Lecture #4

Microcontroller Instruction Set – 2

18-348 Embedded System Engineering

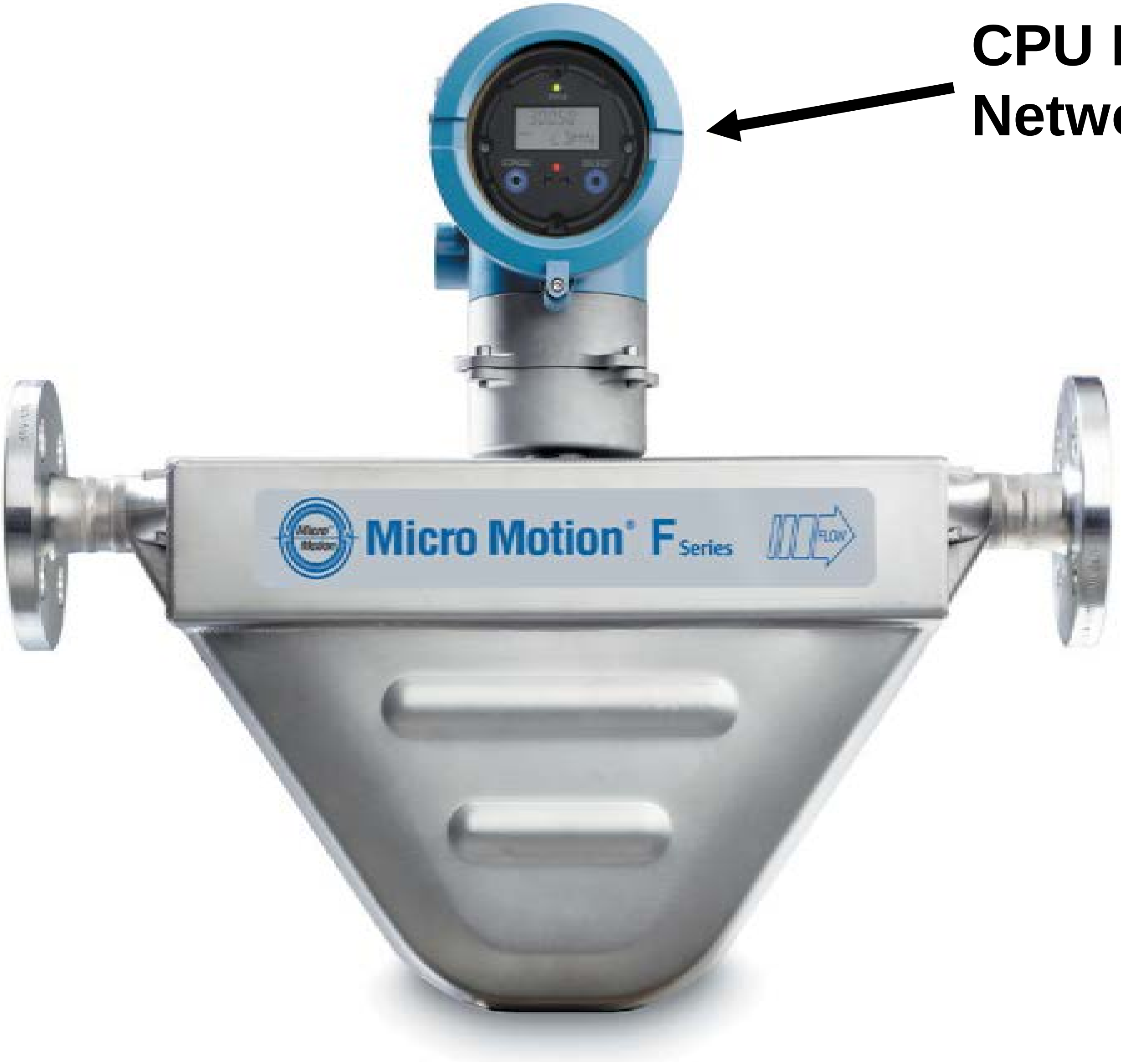
Philip Koopman

Monday, 25-Jan-2016

Electrical&Computer ENGINEERING

Carnegie Mellon

Example Application: Coriolis Mass Flow Meter



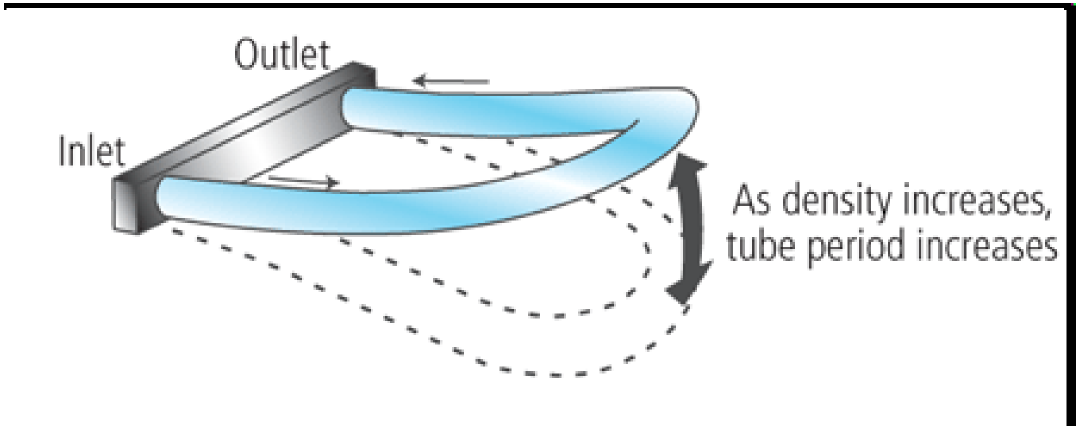
CPU Module &
Network Interface

[Emerson Process Management]

Vibrating The Tube Permits Flow Measurement

- **Used to precisely measure viscous fluids and slurries**

[http://en.wikipedia.org/wiki/Mass_flow_meter]



[www.isa.org]

Where Are We Now?

- **REMINDER: Do Pre-Labs COMPLETELY ON YOUR OWN!**
  - Do not work with your lab partner (or anyone else)
  - Do not talk about it with your lab partner until AFTER you BOTH hand in

- **Where we've been:**
  - Embedded hardware
  - Microcontroller Instruction Set – the basics
- **Where we're going today:**
  - Microcontroller Instruction Set – advanced

  Note: you saw assembly stuff in 18-240, so we're covering it pretty quickly
  - If this stuff is confusing, go to office hours to get help

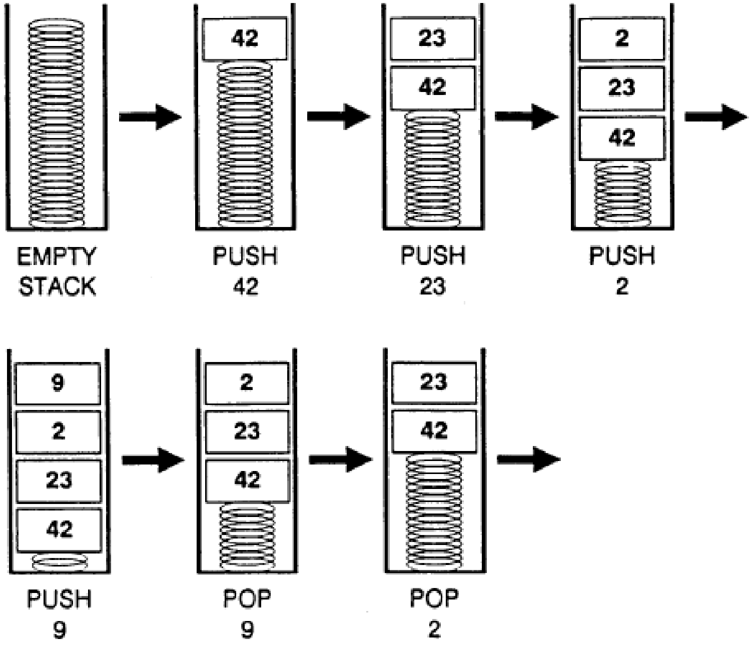
- **Where we're going next:**
  - Engineering process & design
  - Embedded-specific C
  - Coding hacks & multiprecision math
  - …

