Chapter 2: Assemblers

System Software

Department of Computer Science
National Chiao-Tung University
Objectives

- In this section, you will
  - Understand design issues for assembler
    - Hardware dependent
    - Hardware independent
  - Understand data structures and algorithms commonly adopted in assemblers
    - OPTAB
    - SYMTAB
    - On-pass and multi-pass assembling
  - Understand the format of an object file
  - Understand the concept of program relocation
Assemblers

• Translate assembly language into machine codes
  – Machine independent features
    • OPcode lookup
    • Symbol address resolving
    • Program linking
  – Machine dependent features
    • Instruction format encoding
    • Addressing mode assignment

• Design issues
  – One-pass or multi-pass
SIC Assembly

• Index addressing is indicated by adding the modifier “,X”
  – STCH BUFFER,X
• Lines beginning with . are comments
• Assembler directives
  – No executable machine codes will be assembled
  – START: specify the starting address of a program
  – END: specify the end of a program with the address of the first instruction to execute
  – BYTE: generate a one-byte constant
  – WORD: generate a one-word constant
  – RESB: reserve a number of bytes
  – RESW: reserve a number of words
SIC Assembly

• Figure 2.1:
  – Read records from a device 0xf1 and copy them to the other device 0x05
  – Record structure
    • Each records read from 0xf1 is terminated by 0x00
    • If a record begins with 0, then there are no more data to read
  – Operation
    • A record read from 0xf1 is stored in a buffer
      – BYTE BUFFER[4096]
      – LENGTH to indicate data length
    • The record is then written to 0x05
    • Finally, ‘E’O’F’ are written to 0x05
  – Remark
    • The program first remembers L register filled by OS, and then restores it as the return address of the program
<table>
<thead>
<tr>
<th>Line</th>
<th>Source statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>COPY, START 1000</td>
</tr>
<tr>
<td>10</td>
<td>FIRST, STL, RETADR</td>
</tr>
<tr>
<td>15</td>
<td>CLOOP, JSUB RDREC</td>
</tr>
<tr>
<td>20</td>
<td>LDA LENGTH, COMP ZERO</td>
</tr>
<tr>
<td>25</td>
<td>JEQ ENDFIL, EXIT IF EOF FOUND</td>
</tr>
<tr>
<td>30</td>
<td>JSUB WRREC, WRITE OUTPUT RECORD</td>
</tr>
<tr>
<td>35</td>
<td>J CLOOP, LOOP</td>
</tr>
<tr>
<td>40</td>
<td>ENDFIL, LDA EOF</td>
</tr>
<tr>
<td>45</td>
<td>STA BUFFER, INSERT END OF FILE MARKER</td>
</tr>
<tr>
<td>50</td>
<td>LDA THREE, SET LENGTH = 3</td>
</tr>
<tr>
<td>55</td>
<td>STA LENGTH, WRITE EOF</td>
</tr>
<tr>
<td>60</td>
<td>JSUB WRREC, GET RETURN ADDRESS</td>
</tr>
<tr>
<td>65</td>
<td>LDL RETADR, RETURN TO CALLER</td>
</tr>
<tr>
<td>70</td>
<td>RSUB, BYTE C’EOF’</td>
</tr>
<tr>
<td>75</td>
<td>THREE, WORD 3</td>
</tr>
<tr>
<td>80</td>
<td>ZERO, WORD 0</td>
</tr>
<tr>
<td>85</td>
<td>RETADR, RESW 1</td>
</tr>
<tr>
<td>90</td>
<td>LENGTH, RESW 1</td>
</tr>
<tr>
<td>95</td>
<td>BUFFER, RESB 4096</td>
</tr>
</tbody>
</table>

COPY FILE FROM INPUT TO OUTPUT
SAVE RETURN ADDRESS
READ INPUT RECORD
TEST FOR EOF (LENGTH = 0)
EXIT IF EOF FOUND
WRITE OUTPUT RECORD
LOOP
INSERT END OF FILE MARKER
SET LENGTH = 3
WRITE EOF
GET RETURN ADDRESS
RETURN TO CALLER
LENGTH OF RECORD
4096-BYTE BUFFER AREA
SIC Assembly

• Highlights
  – LDL and STL
    • Nested function call must be specially handled in SIC because there is no stack pointer
  – JSUB and RSUB for calling and exiting subroutines, respectively
  – Labels for jump
  – Directives to reserve memory space
  – Directives for program start, program end, the place where the execution begins
    • START and END
    • The first line of a program is not necessarily the first instruction to be executed

• In assembly, it is very hard to know which registers serve as “parameter” to subroutines
  – In practice “push all” and “pop all” are often used
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td></td>
<td>SUBROUTINE TO READ RECORD INTO BUFFER</td>
</tr>
<tr>
<td>115</td>
<td>RDREC</td>
<td>LDX ZERO CLEAR LOOP COUNTER</td>
</tr>
<tr>
<td>120</td>
<td></td>
<td>LDA ZERO CLEAR A TO ZERO</td>
</tr>
<tr>
<td>125</td>
<td>RLOOP</td>
<td>TD INPUT TEST INPUT DEVICE</td>
</tr>
<tr>
<td>130</td>
<td></td>
<td>JEQ RLOOP LOOP UNTIL READY</td>
</tr>
<tr>
<td>135</td>
<td>RD</td>
<td>INPUT READ CHARACTER INTO REGISTER A</td>
</tr>
<tr>
<td>140</td>
<td>COMP</td>
<td>ZERO TEST FOR END OF RECORD (X'00')</td>
</tr>
<tr>
<td>145</td>
<td></td>
<td>JEQ EXIT EXIT LOOP IF EOR</td>
</tr>
<tr>
<td>150</td>
<td>STCH</td>
<td>BUFFER,X STORE CHARACTER IN BUFFER</td>
</tr>
<tr>
<td>155</td>
<td></td>
<td>TIX MAXLEN LOOP UNLESS MAX LENGTH HAS BEEN REACHED</td>
</tr>
<tr>
<td>160</td>
<td></td>
<td>JLT RLOOP</td>
</tr>
<tr>
<td>165</td>
<td>EXIT</td>
<td>STX LENGTH SAVE RECORD LENGTH</td>
</tr>
<tr>
<td>170</td>
<td></td>
<td>RSUB RETURN TO CALLER</td>
</tr>
<tr>
<td>175</td>
<td>INPUT</td>
<td>BYTE X'F1' CODE FOR INPUT DEVICE</td>
</tr>
<tr>
<td>180</td>
<td>MAXLEN</td>
<td>WORD 4096</td>
</tr>
</tbody>
</table>
SUBROUTINE TO WRITE RECORD FROM BUFFER

210  WRREC  LDX  ZERO  CLEAR LOOP COUNTER
215  WLOOP  TD   OUTPUT  TEST OUTPUT DEVICE
220  JEQ    WLOOP  LOOP UNTIL READY
225  LDCH   BUFFER,X  GET CHARACTER FROM BUFFER
230  WD     OUTPUT  WRITE CHARACTER
235  TIX    LENGTH  LOOP UNTIL ALL CHARACTERS
240  JLT    WLOOP  HAVE BEEN WRITTEN
245  RSUB   END   RETURN TO CALLER
250  OUTPUT  BYTE  x'05'  CODE FOR OUTPUT DEVICE
255  END     FIRST
A Simple SIC Assembler

