Assembly Language Programming

Control Flow, Endianess and Registers
Structure of an Assembly Program

An asterisk in the first column indicates a comment.

Assembler directives (these directives tell the assembler what to perform its job).

Label

68K machine level instruction

Source operand

Destination operand

This is an operand that describes the data to the CPU

This is a mnemonic that describes the operation to be carried out

A .W suffix indicates that the data size is 16 bits

ORG $1000
MOVE P, D0
ADD Q, D0
MOVE D0, R
STOP #$2700

* This is the end of the code area

ORG $2000
P
DC.W 2
Q
DC.W 4
R
DS.W 1
END $1000
Anatomy of an Assembly Instruction

- **Start**: Anything beginning in column 1 is a user-supplied label that allows the programmer to refer to this line.

- **Move**: This is a mnemonic that indicates the instruction to be carried out.

- **P, D0**: The first operand, P, defines the source of the data used by the instruction. In this case, the source is the contents of a memory location.

- **Get P**: D0 is the destination operand (in this case a data register).

- **Edges**: The fields of an instruction are separated by at least one space.

- **Comment**: Anything following the instruction is treated as a comment and ignored by the assembler.

- **Size**: Many instruction are terminated by `.B` `.W` or `.L`. These suffixes define the size of the operand (byte, word, or longword).
Assembling a Program

```
00000000
00000000
00000000
00000000
00000000
00000000
1  *-----------------------------------------------------------
2  * Program Number: OUP1
3  * Written by : A. Clements
4  * Date Created : 10 October 2003
5  * Description : Use of assembler directives
6  *
7  *
8  8 ORG $1000
9  9
10  10 MOVE P, D0
11  11 ADD Q, D0
12  12 MOVE D0, R
13  13 STOP #2700
14  14
15  15 ORG $2000
16  16 P DC.W 2
17  17 Q DC.W 4
18  18 R DS.W 1
19  19 END $1000
```

No errors detected.
No warnings generated.

### Symbol Table Information

<table>
<thead>
<tr>
<th>Symbol-name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>2000</td>
</tr>
<tr>
<td>Q</td>
<td>2002</td>
</tr>
<tr>
<td>R</td>
<td>2004</td>
</tr>
</tbody>
</table>

The symbol table relates symbolic names to their addresses.
Define Constant Assembly Directive

(a) ORG $2000 sets the location counter to 2000.

(b) P DC.W 2 puts $0002 in the current location and moves the location counter to the next free location.

(c) Location $2000 has the symbolic value P. Using 'P' in the program is the same as using $2000.
Easy68K Simulator

See box on page 234 for a list of common beginner’s errors.
Unconditional Branch

The branch instruction forces the instruction at 2000 to be executed next.

BRA 2000

2000
Instruction N
2004
Instruction N + 1
2008
Instruction N + 2
200C
Instruction N + 3
2010
Instruction N + 4
2014
Instruction N + 5

1000 Instruction 1
1004 Instruction 2
1008 Instruction 3
100C Instruction 4
1010 Instruction 5
1014 Instruction 6
1018 Instruction 7
101C Instruction 8
1020 Instruction 9
1024 Instruction 10
1028 Instruction 11
102C Instruction 12
1030 Instruction 13
1034 Instruction 14
1038 Instruction 15
103C Instruction 16
1040 Instruction 17
1044 Instruction 18
Conditional Branch