Preview

- **Stack usage**
  - Pushing & popping with stack
  - Subroutine linkage
- **Other assembler operations**
  - Position, memory, and other management
  - Labels
  - Macros
- **More on timing**
  - Cycle-accurate simulation

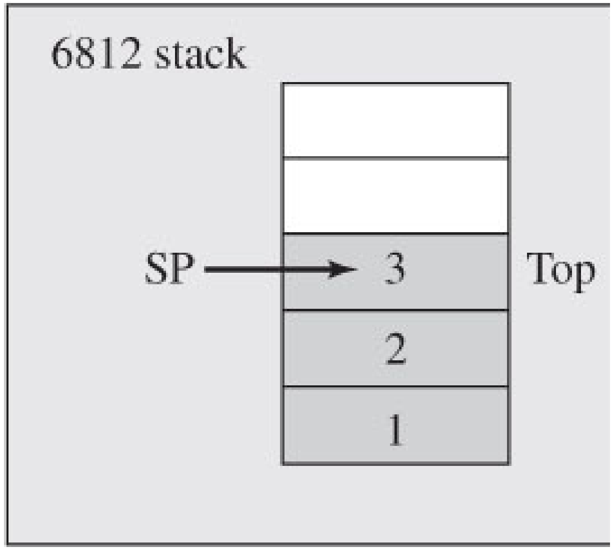
The Stack – Concept & Implementation

- **Concept:**



[Koopman]

- **Implementation:**
  - Uses a pointer to memory
  - The pointer moves up and down as top of stack, not the memory contents!
  - Points to top of stack



[Valvano]

Pushing To The Stack

- **“PSH” instructions – pushes a register onto the stack**
  - PSHA, PSHB, PSHD, PSHX, PSHY
  - PSHC – condition codes (will get to that in a moment)

Example:

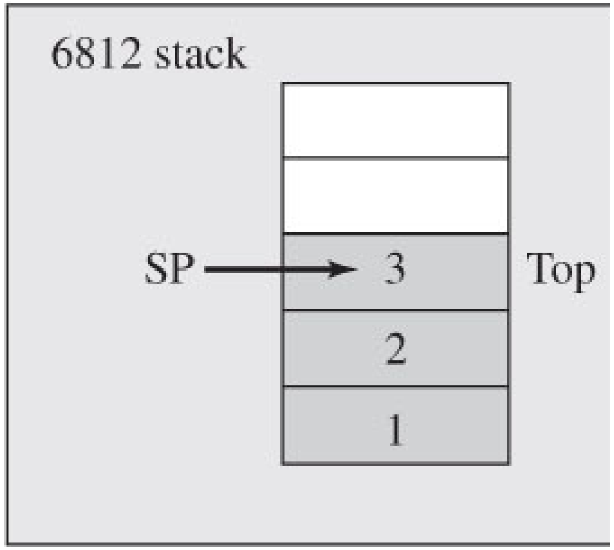
```
    LDAA #1
    LDAB #2
    PSHA
    PSHB
    LDAA #3
    PSHA
```

At end, A=3, B=2

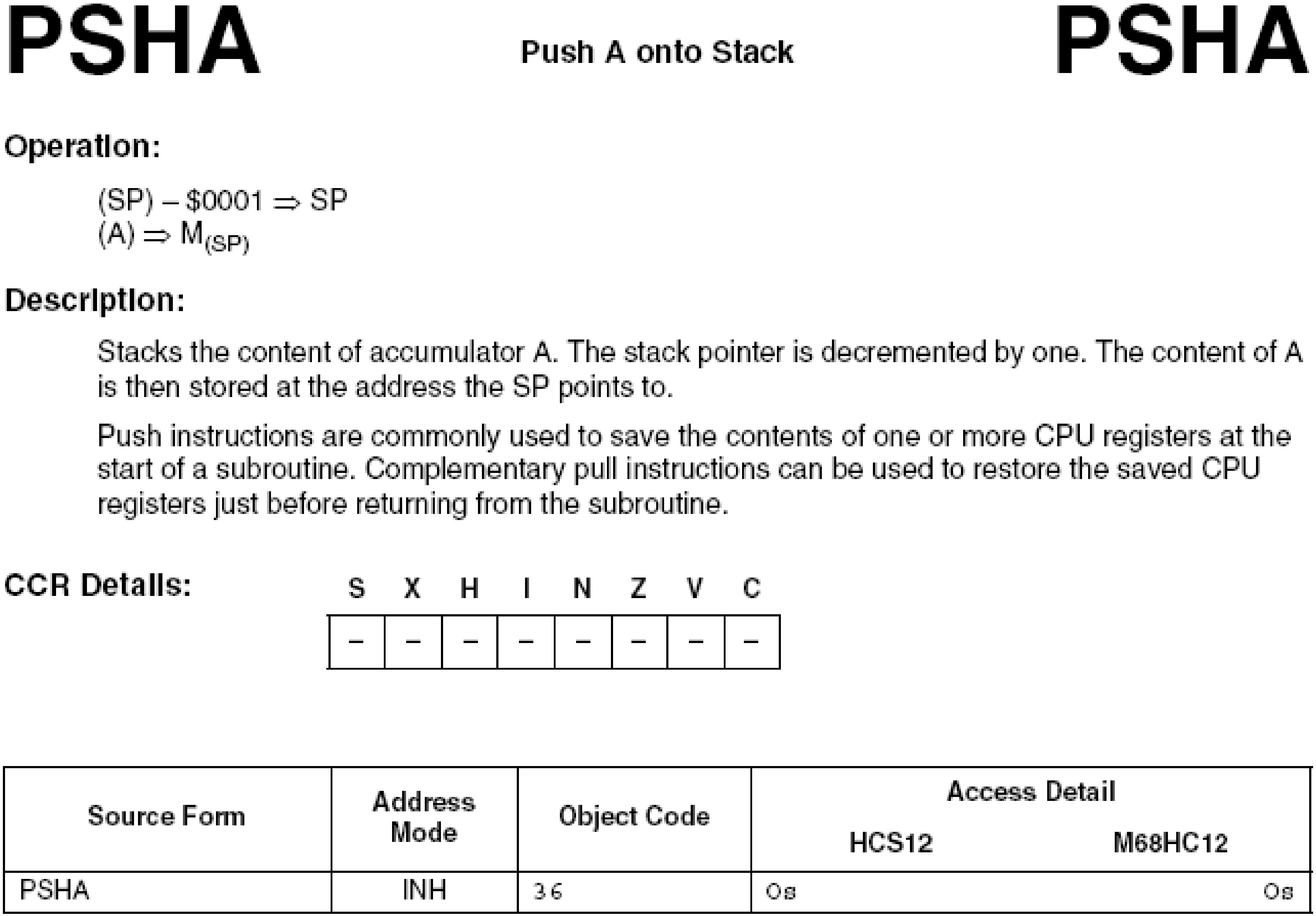
- PSH doesn’t change register values

(On 6812, SP always points to top-most element in use)

End value:



[Valvano]



PSHA — Push A onto Stack — PSHA

Operation:

$(SP) - \$0001 \Rightarrow SP$
$(A) \Rightarrow M_{(SP)}$

Description:

Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stored at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHA	INH	36	Os	Os

[Freescale]

Pulling ("Popping") From The Stack

- **"PUL" instructions – pulls a register value from the stack**
  - PULA, PULB, PULD, PULX, PULY
  - PULC – condition codes (will get to that in a moment)

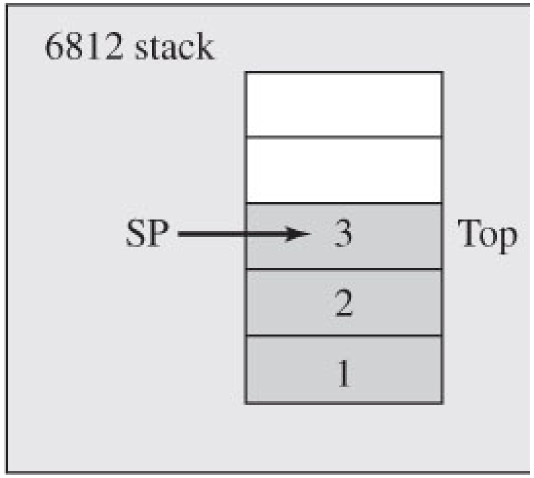
Example:

PULA

PULB

PULB

Start:



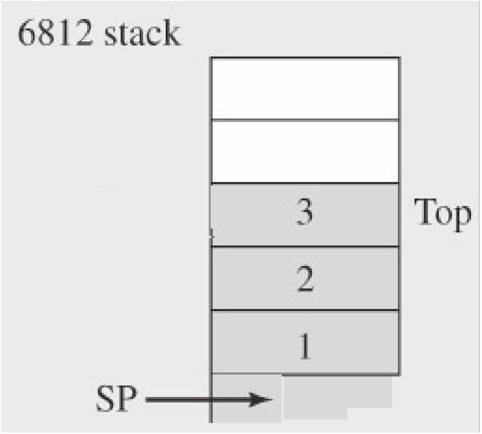
[Valvano]

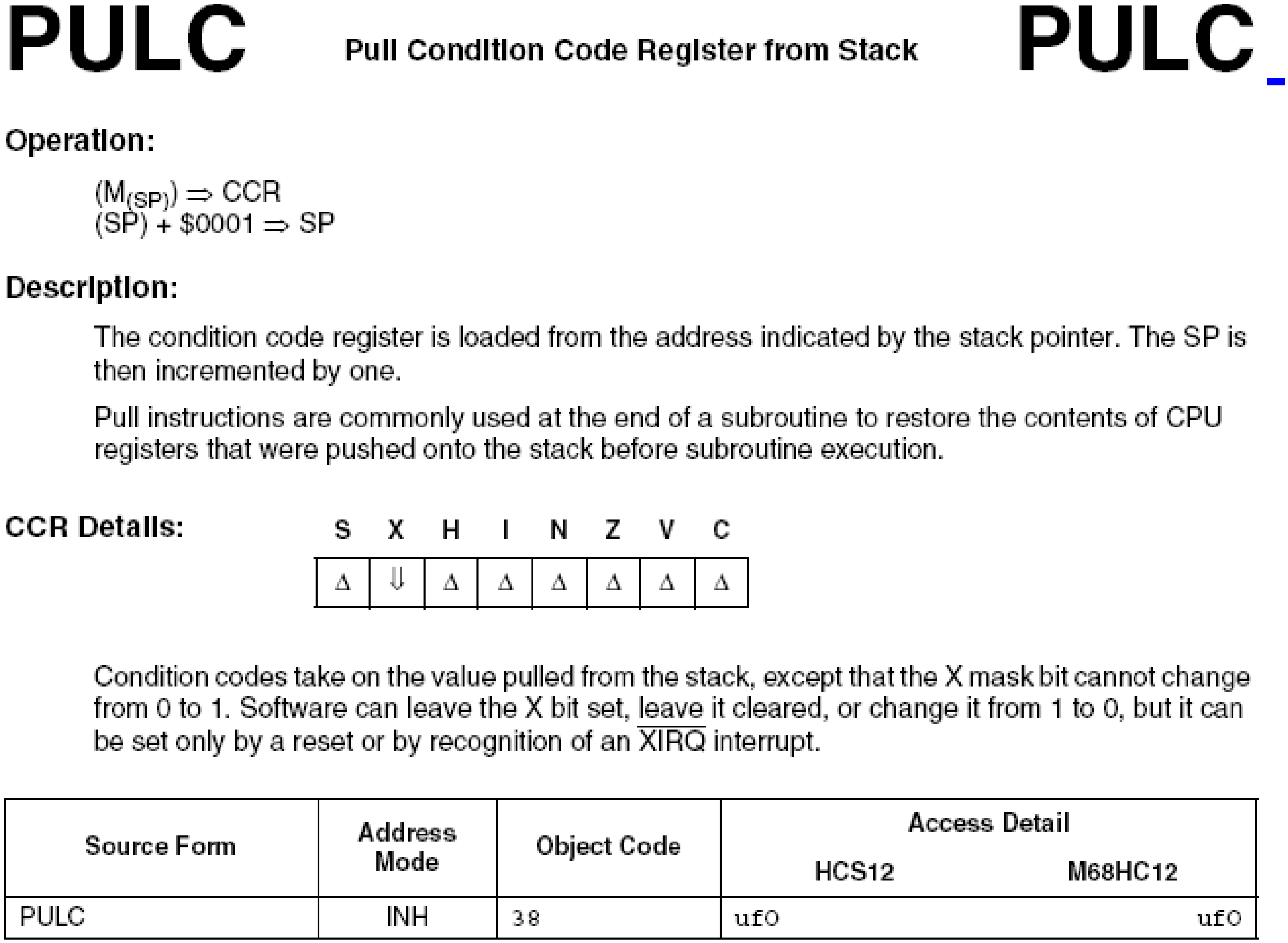
At end, A=3, B=1

- PUL doesn't erase memory values BUT unsafe to access them after PUL due to interrupts!!!

(On HC12, SP always points to top-most element in use)

End:





PULC Pull Condition Code Register from Stack PULC

Operation:

$(M_{(SP)}) \Rightarrow CCR$
$(SP) + \$0001 \Rightarrow SP$

Description:

The condition code register is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

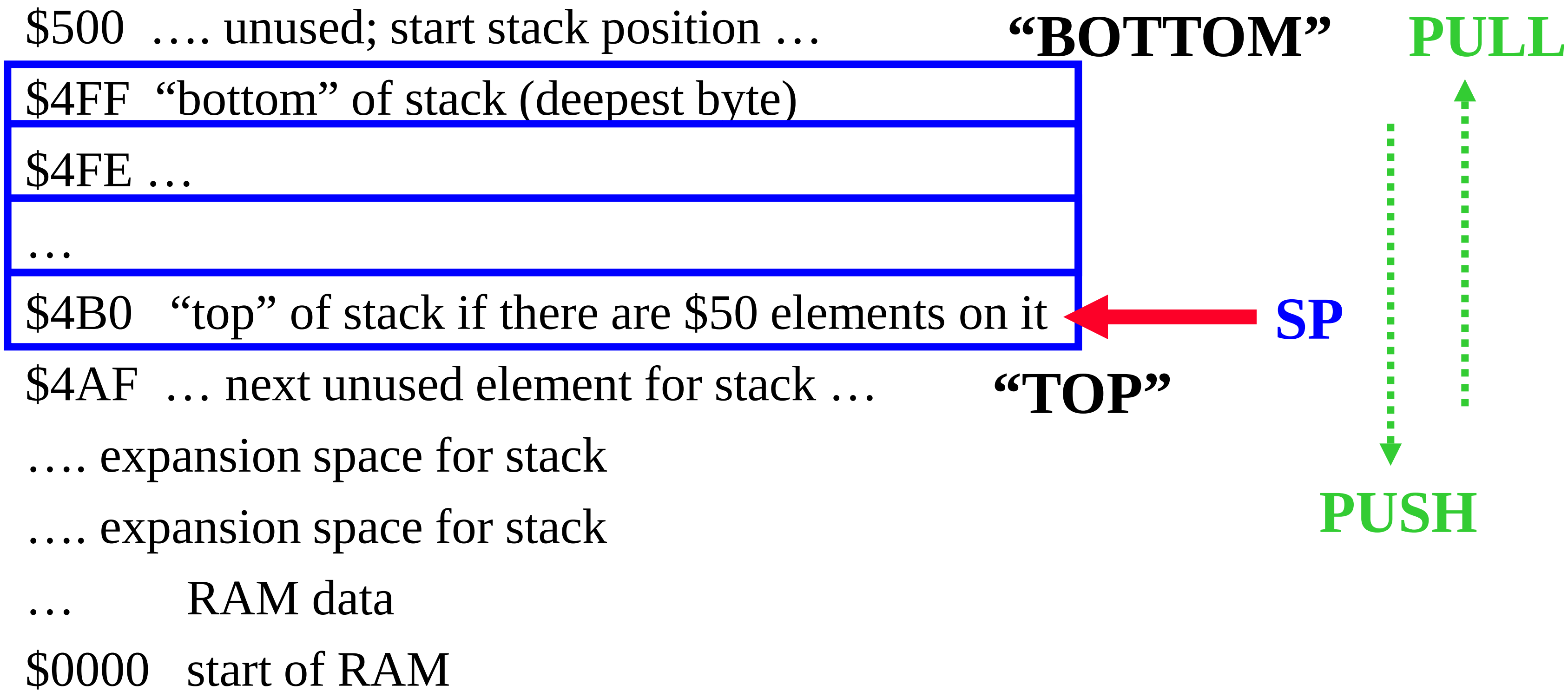
Condition codes take on the value pulled from the stack, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an $\overline{XIRQ}$ interrupt.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULC	INH	38	ufO	ufO

