HCS12 Assembly Programming

- Addressing Modes
- Stack Operations
- Subroutines

Addressing Modes

- Inherent
- Immediate
- Direct Addressing
- Extended Addressing
- Indexed Addressing
- PC Relative Addressing

Inherent Addressing

- No operand field (operands are implicit)
- Instructions have no operand or all operands are internal CPU registers

Examples:
- `mul` - `D = A * B`
- `idiv` - `X = D / X` and `D = D % X`
- `inc` - `A = A + 1`
- `nop` - No operation

Immediate Addressing

- Operands are constant values
- Operand is included in the instruction stream (ROM)
- No additional memory is needed (other than fetching instruction itself)
- Can be 8-bit or 16-bit value (usually depends on instruction)

Examples:
- `cmpa #10` - Compare A with 8-bit constant value 10
  - Machine Code: $81 04
- `adda #$10` - Add 8-bit constant value 16 to A
  - Machine Code: $86 01
- `cpd #$0100` - Compare D with 16-bit constant value 256
  - Machine Code: $8c 01 00
Direct Addressing

- 8-bit address provided along with instruction
- Address directly points to memory location of operand
- 8-bit addresses can access memory locations from $0000$ to $00FF$

Examples:
- `ldaa 50` – Load A with Mem[50]
  - Machine Code: $96 32$
- `adaa $FF` – Add to A the value stored at Mem[255]
  - Machine Code: $9B 7F$
- `staa 128` – Store A to Mem[128]
  - Machine Code: $5A 80$

Extended Addressing

- Similar to direct addressing but with 16-bit address
- Address directly points to memory location of operand
- 16-bit addresses can access memory locations from $0000$ to $FFFF$

Examples:
- `ldaa $1000` – Load A with Mem[4096]
  - Machine Code: $B6 10 00$
- `adaa $0100` – Add to A the value stored at Mem[256]
  - Machine Code: $B7 01 00$
- `staa $3800` – Store A to first RAM location of MC9S12C32 (Mem[14336])
  - Machine Code: $7A 38 00$

PC Relative Addressing

- Used for branch instructions
- 8-bit signed offset used for short branch instructions
- 16-bit signed offset used for long branch instructions
- When instruction is fetched, the PC already ready points to the next instruction
  - PC + 2 for most short branch instructions
  - PC + 4 for most long branch instructions

Examples:
- `beq $04` – Branch if equal to PC + 2 + 4
  - Machine Code: $27 04$
- `beq SFC` – Branch if equal to previous instruction at PC + 2 - 4
  - -4 = SFC (11111110B)
  - Machine Code: $27 7C$

Assembly Code:

```
ldaa #0     ; Initialize j
Loop: cmpa #10    ; Compare j to 10
  bge EndLoop ; Else !(j<10)
  ; do something
  adda #1     ; Increment j
  bra Loop    ; Repeat Loop
EndLoop:          ; do something else
```

How do we calculate these values?
ECE 372 – Microcontroller Design
Assembly Programming – Addressing Modes

**PC Relative Addressing**

**Assembly Cycle:**
- **Loop:** Loop: cmp #10
  - Compare j to 10
  - Else !!(j<10)
  - bra Loop
  - Repeat Loop
  - Do something else

**Assembly Code:**
- Loop: cmpa #10 ; Compare j to 10
  - bra Loop ; Repeat loop
  - Do something else

**Indexed Addressing**

**Address is based on a value stored within register (S)**
- Typically based on X or Y index registers (not required)

**Many index addressing modes exist**
- 5/9/16-bit Constant Offset
- Auto Pre/Post Decrement/Increment
- 16-bit Constant Offset
- 16-bit Indirect Indexed
- Accumulator Offset
- Accumulator D Indirect

**5/9/16-bit Constant Offset Indexed Addressing**

- Calculates address based on index register, stack pointer, or PC
- Address is 5/9/16-bit offset added to address within X/Y/SP/PC
- Offsets are assumed to be signed values