- The translating of source programs (assembly) to object code (binary) requires:
  - Convert mnemonics to binary code
    - STL \( \rightarrow 0x14 \)
  - Resolving symbols into target addresses
    - RETADDR \( \rightarrow 0x1033 \)
  - Evaluating constants
  - Assemble instructions with correct formats
<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1000</td>
<td>COPY START 1000</td>
<td>141033</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>FIRST STL RETADR</td>
<td>482039</td>
</tr>
<tr>
<td>15</td>
<td>1003</td>
<td>CLOOP JSUB RDREC</td>
<td>001036</td>
</tr>
<tr>
<td>20</td>
<td>1006</td>
<td>LDA LENGTH</td>
<td>281030</td>
</tr>
<tr>
<td>25</td>
<td>1009</td>
<td>COMP ZERO</td>
<td>301015</td>
</tr>
<tr>
<td>30</td>
<td>100C</td>
<td>JEQ ENDFIL</td>
<td>482061</td>
</tr>
<tr>
<td>35</td>
<td>100F</td>
<td>JSUB WRREC</td>
<td>3C1003</td>
</tr>
<tr>
<td>40</td>
<td>1012</td>
<td>J CLOOP</td>
<td>00102A</td>
</tr>
<tr>
<td>45</td>
<td>1015</td>
<td>ENDFIL LDA EOF</td>
<td>0C1039</td>
</tr>
<tr>
<td>50</td>
<td>1018</td>
<td>STA BUFFER</td>
<td>00102D</td>
</tr>
<tr>
<td>55</td>
<td>101B</td>
<td>LDA THREE</td>
<td>0C1036</td>
</tr>
<tr>
<td>60</td>
<td>101E</td>
<td>STA LENGTH</td>
<td>482061</td>
</tr>
<tr>
<td>65</td>
<td>1021</td>
<td>JSUB WRREC</td>
<td>081033</td>
</tr>
<tr>
<td>70</td>
<td>1024</td>
<td>LDL RETADR</td>
<td>4C0000</td>
</tr>
<tr>
<td>75</td>
<td>1027</td>
<td>RSUB</td>
<td>454F46</td>
</tr>
<tr>
<td>80</td>
<td>102A</td>
<td>EOF BYTE C'EOF'</td>
<td>000003</td>
</tr>
<tr>
<td>85</td>
<td>102D</td>
<td>THREE WORD 3</td>
<td>000000</td>
</tr>
<tr>
<td>90</td>
<td>1030</td>
<td>ZERO WORD 0</td>
<td>1</td>
</tr>
<tr>
<td>95</td>
<td>1033</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1036</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>1039</td>
<td>BUFFER RESB 4096</td>
<td></td>
</tr>
</tbody>
</table>
SUBROUTINE TO READ RECORD INTO BUFFER

110  2039  RDREC  LDX  ZERO  041030
115  203C  LDA  ZERO  001030
120  203F  RLOOP  TD  INPUT  E0205D
125  2042  JEQ  RLOOP  30203F
130  2045  RD  INPUT  D8205D
135  2048  COMP  ZERO  281030
140  204B  JEQ  EXIT  302057
145  204E  STCH  BUFFER,X  549039
150  2051  TIX  MAXLEN  2C205E
155  2054  JLT  RLOOP  38203F
160  2057  EXIT  STX  LENGTH  101036
165  205A  RSUB  4C0000
170  205D  INPUT  BYTE  X'F1'  F1
175  205E  MAXLEN  WORD  4096  001000

SUBROUTINE TO WRITE RECORD FROM BUFFER

210  2061  WRREC  LDX  ZERO  041030
215  2064  WLOOP  TD  OUTPUT  E02079
220  2067  JEQ  WLOOP  302064
225  206A  LDCH  BUFFER,X  509039
230  206D  WD  OUTPUT  DC2079
235  2070  TIX  LENGTH  2C1036
240  2073  JLT  WLOOP  382064
245  2076  RSUB  4C0000
250  2079  OUTPUT  BYTE  X'05'  05
255  END  FIRST
A Simple SIC Assembler

• Resolving target addresses of operands
  – Most assemblers read and process source programs line by line
  – Some operands could be used before they are actually defined
    • A.K.A. Forward reference
      FIRST STL RETADR
    • Many assemblers take scan the source program so as to deal with forward reference
      – We shall show how to do it later…
A Simple SIC Assembler

- Generate object code
  - Object files contain machine interpretable binary data
    - Binary data could be instructions or data
  - Binary data are loaded into memory for the processor to execute
    - How data are loaded is controlled by special records in the object file
      - Where and how long
  - Header records, text records, and end records
### A Simple SIC Assembler

#### Record formats

<table>
<thead>
<tr>
<th>Header Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col. 1</td>
</tr>
<tr>
<td>Col. 2-7</td>
</tr>
<tr>
<td>Col. 8-13</td>
</tr>
<tr>
<td>Col. 14-19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col. 1</td>
</tr>
<tr>
<td>Col. 2-7</td>
</tr>
<tr>
<td>Col. 8-9</td>
</tr>
<tr>
<td>Col. 10-69</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>End Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col. 1</td>
</tr>
<tr>
<td>Col. 2-7</td>
</tr>
</tbody>
</table>
A Simple SIC Assembler

- ^ is for illustration only
- There is no object code to load between 0x1033-0x2038
  - The loader could reserve the space for variables, as the assembly program indicates

```
HCOPY 00100000107A
T0010001E14103348203900103628103010154820613C100300102A0C103900102D
T00101E150C10364820610810334C000454F46000003000000
T0020391E041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T0020571C1010364C000F1001000041030E02079302064509039DC20792C1036
T002073073820644C000005
E001000
```
A Simple SIC Assembler

- Given formats of input (assembly programs) and output (object files), a simple assembler could be
  - Pass 1: define symbols
    - Assign addresses for instructions
    - Assign addresses for symbols
    - Perform assembler directives such as START, RESB, RESW, WORD, BYTE...
  - Pass 2: assemble instructions and write object program
    - Encode instructions in terms of opcodes, target addresses, addressing modes as different formats
    - Generate data for constants
    - Perform some directives not handled in pass 1
    - Write object programs and listing files
Assembler Algorithms and Data Structures

- **OPTAB**
  - Translate mnemonics into binary opcodes
    - ADD → 0x18
  - Usually read-only
  - Should be efficient on retrieval
  - Also remembers formats and lengths of corresponding instructions
    - E.g., ADDR → 0x90, format 2, 2 bytes
  - Usually implemented as a static hash table
Assembler Algorithms and Data Structures

• **SYMTAB**
  - Translate symbol names into target addresses
    - E.g., in Fig 2.6, CLOOP→0x0006 and BUFFER→0x0036
  - Both lookups and insertions are needed
  - Remembers types (byte or word) and length of symbols
    - E.g., symbol EOF is of 3 bytes and RETADR is of 1 word
  - Usually implemented as a hash table
    - Must be efficient to deal with common prefix

• **LOCCTR**
  - Help to assign addresses to symbols
  - Initialized as 0
  - Incremented by the length in bytes of the instruction just processed
Assembler Algorithms and Data Structures

- Intermediate files
  - Files storing temporary information between pass 1 and pass 2

- Listing files
  - Have detailed information about symbol addresses, encoded object code
  - Source code and object code are listed side-by-side for reference
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #\[OPERAND\] as starting address
      initialize LOCCTR to starting address
      write line to intermediate file
      read next input line
    end {if START}
  else
    begin
      initialize LOCCTR to 0
    end {else}
  while OPCODE <> 'END' do
    begin
      if this is not a comment line then
        begin
          if there is a symbol in the LABEL field then
            begin
              search SYMTAB for LABEL
              if found then
                set error flag (duplicate symbol)
              else
                insert (LABEL, LOCCTR) into SYMTAB
              end {if symbol}
              search OPTAB for OPCODE
              if found then
                add 3 \{instruction length\} to LOCCTR
              else if OPCODE = 'WORD' then
                add 3 to LOCCTR
              else if OPCODE = 'RESW' then
                add 3 * #\[OPERAND\] to LOCCTR
              else if OPCODE = 'RESB' then
                add #\[OPERAND\] to LOCCTR
              else if OPCODE = 'BYTE' then
                begin
                  find length of constant in bytes
                  add length to LOCCTR
                end {if BYTE}
              else
                set error flag (invalid operation code)
            end {if not a comment}
          write line to intermediate file
          read next input line
        end {while not END}
      write last line to intermediate file
      save (LOCCTR - Starting address) as program length
    end {Pass 1}
begin
  read first input line {from intermediate file}
  if OPCODE = 'START' then
    begin
      write listing line
      end {if START}
    end {if START}
  write Header record to object program
  initialize first Text record
  while OPCODE <> 'END' do
    begin
      if this is not a comment line then
        begin
          search OPTAB for OPCODE
          if found then
            begin
              if there is a symbol in OPERAND field then
                begin
                  search SYMTAB for OPERAND
                  if found then
                    store symbol value as operand address
                  else
                    begin
                      store 0 as operand address
                      set error flag (undefined symbol)
                    end
                  end {if symbol}
                else
                  store 0 as operand address
                  assemble the object code instruction
                  end {if opcode found}
                else if OPCODE = 'BYTE' or 'WORD' then
                  convert constant to object code
                  if object code will not fit into the current Text record then
                    begin
                      write Text record to object program
                      initialize new Text record
                    end
                    add object code to Text record
                  end {if not comment}
                end {if OPCODE <> 'END'}
              write last Text record to object program
              write end record to object program
            end
          end {while not END}
        end
      end
  end {Pass 2}

