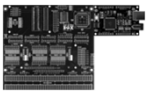
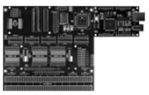


EEL 3744

Today's Menu

- Program File Structure
- Data Structures
- Program Structures
  - >Sequence, Selection, Repetition
- Transition from a "main" program to a "subroutine"
- Subroutine VADD

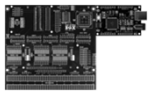
See examples on web:
`Stack1.asm,`
`VectAdd.asm,`
`doc8331, doc0856`



EEL 3744

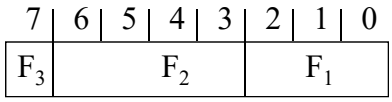
Data Structures

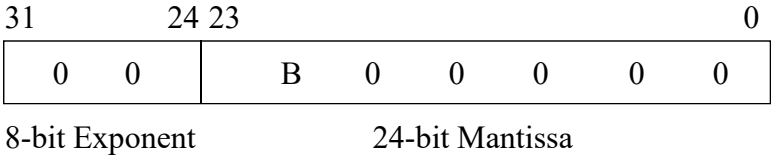
- Bit Fields
- Floating Point
- Sequential List
- Matrix
- Linked List
- Stack
- Queue

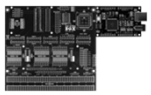


EEL 3744

Data Structures: Bit Fields & Floating Point





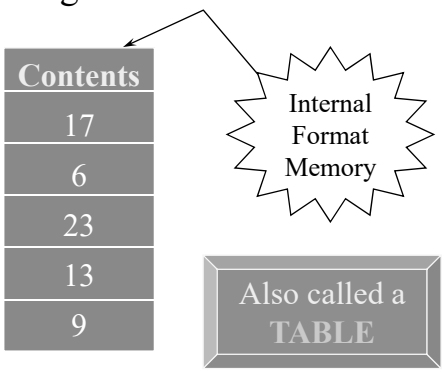


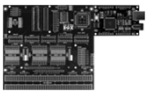
EEL 3744

Data Structures: Sequential List

- A ***sequential list*** of data is a collection of data mapped into successive locations in storage, starting at some initial location called the base address. The order of the elements may or may not have a particular significance.

Time	Experimental Data	Location
1	17	122
2	6	123
3	23	124
4	13	125
5	9	126





EEL 3744
Data Structures: Matrix

$$A = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$$

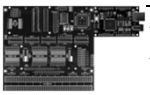
Memory

	Location	Contents
A →	322	4
	323	2
	324	1
	325	3

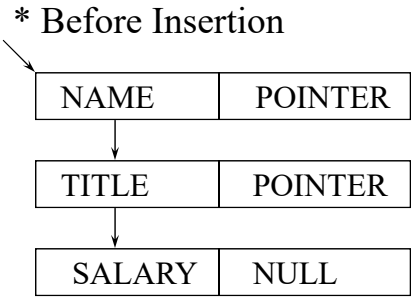
A is an m row by n column matrix

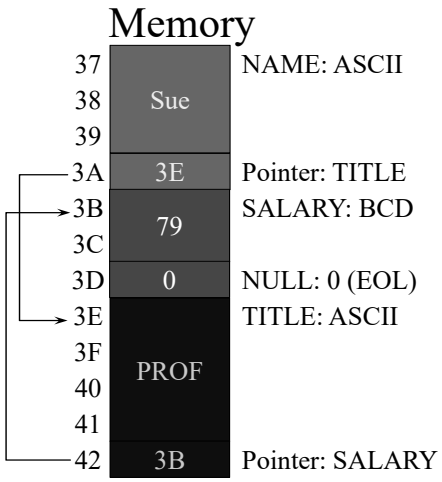
a_{ij} = element in row i column j

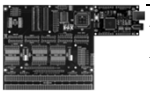
Location of a_{ij} = A+(i-1)*n+(j-1)