When instruction 8 is executed, the next instruction is at 1020 if the condition is false and 2000 if the condition is true.
# 68K’s Conditional Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>Equal</td>
<td>Z=1</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Z=0</td>
</tr>
<tr>
<td>BCS/BHS</td>
<td>Carry set/higher or same</td>
<td>C=1</td>
</tr>
<tr>
<td>BCC/BLO</td>
<td>Carry clear/lower</td>
<td>C=0</td>
</tr>
<tr>
<td>BMI</td>
<td>negative</td>
<td>N=1</td>
</tr>
<tr>
<td>BPL</td>
<td>Positive or zero</td>
<td>N=0</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>V=1</td>
</tr>
<tr>
<td>BVC</td>
<td>Overflow clear</td>
<td>V=0</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher than (signed)</td>
<td>(C=1)(\vee)(Z=0)</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same (signed)</td>
<td>(C=0)(\vee)(Z=1)</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater than or equal (signed)</td>
<td>N=V</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than (signed)</td>
<td>N(\neq)V</td>
</tr>
<tr>
<td>BGT</td>
<td>Greater than (signed)</td>
<td>(Z=0)(\vee)(N(\neq)V)</td>
</tr>
<tr>
<td>BLE</td>
<td>Less than or equal (signed)</td>
<td>(Z=1)(\vee)(N(\neq)V)</td>
</tr>
</tbody>
</table>
An Example

- A and B are two $n$-component vectors
- Write the code that computes the inner product of A and B

$$s = A \cdot B = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 + \ldots + a_n \cdot b_n$$
Example code

CLR.L D6 ; clear initial sum in D6
MOVE.B #24, D5 ; load loop counter with n (assume n=24 here)
MOVE.L #A, A0 ; A0 points to first position of vector A
MOVE.L #B, A1 ; A1 points to first position of vector B

Next
MOVE (A0), D2 ; Repeat: get Ai and update pointer to A
ADD.L #2, A0 ; Point to the next element in A
MOVE (A1), D3 ; get Bi and update pointer to B
ADD.L #2, A1 ; point to next element of B
MULU D2, D3 ; Ai x Bi
ADD.L D3, D6 ; s = s + Ai x Bi
SUB #1, D5 ; decrement loop counter
BNE Next ; repeat n times
A Subroutine Call

- MOVE.W #4, D0
- BSR ABC

Set up a parameter in D0

The subroutine call

Return point (next instruction after the subroutine)

ABC MULU D0, D0
ASL.L #1, D0
RTS

The subroutine

Return from subroutine

Body of the code

MOVE.W #4, D0
BSR ABC

Subroutine call

Return from subroutine

Subroutine

ABC MULU D0, D0
ASL.L #1, D0
RTS

Return
Multiple Subroutine calls
Soft-Boiled Eggs
Endianess

- **big-endian**: in Jonathan Swift's 1726 novel *Gulliver’s Travels* there are tensions in Lilliput and Blefuscu because a faction called the Big-endians prefer to crack open their soft-boiled eggs from the big end, contrary to a Lilliputian royal edict.

(Source: Wikipedia)

“In the matter of breaking eggs they were also divided into parties: those that broke their eggs at the smaller end were in power and had banished their opponents who broke eggs at the bigger end. This had all happened because the present Emperor's grandfather while he was a boy and going to eat his egg, and broke it according to the ancient practice, had happened to cut one of his fingers. Then his father published an edict commanding all his subjects, upon great penalties, to break the smaller end of their eggs.” (from www.hanskokhuis.nl/Lilliput.html)
Endianess dilemma

- If you store the 32-bit long word $12345678$ in memory location $1000$, where does the byte $12$ goes?
  - Big Endian faction: it goes in location $1000$, the big end of the word goes first.
  - Little Endian faction: it goes in location $1003$, the little end of the word goes first.

The 68K is a Big Endian machine.
The 68K’s Byte-Addressable Memory Space

- **8 bits**
  - Byte-wide memory

- **16 bits**
  - Word-wide memory

- **32 bits**
  - Longword-wide memory
The Endianess Divide

Both Big Endianness and Little Endianness are shown in the diagram. The computers listed as Big Endianness include IBM 370, Motorola 68K, and Sun SPARC. The Little Endianness computers listed are Pentium, VAX, and Intel 86 series. Additionally, the IA64, Power PC, DEC Alpha, MIPS, and PA-RISC processors are noted for their switchable endianness.
Data Registers

- The suffix in an instruction indicates in how many bits the instruction operates
  - .L (long word): 32 bits
  - .W (word): 16 bits
  - .B (byte): 8 bits

- The 68K data registers are 32-bit long.
  - What happens with the other bits when a .B or a .L instruction is executed?
Data Register (cont.)