How about RESB, RESW?
Machine-Dependent Assembler Features

- SIC/XE is used to explain machine-dependent issues
  - Indirect addressing is indicated by prefix @
    J @RETADR
  - Immediate values are with prefix #
    LDA #3
  - The extended instruction format is indicated by prefix +
    +LDT #4096
    • Format 3 has only 12 bits for operands
  - By default, SIC/XE assembly uses format-3 relative addressing
    • Sometimes target addresses are not reachable. In this case, programmers should consider to use format-4 instructions.
    • The assembler won’t automatically convert instructions from format-3 to format-4!!!
<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000</td>
<td>COPY START 0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0000</td>
<td>FIRST STL RETADR</td>
<td>17202D</td>
</tr>
<tr>
<td>12</td>
<td>0003</td>
<td>LDB #LENGTH</td>
<td>69202D</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>BASE LENGTH</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0006</td>
<td>CLOOP +JSUB RDREC</td>
<td>4B101036</td>
</tr>
<tr>
<td>20</td>
<td>000A</td>
<td>LDA LENGTH</td>
<td>032026</td>
</tr>
<tr>
<td>25</td>
<td>000D</td>
<td>COMP #0</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0010</td>
<td>JEQ ENDFIL</td>
<td>332007</td>
</tr>
<tr>
<td>35</td>
<td>0013</td>
<td>+JSUB WRREC</td>
<td>4B10105D</td>
</tr>
<tr>
<td>40</td>
<td>0017</td>
<td>J CLOOP</td>
<td>3F2FEC</td>
</tr>
<tr>
<td>45</td>
<td>001A</td>
<td>ENDFIL LDA EOF</td>
<td>032010</td>
</tr>
<tr>
<td>50</td>
<td>001D</td>
<td>STA BUFFER</td>
<td>0F2016</td>
</tr>
<tr>
<td>55</td>
<td>0020</td>
<td>LDA #3</td>
<td>010003</td>
</tr>
<tr>
<td>60</td>
<td>0023</td>
<td>STA LENGTH</td>
<td>0F200D</td>
</tr>
<tr>
<td>65</td>
<td>0026</td>
<td>+JSUB WRREC</td>
<td>4B10105D</td>
</tr>
<tr>
<td>70</td>
<td>002A</td>
<td>J @RETADR</td>
<td>3E2003</td>
</tr>
<tr>
<td>80</td>
<td>002D</td>
<td>EOF BYTE C’EOF’</td>
<td>454F46</td>
</tr>
<tr>
<td>95</td>
<td>0030</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0033</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0036</td>
<td>BUFFER RESB 4096</td>
<td></td>
</tr>
</tbody>
</table>
SUBROUTINE TO READ RECORD INTO BUFFER

125  1036   RDREC   CLEAR   X       B410
130  1038   CLEAR   A       B400
132  103A   CLEAR   S       B440
133  103C   +LDT   #4096   75101000
135  1040   RLOOP   TD      INPUT   E32019
140  1043   JEQ      RLOOP   332FFA
145  1046   RD      INPUT   DB2013
150  1049   COMPR   A,S     A004
155  104B   JEQ      EXIT    332008
160  104E   STCH     BUFFER,X 57C003
165  1051   TIXR     T       B850
170  1053   JLT      RLOOP   3B2FEA
175  1056   EXIT     STX     LENGTH 134000
180  1059   RSUB     RSUB    4F0000
185  105C   INPUT    BYTE    X'F1'  F1

SUBROUTINE TO WRITE RECORD FROM BUFFER

195

200  105D   WRREC   CLEAR   X       B410
205  105F   LDT      LENGTH   774000
210  1062   WLOOP    TD      OUTPUT   E32011
215  1065   JEQ      WLOOP   332FFA
220  1068   LDCH     BUFFER,X 53C003
225  106B   WD       OUTPUT   DF2008
230  106E   TIXR     T       B850
235  1070   JLT      WLOOP   3B2FEF
240  1073   RSUB     RSUB    4F0000
245  1076   OUTPUT   BYTE    X'05'  05
250  1079   END      FIRST
255
Instruction Formats and Addressing Modes

• The START specifies the begin address at 0
  – It is to generate re-locatable programs
  – The loader takes care of where the begin address of programs

• Register-to-register instructions are heavily used
  – Register names are usually put in SYMTAB

• Register-to-memory instructions use extended format or relative addressing
  ✓ If extended format is specified, use it
  ✓ Or target addresses are calculated relatively to PC or B
  ✓ If target addresses are too far from the reach of PC or B, errors are generated
Instruction Formats and Addressing Modes

- Extended instruction format could be used to address data residing far from (B) and (PC)

  15 0006 CLOOP +JSUB RDREC  4B101036
  - The symbol “BUFFER” of 4096 bytes gets in the way, too far to reach by relative addressing
  - Long instructions could also be used to load large immediate values

  113 103C +LDT #4096 75101000
Instruction Formats and Addressing Modes

- If format-4 prefix is not specified, then
  - **PC-relative** addressing is considered
    - -2048~+2047
  - If the target address is can’t be reached by PC-relative addressing, **base-relative** addressing is then considered
    - 0~+4095
  - If both fail, an **error** is generated
    - It’s programmers’ responsibility to
      - set the base register with proper target address
      - Put data near instructions which access the data
Instruction Formats and Addressing Modes

• PC-relative addressing

10 0000 FIRST STL RETADR 17202D
40 0017 J CLOOP 3F2FEC

• Program counter (PC) is incremented after an instruction is fetched and before the instruction is executed

  – Fetch ➔ PC+(n), Decoding ➔ Execute
    • Fetch ➔ PC+(n), Decoding ➔ [Fetch operands] ➔ Execute ➔ [Store result]
  – When fetching operands from target addresses, the PC had already been incremented
    • 2D = 30-3, 6-1A = -14
Instruction Formats and Addressing Modes

• Base-relative addressing
  – Register B could be used as either a temporary storage or a reference for relative addressing
  • PC always refers to the memory address of the next instruction to execute
  • (B) can be configured as a reference for relative addressing by using assembler directive BASE
    – The programmer has his freedom to refer B to any legal memory address by explicitly loading content into B
• If B is used for storage, use NOBASE to tell assembler not to encode instructions in terms of base-relative addressing
Instruction Formats and Addressing Modes

12 0003       LDB  #LENGTH   69202D
13            BASE LENGTH

• Line 12: read the address of LENGTH instead of its content
  – Note that this instruction uses PC-relative addressing
• Line 13: tell the assembler the following instructions could be encoded in terms of base-relative addressing, where (B) is the address of symbol LENGTH

160 104E      STCH BUFFER,X  57C003

• Line 160: symbol BUFFER is out of the reach of PC and therefore base-relative addressing is used
  – In conjunction with indexed addressing