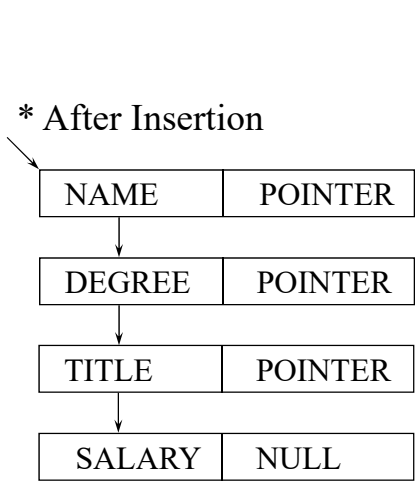
EEL 3744 Data Structures: Linked List

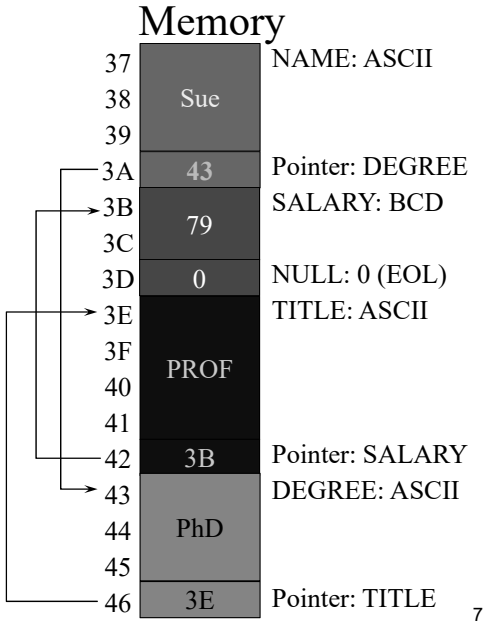


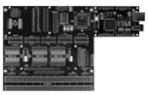




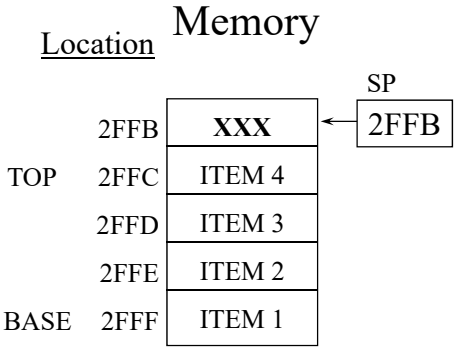
EEL 3744 Data Structures: Linked List

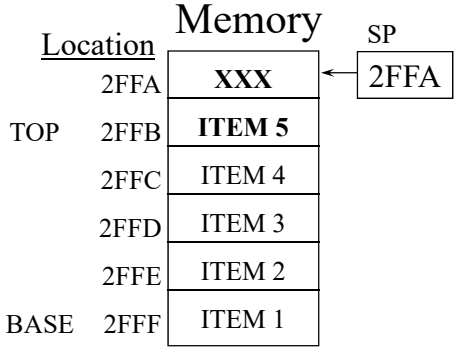


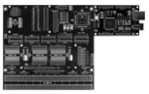




EEL 3744 Data Structures: Stack (**XMEGA** Format [Identical to 68HC11])







EEL 3744
Data Structures: Stack (**6812** Format)

* **Before** pushing a 5th item

Memory

Location		
	121	**XXX**
TOP	122	ITEM 4
	123	ITEM 3
	124	ITEM 2
BASE	125	ITEM 1

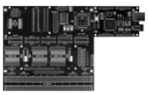
SP → 122

* **After** pushing a 5th item

Memory

Location		
	120	**XXX**
TOP	121	**ITEM 5**
	122	ITEM 4
	123	ITEM 3
	124	ITEM 2
BASE	125	ITEM 1

SP → 121



EEL 3744
Data Structures: Stack (**F2833x DSC** Format)

* **Before** pushing a 5th item

Memory

Location		
BASE	400	ITEM 1
	401	ITEM 2
	402	ITEM 3
TOP	403	ITEM 4
	404	**XXX**

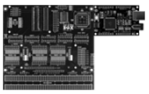
SP → 404

* **After** pushing a 5th item

Memory

Location		
BASE	400	ITEM 1
	401	ITEM 2
	402	ITEM 3
	403	ITEM 4
TOP	404	**ITEM 5**
	405	**XXX**

SP → 405



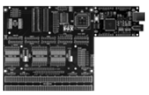
EEL 3744

doc0856

Stack PUSH/POP **XMEGA**

- Data is pushed and popped from the stack using PUSH (decreases SP) and POP (increases SP)