[Freescale]

Stack Implementation

- **Implementation of stack grows from top of memory down**



$500 …. unused; start stack position …

$4FF “bottom” of stack (deepest byte)

$4FE …

…

$4B0 “top” of stack if there are $50 elements on it

$4AF … next unused element for stack …

…. expansion space for stack

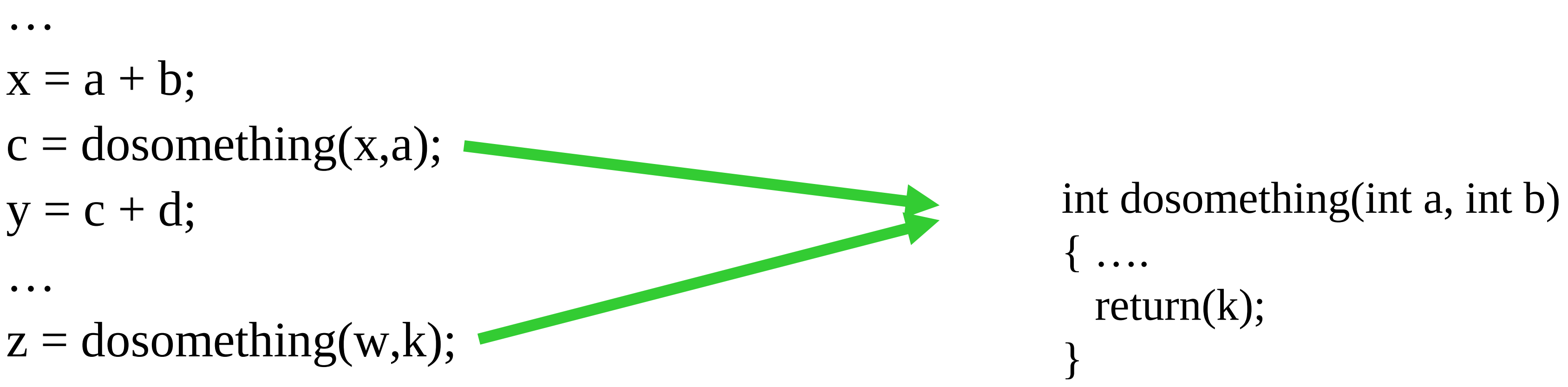
…. expansion space for stack

… RAM data

$0000 start of RAM

Hardware Support For Subroutines

- **Allows use of a procedure (or method in Object Oriented terminology)**



```
…
x = a + b;
c = dosomething(x,a);
y = c + d;
…
z = dosomething(w,k);
```

```
int dosomething(int a, int b)
{ ….
   return(k);
}
```

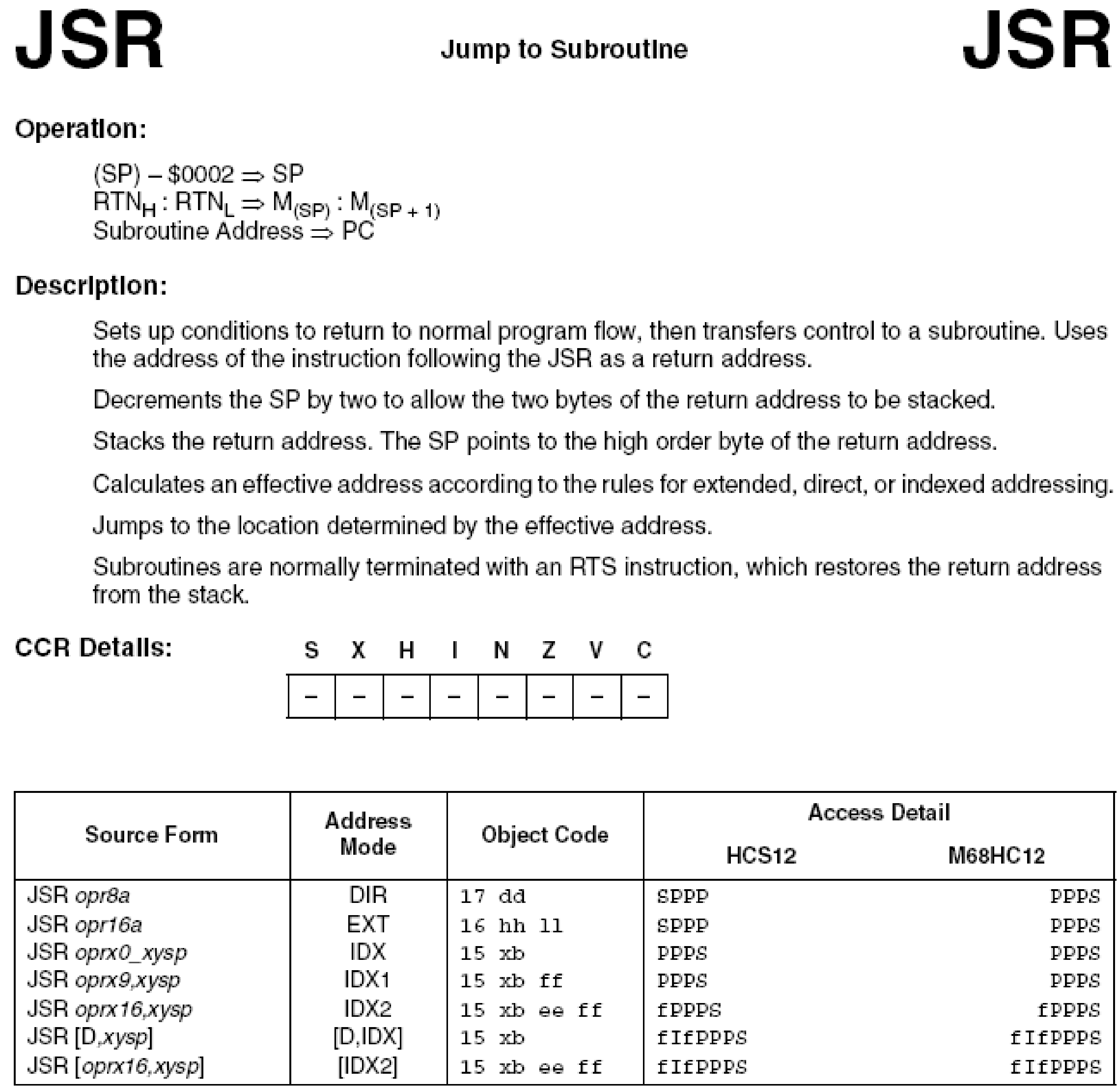
- **What has to happen to make this work?**
  - Prepare parameters for use
    - It isn’t always the same variables passed to the subroutine
  - Unconditional branch to subroutine
  - Execute subroutine
  - Prepare return value
    - The result doesn’t always go in the same output variable
  - Return to calling point to resume caller execution
    - How do we know where that is?

Subroutine Calls

- **Hardware support: JSR, BSR**
  - JSR – full, 16-bit address mode subroutine call
  - BSR – REL mode branch (8-bit PC-relative address), otherwise same as JSR
  - They pretty much do the same thing
    - BSR saves a byte of memory for instruction…
      … but still uses **2 bytes of stack space for return address**

- **JSR (and BSR) operations:**
  - PUSH current program counter onto stack (2-byte value)
  - Put address of subroutine into the PC
  - Start executing code at new PC value (the subroutine)

  - This takes care of saving return address and the actual jump
  - But, doesn't help with parameter values



JSR Jump to Subroutine JSR

Operation:

$(SP) - \$0002 \Rightarrow SP$
$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP + 1)}$
$\text{Subroutine Address} \Rightarrow PC$

Description:

Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction following the JSR as a return address.