Format 3 (3 bytes):

```
<table>
<thead>
<tr>
<th>op</th>
<th>n</th>
<th>i</th>
<th>x</th>
<th>b</th>
<th>p</th>
<th>e</th>
<th>disp</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```
Instruction Formats and Addressing Modes

```
175 1056 EXIT STX LENGTH 134000
```

- Line 175: displacement is 0

```
20 000A  LDA LENGTH 032026
```

- Line 20: symbol LENGTH is reachable from PC (0D→33) and therefore PC-relative addressing is used
Program Relocation

• It is desirable to load multiple programs into memory
  – multiprogramming

• In most cases, the invocations of programs are not known until run-time
  – Absolute addressing causes conflictions among storage spaces of programs

• It is desirable to make programs relocatable anywhere in memory
Program Relocation

20 1006      LDA LENGTH    001036

- Direct addressing (SIC), not re-locatable

20 000A      LDA LENGTH    032026

- PC-relative addressing (SIC/XE format 3), re-locatable
Program Relocation

• Absolute addressing could be used by
  – SIC instructions
  – Format 4 SIC/XE instructions

• Immediate addressing and register-to-register addressing are, surely, re-locatable

• If an instruction uses absolute addressing, its target address for the operand must be “adjusted” for relocation
  – It is impossible to tell re-locatable instructions from non re-locatable ones, unless
    • the loader simulates program execution. It is impractical.
  – Assemblers generate some special records in object programs to direct loaders to perform such adjustments
Diagram showing the layout of memory addresses and instructions with offsets. Sections labeled (a), (b), and (c) illustrate the hierarchical structure of memory space with specific addresses and offsets marked. Instructions and data are arranged to demonstrate the flow and structure of the program.
Program Relocation

- When an instruction using absolute addressing is assembled, the assembler should
  - calculate the target address of the operand relative to the beginning of the program (i.e., 0)
  - Generate a record so that the instruction can be “modified” by the loader upon loading

<table>
<thead>
<tr>
<th>Modification Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col. 1</td>
</tr>
<tr>
<td>Col. 2-7</td>
</tr>
<tr>
<td>Col. 8-9</td>
</tr>
</tbody>
</table>
Program Relocation

• The beginning of the address field could be of odd half-bytes, in the case
  – the beginning of this field is on half-byte boundaries
  – the offset field of an M record (in terms of bytes) is related to the first half-byte rather than the zeroth half-byte

![Diagram showing the difference between odd and correct half-bytes in the address field.](image)
Program Relocation

- A re-locatable object program
Program Relocation

• Not all format-4 instructions need modification records
  – +LDA #4096

• Of course, instructions using PC-relative, base-relative addressing, and register-to-register addressing need no modification records

• For an SIC assembly program, almost all instructions need modification records

• With modification records, the starting address of text records are interpreted as being relative to the beginning of the program
Machine Independent Assembler Features

- Most of machine independent features are for memory address calculations and symbol defining
  - Literals
  - Symbol-defining directives
  - Expressions
  - Program blocks
  - Control section and linking
<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000</td>
<td>COPY START 0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0000</td>
<td>FIRST STL RETADR</td>
<td>17202D</td>
</tr>
<tr>
<td>13</td>
<td>0003</td>
<td>LDB #LENGTH</td>
<td>69202D</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>BASE LENGTH</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0006</td>
<td>CLOOP +JSUB RDREC</td>
<td>4B101036</td>
</tr>
<tr>
<td>20</td>
<td>000A</td>
<td>LDA LENGTH</td>
<td>032026</td>
</tr>
<tr>
<td>25</td>
<td>000D</td>
<td>COMP #0</td>
<td>290000</td>
</tr>
<tr>
<td>30</td>
<td>0010</td>
<td>JEQ ENDFIL</td>
<td>332007</td>
</tr>
<tr>
<td>35</td>
<td>0013</td>
<td>+JSUB WRREC</td>
<td>4B10105D</td>
</tr>
<tr>
<td>40</td>
<td>0017</td>
<td>J CLOOP</td>
<td>3F2FEC</td>
</tr>
<tr>
<td>45</td>
<td>001A</td>
<td>ENDFIL LDA =C’EOF’</td>
<td>032010</td>
</tr>
<tr>
<td>50</td>
<td>001D</td>
<td>STA BUFFER</td>
<td>0F2016</td>
</tr>
<tr>
<td>55</td>
<td>0020</td>
<td>LDA #3</td>
<td>010003</td>
</tr>
<tr>
<td>60</td>
<td>0023</td>
<td>STA LENGTH</td>
<td>0F200D</td>
</tr>
<tr>
<td>65</td>
<td>0026</td>
<td>+JSUB WRREC</td>
<td>4B10105D</td>
</tr>
<tr>
<td>70</td>
<td>002A</td>
<td>J @RETADR</td>
<td>3E2003</td>
</tr>
<tr>
<td>93</td>
<td>002D</td>
<td>LTORG</td>
<td>454F46</td>
</tr>
<tr>
<td>95</td>
<td>0030</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0033</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0036</td>
<td>BUFFER RESB 4096</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>1036</td>
<td>BUFEND EQU *</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>1000</td>
<td>MAXLEN EQU BUFEND-BUFFER</td>
<td></td>
</tr>
</tbody>
</table>
SUBROUTINE TO READ RECORD INTO BUFFER

1036 RDREC CLEAR X B410
1038 CLEAR A B400
103A CLEAR S B440
103C +LDT #MAXLEN 75101000
1040 RLOOP TD INPUT E32019
1043 JEQ RLOOP 332FFA
1046 RD INPUT DB2013
1049 COMPR A, S A004
104B JEQ EXIT 332008
104E STCH BUFFER, X 57C003
1051 TIXR T B850
1053 JLT RLOOP 3B2FEA
1056 EXIT STX LENGTH 134000
1059 RSUB 4F0000
105C INPUT BYTE X’F1’ F1

SUBROUTINE TO WRITE RECORD FROM BUFFER

105D WRREC CLEAR X B410
105F LDT LENGTH 774000
1062 WLOOP TD =X’05’ E32011
1065 JEQ WLOOP 332FFA
1068 LDCH BUFFER, X 53C003
106B WD =X’05’ DF2008
106E TIXR T B850
1070 JLT WLOOP 3B2FEF
1073 RSUB 4F0000
1076 * =X’05’ 05
Literals

- Literals representing constants in instructions without explicitly defining a symbol and reserving memory space for the constant
  - Must be careful with the differences among symbols, immediate values, and literals

- The value of a literal could be directly interpreted from itself

  45  001A ENDFIL LDA  =C’EOF’  032010
  215 1062 WLOOP  TD  =X’05’  E32011

  - Prefix ‘=‘ is used literals
  - It could be noticed that literals are not encoded as immediate values
Literals

- Brain damage
  - Immediate operands: The target addresses of the operands or the constant they specify are encoded as a part of instructions
  - Literals: memory space is implicitly reserved and constants are accessed via memory
    - Symbols with content defined by themselves
- Literals in fact provide only convenience to programmers
  
  ```
  ENDFIL LDA =C’EOF’
  ENDFIL LDA EOF
  EOF RESB C’EOF’
  ENDFIL LDA #X’454F46’ ← if permitted?
  ```
- Immediate operands, on the other hand, decide how instructions should be encoded
Literals

- The assembler will gather all literals used in a program and reserve memory space for them in a literal pool
  - Normally the literal pool is implicitly put at the end of a program
  - A program could have one or more literal pools
    - Literal pools could be explicitly introduced by programmers
    - The reason to do so is to prevent from literals being too far away from instructions using them