- The bits not affected by an instruction remain unchanged.
  - The result of CLR.B D1 is:
    - D1 ← XXXX XXXX XXXX XXXX XXXX XXXX XXXX 0000 0000
    - Where X represent the old value of the corresponding bit.

- The carry bits and other CCR bits are determined only by the result in the area affected by the operation.
Address Registers

- Only .W and .L operations are allowed in address registers.
  - The result of .W operations are sign extended to 32 bits.
    - Examples:
      - MOVEA.W #$8022, A3 results in
        \([A3] \leftarrow \$FFFF8022\)
      - MOVEA.W #$7022, A3 results in
        \([A3] \leftarrow \$00007022\)
Instructions to Operate with Address Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDA.L D1, A3</td>
<td>ADDA = add to address register</td>
</tr>
<tr>
<td>MOVEA.L D1, A2</td>
<td>MOVEA = move to an address register</td>
</tr>
<tr>
<td>SUBA.W D1, A3</td>
<td>SUBA = subtract from an address register</td>
</tr>
<tr>
<td>CMPA.L A2, A3</td>
<td>SUBA = compare with an address register</td>
</tr>
</tbody>
</table>
Address Registers are Pointers

MULU #12, D0 ; Calculate the offset into the data structure

MOVEA.L A0, A1 ; Copy A0 to A1

ADDA.L D0, A1 ; Add the offset to A1
EXG and SWAP Instructions

(a) Effect of an exchange operation.

(b) Effect of a swap operation.
Data Movement Instructions

- The textbook provides examples of data movement (pp. 237-241) and arithmetic (pp. 241-244) instructions and illustrates their effect via the simulator.

  - You may want to complement your reading of these sections by actually trying the program segments in the Simulator yourself.
Shift and logical operations

- Problem: D0 contains an 8-bit byte in D0 with the format `xxyyyyyzzz`. We want to find out the value of the field `yyy`.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSR.B</td>
<td>#3, D0</td>
<td>Shift D0 right 3 places to get <code>000xxyyy</code> in D0</td>
</tr>
<tr>
<td>AND.B</td>
<td>#%00000111, D0</td>
<td>Clear bits 3 to 7 of D0 to get <code>00000yyyy</code></td>
</tr>
</tbody>
</table>
Shift and logical operations

- **Problem:** We want to clear bits 0, 1, and 2, set bits 3, 4, and 5, and toggle bits 6 and 7 of the byte in D0

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mask</th>
<th>D0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND.B</td>
<td>#%11111000</td>
<td>D0</td>
<td>Clear bits 0, 1, and 2</td>
</tr>
<tr>
<td>OR.B</td>
<td>#%00111000</td>
<td>D0</td>
<td>Set bits 3, 4, and 5</td>
</tr>
<tr>
<td>EOR.B</td>
<td>#%11000000</td>
<td>D0</td>
<td>Toggle bits 6 and 7</td>
</tr>
</tbody>
</table>
Conditional Branches

- **Problem**: Subtract D1 from D2 and branch if the result is negative

```plaintext
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB D1, D2</td>
<td>Subtract D1 from D2</td>
</tr>
<tr>
<td>BMI ERROR</td>
<td>Jump if the result was negative</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ERROR ...</td>
<td></td>
</tr>
</tbody>
</table>
```

What is wrong with this code?
Conditional Branches

- **Problem:** Subtract D1 from D2 and branch if the result is negative

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB D1, D2</td>
<td>Subtract D1 from D2</td>
</tr>
<tr>
<td>BMI ERROR</td>
<td>Jump if the result was negative</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>BRA JOIN</td>
<td>Unconditional branch to skip the error handling code</td>
</tr>
<tr>
<td>ERROR …</td>
<td></td>
</tr>
<tr>
<td>JOIN</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Branches

