know combo is bad
        decb
        bpl     checkl  ; B ranges from 4 to 0
; If "fall through", all digits match
        sec             ; return with CF = 1
        pshy            ; restore return address saved in Y
        rts             ; note that X is unchanged
; If "mismatch" occurs, de-allocate remainder of combination
;   that was passed on the stack and return with CF = 0
checkb
        leas    b,sp    ; de-allocate rest of combination
        clc             ; return with CF = 0
        pshy            ; restore return address saved in Y
        rts             ; note that X is unchanged
```

STEP 2: **Write a main program that prints a welcome message, prompts the user to enter a PIN, checks the PIN using the checkp routine, gives the user a second chance if necessary, checks the second PIN entered using checkp, and prints a sarcastic message if the user fails to enter a valid PIN the second time. The main program should also provide storage for the valid 5-digit combination (of your choice). The main program should simply terminate with a STOP instruction after the "access granted" or "access denied" message is printed.**

```
; ATM access code verifier
;
; Tests 5-digit code entered by user
;   and compares it with 5-digit code
;   stored in memory

CR     equ    0dh
LF     equ    0ah
NULL   equ    00h
NDIGS  equ    5       ; number of digits in combination
NTRYS  equ    2       ; number of trys allowed

; Start of main program
;
; 1. Welcome user
; 2. Prompt for PIN
; 3. Input 5-digit PIN (echo "*" as code entered)
; 4. Call checkp to see if PIN is valid
; 5. If PIN is not valid then
;       if this is the second try then
;           print "access denied" and exit
;       else
;       print "try again" and goto 2
;       endif
;    else print "access granted" and exit
;    endif
```

```
        org     800h
main
        movb    #NTRYS,ntry     ; initialize number of trys
        jsr     pmsg
        fcb     CR,LF
        fcb     "Welcome to the MegaMoney ATM"
        fcb     CR,LF,NULL
prompt
        jsr     pmsg
        fcb     "Enter your 5-digit PIN: ",NULL
        ldab    #NDIGS
pmptlp
        jsr     inchar  ; get character
        jsr     atoh    ; convert ASCII to HEX
        psha            ; place on stack
        ldaa    #'*'
        jsr     outchar ; echo *
        dbne    b,pmptlp
        ldx     #combo  ; X points to valid combination
        jsr     checkp
        lbcs    wasgood
        dec     ntry
        beq     goaway  ; give user another chance
                        ;   if still has any trys left
                        ; else, deny access to user
        jsr     pmsg
        fcb     CR,LF
        fcb     "Error in PIN - try again"
        fcb     CR,LF,NULL
        bra     prompt
```

```
; Uh oh...user ran out of chances
goaway
        jsr     pmsg
        fcb     CR,LF
        fcb     "Sorry - Access Denied"
        fcb     CR,LF
        fcb     "Your Money is Our Money Until You Can Figure Out Your PIN!"
        fcb     CR,LF,NULL
        stop
wasgood
        jsr     pmsg
        fcb     CR,LF
        fcb     "Access granted"
        fcb     CR,LF,NULL
        stop

; Combination declaration
combo   fcb     7,6,5,9,4
ntry    rmb     1       ; number of trys
```

Clicker Quiz

```
dlyv  ldaa   #N*16-1     [2]
loopo ldab   #N*16-1     [2]
loopi nop                [1]
      nop                [1]
      dbne    b,loopi    [3]
      dbne    a,loopo    [3]
      rts                [5]
```

1. The maximum value N can be is:
   A. 4
   B. 16
   C. 64
   D. 256
   E. none of the above