  - Literal pool could be generated by LTORG directive
Literals

• **LTORG**
  - Reserve memory space for new literals since the last LTORG or the beginning of program
    
    93                LTORG
    002D   *      =C’EOF’    454F46
  
  - An implicit LTORG directive is handled at the end of program to flush literals which are not handled so far
  
  - Missing LTORG in line 93 of Figure 2.10 will cause the literal =C’EOF’ being too far away from line 45 for PC-relative addressing
    
    • Because 4096B BUFFER gets in the way
Literals

• Literals with different strings sometimes refer to the same value
  – =C’EOF’ and =X’454F46’
  – To identify (logically) identical literals is a complicated task for assemblers

• Sometimes literals of the same name refer to different values
  – Symbol * refers to the current value of program counter

\[
\begin{align*}
\text{BASE} & \quad * \\
\text{LDB} & \quad *=
\end{align*}
\]
  – Symbol * refers to the current value of program counter, on line 13 and line 55 refer to 0003 and 0020, respectively
Literals

- A new data structure LITTAB is adopted for the handling of literals
  - Indexed by literal names (which actually refer to values of them)

- Pass 1
  - When a literal is recognized in operands, LITTAB is checked
    - If found, nothing to do
    - If not found, add a new entry for it
  - A literal pool is created for literals which are not assigned to memory addresses when
    - LTORG is handled or
    - the end of program is reached

- Pass 2
  - When a literal is recognized in operands, LITTAB is checked
    - Get its target address assigned in Pass 1 from LITTAB
    - If literals are location-dependent (such as *), modification records are generated accordingly
      - Why?
Symbol-Defining Directives

• Why symbol-defining directives are useful?
  – We could assign meaningful names to constants to improve readability
    
    +LDT #4096
    
    MAXLEN EQU 4096
    
    +LDT #MAXLEN
    
    MAXLEN EQU 4096

  – Instead of stating the value of a constant everywhere where it is referenced, a symbol could be created to unify them
    • Avoid potential errors when the program is revised with a new value of the constant
      – Since programmers might forget to modify some of them and, consequently, run-time errors could be suffered from
Symbol-Defining Directives

- SYMTAB could also be used to deal with symbols defined by directive EQU
- A symbol must be defined as a previously defined symbol, a constant, an expression, or a literal
  - MAXLEN EQU 4096
  - BUFEND EQU *
  - MAXLEN EQU BUFEND–BUFFER

- No memory space is reserved for them
Symbol-Defining Directives

• Forward reference is not permitted!

\[
\begin{align*}
\text{ALPHA} & \equiv \text{BETA} \\
\text{BETA} & \equiv \text{DELTA} \\
\text{DELTA} & \equivsw 1 \\
\end{align*}
\]

– The target address of a symbol might depend on another symbol
  • ALPHA depends on BETA, and BETA in turn depends on DELTA

– Not target addresses of all symbols could be resolved in the first pass
  • Multiple passes might be required

• Cyclic dependency is an error
Symbol-Defining Directives

- An example that exercises EQU:
  - Accessing a collection of 100 structures, in which one structure is of fields SYMBOL, VALUE, and FLAGS

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>VALUE</th>
<th>FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- LDA VALUE, X
Symbol-Defining Directives

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>VALUE</th>
<th>FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• A collection of structures could better be represented by using a new directive ORG
  – ORG is to reset LOCCTR to a desired value or the target address of an operand
    • ORG value
  – Simply specifying “ORG” in assembly tells the assembler to restore the old value of LOCCTR before “ORG something” is processed

• We could define symbols over previously reserved memory space by using ORG
Symbol-Defining Directives

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAB</td>
<td>RESB</td>
<td>1100</td>
</tr>
<tr>
<td>ORG</td>
<td>STAB</td>
<td></td>
</tr>
<tr>
<td>SYMBOL</td>
<td>RESB</td>
<td>6</td>
</tr>
<tr>
<td>VALUE</td>
<td>RESW</td>
<td>1</td>
</tr>
<tr>
<td>FLAGS</td>
<td>RESB</td>
<td>2</td>
</tr>
<tr>
<td>ORG</td>
<td>STAB+1100</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>VALUE,X</td>
<td></td>
</tr>
</tbody>
</table>

- The type and size of a field could better be stated in assembly in terms of readability.
- No forward reference should be applied to ORG, because it causes that LOCCTR depends on the symbol specified in the operand.
Expressions

- The value of a symbol could be an arithmetic expression over several terms
  - +, -, *, /
  - Constants, symbols, special terms (e.g., *)

**Be very careful with the terminology in the following contents**

- A relative term refers one being relevant to the beginning location of the program
  - Relative terms are not allowed to be involved in * or / → (a)
  - Any expression must have at most one relative term and it must be associated with + → (b)

- An absolute term is independent of where the program is loaded
  - If relative terms are paired in opposite operations (+ and -) then the resulted expression is absolute
    - To cancel the dependency of the beginning location of the program

- Violating either (a) or (b) is an error
Expressions

107 \text{ MAXLEN} \text{ EQU} \text{ BUFFEND-BUFFER}

- MAXLEN is an absolute term, calculating the difference between BUFFEND and BUFFER

- \text{BUFFEND+BUFFER}
- 100-\text{BUFFER}
- 3*\text{BUFFER}

- The above 3 expressions are considered as errors
Expressions

• The assembler must have types of symbols stored in SYMTAB
  – Either relative or absolute
  – For the generation of modification records

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>TYPE</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETADR</td>
<td>R</td>
<td>0030</td>
</tr>
<tr>
<td>BUFFER</td>
<td>R</td>
<td>0036</td>
</tr>
<tr>
<td>BUFFEND</td>
<td>R</td>
<td>1036</td>
</tr>
<tr>
<td>MAXLEN</td>
<td>A</td>
<td>1000</td>
</tr>
</tbody>
</table>
Program Blocks

• So far the generated instructions and data appeared in the same order as they are in the source assembly
  – Programmers put data and codes that are logically close to one another together in source assembly
  – In hardware point of view, it usually is not a good placement
    • Large data area usually gets in the way of relative addressing

• Program blocks are segments of code that can be re-arranged when they are loaded
  – Order of program blocks in memory could be different from that in source assembly
Program Blocks

• Directive “USE” indicates the program block which the following instructions and data are assigned to
  – Source assembly is assumed to use the unnamed (default) block unless explicitly specified
  – When specifying a different block, the following instructions and data are switched from the old block to the specified block (Line 92)
  – The previously discontinued block could be resumed by specifying its name again (Line 123)
<table>
<thead>
<tr>
<th>Line</th>