Decrements the SP by two to allow the two bytes of the return address to be stacked.

Stacks the return address. The SP points to the high order byte of the return address.

Calculates an effective address according to the rules for extended, direct, or indexed addressing.

Jumps to the location determined by the effective address.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
JSR *opr8a*	DIR	`17 dd`	`SPPP`	`PPPS`
JSR *opr16a*	EXT	`16 hh ll`	`SPPP`	`PPPS`
JSR *oprx0_xysp*	IDX	`15 xb`	`PPPS`	`PPPS`
JSR *oprx9,xysp*	IDX1	`15 xb ff`	`PPPS`	`PPPS`
JSR *oprx16,xysp*	IDX2	`15 xb ee ff`	`fPPPS`	`fPPPS`
JSR [D,*xysp*]	[D,IDX]	`15 xb`	`fIfPPPS`	`fIfPPPS`
JSR [*oprx16,xysp*]	[IDX2]	`15 xb ee ff`	`fIfPPPS`	`fIfPPPS`

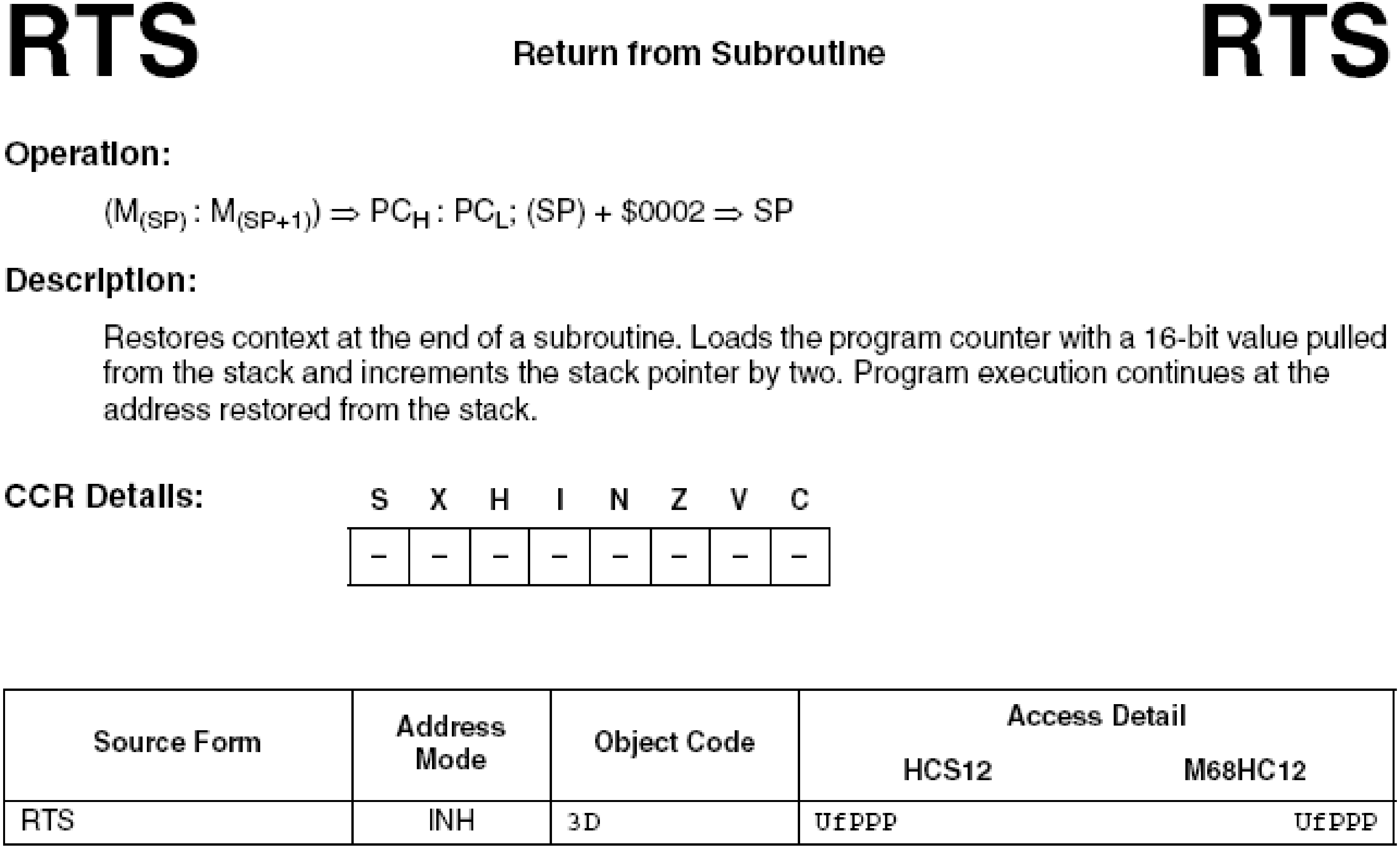
[Freescale]

Subroutine Returns

- **Hardware support: RTS**
  - RTS – INH address mode (how do you know the return address?)

- **RTS operations:**
  - POP top of stack and put it into PC
  - Start executing program at that new PC value (back to calling program)

  - This takes care of jumping back to calling program
  - But, doesn't help with parameter values



RTS Return from Subroutine RTS

Operation:

$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$

Description:

Restores context at the end of a subroutine. Loads the program counter with a 16-bit value pulled from the stack and increments the stack pointer by two. Program execution continues at the address restored from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RTS	INH	3D	UfPPP	UfPPP

[Freescale]

How Do You Pass Parameters?

Multiple methods, all of which can be useful

- **Put values in particular registers**
  - Example: sqrt(D) => D D register used as both input and output
  - Fast, but very limited by number of registers!
  - In the C language, it is very common to put the single return value in a register

- **Hard-code addresses into subroutine**
  - Easy to do
  - But, makes subroutine less flexible – need a different version for each data structure
  - Can make sense when you are just saving space by avoiding duplicated code

- **Pass parameters on stack**
  - Pass pointers to data structures
  - Pass values of variables
  - Flexible, usual method of passing parameters

- **(Note: we'll do stack frames and C variables in a later lecture …)**

Example: Passing By Registers

```
    LDAA    #47
    LDAB    #63
    BSR     COMPUTE_AVERAGE
    STAA    Average_result
    ….

; note – the below code only works on unsigned numbers!
COMPUTE_AVERAGE:
    ABA                 ;   sum to A, assume both are non-negative
                        ;   top bit of A contains carry-out of add
    LSRA                ;  divide by two for unsigned number sum
    RTS                 ; result is in register A
```

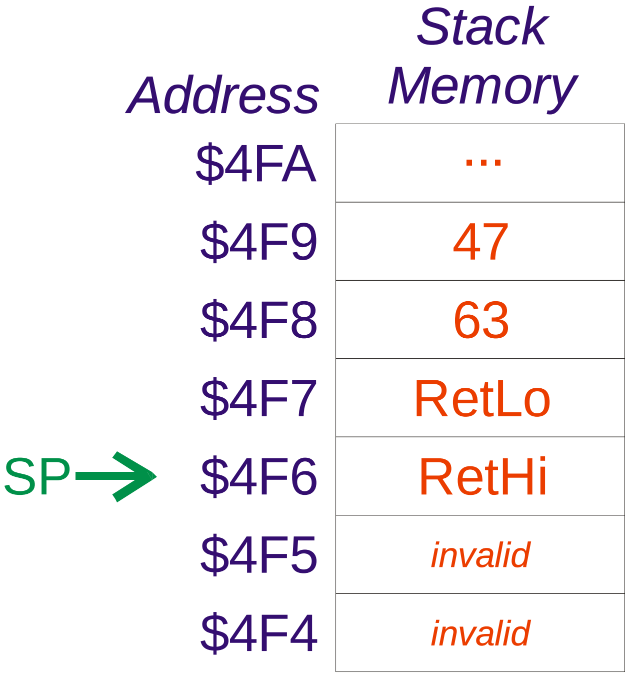
Passing Parameters To A Subroutine (simple version)

