# 6800 Basics

By Ruben Gonzalez

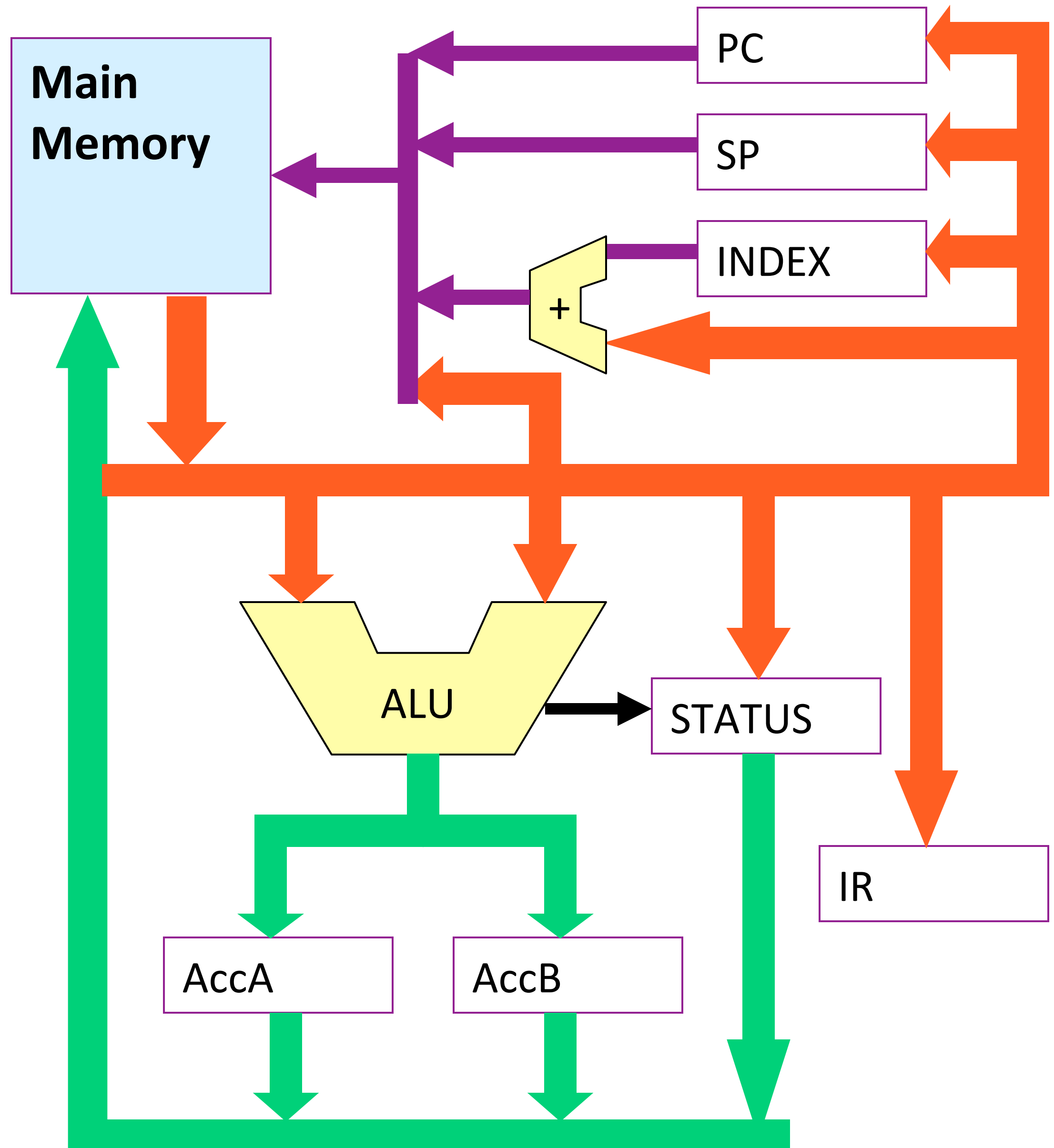
# 6800 Processor

- Uses 8 bit words
- Has addressable main memory of 64k
- Has Memory Mapped I/O and interrupts
- The 6800 has the following main registers:
  - 8-bit Accumulator A (AccA) register
  - 8-bit Accumulator B (AccB) register
  - 16-bit Index Register (X)
  - 16-bit Stack Pointer Register (SP)
  - 16-bit Program Counter (PC) points to next instruction
  - 8-bit Instruction Register (IR) holds instruction
  - 6-bit Status register holds status flags

© 2013 Ruben Gonzalez

# 6800 Processor

- Data paths
- Registers
- Address bus



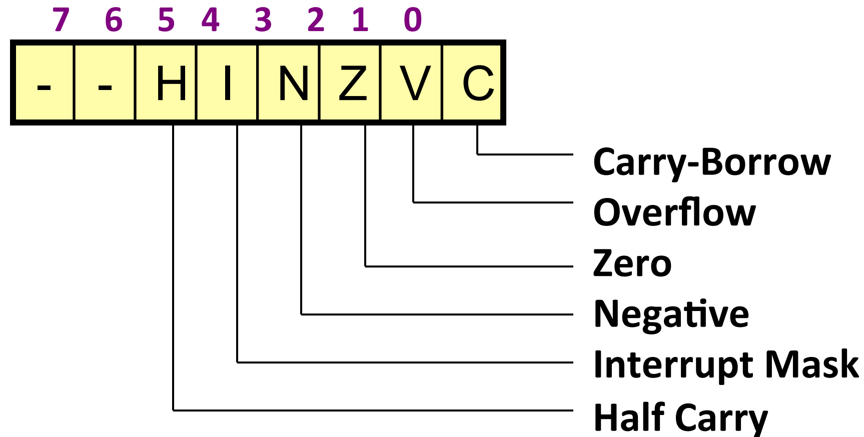

© 2013 Ruben Gonzalez

# 6800 Addressing Modes

- Inherent
  - No operand is provided – implied or not needed
  - Example: clear the value in the accumulator
- Immediate
  - Operand contains required data
- Relative
  - Operand is a value relative to current PC
  - Used for setting which instruction to execute from
- Direct
  - Operand contains 8 bit address of data
- Extended
  - Operand contains 16 bit address of data
- Indexed
  - Operand is an offset relative to the index register
  - Used for processing data arrays

