COM1006 Devices and Networks (Autumn)
COM1090 Computer Architectures

Lecture #10

Assembly language programming

Dr Dirk Sudholt
Department of Computer Science
University of Sheffield

d.sudholt@sheffield.ac.uk
http://staffwww.dcs.shef.ac.uk/~dirk/campus_only/com1006/

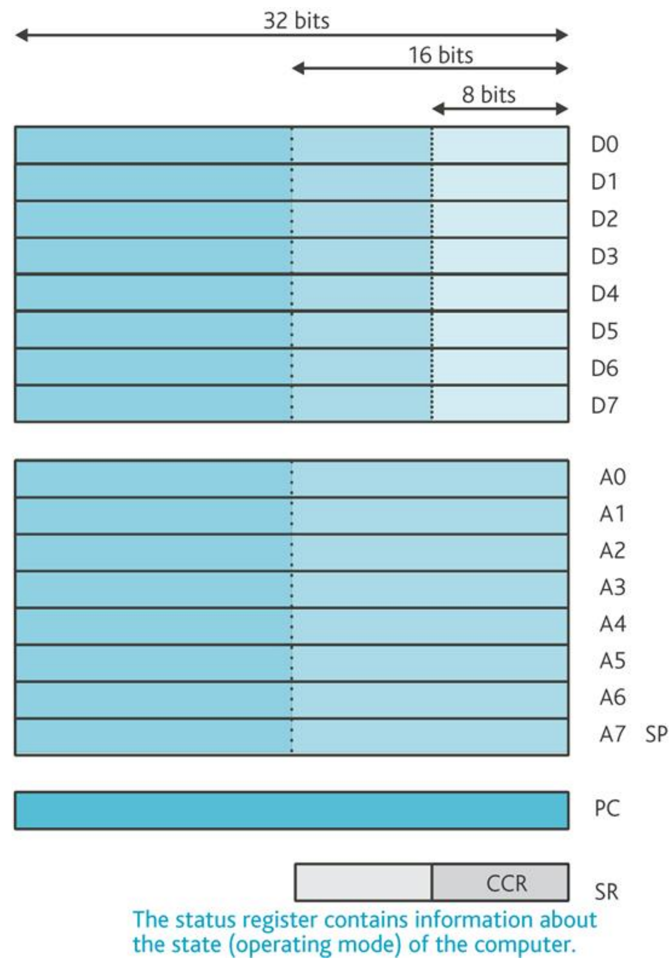
Based on Chapters 5-6 in Clements, Principles of Computer Hardware

Aims of this lecture

- Learning **basics** of assembly language programming on the Motorola 68K as a typical CISC architecture.
- To demonstrate the EASy68K 68K assembler and simulator, which will be used in the tutorials.
- To see how variables are defined and stored in memory.
- To introduce common 68K instructions for
  - data movement
  - logical operations
  - arithmetic operations
  - branching instructions (flow control)

Register sizes

- Data registers are 32 bits wide, but can be treated as if they were 8 or 16 bit wide.
- This is determined by size suffixes:
  - 8-bit: `.B`, e.g. `MOVE.B  D0,`**`D1`**
  - 16-bit: `.W`, e.g. `MOVE.W  D0,`**`D1`**
  - 32-bit: `.L`, e.g. `MOVE.L  D0,`**`D1`**
- Not stating any suffix uses the **default of 16 bits**.



68K Status flags

- The processor status register (PSR) records the outcome of an instruction (hence the state of the machine).
- It contains at least 4 bits (flags) whose values are (usually) set or cleared after each instruction has executed:

Z-bit	Set if the result of the operation is **Zero**
N-bit	Set if the result is **Negative** in a two's complement sense
C-bit	Set if the result yields a **Carry-out**
V-bit	Set if an **oVerflow** in two's complement has occurred

- Status flags are used to implement conditional behaviour.

Variables in 68K

- Variables with given initial values are defined using the DC (Define Constant) command:

```
X   DC.B  12
```

- The suffix specifies **how many bytes** are being allocated.
- The „X“ at the start of the line defines a **label** for this variable. This label can be referred to, e.g. in `ADD X,D0`.
- Strings are defined as sequence of bytes. They are put in single quotes and **must terminate with a byte value 0**:

```
HelloMsg DC.B  'Hello World!', 0
```

- Good practice: keep all variable definitions separate from the code, e.g. at the end of the program.

Demo in EASy68K: intro.X68

- Most popular 68000 Assembler and 68000 Simulator
- Available from http://www.easy68k.com/
- Demo programs will appear on module website.
- Demo shows
  - how the editor works
  - how programs can be assembled and executed step-by-step
  - how registers and flags change
  - where and how program and variables are stored in memory
  - labels like „X“ are replaced by their memory locations
  - to illustrate potential pitfalls with different word sizes

Data movement instructions

- Data movement instructions are the most common types of instructions.

  `MOVE X, Y` [Y] ← [X]

  copies X to Y. One operand must be a register (usually D0-D7).