![Flowchart of conditional branches]

- Start
- IF
  - true: THEN
  - false: ELSE
- JOIN
Template of Control Flow Structures

IF \([D0] = [D1]\) THEN Action1

<table>
<thead>
<tr>
<th>CMP</th>
<th>D0, D1</th>
<th>Perform test</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td>EXIT</td>
<td>IF ([D0] \neq [D1]) THEN exit</td>
</tr>
<tr>
<td>Action1</td>
<td>⋮</td>
<td>ELSE execute Action1</td>
</tr>
<tr>
<td>EXIT</td>
<td>⋮</td>
<td>Exit point</td>
</tr>
</tbody>
</table>

Start

[D0]=[D1]?

Yes

Action1

No

EXIT
Template of Control Flow Structures

IF \([D0] = [D1]\) THEN Action1 ELSE Action2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP D0, D1</td>
<td>Perform test</td>
</tr>
<tr>
<td>BNE Action2</td>
<td>IF ([D0] \neq [D1]) THEN goto Action2</td>
</tr>
<tr>
<td>*** Action1</td>
<td>ELSE execute Action1</td>
</tr>
<tr>
<td>BRA EXIT</td>
<td>Skip around Action2</td>
</tr>
<tr>
<td>*** Action2</td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>Exit point</td>
</tr>
</tbody>
</table>

Start → [D0]=[D1]? → Yes → Action1 → EXIT → No → Action2 → EXIT
Template of Control Flow Structures

FOR K = P TO J
Action1
ENDFOR

What is wrong with this code?

What happens if P > J?

Action1

MOVE #P, D2  Load loop counter

ADD #1, D2  Increment counter

CMP #J+1, D2  Test for end of loop

BNE Action1  IF not end THEN goto next iteration

EXIT  Exit point

D2 ← P

Action1

[D2]=J+1?

Yes

EXIT

No

Start
Template of Control Flow Structures

FOR K = P TO J
  Action1
ENDFOR

MOVE #P, D2  Load loop counter
Test
CMP #J, D2  Compute D2-J
BGT EXIT  Done if D2 is greater than J
Action1
BRA Test
EXIT

[D2] > J?

Yes
No

Action1

EXIT
Template of Control Flow Structures

WHILE [D0] = [D1]
DO Action1

<table>
<thead>
<tr>
<th>Repeat</th>
<th>CMP</th>
<th>D0, D1</th>
<th>Perform test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BNE</td>
<td>EXIT</td>
<td>Done if D0 ≠ D1</td>
</tr>
<tr>
<td>Action1</td>
<td>BRA</td>
<td>Repeat</td>
<td>Repeat loop</td>
</tr>
<tr>
<td>EXIT</td>
<td></td>
<td></td>
<td>Exit point</td>
</tr>
</tbody>
</table>

Diagram:

- Start
- [D0] ≠ [D1]?
  - Yes
  - Repeat
  - EXIT
  - No
  - Action1
  - Repeat loop
  - EXIT

---

Clements, pp. 247
Template of Control Flow Structures

**REPEAT Action1**

**UNTIL** \([D0] = [D1]\)

<table>
<thead>
<tr>
<th>Action1</th>
<th>Perform Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>D0, D1</td>
</tr>
<tr>
<td>BNE</td>
<td>Action1</td>
</tr>
<tr>
<td>EXIT</td>
<td>Exit point</td>
</tr>
</tbody>
</table>

Start ➔ Action1 ➔ [D0] = D[1] ➔ Yes or No ➔ EXIT
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2
...