<th>Loc/Block</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000 0</td>
<td>COPY  START 0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0000 0</td>
<td>FIRST STL RETADR</td>
<td>172063</td>
</tr>
<tr>
<td>15</td>
<td>0003 0</td>
<td>CLOOP JSUB RDREC</td>
<td>4B2021</td>
</tr>
<tr>
<td>20</td>
<td>0006 0</td>
<td>LDA LENGTH</td>
<td>032060</td>
</tr>
<tr>
<td>25</td>
<td>0009 0</td>
<td>COMP #0</td>
<td>290000</td>
</tr>
<tr>
<td>30</td>
<td>000C 0</td>
<td>JEQ ENDFIL</td>
<td>332006</td>
</tr>
<tr>
<td>35</td>
<td>000F 0</td>
<td>JSUB WRREC</td>
<td>4B203B</td>
</tr>
<tr>
<td>40</td>
<td>0012 0</td>
<td>J CLOOP</td>
<td>3F2FEE</td>
</tr>
<tr>
<td>45</td>
<td>0015 0</td>
<td>ENDFIL LDA =C’EOF’</td>
<td>032055</td>
</tr>
<tr>
<td>50</td>
<td>0018 0</td>
<td>STA BUFFER</td>
<td>0F2056</td>
</tr>
<tr>
<td>55</td>
<td>001B 0</td>
<td>LDA #3</td>
<td>010003</td>
</tr>
<tr>
<td>60</td>
<td>001E 0</td>
<td>STA LENGTH</td>
<td>0F2048</td>
</tr>
<tr>
<td>65</td>
<td>0021 0</td>
<td>JSUB WRREC</td>
<td>4B2029</td>
</tr>
<tr>
<td>70</td>
<td>0024 0</td>
<td>J @RETADR</td>
<td>3E203F</td>
</tr>
<tr>
<td>92</td>
<td>0000 1</td>
<td>USE CDATA</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>0000 1</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0003 1</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>0000 2</td>
<td>USE CBLKS</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0000 2</td>
<td>BUFFER RESB 4096</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>1000 2</td>
<td>BUFEND EQU *</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>1000 2</td>
<td>MAXLEN EQU BUFEND-BUFFER</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>SUBROUTINE TO READ RECORD INTO BUFFER</td>
<td></td>
</tr>
</tbody>
</table>
USE

RDREC CLEAR X   B410
CLEAR A   B400
CLEAR S   B440

+LDT #MAXLEN  75101000

RLOOP TD INPUT E32038
JEQ RLOOP 332FFA
RD INPUT DB2032
COMPR A,S A004
JEQ EXIT 332008
STCH BUFFER,X 57A02F
TIXR T B850
JLT RLOOP 3B2FEA

EXIT STX LENGTH 13201F
RSUB 4F0000

USE CDATA

INPUT BYTE X’F1’ F1

SUBROUTINE TO WRITE RECORD FROM BUFFER

USE

WRREC CLEAR X   B410
LDT LENGTH 772017

WLOOP TD =X’05’ E3201B
JEQ WLOOP 332FFA
LDCH BUFFER,X 53A016
WD =X’05’ DF2012
TIXR T B850
JLT WLOOP 3B2FEF
RSUB 4F0000

USE CDATA

LJORG

* =C’EOF’ 454F46
* =X’05’ 05

END FIRST
Program Blocks

• The assembler should gather scattered instructions and data belonging to the same program block
  – When the program is loaded, program blocks are rearranged in the order in which they first appear
  – Physically, the effect is the same as if the programmers did the arrangement in source assembly
Program Blocks

• Implementing the handling of program blocks
  – Pass 1
    • Maintaining different LOCCTR’s for different program blocks
    • A new LOCCTR is created whenever a new program block appears
    • When switching to another program block, LOCCTR of the discontinued program block is saved and LOCCTR of the resumed program block is restored
    • By the end of pass 1, lengths of program blocks could be determined and so are their starting addresses
  – Pass 2
    • Target addresses of operands are relative to the beginning of the program, instead of the beginning of its corresponding program block
Program Blocks

At the end of Pass 1, information about the program blocks are as follows:

<table>
<thead>
<tr>
<th>Block name</th>
<th>Block Number</th>
<th>Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(default)</td>
<td>0</td>
<td>0000</td>
<td>0066</td>
</tr>
<tr>
<td>CDATA</td>
<td>1</td>
<td>0066</td>
<td>000B</td>
</tr>
<tr>
<td>CBLKS</td>
<td>2</td>
<td>0071</td>
<td>1000</td>
</tr>
</tbody>
</table>

20 0006 0 LDA LENGTH 032060

![Diagram showing the program blocks and their addresses and lengths]
Program Blocks

• The benefits of using program blocks are twofold
  – Physically, the need for complicated addressing modes such as format-4 addressing and base-relative addressing could be reduced
    • Move the large memory space reserved for BUFFER to somewhere else
  – Logically, the source assembly could still maintain its readability
    • Because symbol BUFFER is still near instructions accessing it
• The same effect could be achieved by inserting data near instructions accessing them
  – However, programmers must provide jump instructions to get around data
Program Blocks

- Text records should reflect the starting address of program blocks
  - The first four are for (default)
  - The fifth is for CDATA
  - The sixth is for (default)
  - The last is for CDATA
- No text records are generated for CBLKS
HCOPY 000000001071
T000001E1720D34B2010320602900003320064B203B3F2FE0320550F2056010003
T00001E90F20484B20293E203F
T0000271DB410B400B4407510100E3203832FFADB2032A00433006857A02FB850
T000044093B2FEA13201F4F0000
T00006C01F1
T00004D19B41077201E3021B332FFA53A016DF2012B8503B2FEF4F0000
E00000

<table>
<thead>
<tr>
<th>Line</th>
<th>Loc/Block</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000 0</td>
<td>COPY START</td>
<td>0027</td>
</tr>
<tr>
<td>10</td>
<td>0000 0</td>
<td>FIRST STL</td>
<td>125</td>
</tr>
<tr>
<td>15</td>
<td>0003 0</td>
<td>CLOOP JSUB</td>
<td>120</td>
</tr>
<tr>
<td>20</td>
<td>0006 0</td>
<td>LDA #0</td>
<td>130</td>
</tr>
<tr>
<td>25</td>
<td>0009 0</td>
<td>COMP #0</td>
<td>135</td>
</tr>
<tr>
<td>30</td>
<td>000C 0</td>
<td>JEQ ENDIF</td>
<td>140</td>
</tr>
<tr>
<td>35</td>
<td>000F 0</td>
<td>JSUB WRREC</td>
<td>145</td>
</tr>
<tr>
<td>40</td>
<td>0012 0</td>
<td>J CLOOP</td>
<td>150</td>
</tr>
<tr>
<td>45</td>
<td>0015 0</td>
<td>LDA =C’BOF’</td>
<td>155</td>
</tr>
<tr>
<td>50</td>
<td>0018 0</td>
<td>STA BUFFER</td>
<td>160</td>
</tr>
<tr>
<td>55</td>
<td>001B 0</td>
<td>LDA #3</td>
<td>165</td>
</tr>
<tr>
<td>60</td>
<td>001E 0</td>
<td>STA LENGTH</td>
<td>170</td>
</tr>
<tr>
<td>65</td>
<td>0021 0</td>
<td>JSUB WRREC</td>
<td>175</td>
</tr>
<tr>
<td>70</td>
<td>0024 0</td>
<td>J @RETADR</td>
<td>180</td>
</tr>
<tr>
<td>75</td>
<td>0027 1</td>
<td>USE CDATA</td>
<td>185</td>
</tr>
<tr>
<td>80</td>
<td>0030 0</td>
<td>RETADR RESW 1</td>
<td>190</td>
</tr>
<tr>
<td>85</td>
<td>0031 1</td>
<td>LENGTH RESW 1</td>
<td>195</td>
</tr>
<tr>
<td>90</td>
<td>0034 0</td>
<td>USE CJUS</td>
<td>200</td>
</tr>
<tr>
<td>95</td>
<td>0037 0</td>
<td>BUFFER RESB 4096</td>
<td>205</td>
</tr>
<tr>
<td>100</td>
<td>1000 2</td>
<td>BUFEND EQU</td>
<td>210</td>
</tr>
<tr>
<td>105</td>
<td>1000 2</td>
<td>MAXLEN EQU</td>
<td>215</td>
</tr>
<tr>
<td>110</td>
<td>004D 0</td>
<td>USE CDATA</td>
<td>220</td>
</tr>
<tr>
<td>115</td>
<td>004F 0</td>
<td>WRREC LDT</td>
<td>225</td>
</tr>
<tr>
<td>120</td>
<td>0052 0</td>
<td>WLOOP TD</td>
<td>230</td>
</tr>
<tr>
<td>125</td>
<td>0055 0</td>
<td>JEQ VLOOP</td>
<td>235</td>
</tr>
<tr>
<td>130</td>
<td>0058 0</td>
<td>LDTCH BUFFER, X</td>
<td>240</td>
</tr>
<tr>
<td>135</td>
<td>005B 0</td>
<td>NB @’BOF’</td>
<td>245</td>
</tr>
<tr>
<td>140</td>
<td>0060 0</td>
<td>RET 8850</td>
<td>250</td>
</tr>
<tr>
<td>145</td>
<td>0063 0</td>
<td>RSUB 4P0000</td>
<td>255</td>
</tr>
<tr>
<td>150</td>
<td>0066 1</td>
<td>INPUT BYTE F1’ F1’</td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>0069 0</td>
