ECE 372 – Microcontroller Design
Assembly Programming

- HCS12 Assembly Programming
  - Basic Assembly Programming
  - Top Assembly Instructions (Instruction You Should Know!)
  - Assembly Programming Concepts
  - Data Organization

Load and Store Instructions
- ldaa/ldab/ldd – Load A/B/D from memory (or immediate value)
- ldax/lday – Load Index Register X/Y from memory (or immediate value)
- lds – Load Stack Pointer (SP) from memory (or immediate value)
- staa/stab/std – Store A/B/D to memory
- stax/stay – Store Index Register X/Y to memory
- sts – Store Stack Pointer (SP) to memory

Arithmetic Operations
- adda/addb/addr – Add memory (or immediate value) to A/B/D
- aba – Add B to A
- sba – Subtract B from A
- inca/inxb/inxy – Increment A/B/X/Y
  - e.g. A = A + 1
- deca/decb/decx/dey – Decrement A/B/X/Y
  - e.g. A = A - 1

Shift and Rotation Operation
- Bit tests and Bit manipulations
- Stack and Index Pointer Operations
- Condition Code Register Instructions
- Subroutine Calls
- Interrupt Handling Operations
- Miscellaneous Operations
**Arithmetic Operations**
- `cmpa/cmpl/cpd/cpx/cpy` - Compare A/B/D/X/Y with memory (or immediate value)
  - Branch follows (e.g. greater, less, equal)
- `mul` - Performs 8-bit **unsigned** multiplication of A and B, storing 16-bit result in D
  - e.g. \( D = A \times B \)
- `idiv` - Performs 16-bit **unsigned** division of D and Y, storing quotient in X and remainder in D
  - e.g. \( X = D / Y \) and \( D = D \mod Y \)

**Logic Operations**
- `anda/andb` - Perform 8-bit logical AND of A/B with memory (or immediate value)
- `oraa/orab` - Perform 8-bit logical OR of A/B with memory (or immediate value)
- `eora/eorb` - Perform 8-bit logical XOR of A/B with memory (or immediate value)
- `bset/bclr` - Set/Clear bit in memory using provided mask
- `bita/bitb` - Performs 8-bit logical AND of A/B with memory (or immediate value) and modifies CCR
  - Contents of A/B or the memory location are not affected

**Branch Instructions**
- `beq` - Branch if equal (Relative PC Addressing)
- `bne` - Branch if not equal (Relative PC Addressing)
- `bra` - Branch always (Relative PC Addressing)
- `jmp` - Branch always (Absolute Addressing)

**Branch Instructions (Unsigned)**
- `bhi` - Branch if higher (Relative PC Addressing)
  - i.e. unsigned greater than
- `bhs` - Branch if higher or same (Relative PC Addressing)
  - i.e. unsigned greater than or equal
- `blo` - Branch if lower (Relative PC Addressing)
  - i.e. unsigned less than
- `bls` - Branch if lower or same (Relative PC Addressing)
  - i.e. unsigned less than or equal
**Branch Instructions (Signed)**
- bgt – Branch if greater than (Relative PC Addressing)
- bge – Branch if greater than or equal (Relative PC Addressing)
- blt – Branch if less than (Relative PC Addressing)
- ble – Branch if less than or equal (Relative PC Addressing)
- bmi – Branch if minus (Relative PC Addressing)
  i.e. negative result
- bpl – Branch if plus (Relative PC Addressing)
  i.e. positive result

**Long Branch Instructions**
- Similar to other branch instructions
- PC relative addressing
- But with 16-bit relative address
- Few Examples:
  - lbra – Branch always (Long Relative PC Addressing)
  - lb eq – Branch if equals (Long Relative PC Addressing)
  - lb gt – Branch if greater than (Long Relative PC Addressing)

**Shift and Rotate Operations**
- rol a/rol b – Rotate left A/B through carry (C)
- ror a/ror b – Rotate right A/B through carry (C)

