68000 Assembly Basics

Textbook 4.1, 4.4, 4.6, and 4.7

Assembly Language Syntax

- A source file is a sequence of lines
- 68000 assembly language is line oriented, i.e. a complete instruction must be on ONE line.
- Each line has four fields, separated by at least one white space item (tabs/spaces)
    - **Label field** (optional)
    - **Op-code field** (instruction or directive required)
    - **Operand field** (optional, depends on op-code)
    - **Comment field** (optional, white space is allowed within this)

Assembly Language Style Requirements

These must be followed on all assembly language assignments!!

- Line the fields up into columns
- Clearly and consistently indicate labels and comments
- Use blank lines to break code into logical units
- Chose self-documenting label names
- Write good documentation
    - This is for your benefit as much as mine!!!

Example Assembly Code

```
move.w  #40,x
        move.w  #2,y
            move.w  x,d0
          add.w y,d0
    move.w  d0,z
        clr.w   -(sp)
            trap    #1
x ds.w      1
y           ds.w    1
z       ds.w    1
```

This will assemble. It will also get you a zero, because I will not read it!!

Example Readable Assembly Code

```
int x;  // using global
variables, for now
int y;
int sum;
void main()
{
    x = 40;
    y = 2;
    sum = x + y;
}
```

```
; PURPOSE: compute 40 + 2 using 16-bit ints
; NOTE: unoptimized!
start:    move.w    #40,x    ; x = 40
          move.w    #2,y     ; y = 2
          move.w    x,d0
          add.w     y,d0
          move.w    d0,sum   ; sum = x + y
          clr.w     -(sp)
          trap      #1       ; exit
x:        ds.w      1        ; int x
y:        ds.w      1        ; int y
sum:      ds.w      1        ; int sum
```

Label Field

- Must begin with one of:
  - a-z
  - A-Z
  - _ ← underscore
  - @,? ← These are special purpose
- And then be a sequence of
  - the above
  - 0 - 9

Label Field

- Devpac limits labels to **22 characters**, anything beyond this is simply ignored.
- Labels are case-sensitive
- Labels can not be redefined within a module
- Labels must start in column 1, however, this can be overridden if followed by a **:** (colon)
- **Anything starting in column 1 is a label!!!**
    - Except a comment

Comment Field

- Anything that appears after the operand field is a comment
- The comment field can begin early, if preceded by a **;** or a *****
- For readability, most programmers:
  - End all label names with :
  - Begin all comments with either a **;** or ***** (i.e. pick one and stick to it)

Example

```
; PURPOSE: a program that terminates immediately

start:  clr.w   -(sp)
        trap#1  ;request that O/S terminate program
```

Notes:

- Not every line needs to be commented
- Comments MUST be meaningful
    - restating the line of code is NOT meaningful

Important Notes for Assembly

- Assembly time – Creating executable
- Run time – Running executable
- These are important when discussing features of the language
    - Different instructions will be processed at different times

Opcode Field

- The opcode will correspond to one of two possibilities
  1. Assembly Directives
     - Commands for the assembler to follow
  2. Instructions
     - Commands for the CU to run

Opcode Field - Assembly Directives

- Specify an action that the **assembler** is to take
    - Action will occur at **assembly time**!
- Placed in the opcode field
- Assembly directives are not case-sensitive
- Directives are:
    1. dc.x <value list>
    2. ds.x <count>
    3. dcb.x <count>, <value>
    4. even
    5. <name> equ <value>

Assembly Directive - Size Determination

- All of the previous directives had the .x suffix.
- The value in **x** tells the compiler the data size to be used.
- The possible options are:
  - b byte
  - w word
  - l long
- If no size is specified a default size is used
  - Default size is not consistent over all instructions

Define Constant (dc) Directive

dc.x <*value_list*>

- tells the assembler to reserve a block of memory
  - initialized to the given, comma separated, list of constants
  - initialized at the current location
  - each block of memory will be of size **x**
- A poor name since this is really a variable with an initial constant value
- Equivalent to a variable declaration and initialization