1. **PUSH parameters onto stack**
2. **CALL subroutine**
3. **Subroutine reads parameters from stack and does computations**
4. **RTS**
5. **Calling program deletes parameters from stack**
   - Why done here? (look at next slide to understand reason)

Example: Passing Via Stack (simple version)

```
;  Assume SP value is $4FA at this point
     LDAA          #47
     PSHA
     LDAA          #63
     PSHA
     JSR   COMPUTE_AVERAGE
     PULB          ;  discard second parameter  (could also use INS, but that is 2 bytes)
     PULB          ;  discard first parameter (could also use INS, but that is 2 bytes)
     STAA          Average_result
; SP is back to $4FA at this point

; only works on unsigned numbers!
COMPUTE_AVERAGE:
     LDAA   +2,SP  ; second parameter
     ADDA   +3,SP  ; first parameter
     LSRA          ;  divide by two for non-negative
     RTS           ; result is in register A
```



	Address	Stack Memory
	$4FA	...
	$4F9	47
	$4F8	63
	$4F7	RetLo
SP →	$4F6	RetHi
	$4F5	*invalid*
	$4F4	*invalid*

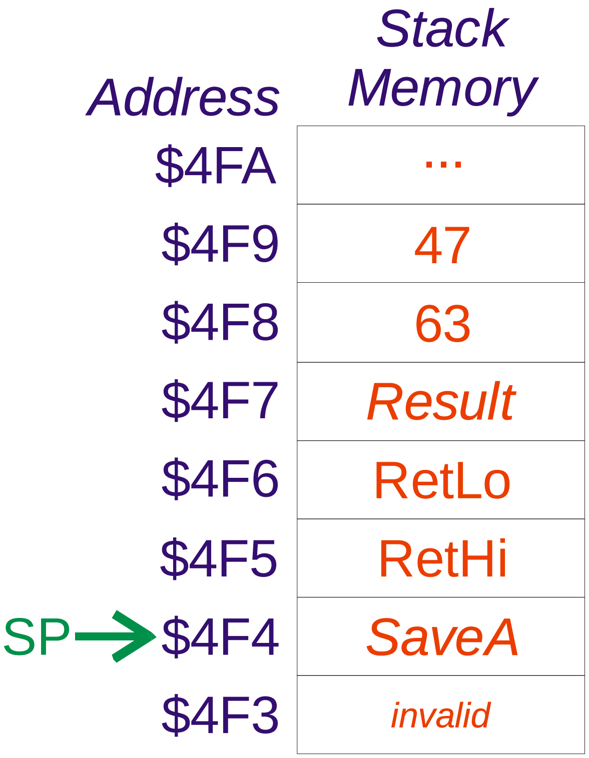
Passing Parameters To A Subroutine (complete)

1. **PUSH parameters onto stack**
2. **CALL subroutine**
3. **Save registers that are going to be modified by subroutine**
   - Avoids unexpected corruption of registers used by the calling program
4. **Subroutine reads parameters from stack and does computations**
5. **Subroutine writes results back to parameters on stack**
6. **Restore registers modified by subroutine**
7. **RTS**
8. **Calling program PULLs parameters from stack**

Passing Via Stack Example (complete version)

```
;  Assume SP value is $4FA at this point
    LDAA            #47
    PSHA
    LDAA            #63
    PSHA
    PSHA ; dummy push to make room for result; could also use DES
    JSR   COMPUTE_AVERAGE
    PULA ;  result stored in third parameter
    STAA  Average_result
    PULB ;  discard second parameter
    PULB ;  discard first parameter
; SP is back to $4FA at this point


; only works on non-negative numbers!
COMPUTE_AVERAGE:
    PSHA            ; make sure A isn’t trashed
    LDAA   +4,SP    ; second parameter
    ADDA   +5,SP    ; first parameter
    LSRA            ;  divide by two for non-negative number sum
    STAA   +3,SP    ; store result in third parameter position
    PULA            ; restore register A
    RTS             ; result is on stack
```



	Address	*Stack Memory*
	$4FA	…
	$4F9	47
	$4F8	63
	$4F7	*Result*
	$4F6	RetLo
	$4F5	RetHi
SP →	$4F4	*SaveA*
	$4F3	*invalid*

Rules For Safe Stack Use

- **PULL as many times as you PUSH**
  - Stack overflow will trash RAM
  - Stack underflow will give invalid PULL values
    - Very often it will also trash RAM
  - Mismatched number results in invalid subroutine return address

- **Don’t access stack memory after that value has been PULLed**
  - Interrupts can change the memory values at random times
    - We’ll talk about interrupts later in course
  - The program will still work *most* of the time – very nasty bug to track down

- **Beware of “stack smashing” attacks**
  - Frequent security vulnerability is someone intentionally over-running data structure to modify return address

Bad Code in a 3rd Party Library

```c
char * getProductName(void)
{
    char productName[128];
    char *cp = productName;

    readNameFromEEPROM(cp, 127);

    return (cp);
}
```

We're returning a pointer to an object on the stack. This is unsafe, yet may appear to work until an ISR runs on the same stack.

Assembler Pseudo-Ops

- **Not everything in a program is “executable code”**
  - By end of this lecture, you should know what everything below is doing…

```
ROMStart    EQU  $C000  ; absolute address to place my code/constant data
RAMStart    EQU  $0     ; absolute address to place my variable data
RAMEnd      EQU  $03FF  ; absolute address of last usable RAM byte

; variable/data section

            ORG RAMStart
 ; Insert here your data definition.
Average_result    DS.B 1

; code section
            ORG   ROMStart
Entry:
            LDS   #RAMEnd+1       ; initialize the stack pointer
            CLI                   ; enable interrupts

   LDAA  #$47
    PSHA
    LDAA #$63
    PSHA
    JSR  COMPUTE_AVERAGE
…
    ORG $FFFE
    DC.W Entry     ; Reset Vector
```

Labels

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

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

- The following all do EXACTLY the same thing:
  - JMP $4713
  - JMP Mylabelb
  - JMP Mylabelc
  - JMP Mylabela+1
- And it is valid to say: LDDA Mylabelb   *(what does this do?)*

ORG ; DS ; DC

- **DS – define storage space, but don’t initialize (RAM usually)**
  - (“Define Space”)

```
DS.B      1     ; allocate 1 byte of storage
DS.W      1     ; allocate one word (2 bytes)
DS.B      370   ; 370 more bytes
DS.W      100   ; 200 more bytes
```

- **DC – define storage space, and initialize with a value (ROM only)**
  - (“Define Constant”)

```
DC.B      13    ; one byte, with value $0D
DC.W      13    ; two bytes, with value $000D
DC.B      370   ; illegal – 8-bit value > 255
```

- **ORG – start laying down bytes at this address (ROM or RAM)**
  - (“Origin”)

```
   ORG $3000
;  next instruction, DS, DC,... is at address $3000
```

EQU

- **EQU is “equate” – means give this label a certain value**
  - This is a “compiler directive” – done at compile time, not at run time!
  - No bytes are deposited in memory!
  - Format: Label EQU Value

```
Foo  EQU  $C000
Bar   EQU   Foo
      LDAA  Foo          ;  same as  LDAA $C000
      LDAA  Bar          ;  same as  LDAA $C000
```

Labels vs. ORG vs .EQU

```
     ORG   $5000
Foo  EQU   $C000
     DS.W  $17
Baz  DC.W  $19
     ORG   Foo
     DS.W  $53
     ORG   Foo+$1000
     DC.B  $54
     DC.W  $5657
```

- **Questions:**
  - What is the address of Baz?
  - What address does the DS.W $53 start at? What value is stored there?
  - What value is at address $D001   (high byte stored first)

  - Note: don’t intermingle DS and DC in real programs – this is just an illustration
    - DS is for RAM; DC is for ROM/Flash memory in our hardware

Does It Create Bytes?