<td>SUBROUTINE TO WRITE RECORD FROM BUFFER</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>0071 0</td>
<td>REFERENCE TO READ RECORD INTO BUFFER</td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>0074 0</td>
<td>END FIRST</td>
<td>454746</td>
</tr>
</tbody>
</table>
Control Sections

• Control sections vs. program blocks
  – Control sections are very like separate assembly source files
    • Files (control sections) can be assembled and loaded independently
    • Where a control section is loaded (and target addresses of symbols in it) is unknown until runtime
  – A control section could be of a number of program blocks
    • Program blocks could be rearranged inside a control section
    • Control sections could be loaded anywhere in memory
  – A control section could be shared by a number of control sections
Control Sections

• Instructions in one control section could refer to a symbol defined in another control section
  – Referred to as external reference
  – The target address of the symbol could not be known until runtime
  – The assembler must generate some auxiliary information to help the linking loader to resolve external references
<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000</td>
<td>COPY 0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>EXTDDEF BUFFER,BUFFEND,LENGTH</td>
<td>172027</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>EXTRFD RDREC,WRREC</td>
<td>4B100000</td>
</tr>
<tr>
<td>10</td>
<td>0000</td>
<td>FIRST RETADR</td>
<td>172027</td>
</tr>
<tr>
<td>15</td>
<td>0003</td>
<td>CLOOP RDREC</td>
<td>03B023</td>
</tr>
<tr>
<td>20</td>
<td>0007</td>
<td>LDA LENGTH</td>
<td>03B023</td>
</tr>
<tr>
<td>25</td>
<td>000A</td>
<td>COMP #0</td>
<td>29B000</td>
</tr>
<tr>
<td>30</td>
<td>000D</td>
<td>JEQ ENDFIL</td>
<td>33B007</td>
</tr>
<tr>
<td>35</td>
<td>0010</td>
<td>+JSUB WRREC</td>
<td>4B100000</td>
</tr>
<tr>
<td>40</td>
<td>0014</td>
<td>J CLOOP</td>
<td>3B2FEC</td>
</tr>
<tr>
<td>45</td>
<td>0017</td>
<td>ENDFIL</td>
<td>03B016</td>
</tr>
<tr>
<td>50</td>
<td>001A</td>
<td>LDA =C’EOF’</td>
<td>03B016</td>
</tr>
<tr>
<td>55</td>
<td>001D</td>
<td>STA BUFFER</td>
<td>01B003</td>
</tr>
<tr>
<td>60</td>
<td>0020</td>
<td>STA LENGTH</td>
<td>0F200A</td>
</tr>
<tr>
<td>65</td>
<td>0023</td>
<td>+JSUB WRREC</td>
<td>4B100000</td>
</tr>
<tr>
<td>70</td>
<td>0027</td>
<td>J @RETADR</td>
<td>3B2000</td>
</tr>
<tr>
<td>95</td>
<td>002A</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>002D</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>LTORG</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0030</td>
<td>* =C’EOF’</td>
<td>45B456</td>
</tr>
<tr>
<td>106</td>
<td>1033</td>
<td>BUFFER RESW</td>
<td>4096</td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>BUFEND EQU *</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>0000</td>
<td>RDREC CSECT</td>
<td></td>
</tr>
</tbody>
</table>
110  
115  SUBROUTINE TO READ RECORD INTO BUFFER
120  
122  EXTREF BUFFER LENGTH BUFOEND
125  0000  CLEAR X     B410
130  0002  CLEAR A     B400
132  0004  CLEAR S     B440
133  0006  LDT MAXLEN  77201F
135  0009  RLOOP TD    INPUT E3201B
140  000C  JEQ RLOOP   332FFA
145  000F  RD INPUT    DB2015
150  0012  COMPR A,S   A004
155  0014  JEQ EXIT    332009
160  0017  +STCH BUFFER X  579D0000
165  001B  TIXR T      B850
170  001D  JLT RLOOP   3B2FE9
175  0020  EXIT +STX LENGTH 13100000
180  0024  RSUB        4F0000
185  0027  INPUT BYTE   X‘F1’  F1
190  0028  MAXLEN WORD   BUFOEND-BUFFER 000000

193  0000  WRREC CSECT
195  
200  SUBROUTINE TO WRITE RECORD FROM BUFFER
205  
207  EXTREF LENGTH BUFFER
210  0000  CLEAR X     B410
212  0002  +LDT LENGTH  77100000
215  0006  WLOOP TD    =X‘05’  E32012
220  0009  JEQ WLOOP   332FFA
225  000C  +LDCH BUFFER, X  539D0000
230  0010  WD =X‘05’    DP2008
235  0013  TIXR T      B850
240  0015  JLT WLOOP   3B2FE9
245  0018  RSUB        4F0000
250  END FIRST
255  001B  * =X‘05’     05
Control Sections

- Three control sections are identified in Fig 2.16
  - COPY, RDREC, WRREC
  - The assembler maintains separate LOCCTR beginning at 0 for different control sections
    - Much like that for program blocks
  - For example, instructions in COPY might reference symbols in RDREC

- Two new directive EXTDEF and EXTREF are introduced
  - EXTDEF names symbols which are defined in this control section and might be referred to in another control section
  - EXTREF names symbols which are referred to in this section and are defined in some other control sections
Control Sections

5 COPY START 0
6 EXTDEF BUFFER, BUFEND, LENGTH

• Exposes three symbols in COPY to other control sections

15 0003 CLOOP +JSUB RDREC 4B100000

• Symbol RDREC is referenced in COPY but defined in RDREC
  – The assembler insert 0 to the target address
  – Relative addressing is not possible!!
    • External references should be used in conjunction with long instructions
  – Necessary information must be provided to linking loader so as to correct this during runtime
Control Sections

160 0017 +STCH BUFFER, X 57900000

• Similar

190 0028 MAXLEN WORD BUFEND–BUFFER 000000

• Neither BUFEND nor BUFFER could be resolved
  – Linking loader needs to perform 2 modifications
    • +BUFEND and then –BUFFER

107 1000 MAXLEN EQU BUFEND–BUFFER

• No object code is generated for this line
  – MAXLEN is defined in both COPY and RDREC but it arises no errors of re-defining symbols
  – Brain damage: if operands of EQU involve external symbols?
Control Sections

- **Two new records** are introduced to direct the linking loader about how external references could be resolved

<table>
<thead>
<tr>
<th>Col. 1</th>
<th>Col. 2-7</th>
<th>Col. 8-13</th>
<th>Col. 14-73</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Name of external symbol defined in this control section</td>
<td>Relative address of symbol within this control section (hex)</td>
<td>Repeat information in Col 2-13 for other external symbols</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Col. 1</th>
<th>Col. 2-7</th>
<th>Col. 8-73</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Name of external symbol referred in this control section</td>
<td>Name of other external reference symbols</td>
</tr>
</tbody>
</table>
Control Sections

- The format of modification record is revised to support the handling of external references.
- Relocation for format-4 instructions without external references could be handled by referring to the control section’s name – M00000705+COPY

<table>
<thead>
<tr>
<th>Col. 1</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col. 2-7</td>
<td>Starting location of the target address to be modified, relative to the beginning of the program (not relative to the first text record)</td>
</tr>
<tr>
<td>Col. 8-9</td>
<td>Length of this record in half-byte</td>
</tr>
<tr>
<td>Col. 10</td>
<td>Modification flag (+ or -)</td>
</tr>
<tr>
<td>Col. 11-16</td>
<td>External symbol whose value is to be added to or subtracted from the indicated field</td>
</tr>
</tbody>
</table>
Assembler Design Options

- One-pass assembler vs. multi-pass assembler
  - The major difference between them is how target addresses of symbols are resolved
  - Multiple passes might be needed for forward references