Direct Constant Directive Examples

Examples

```
x:          dc.w    7                   ;int x = 7;
letter_a:   dc.b    'a'                 ;the letter a
answer:     dc.b    42                  ;one 8-bit int
primes:     dc.w    2,3,5,7,11          ;five 16-bit ints
string:     dc.b    "hello, world!",0   ;null terminated string
```

Character Literals

- There is **no difference** between single and double quotes
- The assembler replaces the sequence of characters with a sequence of 8-bit ASCII values
- Writing the ASCII value directly is bad style

Integer Literals

- **%** prefix means binary (e.g. %10010 = 18_{10})
- **@** prefix means octal (e.g. @22 = 18_{10})
- **$** prefix means hex (e.g. $12 = 18_{10})
- **no prefix** means decimal
- negation of literals is supported (e.g. -$3A)
    - does a 2's comp conversion this would be $C6
    - the negative sign is placed before the prefix

Define Storage (ds) Directive

- **ds.x <*count*>**
- tells the assembler to reserve a block of <*count*> * “size of x in bytes” bytes at the current location all with an initial value of zero

Example:

```
block:    ds.l10   ; reserves 40 bytes (10 longwords)
x:    ds.w    1    ; reserves 2 bytes (1 word)
```

Define Constant Block (dcb) Directive

- **dcb.x <*count*>,<*value*>**
- similar to ds, but the all items of size x in the reserved memory are initialized to <*value*>
- Notice only **ONE** *value* so the entire block is initialized to this value!

Example

```
all_ones:  dcb.w      10,1      ; reserves 20 bytes (10 words) all with the value
                                  of 1 0001,0001,0001, ...
```

Even Directive

This is used to align memory, i.e. waste a byte

```
age:        dc.w    1
greeting:       dc.b    'hi',0
                even
income:         dc.w    1   ; word address must be even!
```

- When allocating variables you should put items of the same size together and order should be:
  - All words or all longs first, then the other
  - Then all bytes
- Avoid mixing as this can lead to run-time errors!!

Equals (equ) Directive

PAY_RATE equ 25 ; pay rate is $25/hr

- defines an **assemble-time** constant
  - the assembler replaces each occurrence of <name> with the literal <value>
  - normally placed near the top of a source file

code, data, end directives

```
                code
start:          move.w          #40,d0
                add.w           #2,d0
                move.w          d0,sum          ; sum = 40 + 2
                clr.w-(sp)
                trap            #1              ; exit
                data
sum:            ds.w            1


                end                             ; this terminates the assembly process
```

If used the assembler will group the various sections together.

Other than end, **which must be the last directive in every file**, these are rarely used

Opcode Field - Instructions

- These are action that will done at runtime.
- Instructions are translated into machine code by the assembler
- The instructions are given as small words or short forms
  - These are referred to as mnemonics
- Instructions are sorted into three general categories:
  1. Data Transfer
  2. Arithmetic/Logic
  3. Program Sequencing & Control

Instructions - Number of Operands

- 68000 instructions can have one of three different number of operands:

Number of Operands	
0	
1	destination
2	source,destination

NOTE: There are **NO** spaces between the operands and the comma

Instruction Results

- Instructions have a **direct result** and an **indirect result**
- The **direct result** is the expected action specified by the instruction
  - i.e. the value placed in the destination
- The **indirect result** is the effect on the status register
  - i.e. the condition codes
- All instructions have a direct result
  - ONLY Data Transfer and Arithmetic/Logic instructions have an indirect result

Operand Field - Addressing Modes

- The **addressing mode** of an operand is the way in which
- The 68000 has 12 addressing modes
  - To start 4 of the simplest modes will be examined
    - register direct
    - immediate
    - absolute
    - register indirect

Addressing Modes - Register Direct Mode

- The data value is the contents of the specified register
  - i.e. the register contains the operand
- Examples:
  -
```
move.l  d0,d1   ;src/dst are both register direct
                ;operation copies the longword contents
                ;of d0 into d1
```
- `Syntax: dN or aN where 0 <= N <= 7`