© 2013 Ruben Gonzalez

# Status Register Ops [00-0F]




- no change
* updated
1 set
0 reset

| Mnemonic | Description | HINZVC |
|---|---|---|
| CLC | Clear Carry | -----0 |
| CLV | Clear Overflow | ----0- |
| CLI | Clear Interrupt Mask | -0---- |
| SEC | Set Carry | -----1 |
| SEV | Set Overflow | ----1- |
| SEI | Set Interrupt Mask | -1---- |
| TAP | Set Status Mask [Flags<=AccA] | ****** |
| TPA | Load Status Mask [AccA<=Flags] | ------ |

© 2013 Ruben Gonzalez

# Accumulator A&B Ops [10-1F]

- (Inherent) Operate on both Accumulators

| | Description | HINZVC |
|---|---|---|
| ABA | Add Accumulators [A <= A + B] | *-**** |
| CBA | Compare Accumulators [A - B] | --**** |
| SBA | Subtract Accumulators [A<=A-B] | --**** |
| TAB | Transfer Accumulator [B <= A] | --**0- |
| TBA | Transfer Accumulator [A <= B] | --**0- |

© 2013 Ruben Gonzalez

# Branch and Jump Ops [20-2F]

- Branch (Relative)

| | Description | | Description |
|---|---|---|---|
| BCS | Branch if Carry Set [C==1] | BRA | Branch Always |
| BCC | Branch if Carry Clear [C==0] | BSR | Branch to Subroutine |
| BMI | Branch if Minus [N==1] | BLT | Branch if < (signed) |
| BPL | Branch if Plus [N==0] | BLE | Branch if <= (signed) |
| BVS | Branch if Overflow Set [V==1] | BGE | Branch if >= (signed) |
| BVC | Branch if Overflow Clear[V=0] | BGT | Branch if > (signed) |
| BEQ | Branch if Equal [Z==1] | BLS | Branch if <= (unsignd) |
| BNE | Branch if Not Equal [Z==0] | BHI | Branch if > (unsgned) |

- Jump (Extended, Indexed)
  - JMP - Jump Absolute address
  - JSR - Jump to Subroutine, **Push PC**
  - RTS - Return from Subroutine, **Pop PC**

© 2013 Ruben Gonzalez

# Stack and Index Ops [30-3F]

- Modes: in**H**erent, **I**mmediate, **D**irect, **E**xtended, inde**X**ed

| | Description | Mode | HINZVC |
|---|---|---|---|
| CPX | Compare X | IDXE | ---*-- |
| DEX | Decrement X [ X <- X - 1] | H | ---*-- |
| INX | Increment X [ X <- X + 1] | H | ---*-- |
| LDX | Load X | IDXE | --**0- |
| STX | Store X | DXE | --**0- |
| TSX | Transfer SP,X [X <= SP + 1] | H | ------ |
| TXS | Transfer X,SP [SP <= X - 1] | H | ------ |
| DES | Decrement SP [SP <- SP - 1] | H | ------ |
| INS | Increment SP [SP <- SP + 1] | H | ------ |
| LDS | Load SP | IDXE | --**0- |
| STS | Store SP | DXE | --**0- |
| **PSHA, PSHB** | **Push Accumulator** | H | ------ |
| **PULA, PULB** | **Pull/Pop Accumulator** | H | ------ |

© 2013 Ruben Gonzalez

# Accumulator or Memory Only

- (Inherent) Operate on either AccA or AccB
- (Index or Extended) Operate on Memory

| | Description | HINZVC |
|---|---|---|
| ASL? | Arithmetic Shift Left Acc | --**** |
| ASR? | Arithmetic Shift Right Acc | --**** |
| LSR? | Logical Shift Right Acc | --0*** |
| ROL? | Rotate Left Acc | --**** |
| ROR? | Rotate Right Acc | --**** |
| CLR? | Clear Accumulator [?<=0] | --0100 |
| COM? | One's Complement | --**01 |
| DEC? | Decrement Acc [? <= ? - 1] | --***- |
| INC? | Increment Acc [? <= ? + 1] | --***- |
| NEG? | Negate Acc [? <= 0 - ?] | --**** |
| TST? | Test Acc [? <= ? - 0] | --**00 |

**Replace ? with A or B or nothing e.g.**

ASL or ASLA or ASLB

TST or TSTA or TSTB

© 2013 Ruben Gonzalez

# Accumulator and Memory Ops

- Immediate, Direct, Indexed and Extended
- **Replace ? With A or B (eg LDAA, LDAB)**

| | Description | Mode | HINZVC |
|---|---|---|---|
| LDA? | Load Accumulator [? <= M] | IDXE | --**0- |
| STA? | Store Accumulator [M <= ?] | DXE | --**0- |
| ADC? | Add with Carry [? <= ?+M+C] | IDXE | --**0- |
| ADD? | Add [? <= ?+M] | IDXE | --**0- |
| SBC? | Subtract w. Carry [? <= ?-M-C] | IDXE | --**0- |
| SUB? | Subtract [? <= ?-M] | IDXE | --**0- |
| BIT? | Bit Test [M <= ?] | IDXE | --**0- |
| AND? | Logical AND [? and M] | IDXE | --**0- |
| EOR? | Exclusive OR [? xor M] | IDXE | --**0- |
| ORA? | Inclusive OR [? or M] | IDXE | --**0- |
| CMP? | Compare Memory [? - M] | IDXE | --**** |

© 2013 Ruben Gonzalez

# Interrupts and I/O

- Reading and writing to memory mapped I/O devices is just like accessing any other memory location
- Problem: How do we know when an input device has new data to be read – Interrupts.
- An interrupt is a special processor function that pauses code execution, saves all of the registers onto the stack and jumps to a function called an interrupt service routine – when it finishes it restores all the registers and continues processing from where it left off. [not fully supported in the emulator]

| | **Description** | **HINZVC** |
|---|---|---|
| NOP | No Operation | ------ |
| RTI | Return from Interrupt [**not supported*] | ****** |
| SWI | Software Interrupt [**not supported*] | -1---- |
| WAI | Wait for Interrupt - pause processing | -1---- |