Instruction	Operands	Description	Operation	#Clocks
PUSH	Rr	Push Register on Stack	STACK ← Rr	2
POP	Rd	Pop Register from Stack	Rd ← STACK	2

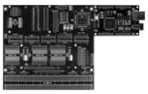


EEL 3744

doc8331
Section 3.8

Stack on the **XMEGA**

- Stack Pointer (SP) is two 8-bit registers to form a 16-bit register
  - > SPL is at 0x0D and SPH is at 0x0E (in code use, **CPU_SPL** and **CPU_SPH**)
- SP is automatically loaded after reset; initial (default) value is the highest address of the internal SRAM (0x3FFF for our chip)
- If SP is changed, must be set above address 0x2000 (the lowest address in internal SRAM) and defined **before** any subroutine calls are executed or **before** interrupts are enabled
- To prevent corruption, a write to SPL will disable interrupts for up to 4 instructions or until the next I/0 memory write
- Stack grows from a higher memory to a lower memory
  - > Pushing data on the stack decreases SP
    - – SP points to next empty memory location for data to be store with **push**
  - > Popping data from the stack increases SP
    - – SP is incremented before data is extracted with **pop**
  - > Stack data is at addresses higher that the stack pointer



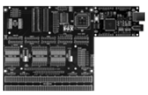
EEL 3744

Stack Initialization on **XMEGA**

- The stack pointer has only 16 bits and can only address the low 64k of data space (0 - 0xFFFF)
  - > After reset, SP points to address 0x3FFF, but do **NOT** assume this, i.e., **always initialize the stack!**

Example:

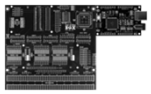
```
.EQU   STACK_ADDR = 0x3FFF

    ldi  R16, low(STACK_ADDR)
    out  CPU_SPL, R16          ;initialize low byte of stack pointer
    ldi  R16, high(STACK_ADDR)
    out  CPU_SPH, R16          ;initialize high byte of stack pointer
```

EEL 3744

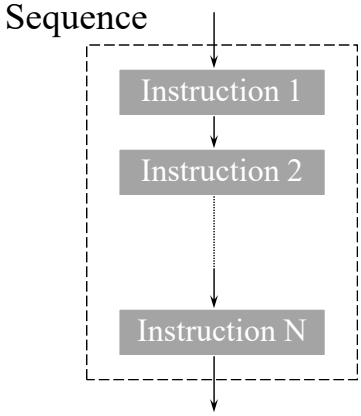
Program Structures and Structured Programming

- Do **not** use tricks to shorten code.
  - > Tricks will "*byte*" you later!
- Program Structures
  - > Sequence
  - > Selection (IF-THEN-ELSE)
  - > Repetition (FOR, WHILE, REPEAT-UNTIL)
  - > Main-Subroutine

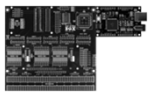


EEL 3744

Sequence




```
        *
        *
ldi     R16, 0xF        ;Instruction 1
sts     PORTQ_DIR, R16
                        ;Instruction 2
        *
        *
        *
add     XL,YL           ;Instruction N
        *
        *
```

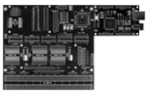



EEL 3744

doc0856

XMEGA Branch Instruction

- The syntax for a branch instruction is as follows:
  BRxxx Label
  - >Label is assemble as a 7-bit signed constant
    - – Values between 63 and -64
- PC calculations
  - >If (COND = true) PC = PC + 1+ signed 7-bit offset
  - >If (COND = false) PC = PC + 1
  - >Note: If (COND = true) then instruction takes 2 cycles.
  - >If (COND = false) then instruction takes 1 cycles.
- Note that there are signed and unsigned branch instructions (see page 10 in doc0856)



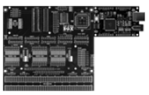
EE

XMEGA
Branch

Signed

doc8331
Section 35
---
doc0856
Page 12
(see also pg 10)