Addressing Modes - Immediate Addressing Mode

- The operand is the data value being used in the operation
- Indicated by a hash (#)
- Example
  - ```
    move.l #5,d1;src=immediate, dst=register direct
                        ;copies the value 5 as a longword into d1
    movea.l #var,a1 ;copies the address of var into a1
    ```
- Integer literals (to specify bin/oct/hex) can be included
- Example
  - `move.l #%101,d1 ;copies the value` $101_2 = 5_{10}$ `as a longword into d1`
    ```

Addressing Modes - Absolute Addressing Mode

- The data value is found at the directly specified **address**
  - i.e. the data is the contents of memory at the specified address
- A numeric value or a label but **NO HASH**
- Examples:
  - `move.l 5,d1` `;src=absolute, dst=register direct`
    `;copies the longword contents of address 5 to d1`
  - `move.l var,a3` `;copies the longword contents of var to a3`

Addressing Modes - Register Indirect Mode

- The data value is the contents of the address in the specified address registers
  - The **address register** contains the address of the operand
  - Indicated by **parentheses**
- Examples:
  - ```
    move.l  #var,a0     ;copies address of var into a0
    move.b  (a0),d2     ;src=register indirect, dst=register direct
                        ;copies 99 into d2
    var:dc.b 99     ;THIS IS A POINTER
    ```
    ```

Instructions

- The next few slides will show some of basics of instructions
- Use the 68000 reference card for the complete list of operations, with all allowable addressing modes and indirect results
- Some instructions are **data register** only:
  - **add** either the source, or destination **MUST** be a data register
- Some instructions are **address register** only:
  - **adda** most of these are address register specific instructions

Instructions

- The **move** instruction is very common
  - Computers spend a large amount of time moving data around! Why?
- **clr.x var** is similar to **move.x #0,var** but faster
- **exg** works only for register operands
- **lea** is similar to the C++ **address of** (&) operation
  - the destination must be an address register
- **neg** computes the 2’s complement conversion
- **not** performs the 1’s complement conversion

Instructions

- **muls** and **mulu** multiply a word by a word to produce a longword
- **divs** and **divu** divide a longword by a word to produce a longword, where MSW = remainder and LSW = quotient
- **asr/asl** and **lsr/lsl** are arithmetic (signed) and logical (unsigned) shift operators which multiply/divide by a power of 2.
    - the only difference is that asr shifts in a copy of the old high bit, while lsr always shifts in 0.
    - Note; that both use the C bit to indicate a loss of significance!

Instructions

- **tst.x var** is similar to **cmp.x #0,var**, but faster
- **ror/rol** rotate the bits in a register (i.e. out one end and in the other)
- **ext** is for widening signed integers.
- **nop** is not useless!

Instruction Results & Instruction Size

- All data and address registers are 32-bits
  - Not all instructions work on all 32-bits
- Size worked on depends on specified size for instruction
- For W,B sizes only the lower order word and byte affected
  - Even for carry out
- Condition code bits set based on bytes used in instruction

Instruction Results & Instruction Size

Example 1:

d0=FFFF9002

add.w d0,d0 ;result is d0=FFFF2004, C=1,V=1,N=0,Z=0

NOTE: the MSW was not affected at all, and didn’t impact CC bits

Instruction Results & Instruction Size

Example 2:

d0=FFFF9002

add.**b** d0,d0 ;result is d0=FFFF9004, C=0,V=0,N=0,Z=0

NOTE: only the LSB was affected, and only the LSB impacted CC bits

Types of Programming Errors

There are several types of assembly programming error:

- assembly-time errors (i.e. syntax errors)
- link-time errors (i.e. unresolved external references)
- run-time “faults” (i.e. crashes, also called abnormal ends - “abends”)
- run-time “behavioral” errors (i.e. not crashes, but logic errors due to programmer error, analysis/design flaws, etc.)

Common Runtime Faults

- **Bus Error**: address doesn't exist, or is protected by hardware & O/S (usually caused by stray pointer). Called "seg. fault" under Unix/Linux. Atari – 2 bombs
- **Address Error**: attempted access of word or longword at odd address (usually caused by stray pointer or careless global data setup). Atari – 3 bombs

Common Runtime Faults

- **Illegal Instruction**: attempted execution of invalid instruction (usually caused by random jump or execution of data). Atari – 4 bombs
- **Privilege Violation**: attempted execution of supervisor-only instruction from user mode (again, usually caused by random jump or execution of data). Atari – 8 bombs
- … THERE ARE OTHERS!!!