```
dlyv   ldaa    #N*16-1      [2]
loopo  ldab    #N*16-1      [2]
loopi  nop                  [1]
       nop                  [1]
       dbne     b,loopi     [3]
       dbne     a,loopo     [3]
       rts                  [5]
```

2. If N = 1, the total number of cycles consumed is:

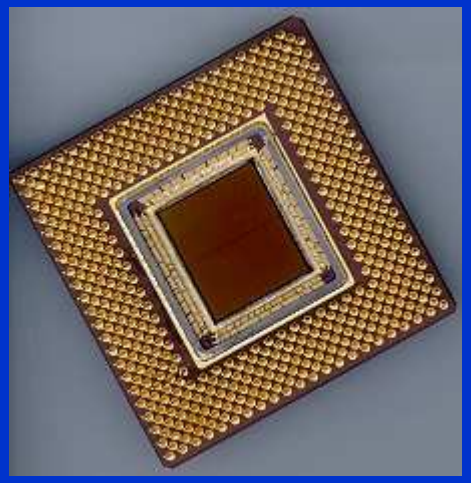
   A. 17

   B. 1207

   C. 1360

   D. 1367

   E. none of the above



Microcontroller-Based Digital System Design

Module 1-G

Assembly Language Programming Techniques: Macros and Structured Programming Methodology

Outline

- **Macros and conditional assembly**
- **Structured programming methodology**
- **Recommended programming procedure**

Learning Objectives

- **<u>define</u> a macro and <u>describe</u> its utility**
- **<u>compare and contrast</u> macros with subroutines**
- **<u>describe</u> the function and utility of conditional assembly**
- **<u>determine</u> the fields of an object file (S-Record)**
- **<u>create</u> an assembly language or C program that performs a prescribed task**
- **<u>diagnose and correct</u> (debug) programming errors**

Macros

- **<u>Definition</u>: A macro associates a *symbol* (name) with a *group (or set) of instructions*, to be *substituted* in a source program where that symbol is used**
- **Form of a *macro definition:***

```
name   MACRO
       {macro body}
       ENDM
```

- **Macro expansion is done at *assembly time* (listing of the expanded macro in the output file can be enabled or disabled)**
- **<u>Note</u>: Before a macro can be *invoked* in a program, it must be *defined***

Macros

- **Where used:**
  - **to define "new" instructions**

    <u>Example</u>: "BAB" ;(B) ← (A) + (B)

    ```
    psha
    addb   1,sp+
    ```

  - **to provide a "short-hand" notation for frequently used operations**

    <u>Example</u>: "RIGHT4A" ;shift (A) right 4 places

    ```
    lsra
    lsra
    lsra
    lsra
    ```

Macros

- **Illustration of *in-line substitution***

```
lls4a       MACRO
            lsla
            lsla
            lsla
            lsla
            ENDM

            org  $800
data        fcb  ____        ; storage location

main
            ldaa data
            llsa4
            staa data
```

```
lsla
lsla
lsla
lsla
```

Macros

- **Macros can also have arguments specified, as well as contain labels (but, labels can become multiply-defined if macro invoked more than once)**
- **<u>Example</u>: *n*-bit logical left shift of (A)**

```
                llsla     n

llsla           MACRO
                pshb
                pshc
                ldab  #\1
loop            lsla
                decb
                bne   loop
                pulc
                pulb
                ENDM
```

Note: *<u>Direct substitution</u>* occurs at *<u>assembly time</u>* with the parameter(s) listed on the macro invocation line

Macros

- **Can avoid use of labels in macros by using symbol for location counter ($)**

$ = starting address of instruction being assembled

```
llsla      n

llsla      MACRO
           pshb
           pshc
           ldab  #\1
addr       lsla        ; desired "loop" address
addr+1     decb
addr+2     bne   $-2
           pulc
           pulb
           ENDM
```

Macros

- **<u>Example</u>: HLL-looking *print* macro**