BRBS	Branch if Status Flag Set	if (SREG(s) = 1) then PC	←	PC + k + 1
BRBC	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC	←	PC + k + 1
BREQ	Branch if Equal	if (Z = 1) then PC	←	PC + k + 1
BRNE	Branch if Not Equal	if (Z = 0) then PC	←	PC + k + 1
BRCS	Branch if Carry Set	if (C = 1) then PC	←	PC + k + 1
BRCC	Branch if Carry Cleared	if (C = 0) then PC	←	PC + k + 1
BRSH	Branch if Same or Higher	if (C = 0) then PC	←	PC + k + 1
BRLO	Branch if Lower	if (C = 1) then PC	←	PC + k + 1
BRMI	Branch if Minus	if (N = 1) then PC	←	PC + k + 1
BRPL	Branch if Plus	if (N = 0) then PC	←	PC + k + 1
BRGE	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC	←	PC + k + 1
BRLT	Branch if Less Than, Signed	if (N ⊕ V= 1) then PC	←	PC + k + 1
BRHS	Branch if Half Carry Flag Set	if (H = 1) then PC	←	PC + k + 1
BRHC	Branch if Half Carry Flag Cleared	if (H = 0) then PC	←	PC + k + 1
BRTS	Branch if T Flag Set	if (T = 1) then PC	←	PC + k + 1
BRTC	Branch if T Flag Cleared	if (T = 0) then PC	←	PC + k + 1
BRVS	Branch if Overflow Flag is Set	if (V = 1) then PC	←	PC + k + 1
BRVC	Branch if Overflow Flag is Cleared	if (V = 0) then PC	←	PC + k + 1
BRIE	Branch if Interrupt Enabled	if (I = 1) then PC	←	PC + k + 1
BRID	Branch if Interrupt Disabled	if (I = 0) then PC	←	PC + k + 1



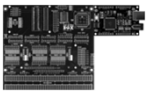
EEL 3744

XMEGA
Branch

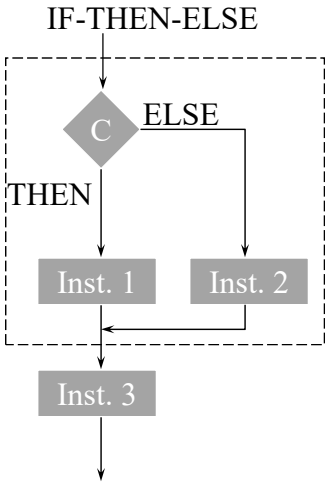
doc0856
Page 10

Test	Boolean	Mnemonic	Complementary	Boolean	Mnemonic	Comment
Rd > Rr	Z•(N ⊕ V) = 0	BRLT(1)	Rd ≤ Rr	Z+(N ⊕ V) = 1	BRGE*	Signed
Rd ≥ Rr	(N ⊕ V) = 0	BRGE	Rd < Rr	(N ⊕ V) = 1	BRLT	Signed
Rd = Rr	Z = 1	BREQ	Rd ≠ Rr	Z = 0	BRNE	Signed
Rd ≤ Rr	Z+(N ⊕ V) = 1	BRGE(1)	Rd > Rr	Z•(N ⊕ V) = 0	BRLT*	Signed
Rd < Rr	(N ⊕ V) = 1	BRLT	Rd ≥ Rr	(N ⊕ V) = 0	BRGE	Signed
Rd > Rr	C + Z = 0	BRLO(1)	Rd ≤ Rr	C + Z = 1	BRSH*	Unsigned
Rd □ Rr	C = 0	BRSH/BRCC	Rd < Rr	C = 1	BRLO/BRCS	Unsigned
Rd = Rr	Z = 1	BREQ	Rd ≠ Rr	Z = 0	BRNE	Unsigned
Rd ≤ Rr	C + Z = 1	BRSH(1)	Rd > Rr	C + Z = 0	BRLO*	Unsigned
Rd < Rr	C = 1	BRLO/BRCS	Rd ≥ Rr	C = 0	BRSH/BRCC	Unsigned
Carry	C = 1	BRCS	No carry	C = 0	BRCC	Simple
Negative	N = 1	BRMI	Positive	N = 0	BRPL	Simple
Overflow	V = 1	BRVS	No overflow	V = 0	BRVC	Simple
Zero	Z = 1	BREQ	Not zero	Z = 0	BRNE	Simple