P = N Action N

P > N ErrorHandler

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP.B #N, P</td>
<td>Test for P out of range</td>
</tr>
<tr>
<td>BGT</td>
<td>ErrorHandler if P &gt; N then ErrorHandler</td>
</tr>
<tr>
<td>MOVE.W P, D0</td>
<td>Move P to D0</td>
</tr>
<tr>
<td>MULU #4, D0</td>
<td>Each address is a long word</td>
</tr>
<tr>
<td>LEA Table, A0</td>
<td>A0 points to table of addresses</td>
</tr>
<tr>
<td>LEA (A0,D0), A0</td>
<td>A0 now points to case P on table</td>
</tr>
<tr>
<td>MOVEA.L (A0), A0</td>
<td>A0 contains address of case P handler</td>
</tr>
<tr>
<td>JMP (A0)</td>
<td></td>
</tr>
</tbody>
</table>

Clements, pp. 247
Template of Control Flow Structures

CASE OF P

Assume P=2.

P = 0 Action0
P = 1 Action1
P = 2 Action2

... 

P = N Action N

P > N ErrorHandler

D0: 
A0: 

CMP.B #N, P Test for P out of range
BGT ErrorHandler If P > N then ErrorHandler
MOVE.W P, D0 Move P to D0
MULU #4, D0 Each address is a long word
LEA Table, A0 A0 points to table of addresses
LEA (A0,D0), A0 A0 now points to case P on table
MOVEA.L (A0), A0 A0 contains address of case P handler
JMP (A0)
## Template of Control Flow Structures

**CASE OF P**

<table>
<thead>
<tr>
<th>P = 0</th>
<th>Action0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P = 1</td>
<td>Action1</td>
</tr>
<tr>
<td>P = 2</td>
<td>Action2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>P = N</td>
<td>Action N</td>
</tr>
<tr>
<td>P &gt; N</td>
<td>ErrorHandler</td>
</tr>
</tbody>
</table>

**Assume P=2.**

<table>
<thead>
<tr>
<th>D0: 2</th>
<th></th>
</tr>
</thead>
</table>

**CMP.B** #N, P  
Test for P out of range

**BGT**  
ErrorHandler  
If P > N then ErrorHandler

**MOVE.W** P, D0  
Move P to D0

**MULU** #4, D0  
Each address is a long word

**LEA** Table, A0  
A0 points to table of addresses

**LEA** (A0,D0), A0  
A0 now points to case P on table

**MOVEA.L** (A0), A0  
A0 contains address of case P handler

**JMP** (A0)  
**Table**  
<table>
<thead>
<tr>
<th>Action0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Action1</td>
<td></td>
</tr>
<tr>
<td>Action2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ActionN</td>
<td></td>
</tr>
</tbody>
</table>

**ErrorHandler**
**Template of Control Flow Structures**

CASE OF P

<table>
<thead>
<tr>
<th>P</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Action0</td>
</tr>
<tr>
<td>1</td>
<td>Action1</td>
</tr>
<tr>
<td>2</td>
<td>Action2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N</td>
<td>Action N</td>
</tr>
</tbody>
</table>

P > N Error Handler

Assume P=2.

<table>
<thead>
<tr>
<th>D0:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0:</td>
<td></td>
</tr>
</tbody>
</table>

**Assembly Code:**

- **CMP.B** #N, P  Test for P out of range
- **BGT**          ErrorHandler  If P > N then ErrorHandler
- **MOVE.W** P, D0  Move P to D0
- **MULU** #4, D0  Each address is a long word
- **LEA** Table, A0  A0 points to table of addresses
- **LEA** (A0,D0), A0  A0 now points to case P on table
- **MOVEA.L** (A0), A0  A0 contains address of case P handler
- **JMP** (A0)
Template of Control Flow Structures

**CASE OF P**

- P = 0 Action0
- P = 1 Action1
- P = 2 Action2
- ... 
- P = N Action N
- P > N ErrorHandler

Assume P=2.

| D0: | 8 |

**CMP.B**  
#N, P  
Test for P out of range

**BGT**  
ErrorHandler  
If P > N then ErrorHandler

**MOVE.W**  
P, D0  
Move P to D0

**MULU**  
#4, D0  
Each address is a long word

**LEA**  
Table, A0  
A0 points to table of addresses

**LEA**  
(A0,D0), A0  
A0 now points to case P on table

**MOVEA.L**  
(A0), A0  
A0 contains address of case P handler

**JMP**  
(A0)
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2
...
P = N Action N
P > N ErrorHandler

Assume P=2.