© 2013 Ruben Gonzalez

# Machine Language

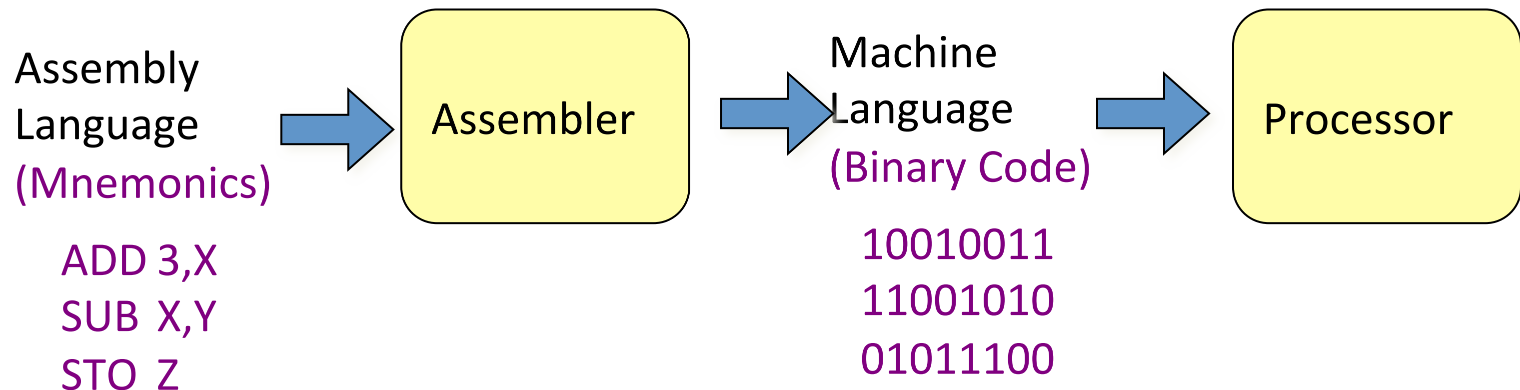
- The set of instruction codes used to control a processor is called an **instruction set**.
- Each instruction needs to be represented by a different codeword/bit pattern (e.g. 001 for add).
- The set of binary instruction codes that make up a computer program is called **machine language.**
- Programming in machine code by hand is difficult
  - Hypothetical 4bit Example: Look at code to add one to one

| Memory Address | Hex Content of Memory | Instruction | Binary Content of Memory |
| --- | --- | --- | --- |
| 000 | 0x81 | Store 1 in Accumulator | 0001 0001 |
| 001 | 0x24 | Add with value at address 4 | 0010 0100 |
| 002 | 0x45 | Store result at address 5 | 0100 0101 |
| 003 | 0xE0 | Clear Accumulator | 1110 0000 |

© 2013 Ruben Gonzalez

# Assembly Language

- Assembly Language was invented to make programming easier by using simple 3 letter Mnemonics to represent instructions.
- These are easy for humans to write & understand but impossible for processors to understand.
- Assemblers translate instructions into the machine language by looking up the mnemonics and replacing with the corresponding binary value




© 2013 Ruben Gonzalez

# Assemblers

- Assemblers create an ***object program*** *file* from mnemonic *source code* in two passes.
- During the first pass, the assembler assembles as much of the program is it can, while it builds a *symbol table* that contains memory references for all symbols in the program.
- During the second pass, the instructions are completed using the values from the symbol table.

© 2013 Ruben Gonzalez

# 6800 Assembly Language

- 6800 Assembly statements contain the following fields: [Label] Operation [operand] [comment]
  - **Label** – (*optional*) "names" that you give to memory locations
  - **Operation** - Defines the opcode or directive. (Not case sensitive)
  - **Operand** - Contains an address or the data for the instruction. (*Ignored with inherent operations*)
  - **Comment** – (*optional*) can be at start of line or end of operand must follow a semicolon
- Examples

```
; This is a comment
label adda #3      ; Add value 3 to AccA
      jmp label    ; Jump to address 'label'
      .end         ; end of program
```

© 2013 Ruben Gonzalez

# Assembler Directives

- Directives tells the assembler how/where to put instructions and data into memory.
- The following directives are supported

| Tag | Description | Example |
|---|---|---|
| .org | Where to put code in memory | .org $200 |
| .equ | Define Constant | label .equ 100 |
| .setw | Preset memory value | .setw $FFFE,10 |
| .rmb | Reserve Memory (# bytes) | .rmb 16 |
| .byte | Create and initialise byte variable<br>Array of bytes | .byte 64<br>.byte 1,2,3 |
| .word | Create and initialise word variable<br>Array of words | .word 5000<br>.word 1,2,3 |
| .str | Create character string | .str "text" |
| .end | End of Program | .end |

© 2013 Ruben Gonzalez

# Labels

- A label can be placed on any memory position.
- Labelling a memory position allows you to access whatever is stored in that memory location.
- Referring to a label refers to the memory position of that label.
- Example

```
loop nop  ; name the location where this instruction
          ; will be stored "loop"
var  .byte 14    ; name the location of this byte in memory "var"
     inc var     ; increment the value named "var"
     jsr loop    ; set PC to location of "loop"
```

© 2013 Ruben Gonzalez

# 6800 Simulator

- Includes an assembler and an emulator with built-in debugging support such as user breakpoints, execution trace, internal register display and a Hex/Bin/Dec number converter.

Assembly Program

```
001 ;==============================
002 ; Sum 1 to 10
003 ;==============================
004 ten      .equ 10
005 sum10    clra           ;A = 0
006          ldab #ten      ;B = 10
007 loop1    aba            ;A = A + B
008          decb           ;B--
009          tstb           ;test (B==0)
010          bne  loop1     ;repeat if B!=
011          staa sum       ;sum = A
012 ;==============================
013 ; Sum Array values, zero terminate
014 ;==============================
015 sumarry ldx array       ;X = array
016          clra           ;A = 0
017          clrb           ;B = 0
018 loop2    adda 0,x       ;A += array[0]
019          inx            ;next item
```

```
Assembling program...
Syntax Check: OK
```