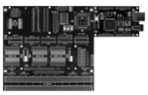
Note: 1. Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr → CP Rr,Rd



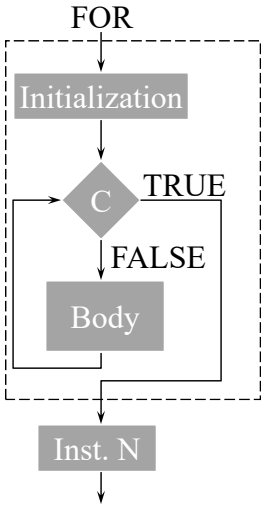
EEL 3744
IF-THEN-ELSE Selection



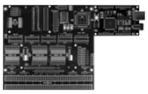

```
                *
                *
    ldi     R16, N1
    ldi     R17, N2
    cp      R16, R17  ;Condition is N1-N2=0
    breq    LY        ;IF condition is TRUE,
*                     ;  THEN branch to LY,
LX: {Inst. 2}         ;ELSE branch to here(LX)
    rjmp    LZ
LY: {Inst. 1}
LZ: {Inst. 3}
                *
                *
```

EEL 3744
FOR Repetition

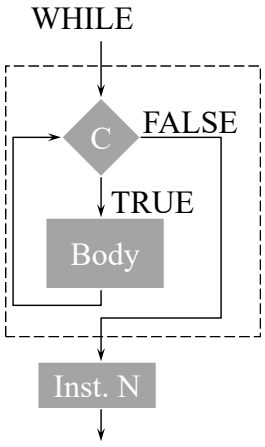



```
                *
                *
    ldi     R16, 101  ;Initialization
    breq    LY        ;The block of instructions of
                      ;  BODY are executed
LX: {BODY}            ;  repeatedly, starting
                      ;  with (R16) equal to 101 and
                      ;  continuing until (R16) is 0
    dec     R16       ;In this case, the condition is
                      ;  R16=0
    brne    LX
LY: {Inst. N}
                *
                *
```

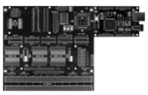


EEL 3744

WHILE Repetition

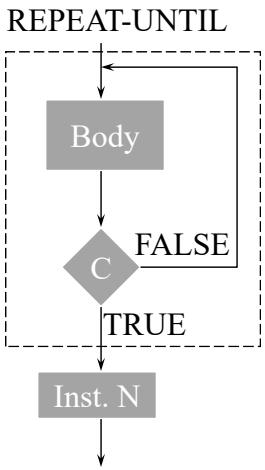



```
        *
    ldi     R16, N1
LX: cp      R16, R17    ; Condition is N1-AR17=0
                        ; The block of instructions of
                        ;  BODY are executed
    brne    LY          ;  repeatedly, WHILE the
                        ;  condition (C) is satisfied
    {BODY}              ; N1-R17=0 must be eventually
                        ;  satisfied
    rjmp    LX
LY: {Inst. N}
        *
        *
```

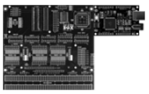



EEL 3744

REPEAT-UNTIL Repetition



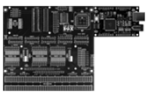

```
LX:     *               ;The block of instructions of
      {BODY}            ;  BODY are executed repeatedly
*       *               ;  UNTIL the condition(C) is
*       *               ;  satisfied.
      cp    R16,R17     ; Condition is R16-R17=0
      breq  LY
      rjmp  LX
LY:   {Inst. N}
        *
        *
```



EEL 3744
Stack on the **XMEGA**

doc8331
Section 3.8

- During subroutine calls and interrupts, the return address is **automatically** pushed on the stack
  - > The return address (for our chip) is **3** bytes [you should try it and verify], and hence the stack pointer is decremented/incremented by **three**
  - > The return address is popped off the stack when returning from each of the following:
    - – Return from subroutines with the **RET** instruction
    - – Return from interrupts with the **RETI** instruction

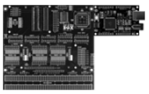


EEL 3744
RCALL/CALL, RET/RETI, and Stack on **XMEGA**

- Use **CALL** (or **RCALL**) instruction to call subroutine
- Use **RET** instruction to return from subroutine calls
- Use **RETI** instruction to return from interrupts