```
            print    "Print this string"

print       MACRO
            jsr   pmsg
            fcb   RET,LF
            fcc   \1
            fcb   RET,LF
            fcb   NULL
            ENDM
RET   equ   $0d
LF    equ   $0a
NULL  equ   $00
```

"most useful" macro!

Comparison of Macros and Subroutines

- **<u>Macros</u>:**
  - **macro invocation generates *in-line code***
  - **macro expansion done at *assembly time***
  - **a given program may contain *many* (expanded) *versions* of the same macro, due to parameter substitution**
- **<u>Subroutines</u>:**
  - **subroutine call causes a *branch* to another part of the program**
  - **subroutine call performed at *execution time***
  - **a given program will usually contain only a *single version* of a given subroutine**

Clicker Quiz

1. Invocation of a macro does <u>not</u>:
   A. cause a branch to another part of the program
   B. generate in-line code
   C. happen at assembly time
   D. allow substitution parameters
   E. none of the above

2. Macro parameter substitution occurs:

A. when the macro is called

B. when the macro is defined

C. when the macro is invoked

D. when the macro is executed

E. none of the above

```
dad   MACRO
      psh\1
      addd   2,sp+
      ENDM
```

3. The function performed by the macro invocation **dad x** is:

A. (D) ← (D) + (X)

B. (D) ← 2∗(X)

C. (D) ← 2∗(D)

D. (D) ← 2∗(SP)

E. none of the above

```
dad   MACRO
      psh\1
      addd  2,sp+
      ENDM
```

4. The function performed by the macro invocation **`dad d`** is:

A. (D) ← (D) + (X)

B. (D) ← 2∗(X)

C. (D) ← 2∗(D)

D. (D) ← 2∗(SP)

E. none of the above

Conditional Assembly

- **The purpose of *conditional assembly directives* in an assembly source file is:**
  - **to allow easy insertion and removal of debugging code**
  - **to allow configuration of a single source file for multiple versions of a target system (e.g., with different feature sets)**

Conditional Assembly

- **IF / ELSE / ENDIF directives are used to define conditional assembly code blocks**

```
IF value1 == value2 (or other conditional)
; block of code assembled if
; conditional is true
ELSE
; block of code assembled if
; conditional is false
ENDIF
```

<u>IMPORTANT</u>: Conditional assembly directives are evaluated at *<u>assembly</u>* time!

Conditional Assembly

- **<u>Example</u>:**

```
IF debug == 1
      ldaa test
      jsr  outchar
ELSE
      ldaa value
      jsr  outchar
ENDIF
```

Clicker Quiz

1. IF / ELSE / ENDIF directives are <u>not</u>:
   A. used to define conditional assembly blocks
   B. evaluated at assembly time
   C. controlled by evaluation of conditionals
   D. used to control execution of code blocks
   E. none of the above

2. IF / ELSE / ENDIF directives do <u>not</u>:

A. assemble into machine code

B. utilize C-like conditionals

C. provide a means of customizing assembly of a source file for different target systems

D. provide a means of controlling the insertion of debugging code

E. none of the above

Structured Programming Methodology

- **As programs get larger, they get:**
  - **harder to organize**
  - **harder to write**
  - **harder to debug**
- **A useful strategy for breaking down a large programming task into small, manageable parts ("code modules") is:**
  - ***top-down* organization, followed by**
  - ***bottom-up* coding/debugging**

Structured Programming Methodology

- **Outline of basic top-down, bottom-up strategy**
  - **write a series of *English* statements (which will eventually become the program's *comments*) describing the *basic steps* the program must perform**
  - **break each of these main tasks into a series of *subtasks* which must be performed (i.e., explain each major task in greater detail, again using *written comments*)**
  - ***further* break down each of these subtasks until your English descriptions (comments) translate more or less one-to-one directly into code**

Structured Programming Methodology

- **Outline of basic top-down, bottom-up strategy**
  - **define subtasks ("subroutines") that are *common* to several major tasks; identify parameters which must be passed, and draw a block diagram illustrating the *hierarchical arrangement* of these subtasks**
  - **flowchart (or outline) each of these subtasks (each should be small enough that it can be flowcharted or outlined on a *single page*)**
  - **finally, write code for each subtask, starting at the *bottom* of the hierarchical tree and working *up* ("bottom-up" coding), *testing* each code module *as it is written* – note that the "main" program should merely be a series of subroutine calls**

Recommended Programming Procedure

- **Read the program specification (several times) to gain an understanding of the *basic functionality* required and how the user will interface with (i.e., provide input to) the program**
- **Based on the program specification, identify *primary tasks* and *data structures*; form a *high-level block diagram* of the program**
- **Based on the program specification, *flowchart* (or *outline*) each routine, starting with "main" – use English statements or pseudo-code, *not* assembly language or register references***

***Why? Because at this point, the program is being *designed*, NOT implemented!**

Recommended Programming Procedure

- **Review the initial program design based on the following:**
  - **logical correctness**
  - **module length**
  - **structural soundness**
  - **relegation of common code segments to subroutines**
- **Code the primary data structures along with each individual routine, based on the flowchart/outline developed previously (begin with "main" and consider register usage before coding each routine)**
- ***Comment as you write code***
- **Debug routines *one at a time* using instruction tracing, breakpoints, and other debugger functions**

Modular Programming Example

- **One of the files generated by the assembler program is an "S-record" (or "object file")**
- **An S-record contains all the information necessary to load object code into the target system's memory**
  - **start code "S"**
  - **record type (1 → data record, 9 → end record)**
  - **number of bytes**
  - **address at which machine code is to be stored**
  - **machine code/data bytes**
  - **checksum byte**
- **A loader program (that typically runs on the target system) translates the S-record file into a memory image**

S-Record Loader

- **Sample S-record:**