- Address registers A0-A7 are used for **indirect addressing**. The instruction

  `LEA Text, A1` [A1] ← Text

  (Load Effective Address) loads the address of "Text" in A1.

Logical operations

- `CLR.L D0` — [D0] ← 0
  clears a register, i.e. sets its contents to 0.

- All common logical operations are available, e.g.:

`AND`	`D0,D1`	[D1] ← [D1] AND [D0]
`OR`	`#$00FF,D2`	set lower 8 bits in D2
`EOR.B`	`#%00000001,D3`	toggle LSB in D3
`NOT`	`D0`	invert all bits in D0(.W)

- $ indicates a **hexadecimal** number, % a **binary** number.

- There are various **shift** operations (left/right) :
  - logical shift 10011001 → 00100110 (fill with zeros)
  - arithmetic shift **1**0011001 → **111**00110 (replicate sign bit)
  - rotation 100110**01** → **01**100110 (wrap around)

Arithmetic instructions

- Addition:

  `ADD.L #2,`**`A0`** $[A0] \leftarrow [A0] + 2$

  Multiplication of two unsigned/signed register values, resp.:

  `MULU D2,`**`D3`** $[D3] \leftarrow [D2] \times_{unsigned} [D3]$

  `MULS D2,`**`D3`** $[D3] \leftarrow [D2] \times_{signed} [D3]$

  Note: operands are the lower 16 bits, the result is 32 bits.

  Decrementing a register:

  `SUB #1,`**`D5`** $[D5] \leftarrow [D5] - 1$

- ❓ Why don't we have ADDU vs. ADDS and SUBU vs. SUBS?

Division

Division of two unsigned/signed register values, resp.:

`DIVU D2,`**`D3`** $[D3] \leftarrow [D3] /_{\text{unsigned}} [D2]$

`DIVS D2,`**`D3`** $[D3] \leftarrow [D3] /_{\text{signed}} [D2]$

Target register contains **result in lower 16 bits** and **remainder in upper 16 bits**.

- Remainder can be cleared away using

`AND.L #$0000FFFF,`**`D3`**

- Access the remainder using a right shift or

`SWAP` **`D3`** swaps upper and lower 16 bits

Division fails if division by 0 or if result does not fit in 16 bits.

Input/Output in Motorola 68K

- Input/Output is performed by
  1) putting an instruction code in D0
  2) and calling the instruction

     ```
     TRAP #15
     ```

- For instance, if D0 contains #13 then a null-terminated string at [A1] is printed on the screen.
- See demo hello-world.X68.
- Find a table of instruction codes in EASy68K under

  Help → Help → Simulator I/O

Flow control instructions

- An **unconditional jump** continues the program at a given address:

```
JMP        GoHere       [PC] ← GoHere
...
GoHere:    ADD D1,D0
```

- The assembler translates the label "`GoHere`" to the memory location where the instruction `ADD D1,D0` is stored in memory.

Conditional branches

- Conditional branches execute jumps depending on status flags. Status flags are set after an arithmetic operation or a **comparison**. Example of a do-while loop:

```
Loop     ...
         ADD #1, D5          increment D5
         CMP #10, D5         is D5==10?
         BNE Loop            if so, go to Loop
```

- `CMP` sets the status flags according to the result of the subtraction [D5]-10.
- `BNE` tests the value of the Z-bit. If it is not set (i.e. [D5]-10 is not 0) then we jump to `Loop` and do another iteration.
- Other branch commands test other bits (N, Z, V, C): BGE ("≥"), BGT (">"), BE ("="), BLT ("<"), BLE ("≤"), ...

Subroutines

- Subroutines can be called from anywhere in the program (like methods in Java).
- `BSR` and `JSR` (branch/jump to subroutine) store the address of the next instruction on the stack before branching.
- `RTS` (return from subroutine) pops the return address from the stack and returns there.

```
      BSR    Sub          [top of stack]←Next, [PC]←Sub
Next: ADD    D0,D1
      ...
Sub:  MOVE   #4,D0
      TRAP   #15
      RTS                 [PC]←[top of stack]
```

- Subroutines can save registers on the stack (not discussed here).

Things to remember

- For the assessment you should be able to **understand** and **write** simple Motorola 68K assembly language programs.
- Remember common Motorola 68K instructions.
- Remember to always add a „#“ to immediate values:

```
ADD   #4,D0
```

- Remember that the first operand is the source, the second one is the destination.
- Be careful when dealing with different word lengths – make sure to clear registers before using them.

Summary

- Learned common Motorola 68K instructions.
- Seen the EASy68K assembler and simulator in action.
- Motorola 68K uses different word lengths for registers.
- Program and variables are stored alongside in memory.
- Input/Output routines can be called using `TRAP #15`.
- Branching instructions implement if-statements, loops, etc.
- Subroutines can be called from anywhere in the program; the return address for the program counter is stored on the stack.