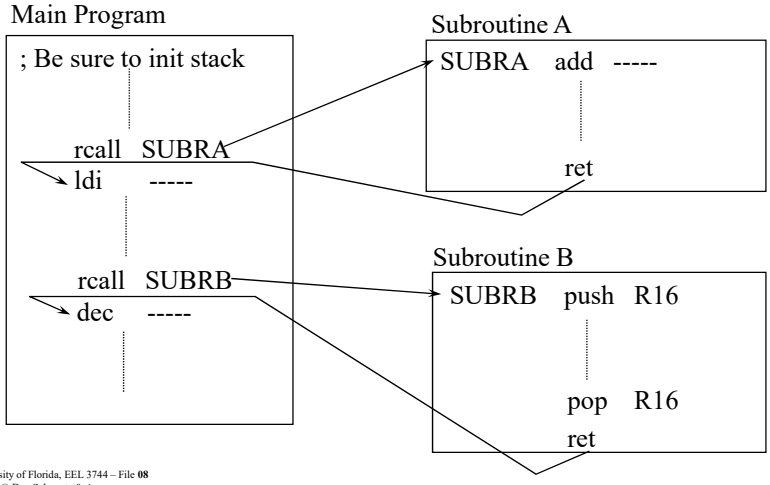
doc0856

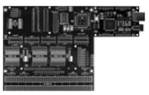
Instruction	Operands	Description	Operation	# Clocks
RCALL	k	Relative Call Subroutine	PC ← PC + k +1	3/4
CALL	k	Call Subroutine	PC ← k	4/5
RET	None	Subroutine Return	PC ← STACK	4/5
RETI	None	Interrupt Return	PC ← STACK	4/5



EEL 3744
Main-Subroutine

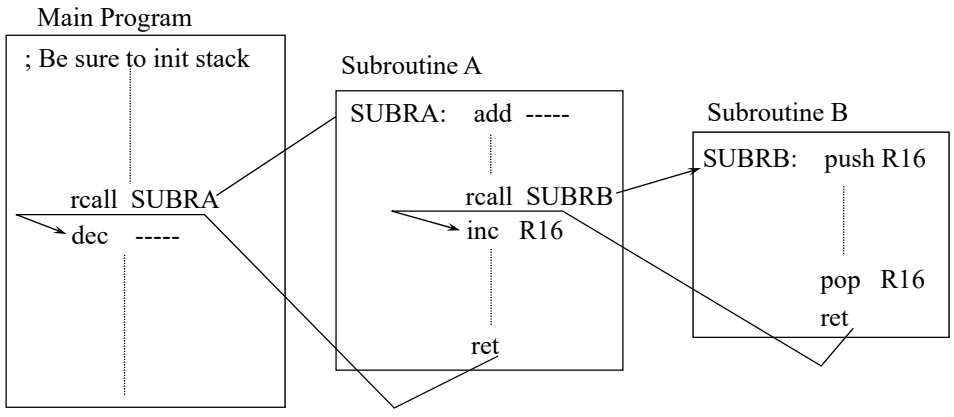
- A program which calls two subroutines; think about the stack

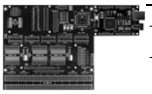




EEL 3744
Main-Subroutine (Nesting)

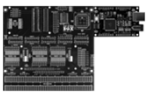
- Two levels of nested subroutine calls; think about the stack





EEL 3744 CALL and RCALL on **XMEGA**

- With the XMEGA, both RCALL and CALL store **3-bytes** (not 2-bytes) onto the stack
- The most significant byte for our processor is **ALWAYS** 0x00 because we are limited to 128k (shift right 1-bit ➔ 64k)
- The RET works as it should, i.e., a 3-byte address is used for the return
- RCALL take 2-bytes (1 word) of program memory
  - >Can go -2048 to 2047 addresses from the next address
- CALL takes 4-bytes (2 words) of program memory
  - >Can go anywhere in the addressable space (even for larger XMEGAs)