**Logical Shift Operations**
- lsla/lslb/lsl d – Logical shift left A/B/D into carry (C)
  - Same as asla/aslb/asld instructions
  - Least significant bit is loaded with 0
  - Also performs multiply by 2
- lrsa/lrsb/lrs d – Logical shift right A/B/D into carry (C)
  - NOT the same asra/asrb/asrd
  - Most significant bit is loaded with 0
  - Also performs unsigned divide by 2
Shift and Rotate Operations

- `asla/aslb/asld` – Arithmetic shift left A/B/D into carry (C)
  - Same as `lsla/lslb/lsld` instructions
  - Least significant bit is loaded with 0
  - Also performs multiply by 2
- `asra/asrb/asrd` – Arithmetic shift right A/B/D into carry (C)
  - NOT the same as `lsra/lsrb/lsrd`
  - Most significant bit is loaded with previous most significant bit value
  - Also performs unsigned/signed divide by 2

Arithmetic Shift Right:

```
```

If-Then C Code:
```
if (cond) {
    // if body
}
```

If-Then Assembly Code:
```
cmp ??       ; If Cond.
b ?? EndIf  ; Branch !cond
            ; If Body
EndIf:
```

If-Then-Else C Code:
```
if (cond) {
    // if body
} else {
    // else body
}
```

If-Then-Else Assembly Template:
```
cmp ??       ; If-Else Cond.
b ?? Else   ; Branch !cond
            ; If Body
bra EndIf  ; Branch EndIf
            ; Else Body
EndIf:
```

If-Then Assembly Code:
```
cmpa, #20   ; If Cond. (A>=20)
blo EndIf  ; Branch !(A>=20)=>(A<20)
            ; If Body
EndIf:
```

If-Then-Else Assembly Template:
```
cmp ??       ; If-Else Cond.
b ?? Else   ; Branch !cond
            ; If Body
bra EndIf  ; Branch EndIf
            ; Else Body
Else:
    ; Else Body
EndIf:
```
If-Then-Else C Code:
```c
if (a>=20) {
    // if body
} else {
    // else body
}
```

If-Then-Else Assembly Code:
```assembly
cmpa, #20    ; If-Else Cond. (a>=20)
blz Else     ; Branch !(a>=20) => (a<20)
bra EndIf    ; If Body
Else:        ; Else Body
EndIf:
```

For Loop C Code:
```c
for(init; cond; iter) {
    // loop body
}
```

For Loop Assembly Template:
```assembly
??          ; Loop Init.
Loop: cmp??   ; Loop Cond.
   b?? EndLoop ; Branch !cond
; Loop Body
??          ; Loop Iter. (iter++)
bra Loop    ; Repeat Loop
EndLoop:
```

While Loop C Code:
```c
while (cond) {
    // loop body
}
```

While Loop Assembly Template:
```assembly
Loop: cmp?? ; Loop Cond.
   b?? EndLoop ; Branch !cond
; Loop Body
bra Loop    ; Repeat Loop
EndLoop:
```
While Loop C Code:

```c
while(i!=0)
{
    // loop body
}
```

While Loop Assembly Code:

```assembly
Loop: cmpa, #0    ;
      beq EndLoop ; Branch !(i!=0)=>(i==0)
      bra Loop    ; Repeat Loop
EndLoop:
```

Data Organization

- **Heap**
  - Block of memory for allocating storage whose lifetime is not related to the execution of the current routine
  - Often used for dynamically allocated data (malloc)
  - Typically located at lowest memory address after global variables

- **Stack**
  - Generally - A data structure supporting two basic operations, push and pop
  - **Push** - Adds an item to the top of the stack
  - **Pop** - Removes an item from the top of the stack
  - **LIFO** - Last In First Out data structure (opposite of queue)
  - **Stack Pointer (SP)**
    - Points to memory location corresponding to the top of the stack
    - Must be initialized within your assembly program