Clear | Load | Save

Memory | Display | Reference    ADDRESS: 0002

```
0000: 4F 00 C6 0A 1B 00 5A 00 5D 00 26 F8 B7 00 B6 FE
0010: 00 B7 4F 00 5F 00 AB 00 08 00 6D 00 26 F8 B7 00
0020: B6 86 05 16 00 5A 00 5D 00 27 1D 37 00 BD 00 3A
0030: 33 00 5D 00 26 EF 7E 00 48 00 97 39 4F 00 9B 39
0040: 5A 00 5D 00 26 F8 39 00 B7 00 B6 86 05 16 00 5A
0050: 00 5D 00 27 24 37 00 B7 00 69 F7 00 6A BD 00 6B
0060: 33 00 5D 00 26 E9 7E 00 79 00 00 4F 00 D6 6A 9B
0070: 69 5A 00 5D 00 26 F8 39 00 B7 00 B6 86 05 16 00
0080: 5A 00 5D 00 27 2D 37 00 30 00 09 00 36 00 37 00
0090: BD 00 A1 33 00 33 00 33 00 5D 00 26 E3 7E 00 B3
00A0: 00 A6 00 E6 01 97 A0 4F 00 9B A0 5A 00 5D 00 26
00B0: F8 39 00 B7 00 B6 00 03 05 07 00 00 00 00 00 00
00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Program Line #

- Break Disabled
- Program Line #
- PC Address =
- SP Address =
- X Register =
- Accumulator A =
- Accumulator B =
- Memory Changed

Step | Run

PC 0002  X 0000
SP F000  IR 004F
CLRA

ACCUMULATOR
A 00  B 00

Status Flags

| H | I | N | Z | V | C |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |

Base Converter

| 7 | 8 | 9 | F |
|---|---|---|---|
| 4 | 5 | 6 | E |
| 1 | 2 | 3 | D |
| 0 | A | B | C |

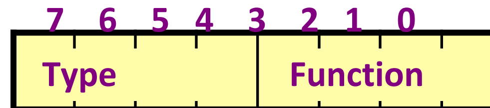
Hex | Dec | Bin

© 2013 Ruben Gonzalez

# 6800 Instruction Format

- Instruction Formats (8-bit Opcode)
  - Inherent: [Opcode] (data is implied)
  - Indexed, Relative: [Opcode] [Operand] (8-bit offset from X or PC)
  - Immediate, Direct: [Opcode] [Operand] (8-bit data or address)
  - Extended [Opcode] [16 bit Operand] (16-bit address of data)

- Opcode Format:




***Roughly**

- 00-0F: (INH) status reg ops
- 10-1F: (INH) Acc A&B ops
- 20-2F: (REL) Branch & Jump
- 30-3F: (INH) Stack & Index
- 40-4F: (INH) AccA only
- 50-5F: (INH) AccB only
- 60-6F: (IDX) memory only
- 70-7F: (EXT) memory only

- 80-8F: (IMM) AccA + Memory
- 90-9F: (DIR) AccA + Memory
- A0-AF: (IDX) AccA + Memory
- B0-BF: (EXT) AccA + Memory
- C0-CF: (IMM) AccB + Memory
- D0-DF: (DIR) AccB + Memory
- E0-EF: (IDX) AccB + Memory
- F0-FF: (EXT) AccB + Memory

© 2013 Ruben Gonzalez

# 6800 Assembly Programming

- When 6800 starts execution, the program counter will always start with the value 0. So always put the first instruction at location 0.
- Furthermore, at the start of execution, all registers (except PC) and unspecified memory locations are in an unknown state. Do not assume they have the value zero.
- Normally declare all constants (.equ) at the start of program and variables at the end (.byte)

© 2013 Ruben Gonzalez

# Assembly Language Syntax

- Number Formats:

| Prefix | Description | Example |
|---|---|---|
| | Decimal | 320 |
| $ | Hexadecimal | $240 |
| % | Binary | %110101 |
| ’ | Single character | ‘m |

- Operand Formats:

| Syntax | Format | Examples |
|---|---|---|
| | Inherent | CLRA |
| #<data> | Immediate | LDDA #4 |
| <data> | Relative | BRA 10 |
| <data> | Direct | LDDA 4 |
| <data> | Extended | LDDA label |
| <data>,X | Indexed | LDDA 4,X |

© 2013 Ruben Gonzalez

# Load Example

- A simple 6800 code fragment to load the value 5 to the accumulator A.

```
    ldaa #5   ; load accumulator A immediate
    .end          ; end of code directive
```

- Alternatively

```
five  .equu  5   ; define const with value of 5
    ldaa #five    ; load the value at label five
    .end          ; end of code directive
```

© 2013 Ruben Gonzalez

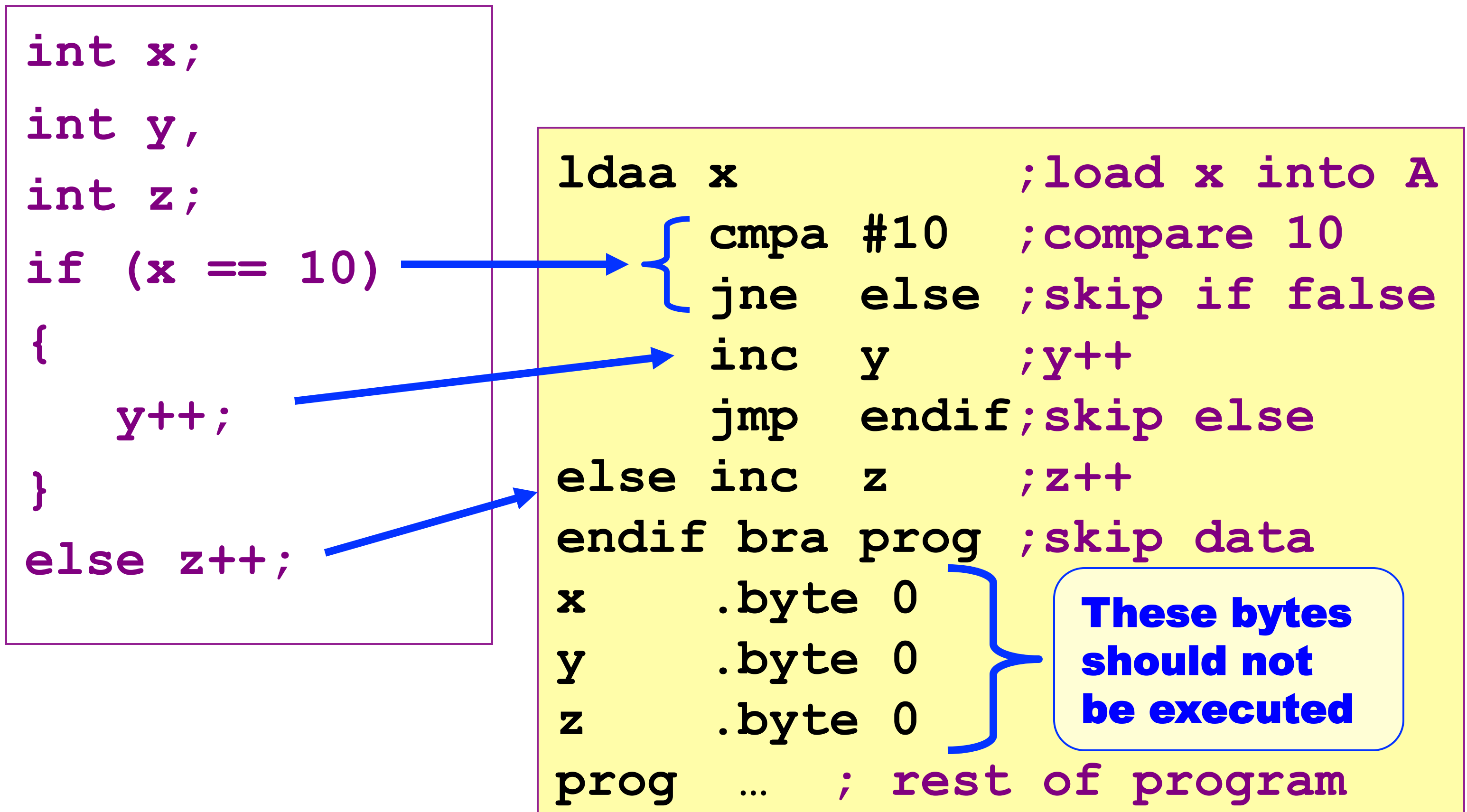
# Addition & subtraction Examples