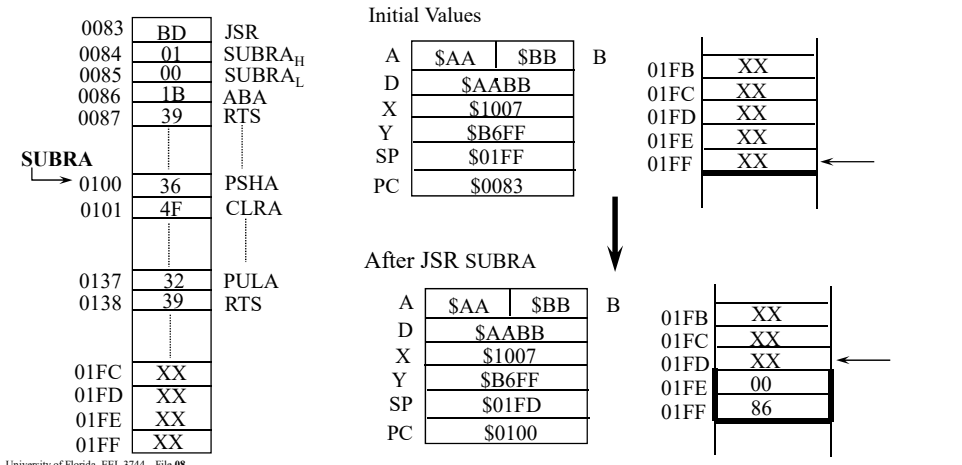
EEL 3744 Subroutine Control Instructions for **68HC11**

2: for Direct or Indexed X
3: for Extended or Indexed Y

- BSR (Branch to Subroutine)
  - > General format: BSR offset
  - > Addressing Mode: PC Relative (−128 ≤ offset ≤ 127)
  - > Description: (PC) ← (PC) + 2; ((SP)) ← (PC_L); (SP) ← (SP) – 1; ((SP)) ← (PC_H); (SP) ← (SP) – 1; PC ← PC + offset
- JSR (Jump to Subroutine)
  - > General format: JSR address (or label)
  - > Addressing Mode: Direct, Extended, Indexed X, Indexed Y
  - > Description: (PC) ← (PC)+2/3; ((SP)) ← (PC_L); (SP) ← (SP) –1; ((SP)) ← (PC_H); (SP) ← (SP) – 1; PC ← addr
- RTS (Return from Subroutine)
  - > General format: RTS
  - > Addressing Mode: Inherent
  - > Description: (SP)←(SP)+1; (PC_H)←((SP)); (SP)←(SP)+1; (PC_L)←((SP))

```
        ...
        JSR SUBRA
        ABA
        RTS
        ...
SUBRA   PSHA
        CLRA
        ...
        PULA
        RTS
```

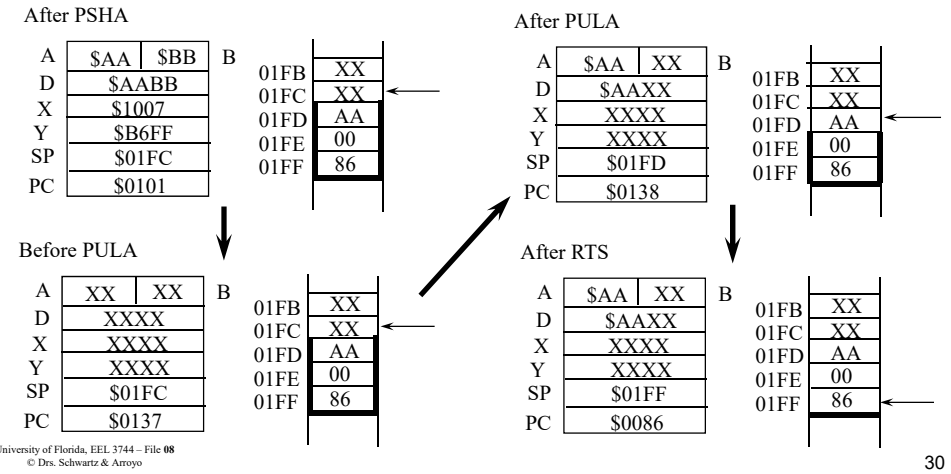
68HC11 Subroutine Control Instructions Example