**Example**
- ldaa 0,X – Load A with value stored at Mem[X+0] (5-bit)
- stab -8, Y – Store B to Mem[Y-8] (5-bit)
- stab -20, Y – Store B to Mem[Y-20] (9-bit)
Accumulator Offset Indexed Addressing
- Calculates addressed based on index register, stack pointer, or PC and accumulators A, B, or D
- Address is the unsigned A/B/D offset added to address within X/Y/SP/PC

Example
- ldab X – Load A with value stored at Mem[X+B]
- stbd Y – Store B to Mem[Y+D]

16-bit Constant Indirect Indexed Addressing
- Calculates indirect address based on index register, stack pointer, or PC and a 16-bit offset
- Indirect address points to memory location of pointer providing memory address of operand
- Direct address is value stored in memory location provided by indirect address

Example
- ldab [10,X] – Load A with value stored at Mem[Mem[X+10]]
  - If X = $1000 and Mem[$100A:$100B] = $2000
  - Indirect address = X+10 = $1000 + $0A = $100A
  - Direct Address = Mem[$100A] = $2000
  - A = Mem[$2000]

Accumulator D Indirect Indexed Addressing
- Calculates indirect address based on index register, stack pointer, or PC and 16-bit accumulator D
  - Indirect address points to memory location of pointer providing memory address of operand
  - Direct address is value stored in memory location provided by indirect address

Example
- ldad [D,X] – Load A with value stored at Mem[Mem[X+D]]
  - If X = $1000, D = $0C, and Mem[$100C:$100D] = $3800
  - Indirect address = X+12 = $1000 + $0C = $100C
  - Direct Address = Mem[$100C] = $3800
  - A = Mem[$3800]

Auto Pre/Post Decrement/Increment Indexed Addressing
- Calculates address based on index register or stack pointer
- Provides four methods for automatically adjusting the index register during the instruction
  - Pre-decrement/Pre-Increment
    - Decrements or increments the index register before calculating address
  - Post-decrement/Post-Increment
    - Decrements or increments the index register after calculating address
  - Can be decremented/incremented in ranges from -1 to -8 and 1 to 8

Example
- ldai 1,-X – Load A with value stored at Mem[X-1], X = X - 1
- stae 2,X+ – Store A to Mem[X], X = X + 2
Create an assembly program to add together two 32-bit unsigned numbers stored in memory at location $3800 and $3804 storing the result in location $3808.

Stack Pointer (SP)
- Points to memory location corresponding to the top of the stack
- Must be initialized within your assembly program
- Initialize to location $4000 for MC9S12C32

Basic Stack Pointer Instructions
- des – Decrement stack pointer (SP)
- ins – Increment stack pointer (SP)
- psha/pshb/pshd/pshx/pshy – Push A/B/D/X/Y to stack
  - Automatically adjusts stack pointer
- pula/pulb/pulx/puly – Pull (Pop) A/B/D/X/Y from stack
  - Automatically adjusts stack pointer
- lds – Load stack pointer (SP) with from memory (or immediate value)
- sts – Store stack pointer (SP) to memory

Storing and Retrieving from top of Stack
- Utilized stack to store temporary data
- Push data onto stack
- Pop data from stack

Example:
```assembly
ldaa #0      ; A = 0
psha         ; Push A(0) onto stack
ldaa #20     ; A = 20
psha         ; Push A(20) onto stack
ldd #300     ; D = 300
pshd         ; Push D(300) onto stack
...          
puld ; D = 300
pula ; A = 20
pulb ; B = 0
```

```
SP: $3FF8
    00
    $3F9F
    $3FA
    $3FB
    $3FC
    ...  

SP: $3FF8
    00
    $3F9F
    $3FA
    $3FB
    $3FC
```
Storing and Retrieving from top of Stack
- Utilized stack to store temporary data
- Push data onto stack
- Pop data from stack

Example:

```
ldaa #0  ; A = 0
psha ; Push A(0) onto stack
ldaa #20 ; A = 20
psha ; Push A(20) onto stack
ldd #300 ; D = 300
pshd ; Push D(300) onto stack
... 
puld ; D = 300
pula ; A = 20
pulb ; B = 0 
```

SP: $3FFC
unsigned short a[10];

for(j=0; j<10; j++)
{
  if(a[j]==1) PORTT=0x04;
  else PORTT=0x00;
}

Array For Loop Example:
1. Initialize J
2. Compare J to 10
3. If Not Less than 10,
   1. End Loop
4. Else
   1. Load a[j]
   2. If a[j] == 1
      1. PORT T = 4
   3. Else
      1. PORT T = 0
   4. Increment J
5. Repeat Loop (Step 2)

We could store these values on the stack instead (temporary value).