```
;*** Result =  5 + 3 ***
;*** direct addressing ***
three .equ  3
five  .equ  5
       ldaa #five
       adda #three
       staa result
result.byte 0
       .end
```

```
;*** Result = 5 – 3 ***
;*** immediate address
;
;
       ldaa  #5
       suba  #3
       staa result
result .byte 0
       .end
```

© 2013 Ruben Gonzalez

# If … else … example




- and TST instructions only affect the status flags and are used to control branching/jumps

© 2013 Ruben Gonzalez

# Allocating Data

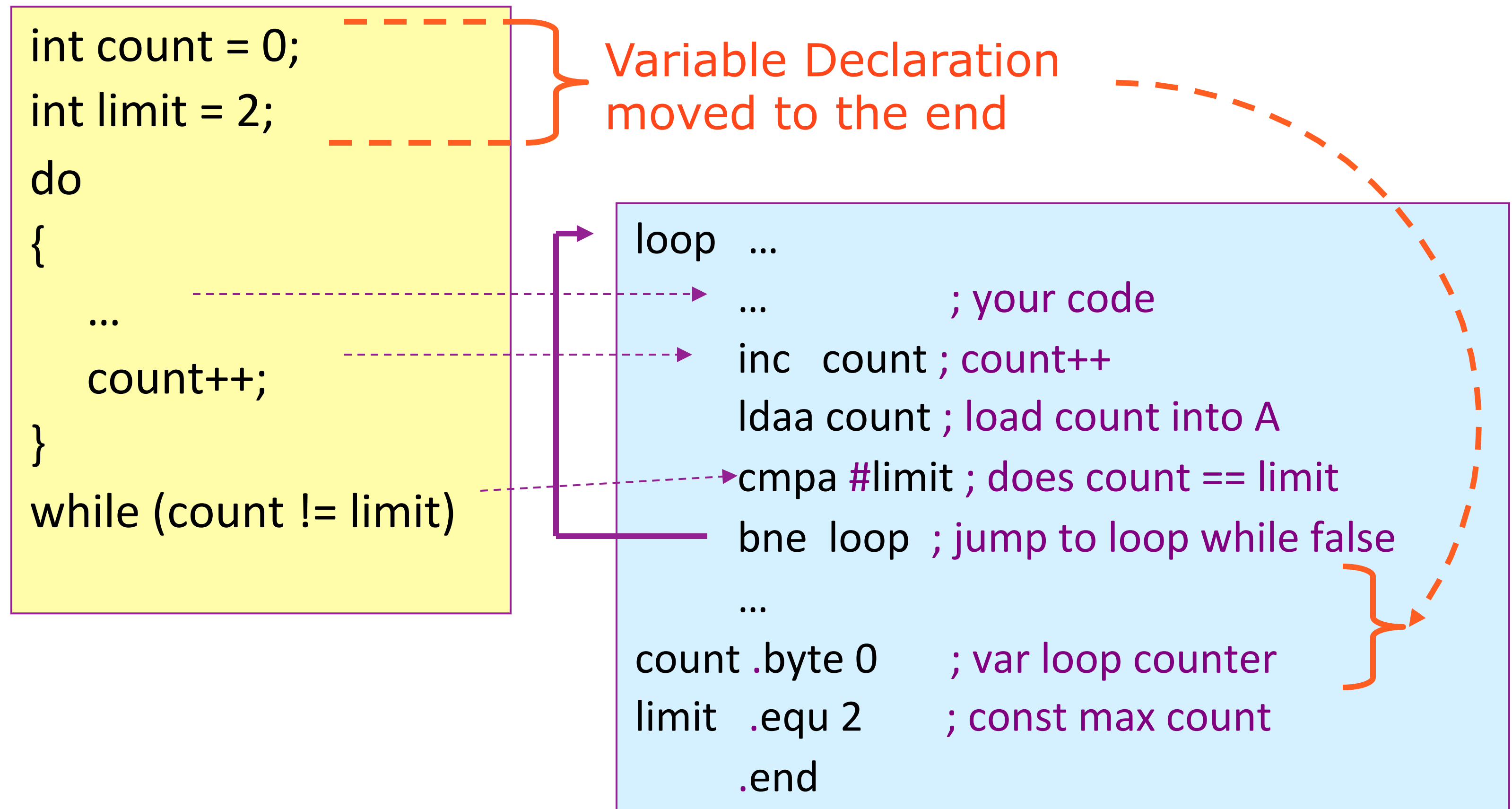
- Be careful not to allocate data at the start of a program before the code as it will be interpreted as instructions and executed
- Data can be allocated at the end of the program or after a BRA instruction so that you can jump over it and continue execution

| Addr | Unsafe | Addr | Safe | Safe |
|---|---|---|---|---|
| 0x00 | A .byte 3 | - | A .equ 3 | - |
| 0x01 | ldaa A | 0x00 | ldaa A | ldaa A |
| 0x02 | nop | 0x01 | bra 1 | ldaa B |
| 0x03 | B .byte 0 | 0x02 | B . byte 0 | A .byte 3 |
| 0x04 | ldab B | 0x03 | ldaa B | B .byte 0 |

© 2013 Ruben Gonzalez

# Program Loops in Assembly

- How to construct a loop?




- **z and TST instructions only affect the status flags and are used to control branching/jumps**

© 2013 Ruben Gonzalez

# Loop Execution Example

| Instruction | Comment | PC | Zero Flag | count | limit |
|---|---|---|---|---|---|
| loop … | First instruction of loop | 0 | x | 0 | 2 |
| … | Instructions in loop body | … | x | 0 | 2 |
| inc count | Increment count | Z | 1 | 1 | 2 |
| cmpa limit | Does count == limit? | z+2 | 0 | 1 | 2 |
| bne loop | Jump to loop if false | 0 | 0 | 1 | 2 |
| 2nd Pass | | | | | |
| loop … | First instruction of loop | 0 | x | 1 | 2 |
| … | Instructions in loop body | … | x | 1 | 2 |
| inc count | Increment count | z | 1 | 2 | 2 |
| cmpa limit | Does count == limit? | z+2 | 1 | 2 | 2 |
| bne loop | Jump to loop if false | z+4 | 1 | 2 | 2 |
| … | Continue Execution | z+6 | x | 2 | 2 |

© 2013 Ruben Gonzalez

# For Loop Operation

- Consider the following high level loop:

```
for (count =0; count <10; count ++)
{
    // do something
}
```

- This is the same as the do-while loop:

```
loop,  …
       …            ;do something
       inc   count  ;count++
         ldaa  count ;load count into A
       cmpa limit   ;does count == limit
       bne  loop   ;jump to loop while false
       …
count .byte 0       ;loop counter
limit   .byte 10     ;max count
       .end
```

© 2013 Ruben Gonzalez

# Simple Loop Example

- Sum the numbers from 1 to 10

```
ten       .equ 10
sum10     clra        ;A = 0
          ldab #ten   ;B = 10
loop1     aba         ;A = A + B
          decb        ;B--
          tstb        ;test (B==0)
          bne loop1        ;repeat if B!=0
          staa sum    ;sum = A
;--------- variables ----------
sum       .rmb 1           ;same as .byte
          .end
```

© 2013 Ruben Gonzalez

# Using Directives

- Normally declare constants (.equ) at start of program and variables (.byte etc) at the end
- You can assign other labels to any constant but no labels can be assigned to variables

```
a   .equ 8
b   .equ a    ; Ok b = 8
c   .byte a   ; Illegal
```

```
a   .byte 8
b   .equ a    ; Ok b = a's addr
c   .byte a   ; Illegal
```

- Use to .org to locate where the next instruction will be placed in memory
- Declaring a constant with value '*' assigns it the current memory location

```
    .org $40
m   .equ *
    ldaa #m
```

**AccA = $ 0**

- Using Direct addressing with a constant gives undefined results since constants contain no data

© 2013 Ruben Gonzalez

# Addressing

- 6800 Assembly provides many addressing modes with and without labels (for variables - not constants)

| Example | Mode | Description |
|---|---|---|
| ldaa #3 | Immediate | Load number 3 |
| ldaa 3 | Direct | Load content of memory location 3 |
| ldaa #label | Immediate | Load memory address of Label |
| ldaa label | Direct | Load content of memory at label |
| ldaa ,x | Indexed | Load content of memory at 0+index |
| ldaa 3,x | Indexed | Load content of memory at 3+index |
| ldaa label,x | Indexed | Load content at label + index |

- Indexed Addressing
  - Can't use # symbol (no mixing indexed and immediate)
  - Using a constant label (.equ) is correct
  - Using a variable label adds the address of the variable to the index

© 2013 Ruben Gonzalez

# Addressing Examples

Index = 0

| Addr | Memory | Assembly | AccA |
|---|---|---|---|
| 0000 | 86 04 | ldaa #4 ;load num 4 | 4 |
| 0002 | 96 04 | ldaa 4 ;load Mem[4] | 86 |
| 0004 | 86 06 | ldaa #a ;load .equ 6 | 6 |
| 0006 | B6 00 06 | ldaa a ;load Mem[#a] | B6 |
| 0009 | 86 22 | ldaa #b ;load addr of b | 22 |
| 000B | B6 00 22 | ldaa b ;load Mem[#b] | 6 |
| 000E | 86 06 | ldaa #c ;load a .equ 6 | 6 |
| 0010 | B6 00 06 | ldaa c ;load Mem[#c] | B6 |
| 0013 | 86 22 | ldaa #d ;load addr of b | 22 |
| 0015 | B6 00 22 | ldaa d ;load Mem[#d] | 6 |
| 0018 | A6 03 | ldaa 3,x ;load Mem[3+0] | 4 |
| 001A | A6 06 | ldaa a,x ;load Mem[6+0] | B6 |
| 001C | A6 22 | ldaa b,x ;load Mem[22+0] | 6 |
| 001E | A6 06 | ldaa c,x ;load Mem[6+0] | B6 |
| 0020 | A6 22 | ldaa d,x ;load Mem[22+0] | 6 |
| – | – | a .equ 6 | |
| 0022 | 06 | b .byte 6 | |
| – | – | c .equ a ;a = 6 | |
| – | – | d .equ b ;addr of b!! | |

© 2013 Ruben Gonzalez

# Array Handling

- Use indexed addressing  and INX instruction
- Example: Sum Array of values, zero terminated