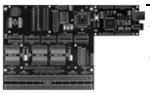
29

```
        ...
        JSR SUBRA
        ABA
        RTS
        ...
SUBRA   PSHA
        CLRA
        ...
        PULA
        RTS
```

68HC11 Subroutine Control Instructions Example (continued)



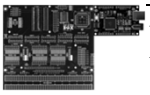
30



EEL 3744 Subroutine Control Instructions for **XMEGA**

- **rcall** (Relative Call to Subroutine)
  - > General format: **rcall** LABEL (or address) [assembler calculates **offset**]
  - > Addressing Mode: PC Relative (–2048 ≤ **offset** ≤ 2047)
  - > Description: (PC) ← (PC) + 1; ((SP)) ← (PC_L); (SP) ← (SP) – 1; ((SP)) ← (PC_M); (SP) ← (SP) – 1; ((SP)) ← (PC_H); (SP) ← (SP) – 1; PC ← PC + **offset**
- **call** (Call Subroutine)

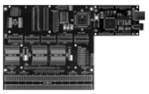
  PC is 22-bits
  $PC = PC_H \mid PC_M \mid PC_L$
  - > General format: **call** LABEL (or address)
  - > Addressing Mode: Extended
  - > Description: (PC) ← (PC) + 2; ((SP)) ← (PC_L); (SP) ← (SP) – 1; ((SP)) ← (PC_M); (SP) ← (SP) – 1; ((SP)) ← (PC_H); (SP) ← (SP) – 1; PC ← addr
- **ret** (Return from Subroutine)
  - > General format: **ret**
  - > Addressing Mode: Inherent
  - > Description: (SP)←(SP)+1; (PC_H)←((SP)); (SP)←(SP)+1; (PC_M)←((SP)); (SP)←(SP)+1; (PC_L)←((SP));



EEL 3744 XMEGA Stack Example with Subroutine

- See example on website: **Stack1.asm**
  - >View code and **simulate**
    - – Watch stack, stack pointer (SP)

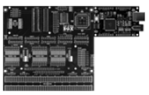
Stack1.asm



EEL 3744

Example (Add two Matrices)

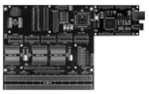
```
      Dimension A(2,3),B(2,3),C(2,3)
      Data A/1,2,3,4,5,6/,
           B/48,32,16,112,96,80/
      Call MADD(A,B,C,2,3)
      END
      Subroutine MADD(A,B,C,m,n)
      Dimension A(m,n), B(m,n),
        C(m,n)
      DO 10 i=1,m
      DO 10 j=1,n
      C(i,j) = A(i,j) + B(i,j)
   10 Continue
      End
```

EEL 3744

Example (Add two Vectors)

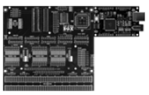
```
      Dimension A(6),B(6),C(6)
      Data A/1,2,3,4,5,6/,
        B/48,32,16,112,96,80/
      Call VADD(A,B,C,6)
      END
      Subroutine VADD(A,B,C,na)
      Dimension A(na), B(na), C(1)
      DO 10 k=1,na
      C(k) = A(k) + B(k)
   10 Continue
      End
```



EEL 3744

Example (Add two Vectors)

```
void Add_Vectors(int a[3], int b[3], int result[3]);
void main(void)
{
    int A[3] = {1, 3, -4};
    int B[3] = {0, 2, 6};
    int C[3];
    Add_Vectors(A, B, C);
}
void Add_Vectors(int a[3], int b[3], int result[3])
{
    int i;
    for(i=0; i<3; i++)
    {
        result[i] = a[i] + b[i];
    }
}
```

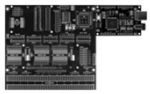
EEL 3744

VectorAdd.asm

ASM Example: Description **(for XMEGA)**

```
**********************************************
* Calls a subroutine, VADD, that adds two
* contiguous N-element vectors to form the
* resulting vector, VC = VA + VB.  The
* subroutine inputs and outputs are below.
* Inputs:  Z = address of the first vector (VA)
*          R16 = N, the number of elements
*          Z+N is address of the 2nd vector (VB)
* Outputs: X = address of resulting vector sum
*          R16=0 if successful, else non-zero
**********************************************
```

VectorAdd.asm


EEL 3744

The End!