- **These DO NOT create bytes of data in memory**
  - Label – creates a value for use by the assembler, no run-time effect
  - EQU – creates a value for use by the assembler, no run-time effect
    - (Really, it's just a more general way to create a "label" value)
  - ORG – directs where the next byte goes
  - DS.B; DS.W – allocates storage space, but doesn't put in any values

- **These DO create bytes of data in memory**
  - Instructions – these put opcode etc. for instructions in ROM
  - DC.B, DC.W – these store a "constant" value (pre-initialized variable, etc.) in ROM

Now Do We Know What All This Means?

```
ROMStart    EQU  $C000  ; absolute address to place my code/constant data
RAMStart    EQU  $0     ; absolute address to place my variable data
RAMEnd      EQU  $03FF  ; absolute address of last usable RAM byte

; variable/data section

            ORG RAMStart
 ; Insert here your data definition.
Average_result    DS.B 1

; code section
            ORG   ROMStart
Entry:
            LDS   #RAMEnd+1       ; initialize the stack pointer
            CLI                   ; enable interrupts

   LDAA  #$47
    PSHA
    LDAA #$63
    PSHA
    JSR  COMPUTE_AVERAGE
…
    ORG $FFFE
    DC.W Entry    ; Reset Vector
```

Other Info

- **http://www.ece.utep.edu/courses/web3376/Directives.html**
  - (“essential code warrior syntax” for assembly)

- **Codewarrior documentation is available on the course web site**
  - http://www.ece.cmu.edu/~ece348/reading/index.html

  - Assembler manual
  - C compiler manual
  - Build tools manual
  - Debugger manual

Bad Code in a Telematics Application

```
void version_send(void)
{
    char * my_ver = “Version X.Y.Z”;
    my_ver[8]  = ‘0’ + (major_version & 0x07);
    my_ver[10] = ‘0’ + (minor_version & 0x07);
    my_ver[12] = ‘0’ + (revision_code & 0x07);
    output_version(my_ver);
}
```

String literal is ROMable

Hint: This function worked fine during development, when it was consistently executed out of RAM.

Rebuilt for flash download, the program bus faulted.

Cycle Counting for Branches

- **Some instructions have variable execution times – especially branches**

- **Branch timing cases:**
  - Branch not taken
    - Just continues along as if it were a no-op
  - Branch taken
    - Must refill instruction prefetch queue to get back to normal operation
    - (Remember, small microcontrollers don’t have cache memory, don’t speculate, etc.)

BEQ Branch if Equal BEQ

Operation:

If Z = 1, then (PC) + $0002 + Rel ⇒ PC

Simple branch

Description:

Tests the Z status bit and branches if Z = 1.

See 3.8 Relative Addressing Mode for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

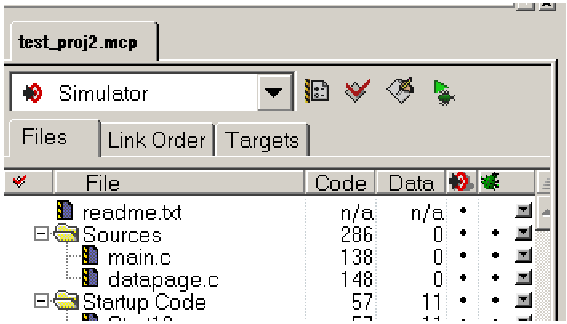
Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BEQ *rel8*	REL	27 rr	PPP/P[1]	PPP/P[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

[Freescale]

Lab Cycle-Accurate Simulator

- **The IDE simulator provides several useful capabilities**
  - Run code without a module
  - Easy single-step debugging
  - Count CPU cycles

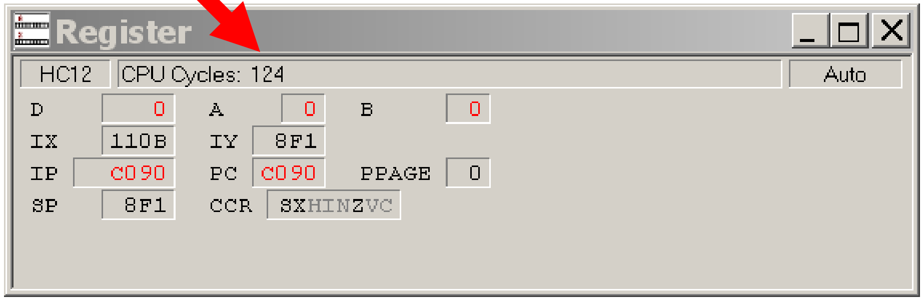


test_proj2.mcp

Simulator

File	Code	Data
readme.txt	n/a	n/a
Sources	286	0
main.c	138	0
datapage.c	148	0
Startup Code	57	11

- **Generally a simulator is the only way to get exact CPU cycle counts**
  - But only if the simulator is actually accurate (a problem on complex CPUs)
  - And only if the actual run-time environment matches the simulation



HC12 | CPU Cycles: 124 | Auto

```
D      0     A      0     B       0
IX  110B     IY   8F1
IP  C090     PC  C090     PPAGE   0
SP   8F1     CCR SXHINZVC
```

[Code Warrior Screenshots]

Simulator vs. Real Hardware vs. Hand Counting

- **Motivation: need to know execution time for real time scheduling**

- **Hand counting**
  - Doesn’t require fancy tools
  - Is tedious
  - Is OK for “best case” but often humanly impossible for worst case
    - in critical systems worst case is the important case!
- **Simulator**
  - If you have a simulator (luckily we do!) counting isn’t so bad
    - Single-step through program and subtract start count from end count
    - Can use “break points” (covered in debugging lecture)
- **Real hardware**
  - Can use hardware timers to assist (covered in counter/timer lecture)
  - Can use a stopwatch if timing is repeatable
  - Other approaches (covered in debugging lecture)
  - Issue: hard to get really precise and accurate times

NOP Timing Loops

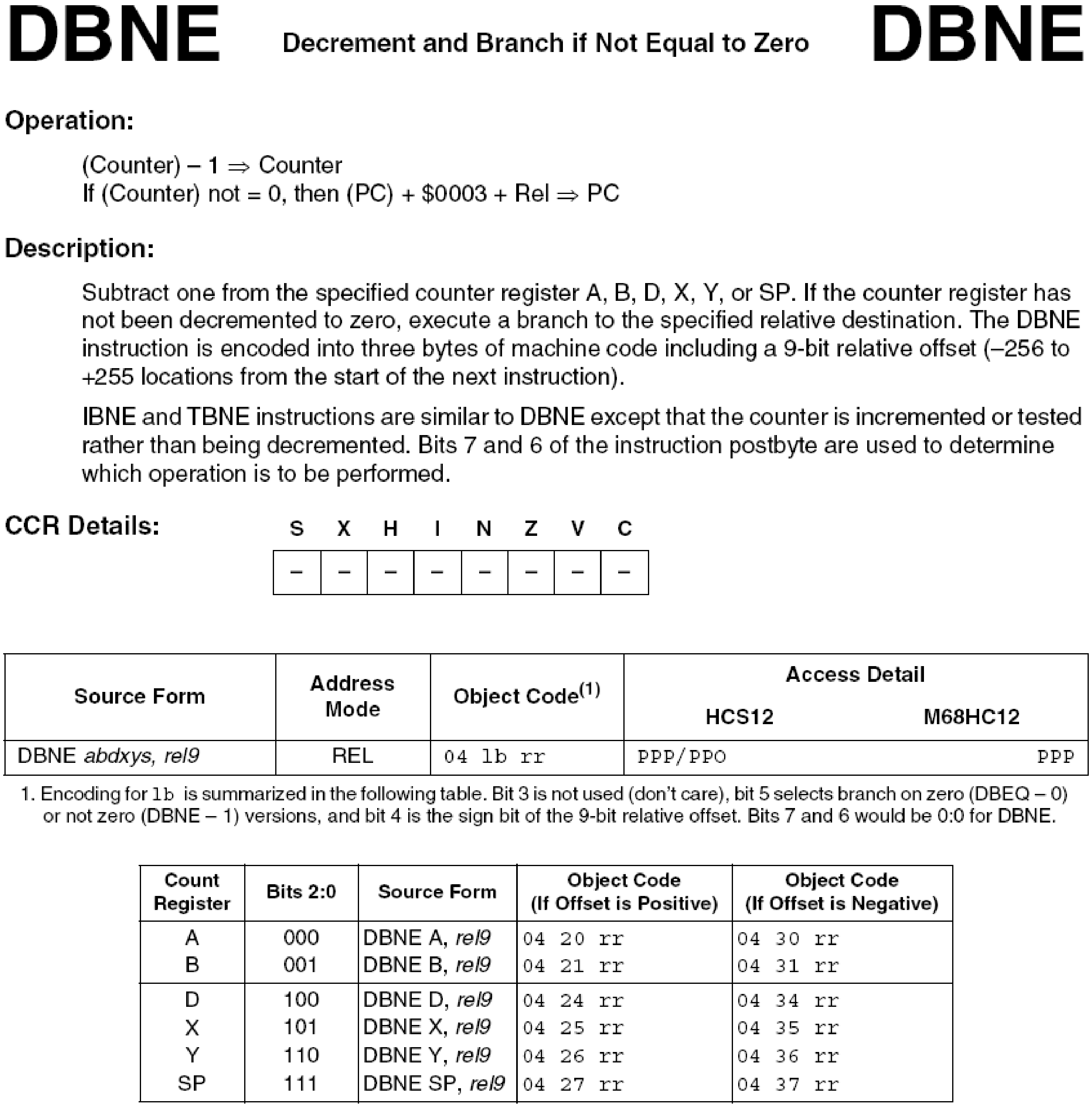
What if your CPU doesn’t even have a timer?