```
sumarry   ldx #array    ; X = array
          clra          ; A = 0
          clrb          ; B = 0
loop2     adda ,x       ; A += array[0]
          inx           ; next item
          tst  ,x       ; check item
          bne  loop2    ; repeat if B!=0
          staa sum      ; sum = A
sum       .byte   0
array     .byte   3,5,7,0
          .end
```

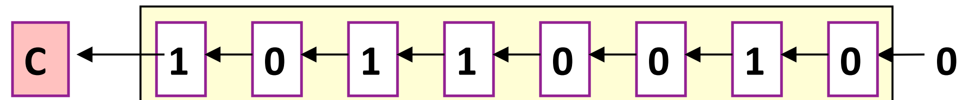
Pseudo Code

```
int sum = 0;
int i = 0;
while (a[i] != 0)
{
   sum = sum + a[i];
   i = i +1;
}
```

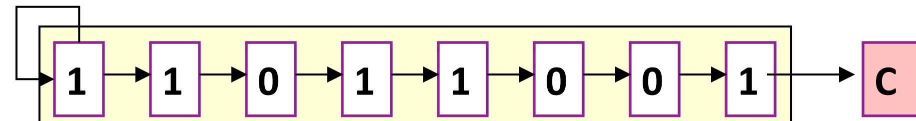
© 2013 Ruben Gonzalez

# Bit Manipulation Instructions

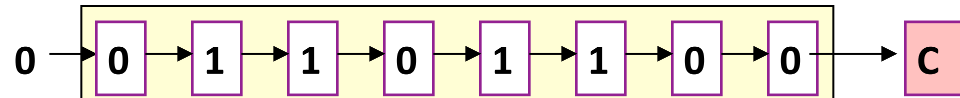
- Complement
  - LDDA 00110001
  - COMA => 11001110
- Rotate and shift (uses carry flag)
  - Arithmetic shift Left



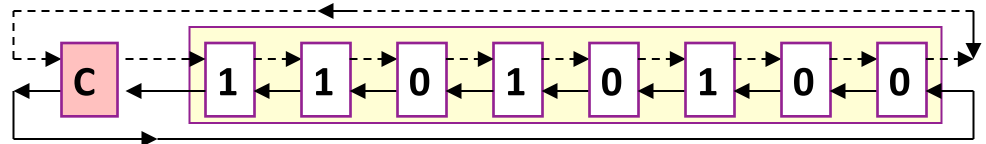

  - Arithmetic shift right (replicates the left bit)




  - Logical shift right




  - Rotate left / right




- Bit Test
  - LDDA 00110001
  - BITA 00100000 => true

© 2013 Ruben Gonzalez

# A Sample Program

- Count the number bits in a word which are 1.
  - E.g: $CD_{16}$ or $11001101_2$ has 5 Ones.
- Program
  - ;--- Program to count the number of 1s in a word
  - ;--- shift each bit into carry & increment count if carry=1
  - value .equ $CD ; the word to be counted
  - clc ; clear carry
  - ldaa value
  - loop asla ; shift left
  - bcc next ; jmp if not carry
  - inc count; ; else count++
  - next dec nbits ; nbits—
  - tst nbits ; is nbits == 0?
  - bne loop ; if false loop
  - count .byte 0 ; initial count
  - nbits .byte 8 ; how many bits to pro
  - .end

<u>Basic Algorithm</u>

```
load the word to ACC;
int count = 0;
for (n =wordsize; n >0; n--)
{
    shift ACC once to the left;
    if (carry == 1) count++;
}
```

© 2013 Ruben Gonzalez

# Sample program Execution

- **First pass**

| Instruction | Comment | PC | Flags Z, C | Acc | nbits | count |
|---|---|---|---|---|---|---|
| value .equ $CD | Constant to evaluate | - | | | | |
| clc | Clear carry flag | 2 | -, 0 | | 8 | 0 |
| ldaa value | Load word into Acc | 3 | 0, 0 | $CD | 8 | 0 |
| loop asla | Shift left into carry | 5 | 0, 1 | $9A | 8 | 0 |
| bcc next | Skip if carry = 0 | 6 | 0, 1 | $9A | 8 | 0 |
| inc count | Increment count | 8 | 0, 1 | $9A | 8 | 1 |
| next dec nbits | Decrement nbits | 10 | 0, 1 | $9A | 7 | 1 |
| tst nbits | Does nbits = 0? | 12 | 0, 0 | $9A | 7 | 1 |
| bne loop | If false jump to loop | 14 | 0, 0 | $9A | 7 | 1 |
| count .byte 0 | First byte in memory | 0 | | | | 0 |
| nbits .byte 8 | Second byte in memory | 1 | | | 8 | 0 |

Note we have lost Carry Flag

© 2013 Ruben Gonzalez

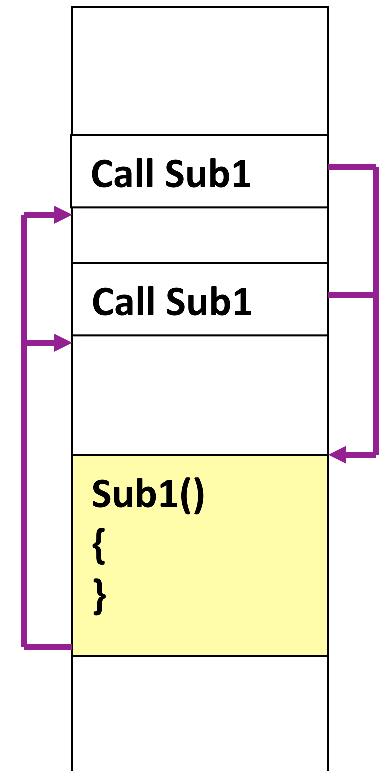
# Sample passes: 2nd & 3rd

| Instruction | Comment | PC | Flags Z, C | Acc | nbits | count |
|---|---|---|---|---|---|---|
| loop asla | Shift left into carry | 5 | 0, 1 | $34 | 7 | 1 |
| bcc next | Skip if carry = 0 | 6 | 0, 1 | $34 | 7 | 1 |
| inc count | Increment count | 8 | 0, 1 | $34 | 7 | 2 |
| next dec nbits | Decrement nbits | 10 | 0, 1 | $34 | 6 | 2 |
| tst nbits | Does nbits = 0? | 12 | 0, 0 | $34 | 6 | 2 |
| bne loop | If false jump to loop | 14 | 0, 0 | $34 | 6 | 2 |
| | | | | | | |
| loop asla | Shift left into carry | 5 | 0,0 | $68 | 6 | 2 |
| bcc next | Skip if carry = 0 | 6 | 1,0 | $68 | 6 | 2 |
| inc count | Increment count | 8 | 1,0 | $68 | 6 | 2 |
| next dec nbits | Decrement nbits | 10 | 1,0 | $68 | 5 | 2 |
| tst nbits | Does nbits = 0? | 12 | 0,0 | $68 | 5 | 2 |
| bne loop | If false jump to loop | 14 | 0,0 | $68 | 5 | 2 |
| | | | | | | |
| loop asla | …. Until nbits == 0 | | | | | |

© 2013 Ruben Gonzalez

# Subroutines

- There are two main issues when invoking subroutines
  - How is subroutine linkage achieved?
  - How is data passed in and out of a subroutine?
- Subroutine linkage requires:
  - control to be transferred to start of a subroutine
  - control to be transferred back at subroutine end
  - Since a subroutine can be called from more than one place the return code must be flexible enough to compute its return address.
- Parameters can be either passed:
  - by copying values to the accumulator
  - by copying values on to the stack
  - By using program variables (.byte etc)




© 2013 Ruben Gonzalez

# Calling Subroutines: JSR & BSR

- Call Subroutines using the JSR or BSR instruction:
  - JSR label or BSR label

JSR has the following functionality:

(stack) ← PC

This places the 16 bit return address on the stack

First the top byte, then the bottom byte

PC ← label

BSR has the following functionality:

(stack) ← PC

This places the 16 bit return address on the stack

First the top byte, then the bottom byte

PC ← PC + value

Change PC to point somewhere relative to its current location

© 2013 Ruben Gonzalez

# Subroutine Example

- The RTS instruction is used to return control to the calling function and restore the stack as it was

- Write a subroutine to decrement a value (in acc).

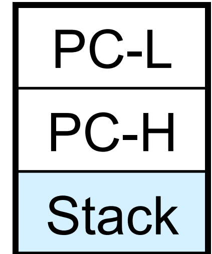
```
value .byte 10 ; value to be decremented
        ldaa value       ; load parameter to pass into acc
        jsr subr   ; jump to decrement subroutine
        staa  value