```
S1070800F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Sample S-record:**

```
S1070800F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Sample S-record:**

```
S1070800F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Sample S-record:**

```
S1070800F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Sample S-record:**

```
S10708 00F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Sample S-record:**

```
S1070800F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Sample S-record:**

```
S1070800F236723D19
S9030000FC
```

Record No.	Start Code	Record Type	No. of Bytes	Address Field	Data Field	Checksum
1	S	1	07	0800	F236723D	19
2	S	9	03	0000	none	FC

S-Record Loader

- **Checksum calculation:**

```
S1070800F236723D19
S9030000FC
```

```
  07
  08
  00
  F2
  36
  72
 +3D
 ----
 E6h
```

in binary: 11100110B

complement: 00011001B (checksum)

checksum = 19h

Clicker Quiz

1. For the S-Record S1 06 09 01 02 03 04 __, the checksum should be:

A. $19

B. $25

C. $DA

D. $E6

E. none of the above

2. For the S-Record S1 06 09 01 02 03 04 E6, the value loaded into location $902 is:

A. $01

B. $02

C. $03

D. $04

E. none of the above

Basic Algorithm for S-Record Loader

- **Loop until “S” (start code) is received**
- **Read record type (1 or 9)**
- **Read the total number of record bytes (single byte)**
- **Read the load address (two bytes)**
- **Read the data fields and store each value in consecutive locations, starting at the specified address**
- **Read and process the checksum**

Routines That Need to be Written

- **FNDSTRT – find start of S-record**
- **RECTP – read and check validity of record type**
- **GETNUM – read the data byte count**
- **GETADDR – read in memory load address**
- **GETDAT – read in data fields of record (up to the checksum byte)**
- **CHECK – read and process the checksum byte**
- **PMSG – print message string**
- **GETBYTE – read two ASCII characters and return value as a hex byte**
- **GETWORD – read four ASCII characters and return value as a hex word**
- **ATOH – convert ASCII character to hex equivalent**

Outline of “main” Program

```
;
; main -- Implements main record-processing loop.  Checks for
;         end record and download errors.
main    clr     error       ; Clear the error flag
        jsr     pmsg
        fcb     'Ready to load S-record...'
        fcb     RET,LF,NULL

recloop jsr     getrec      ; Read and process next record
        ldaa    rectyp      ;   Get type of last record loaded
        cmpa    #ENDREC     ; Was it an end record?
        bne     recloop     ;   No: Then process next record

        ldaa    error       ; Check error counter
        bne     errmsg      ; If error count not zero, print message
        jsr     pmsg
        fcb     '*** Your file loaded correctly!'
        fcb     RET,LF,NULL
        stop
errmsg  jsr     pmsg
        fcb     '*** An error occurred while loading your file...'
        fcb     RET,LF,NULL
        stop
```

Outline of “main” Program