D0: 8
A0:

CMP.B #N, P Test for P out of range
BGT ErrorHandler If P > N then ErrorHandler
MOVE.W P, D0 Move P to D0
MULU #4, D0 Each address is a long word
LEA Table, A0 A0 points to table of addresses
LEA (A0,D0), A0 A0 now points to case P on table
MOVEA.L (A0), A0 A0 contains address of case P handler
JMP (A0)
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2
...
P = N Action N

P > N ErrorHandler

Assume P=2.

D0: 8
A0: Table

CMP.B #N, P Test for P out of range
BGT ErrorHandler If P > N then ErrorHandler
MOVE.W P, D0 Move P to D0
MULU #4, D0 Each address is a long word
LEA Table, A0 A0 points to table of addresses
LEA (A0,D0), A0 A0 now points to case P on table
MOVEA.L (A0), A0 A0 contains address of case P handler
JMP (A0)

Clements, pp. 247

© 2006
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2
... 
P = N Action N

P > N ErrorHandler

Assume P=2.

D0: 8
A0:

CMP.B #N, P      Test for P out of range
BGT ErrorHandler If P > N then ErrorHandler
MOVE.W P, D0     Move P to D0
MULU #4, D0      Each address is a long word
LEA Table, A0    A0 points to table of addresses
LEA (A0,D0), A0  A0 now points to case P on table
MOVEA.L (A0), A0 A0 contains address of case P handler
JMP (A0)
### Template of Control Flow Structures

**CASE OF P**

<table>
<thead>
<tr>
<th>P = 0</th>
<th>Action 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P = 1</td>
<td>Action 1</td>
</tr>
<tr>
<td>P = 2</td>
<td>Action 2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>P = N</td>
<td>Action N</td>
</tr>
<tr>
<td>P &gt; N</td>
<td>ErrorHandler</td>
</tr>
</tbody>
</table>

Assume P = 2.

- **D0:** 8
- **A0:** Table + 8

```assembly
CMP.B  #N, P  ; Test for P out of range
BGT    ErrorHandler ; If P > N then ErrorHandler
MOVE.W P, D0  ; Move P to D0
MULU   #4, D0  ; Each address is a long word
LEA    Table, A0  ; A0 points to table of addresses
LEA    (A0,D0), A0  ; A0 now points to case P on table
MOVEA.L (A0), A0  ; A0 contains address of case P handler
JMP    (A0)  ; ErrorHandler
```
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2

P = N Action N

P > N ErrorHandler

Assume P=2.

D0: Table + 8

To set up the code for P = 2:

1. CMP.B #N, P: Test for P out of range
2. BGT ErrorHandler: If P > N then ErrorHandler
3. MOVE.W P, D0: Move P to D0
4. MULU #4, D0: Each address is a long word
5. LEA Table, A0: A0 points to table of addresses
6. LEA (A0,D0), A0: A0 now points to case P on table
7. MOVEA.L (A0), A0: A0 contains address of case P handler
8. JMP (A0): Jump to the address of case P handler
Template of Control Flow Structures

**CASE OF P**

- **P = 0** Action0
- **P = 1** Action1
- **P = 2** Action2
- ***
- **P = N** Action N
- **P > N** ErrorHandler

Assume P=2.