subr  deca                ; subroutine
        rts          ; return by restoring PC from stack
        .end
```

© 2013 Ruben Gonzalez

# Subroutine Example

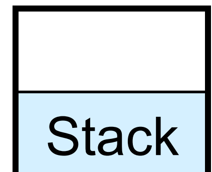
- The subroutine body executes the next instructions (DECA) normally
- Control is returned to the instruction following the JSR using the RTS instruction by popping the PC value from the stack

| Addr | value | .byte 10 |
| --- | --- | --- |
| 1 | | ldaa value |
| 3 | | jsr subr |
| 5 | | staa value |
| 7 | subr | deca |
| 8 | | rts |




PC = 8

| Addr | value | .byte 10 |
| --- | --- | --- |
| 1 | | ldaa value |
| 3 | | jsr subr |
| 5 | | staa value |
| 7 | subr | deca |
| 8 | | rts |




PC = 5

© 2013 Ruben Gonzalez

# Argument Passing in Registers

- Calculate Factorial(n) passing arguments in AccA & AccB
- Subroutine result returned in AccA

```
;--- Calculate Factorial(5) ---
main ldaa #5      ; A = 5
     tab    ; B = A
loop decb ; B = B - 1
     tstb   ; test (B==0)
     beq  fin     ; exit loop if B=0
     jsr  mult    ; call subroutine
     tstb   ; test (B==0)
     bne loop     ; repeat if B!=0
fin  staa sum     ; sum = A
```

```
;---- Subroutine Multiply(A*B) ----
var  rmb 1        ; variable A
mult pshb ; save B  ****
     staa var     ; var = A
     clra   ; A = 0
more adda var     ; A += var
     decb ; B--
     tstb   ; test (B==0)
     bne more; repeat if B!=0
     pulb ; restore B  ****
     rts    ; return
```

© 2013 Ruben Gonzalez

# Argument Passing in Variables

- Calculate Factorial(n) passing arguments as variables
- Subroutine result returned in AccA

```
;--- Calculate Factorial(5) ---
main ldaa #5      ; A = 5
     tab    ; B = A
loop decb ; B = B - 1
     tstb   ; test (B==0)
     beq  fin     ; exit loop if B=0
     staa var1    ; var1 = A
     stab var2    ; var2 = B
     jsr   mult   ; call subroutine
     tstb   ; test (B==0)
     bne loop     ; repeat if B!=0
fin  staa sum     ; sum = A
```

```
;---- Subroutine Multiply(A*B) ----
var1 rmb 1       ; variable A
var2 rmb 1       ; variable B
mult pshb ; save B
     clra   ; A = 0
     ldab var2  ;
moreadda var1; A += var
     decb ; B--
     tstb   ; test (B==0)
     bne more; repeat if B!=0
     pulb ; restore B
     rts    ; return
```

© 2013 Ruben Gonzalez

# Argument Passing on the Stack

- Calculate Factorial(n) passing arguments on stack
- Subroutine result returned in AccA

```
main ldaa #5    ; A = 5
     tab   ; B = A
loop decb ; B = B - 1
     tstb  ; test (B==0)
     beq fin    ; exit loop if B=0
     pshb ; save B
     tsx   ; X = SP
     dex
     pshb ; var B
     psha ; var A
     jsr  mult ; call subroutine
     pulb  ; clear stack
     pulb  ; clear stack
     pulb  ; restore B
     tstb  ; test (B==0)
     bne loop   ; repeat if B!=0
fin  staa sum   ; sum = A
```

```
;--- Subroutine Multiply(A*B) ---
var   rmb  1     ;variable A
mult3      ldaa 0,x    ;A = var1
      ldab 1,x    ;B = var2
      staa var    ;var = A
      clra  ;A = 0
more       adda var;A += var
      decb ;B--
      tstb  ;test (B==0)
      bne  more;repeat if B!=0
      rts   ;return
```

Need to use index register to get arguments from the stack since they are under the saved PC

© 2013 Ruben Gonzalez