```
;
; Global variables and equates
;
DATREC  equu    1       ; Data record type
ENDREC  equ     9       ; End record type
NULL    equ     0       ; ASCII null character
RECTYP  rmb     1       ; S-Record type (DATREC or ENDREC)
BYTSUM  rmb     1       ; Current byte sum (for checksum calculation)
ERROR   rmb     1       ; Error flag byte (00 = OK, 01 = error)
```

Note: The variables (RECTYP, BYTSUM, ERROR) must be stored in SRAM

Loader Program Subroutines

```
;
; getrec -- Process an individual S-record
;
getrec  jsr     fndstrt         ; Loop until (S)tart char found
        jsr     rectp           ; Read the record type
        jsr     getnum          ; Read in count of bytes in a record
        jsr     getaddr          ; Read in memory storage address
        jsr     getdat          ; Read in record data bytes
        jsr     check           ; Add to checksum value
        rts                     ; Exit routine

;
; findstrt -- Just loop until an S (for start of record) is found on
;             serial input
fndstrt jsr     inchar          ; Read in next character
        cmpa    #'S'            ; Is it the start of a record?
        bne     fndstrt         ;   No: Then keep waiting for S
        rts                     ;   Yes: Exit routine
```

Loader Program Subroutines

```
;
; rectp -- Read in 1-byte record type, check for validity, and store
;          in memory
;
rectp   jsr     inchar          ; Read record type from serial input
        suba    #'0'            ; Convert record type from ASCII to hex
        staa    RECTYP          ; Store record type in memory
        cmpa    #ENDREC         ; Is it a valid record type (end)?
        beq     endtp           ;   Yes: Then exit routine
        cmpa    #DATREC         ;   No: Is it a data record type?
        beq     endtp           ;     Yes: The exit routine
        inc     ERROR           ;     No: Then set error flag
endtp   rts                     ; Exit routine
```

Loader Program Subroutines

```
;
; getnum -- Read in count of data bytes (plus two for record
;           destination address) to load.  Note that the byte count
;           is returned in the B reg.  Also, reset the byte sum.
;
getnum  jsr     getbyte         ; Read in two ASCII chars (byte count)
        staa    BYTSUM          ; Store count in memory
                                ;   (for byte summing)
        tfr     a,b             ; Save byte count in B for later use
        rts                     ; Exit routine
```

Loader Program Subroutines

```
;
; getaddr -- Read in the destination memory-address of the data
;            about to be loaded.  Also add each byte received to
;            byte sum and decrement byte count by 2.  Routine
;            returns the target in the X register.
;
getaddr pshb                    ; Save byte count
        jsr     getword         ; Get the 16-bit target address
        tfr     d,x             ; Store address in X for later use
        adda    BYTSUM          ; Include upper byte of addr in sum
        staa    BYTSUM          ; Update byte sum
        addb    BYTSUM          ; Include lower byte of addr in sum
        stab    BYTSUM          ; Update byte sum
        pulb                    ; Restore byte count
        subb    #02h            ; Dec count by 2 since addr is 2 bytes
        rts                     ; Exit routine
```

Loader Program Subroutines

```
;
; getdat -- Read in the record data bytes, and place them in memory
;           (as pointed to by X).  Update the byte count and stop
;           reading data when only the checksum value remains.
;
getdat  ldaa    RECTYP          ; Get record type
        cmpa    #ENDREC         ; Is this an end record?
        beq     yesend          ;   Yes: Then no data to receive
                                ;   No: Then get the record data
getlp   jsr     getbyte         ; Read in the next data byte
        staa    1,x+            ; Store in memory, and inc pointer
        adda    BYTSUM          ; Add received byte to byte sum
        staa    BYTSUM          ; Update byte sum in memory
        decb                    ; Dec count of bytes left to receive
        cmpb    #01             ; Last data byte received?
        bne     getlp           ;   No: Get next data byte
yesend  rts                     ;   Yes: Then exit routine
```

Loader Program Subroutines