What if you need just a few microseconds of delay?

- **Sometimes (and frequently in old systems) => NOP Timing Loops**

```
      LDAA #$FF
Start_loop:
      NOP
      NOP
      NOP
      DBNE  A,Start_loop   ; Cool loop instruction
```

- **Number of NOP instructions and index values used to tune time**
  - Very commonly used in 1980s era embedded systems
    - BUT – really a problem if you have cache memory, interrupts, etc. etc.
    - What happens if you start using a new chip that is faster/different timing?
  - We still use it in this class until you know more advanced techniques
    - BUT – **dangerous to use in production systems unless you are really sure it is OK!**



DBNE Decrement and Branch if Not Equal to Zero DBNE

Operation:

(Counter) – 1 ⇒ Counter
If (Counter) not = 0, then (PC) + $0003 + Rel ⇒ PC

Description:

Subtract one from the specified counter register A, B, D, X, Y, or SP. If the counter register has not been decremented to zero, execute a branch to the specified relative destination. The DBNE instruction is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

IBNE and TBNE instructions are similar to DBNE except that the counter is incremented or tested rather than being decremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
DBNE *abdxys, rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

[1] Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (DBEQ – 0) or not zero (DBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 would be 0:0 for DBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	DBNE A, *rel9*	`04 20 rr`	`04 30 rr`
B	001	DBNE B, *rel9*	`04 21 rr`	`04 31 rr`
D	100	DBNE D, *rel9*	`04 24 rr`	`04 34 rr`
X	101	DBNE X, *rel9*	`04 25 rr`	`04 35 rr`
Y	110	DBNE Y, *rel9*	`04 26 rr`	`04 36 rr`
SP	111	DBNE SP, *rel9*	`04 27 rr`	`04 37 rr`

[Freescale]

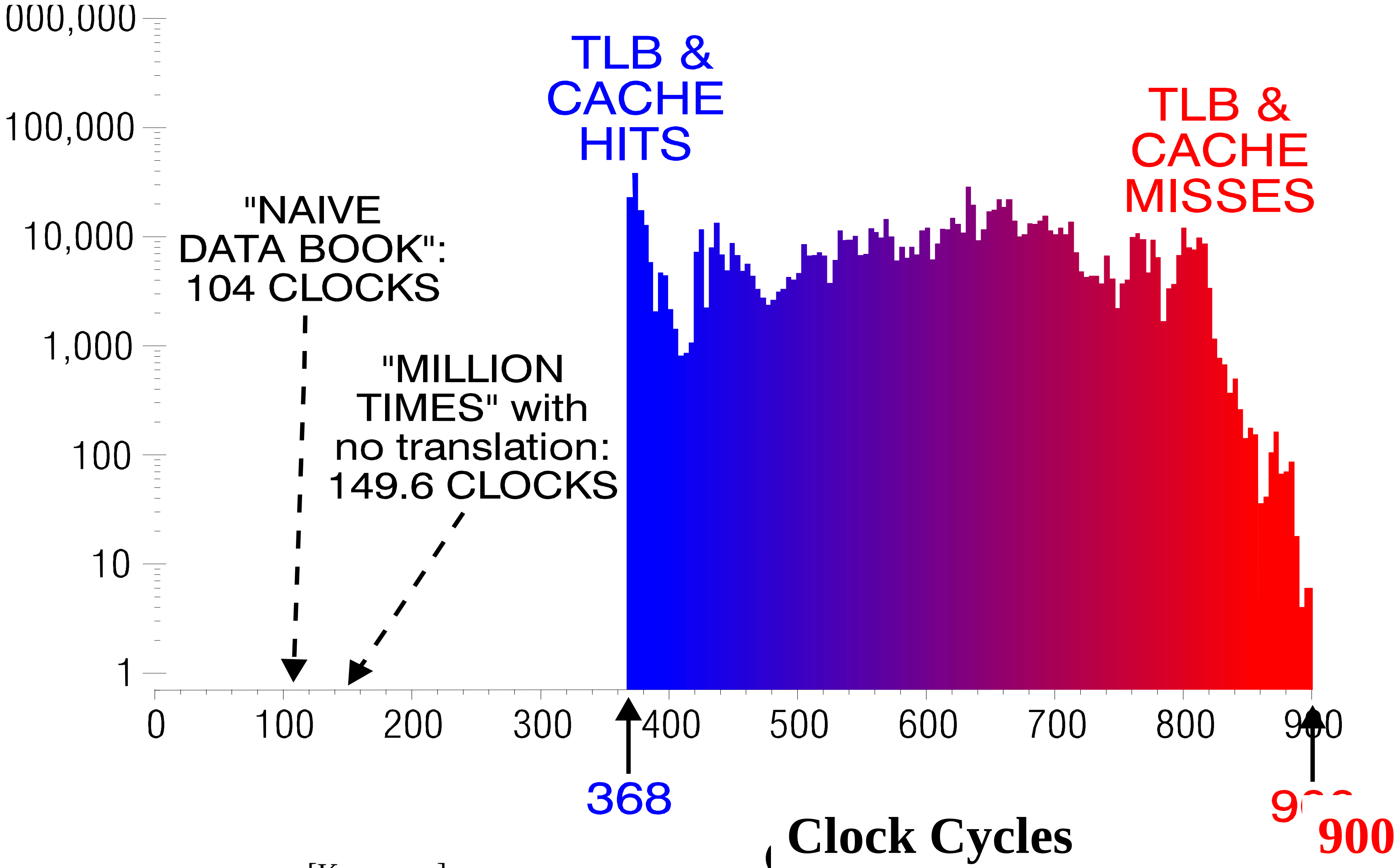
Advance Processors & Timing Prediction

- **Fancy CPUs and systems have practically unpredictable timing**
    - Speculative execution
    - Cache memory
    - Virtual memory
    - Variable timing on multiplication and division
    - DRAM refresh delays
    - System-level interrupts
    - Operating system latencies
    - …

- **Timing analysis for complex systems is a tough problem**
    - Something to NOT do – "run loop 1 million times and divide by 1 million"
    - Why?

Intel 80386 Timing For Simple Loop

- **Interrupt Service Routine that puts bytes into a queue**
  - "Memory sweeper" task running in foreground, including Virtual Memory



[Koopman]

Review

◆ Stack usage

- Pushing & popping with stack
- Subroutine calls
- Parameter passing to/from subroutines
- SP-relative loads and stores

◆ Other assembler operations

- Position, memory, and other management
- Labels
- Differences among label, EQU, DS, DC, ORG

◆ More on timing

- Cycle-accurate simulation
- Nop timing loop

Lab Skills

- **Register-based subroutine interface**
  - Write a program that uses registers to pass values
- **Stack-based subroutine interface**
  - Write a program that uses the stack to pass values
- **Timing**
  - Hand compute timing
  - Simulation-based timing
  - Stop-watch based timing