Assembly Code:
ldaa #0 ; Initialize J
ldx #$3800 ; Initialize index to A[0]
Loop:
  cmpa #10 ; Compare J to 10
  bge EndLoop ; Else !(J<10)
  staa $3814 ; Store J to RAM
  ldd 0,X ; load A[J]
  cpd #1 ; Compare J to 1
  bne Else ; Else !(A[J]==1)
  ldab #4 ; Value to write to PORT T
  bme EndIf ; Write value to PORT T
  ldd $3814 ; Read J from RAM
  adda #1 ; Increment J
  inc ; Increment A[J]
  inc ; Need to increment by 2
  bra Loop ; Repeat Loop
Else:
  ldab #0 ; Value to write to PORT T
  EndIf:
  pula ; Pull(J) from Stack
  adda #1 ; Increment J
  inc ; Increment A[J]
  inc ; Need to increment by 2
  bra Loop ; Repeat Loop
EndLoop:
  ; do something else
  ; do something else

Subroutines
- Similar to functions in C/C++
- Allows program to call routine
- Parameter passing must be done explicitly

Basic Subroutine Instructions
- bsr - Branch to subroutine (PC Relative Addressing)
  - Return address will be pushed onto stack
  - Uses 16-bit address of following instruction (PC + 2)
- jsr - Jump to subroutine
  - Return address will be pushed onto stack
  - Uses 16-bit address of following instruction (varies depending on addressing mode)
- rts - Return from subroutine
  - PC will pulled (popped) from top of stack

Accessing items within stack (not at the top)
- Push/Pull only provide access to top of stack
- Can use index addressing to access stack items not at top of stack

ldab 1,SP ; B = Mem[SP-1] = $2C
ldd 2,SP ; D = Mem[SP-2] = $2000
...
ECE 372 – Microcontroller Design
Assembly Programming – Subroutines

**Simple Subroutine Example**

```c
void FuncA() {
    // do something
}
```

**Assembly Code:**

```
ldaa #0 ; Initialize j
Loop: cmpa #10 ; Compare j to 10
    bge EndLoop ; Else !(j<10)
    bsr FuncA ; Call FuncA
    adda #1 ; Increment j
    bra loop ; Repeat Loop
EndLoop:
FuncA: ; do something
    rts ; Return from FuncA
```

How do we calculate offset?

PC = PC + Rel
$4014 = 4004 + Rel
Rel = 16 ($10)
Subroutines with Parameters

- Utilize stack to pass parameters (in addition to return address)
- Caller Initialize Steps:
  - Push function parameters onto stack
  - Allocate space on stack for return value
  - Call subroutine
- Caller Steps:
  - Access parameters on using index addressing
- Caller Finishing Steps:
  - Pull (Pop) return value from stack
  - Deallocate space on stack previously used by parameters

C Function Example:
```
for(j=0; j<10; )
{
  j = FuncA(j);
}
```
```
byte FuncA(byte a) {
  return a+2;
}
```

Assembly Code:
```
ldaa #0     ; Initialize j
Loop: cmpa #10    ; Compare j to 10
       bge EndLoop ; Else !(j<10)
psha ; Push parameter J onto Stack
des         ; Allocate space for return value
bsr FuncA ; Call FuncA
pula        ; Pull return value from stack
ins         ; Deallocate space from stack
bra Loop    ; Repeat Loop
EndLoop:
FunctA:
ldaa 3,SP   ; Load parameter from stack
adda #2     ; Add 2 to A
staa 2,SP   ; Store return value to stack
rts         ; Return from FunctA
```

SP:  
RetVal: 
Param: 
Initial SP: 
Stack Top: $3FFB
$3FFC $3FFF $4000