```
;
; check -- Reads the checksum byte, compares with computed one's
;          complement, and set error flag (if necessary) to indicate
;          parity error.
;
check   jsr     getbyte; Read in the checksum byte
        adda    BYTSUM  ; Add checksum to the byte sum in memory
        cmpa    #0ffh   ; Is there a checksum error?
        beq     endck   ;   No: Then don't set error flag
        inc     ERROR   ;   Yes: Then set error flag
endck   rts             ; Exit routine
```

Loader Program Subroutines

```
;
; pmsg -- Print string following call to routine.  Note that
;         subroutine return address points to string, and is
;         adjusted to point to next valid instruction
;
pmsg    pulx            ; Get pointer to string (return addr)
ploop   ldaa    1,x+    ; Get next character of string
        cmpa    #NULL   ; Test for string termination
        beq     pexit   ; Exit if ASCII null encountered
        jsr     outchar ; Print character on terminal screen
        bra     ploop   ; Process next string character
pexit   pshx            ; Place corrected return address on stack
        rts             ; Exit routine
```

Loader Program Subroutines

```
; getbyte -- Inputs two ASCII characters from the HC12 SCI and
;            converts them to two hexadecimal digits packed into a
;            single byte.  Returns byte equivalent in A register.
getbyte     pshc
            jsr     inchar    ; get first ASCII character
            jsr     outchar   ; echo character
            jsr     atoh      ; convert ASCII character to hex
            bcs     errhex1   ; if not hex, go to error routine
            asla              ; shift converted hex digit
            asla              ;   to upper nibble
            asla
            asla
            psha              ; save on stack temporarily
get2        jsr     inchar    ; get second ASCII character
            jsr     outchar   ; echo to screen
            jsr     atoh      ; convert ASCII character to hex
            bcs     errhex2   ; if not hex, go to error routine
            oraa    1,sp+     ; OR converted hex digits together
            pulc
            rts
errhex1     ldaa    #'?'      ; get ? to prompt for new character
            jsr     outchar
            bra     getbyte
errhex2     ldaa    #'?'
            jsr     outchar
            bra     get2
```

Loader Program Subroutines

```
; getword – Get four ASCII characters and return value as a hex word
;           in the D register

getword jsr     getbyte         ; get first byte of the data entered
        bcs     badval          ; is there an error in the first byte?
        tfr     a,b             ; save MSB in B
        jsr     getbyte         ; get second byte of data entered
        bcs     badval          ; is there an error in the second byte?
        exg     a,b             ; put MSB in A and LSB in B
        andcc   #$FE            ; no errors, clear Z flag
        rts

badval  orcc    #$01            ; error, set Z flag
        rts
```

Loader Program Subroutines

```
; atoh -- Converts an ASCII character to a hexadecimal digit
;         ASCII character passed via A register
;         Converted hexadecimal digit returned in A register,
;         CF = 0, result OK; CF = 1, error occurred (invalid input)
atoh       pshb
           pshx
           pshy
           suba    #$30     ; subtract "bias" to get ASCII equivalent
           blt     outhex
           cmpa    #$0a
           bge     cont1
quithx      clc             ; return with CF = 0 to indicate result OK
           puly
           pulx
           pulb
           rts
cont1      suba    #$07
           cmpa    #$09
           blt     outhex
           cmpa    #$10
           blt     quithx
           suba    #$20
           cmpa    #$09
           blt     outhex
           cmpa    #$10
           blt     quithx
outhex      sec            ; set CF = 1 to indicate error
           puly
           pulx
           pulb
           rts
```