**D0:** 8

**A0:** Action2

**Table**

- **Table**
  - **Action0**
  - **Action1**
  - **Action2**
  - ***
  - **ActionN**

**ErrorHandler**

---

**CMP.B** #N, P Test for P out of range
**BGT** ErrorHandler If P > N then ErrorHandler
**MOVE.W** P, D0 Move P to D0
**MULU** #4, D0 Each address is a long word
**LEA** Table, A0 A0 points to table of addresses
**LEA** (A0,D0), A0 A0 now points to case P on table
**MOVEA.L** (A0), A0 A0 contains address of case P handler
**JMP** (A0)
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2

•••

P = N Action N

P > N ErrorHandler

Assume P=2.

D0: 8

A0: Action2

Table

CMP.B #N, P Test for P out of range
BGT ErrorHandler If P > N then ErrorHandler
MOVE.W P, D0 Move P to D0
MULU #4, D0 Each address is a long word
LEA Table, A0 A0 points to table of addresses
LEA (A0,D0), A0 A0 now points to case P on table
MOVEA.L (A0), A0 A0 contains address of case P handler
JMP (A0)
Template of Control Flow Structures

CASE OF P

P = 0 Action0
P = 1 Action1
P = 2 Action2
...

P = N Action N
P > N ErrorHandler

Assume P=2.

D0: 8

A0: Action2

CMP.B #N, P Test for P out of range
BGT ErrorHandler If P > N then ErrorHandler
MOVE.W P, D0 Move P to D0
MULU #4, D0 Each address is a long word
LEA Table, A0 A0 points to table of addresses
LEA (A0,D0), A0 A0 now points to case P on table
MOVEA.L (A0), A0 A0 contains address of case P handler
JMP (A0)
LOAD EFFECTIVE ADDRESS (LEA)

- Instead of moving the memory content pointed by an address into the register, it moves the address itself.

  Thus if A0 contains the address $4000 and D0 contains the data $0008, then the instruction

  $$\text{LEA (A0,D0), A0} \quad A0 \leftarrow A0+D0$$

  Writes the value $4008 into A0.
Difference Between MOVE and LEA

- It is important to understand the difference between MOVE and LEA:

  LEA  (A0, D0), A1  A1 ← A0+D0
  MOVE (A0,D0), D1  D1 ← (A0+D0)

  LEA writes the effective address into the destination register while MOVE writes the content of the memory position in the address.
Two ways to do the same thing

- The following two instruction sequences do the same thing:

  SEQ1    LEA    (A0,D0), A0    A0 ← A0+D0
          MOVEA.L (A0), A0    A0 ← (A0)

  SEQ2    MOVEA.L (A0,D0), A0    A0 ← (A0+D0)
# How to Implement the Dispatch Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA TABLE, A0</td>
<td>A0 contains memory address of first TABLE entry</td>
</tr>
<tr>
<td>LEA Action1, A1</td>
<td>A1 contains memory address of the code for Action1</td>
</tr>
<tr>
<td>MOVEA.L A1, (A0)+</td>
<td>Write address of Action1 in first TABLE entry and increment the pointer</td>
</tr>
<tr>
<td>LEA Action2, A1</td>
<td>A1 contains memory address of the code for Action2</td>
</tr>
<tr>
<td>MOVEA.L A1,(A0)</td>
<td>Write address of Action2 in second TABLE entry</td>
</tr>
<tr>
<td>BGT ErrorHandler</td>
<td>If P &gt; N then ErrorHandler</td>
</tr>
<tr>
<td>MOVE.W P, D0</td>
<td>Move P to D0</td>
</tr>
<tr>
<td>JMP (A0)</td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>&lt;code that follows the switch-case&gt;</td>
</tr>
</tbody>
</table>

**Actions**

- **Action1**
  - <code for Action1>
  - BRA EXIT
- **Action2**
  - <code for Action2>
  - BRA EXIT

**Table Declaration**

| TABLE | DS.L 2 | Reserves space for two long words in the location TABLE |
How to Implement the Dispatch Table

- **LEA** `TABLE, A0` A0 contains memory address of first TABLE entry
- **LEA** `Action1, A1` A1 contains memory address of the code for Action1
- **MOVEA.L** `A1, (A0)+` Write address of Action1 in first TABLE entry and increment the pointer
- **LEA** `Action2, A1` A1 contains memory address of the code for Action2
- **MOVEA.L** `A1,(A0)` Write address of Action2 in second TABLE entry
- **BGT** `ErrorHandler` If P > N then ErrorHandler
- **MOVE.W** `P, D0` Move P to D0
- **JMP** `(A0)`
- **EXIT** <code that follows the switch-case>
- **Action1** <code for Action1>
- **BRA** EXIT
- **Action2** <code for Action2>
- **BRA** EXIT
- **TABLE** `DS.L 2` Reserves space for two long words in the location TABLE
# How to Implement the Dispatch Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA TABLE, A0</td>
<td>A0 contains memory address of first TABLE entry</td>
</tr>
<tr>
<td>LEA Action1, A1</td>
<td>A1 contains memory address of the code for Action1</td>
</tr>
<tr>
<td>MOVEA.L A1, (A0)+</td>
<td>Write address of Action1 in first TABLE entry and increment the pointer</td>
</tr>
<tr>
<td>LEA Action2, A1</td>
<td>A1 contains memory address of the code for Action2</td>
</tr>
<tr>
<td>MOVEA.L A1,(A0)</td>
<td>Write address of Action2 in second TABLE entry</td>
</tr>
<tr>
<td>BGT ErrorHandler</td>
<td>If P &gt; N then ErrorHandler</td>
</tr>
<tr>
<td>MOVE.W P, D0</td>
<td>Move P to D0</td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>JMP (A0)</td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>&lt;code that follows the switch-case&gt;</td>
</tr>
<tr>
<td>Action1</td>
<td>&lt;code for Action1&gt;</td>
</tr>
<tr>
<td>BRA EXIT</td>
<td></td>
</tr>
<tr>
<td>Action2</td>
<td>&lt;code for Action2&gt;</td>
</tr>
<tr>
<td>BRA EXIT</td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td>DS.L 2 Reserves space for two long words in the location TABLE</td>
</tr>
</tbody>
</table>
How to Implement the Dispatch Table

LEA  TABLE, A0  A0 contains memory address of first TABLE entry
LEA  Action1, A1  A1 contains memory address of the code for Action1
MOVA.L  A1, (A0)+  Write address of Action1 in first TABLE entry and increment the pointer
LEA  Action2, A1  A1 contains memory address of the code for Action2
MOVA.L  A1,(A0)  Write address of Action2 in second TABLE entry
BGT  ErrorHandler  If P > N then ErrorHandler
MOVA.W  P, D0  Move P to D0
.....
JMP  (A0)
EXIT  <code that follows the switch-case>
Action1  <code for Action1>
         BRA  EXIT
Action2  <code for Action2>
         BRA  EXIT
TABLE  DS.L  2  Reserves space for two long words in the location TABLE
How to Implement the Dispatch Table

LEA TABLE, A0  
A0 contains memory address of first TABLE entry

LEA Action1, A1  
A1 contains memory address of the code for Action1

MOVEA.L A1, (A0)+  
Write address of Action1 in first TABLE entry and increment the pointer

LEA Action2, A1  
A1 contains memory address of the code for Action2

MOVEA.L A1,(A0)  
Write address of Action2 in second TABLE entry

BGT ErrorHandler  
If P > N then ErrorHandler

MOVE.W P, D0  
Move P to D0

.....

JMP (A0)

EXIT  
<code that follows the switch-case>

Action1  
<code for Action1>

BRA EXIT

Action2  
<code for Action2>

BRA EXIT

TABLE DS.L 2  
Reserves space for two long words in the location TABLE
Exercise

Write a code segment that reads a byte $B$ from the address 0x8400 0040 and:

a) writes 0x0000 0000 0000 00FF in address 0x8400 0044 if the bit 5 of $B$ is 1;

b) writes 0xFFFF FFFF FFFF FF00 in address 0x8400 0044 otherwise.