```
ALPHA EQU BETA
BETA EQU DELTA
DELTA RESW 1
```
One-Pass Assemblers

• **Forward references** involve data items, labels on instructions, or other symbols
  – To prohibit forward references to data items is not very restrictive
    • E.g., defining all data before instructions (like C-programs do)
  – To eliminate forward references to labels on instructions could be too restrictive
    • E.g., some condition-guarded loops might skip ahead
<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>COPY START 1000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>EOF BYTE C'EOF'</td>
<td>454F46</td>
</tr>
<tr>
<td>2</td>
<td>1003</td>
<td>THREE WORD 3</td>
<td>0000003</td>
</tr>
<tr>
<td>3</td>
<td>1006</td>
<td>ZERO WORD 0</td>
<td>0000000</td>
</tr>
<tr>
<td>4</td>
<td>1009</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100C</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>100F</td>
<td>BUFFER RESB 4096</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>200F</td>
<td>FIRST STL RETADR</td>
<td>141009</td>
</tr>
<tr>
<td>15</td>
<td>2012</td>
<td>CLOOP JSUB RDREC</td>
<td>48203D</td>
</tr>
<tr>
<td>20</td>
<td>2015</td>
<td>LDA LENGTH</td>
<td>00100C</td>
</tr>
<tr>
<td>25</td>
<td>2018</td>
<td>COMP ZERO</td>
<td>281006</td>
</tr>
<tr>
<td>30</td>
<td>201B</td>
<td>JEQ ENDFIL</td>
<td>302024</td>
</tr>
<tr>
<td>35</td>
<td>201E</td>
<td>JSUB WRREC</td>
<td>482062</td>
</tr>
<tr>
<td>40</td>
<td>2021</td>
<td>J CLOOP</td>
<td>302012</td>
</tr>
<tr>
<td>45</td>
<td>2024</td>
<td>ENDFIL LDA EOF</td>
<td>001000</td>
</tr>
<tr>
<td>50</td>
<td>2027</td>
<td>STA BUFFER</td>
<td>0C100F</td>
</tr>
<tr>
<td>55</td>
<td>202A</td>
<td>LDA THREE</td>
<td>001003</td>
</tr>
<tr>
<td>60</td>
<td>202D</td>
<td>STA LENGTH</td>
<td>0C100C</td>
</tr>
<tr>
<td>65</td>
<td>2030</td>
<td>JSUB WRREC</td>
<td>482062</td>
</tr>
<tr>
<td>70</td>
<td>2033</td>
<td>LDL RETADR</td>
<td>081009</td>
</tr>
<tr>
<td>75</td>
<td>2036</td>
<td>RSUB</td>
<td>4C0000</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>2039</td>
<td>INPUT BYTE X'F1'</td>
<td>F1</td>
</tr>
<tr>
<td>122</td>
<td>203A</td>
<td>MAXLEN WORD 4096</td>
<td>001000</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`SUBROUTINE TO READ RECORD INTO BUFFER`
One-Pass Assemblers

• One-pass assemblers could produce object codes either in memory or to external storage
  – One-pass assemblers usually need to modify object code already generated, so whether object code is stored in memory or external storage imposes difference considerations on assembler design

• Load-and-go assemblers produce object code in memory and then start executing immediately
  – Useful for intensive testing
One-Pass Assemblers

• The handling of forward reference for load-and-go assemblers
  – When an instruction referring to an undefined symbol:
    • The symbol is entered in SYMTAB and flagged as a undefined one
    • Its target address is omitted in the object code
  – With respect to a not yet defined symbol, all instructions refer to it are chained after its entry in SYMTAB
    • Once the symbol is defined, the correct target address will be inserted as operands for those instructions
  – If there is any undefined symbol after the entire source assembly is scanned, errors would be generated
<table>
<thead>
<tr>
<th>Memory address</th>
<th>Contents</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>454F4600 00030000 00xxxxxx xxxxxxxx</td>
<td>LENGTH</td>
<td>100C</td>
</tr>
<tr>
<td>1010</td>
<td>xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx</td>
<td>RDREC</td>
<td>*</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>THREE</td>
<td>1003</td>
</tr>
<tr>
<td>2000</td>
<td>xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx14</td>
<td>ZERO</td>
<td>1006</td>
</tr>
<tr>
<td>2010</td>
<td>1009 48-- --00100C 28100630 --48--</td>
<td>WRREC</td>
<td>*</td>
</tr>
<tr>
<td>2020</td>
<td>--3C2012</td>
<td>EOF</td>
<td>1000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>EOF</td>
<td>1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2012</td>
<td>CLOOP</td>
<td>48203D</td>
</tr>
<tr>
<td>30</td>
<td>201B</td>
<td>JEQ</td>
<td>302024</td>
</tr>
<tr>
<td>35</td>
<td>201E</td>
<td>JSUB</td>
<td>482062</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLOOP</td>
<td>2012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JSUB</td>
<td>200F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETADR</td>
<td>1009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BUFFER</td>
<td>100F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLOOP</td>
<td>2012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIRST</td>
<td>200F</td>
</tr>
</tbody>
</table>
One-Pass Assemblers

• For load-and-go assemblers,
  – The actual beginning address of a program is known before the source program is assembled
    • Initial value for LOCCTR will be accordingly adjusted
  – Issues of program relocation could be ignored since no object programs are stored
    • The generated object code is useful only at that time
One-Pass Assemblers

• For one-pass assemblers which store object programs in external storage (e.g., files on disks)
  – We can not assume that the object program can be randomly accessed
    • Maybe object programs are stored on tapes or an archive file?
    • Therefore, random updates to operands’ target addresses (as load-and-go assemblers do) are not permitted
  – For any symbol involved in forward references
    • Once the target address of the symbol is identified, additional text records must be generated to overwrite those previously omitted target addresses
    • Caution: records must be loaded in the same order as they appear in the object program
  – Actually, the handling of forward references are jointly done by the assembler and the linking loader
Fig 2.20: how text is patched by additional text records
Multi-Pass Assemblers

• A multi-pass assembler is able to handle forward reference in directives like EQU
  – Forward references in symbol defining directives introduces additional dependencies
    • From symbols to symbols
    • Forward references in ordinary instructions introduces dependencies from symbols to operands in instructions
      – Dependencies from instructions to anything are obviously not possible
    • Dependencies of symbol→symbol must first be resolved and then dependencies of symbol→instruction
Dependencies, illustrated

- Cycles are not permitted
- No edges from rectangles to anything
- Without dependencies among circles, any path is no longer than 1
  - Multiple passes are needed whenever path length > 1
Multi-Pass Assemblers

• A multi-pass assembler does not necessarily scan the source assembly multiple times
  – Similar techniques as adopted in one-pass assemblers could be used to resolve dependencies among symbols
  – Of course, the two kinds of dependencies (symbols→instructions and symbols→symbols) could be handled in the first pass if proper design is adopted
Multi-Pass Assemblers

1. \texttt{HALFSZ} \texttt{EQU} \texttt{MAXLEN/2}
2. \texttt{MAXLEN} \texttt{EQU} \texttt{BUFEND-BUFFER}
3. \texttt{PREVBT} \texttt{EQU} \texttt{BUFFER-1}

4. \texttt{BUFFER RESB 4096}
5. \texttt{BUFEND EQU *}

- \texttt{SYMTAB} could be revised to handle forward references in symbol defining instructions
1 1 HALFSZ EQU MAXLEN/2
2 2 MAXLEN EQU BUFEND-BUFFER
3 3 PREVBT EQU BUFFER-1
4 4 BUFFER RESB 4096
5 5 BUFEND EQU *
HALFSZ EQU MAXLEN/2
MAXLEN EQU BUFEND-BUFFER
PREVBT EQU BUFFER-1
BUFFER 1034 RESB 4096
BUFEND EQU *
1) \text{HALFSZ} \quad \text{EQU} \quad \text{MAXLEN/2}

2) \text{ALPHA} \quad \text{EQU} \quad \text{BUFEND-BUFFER}

3) \text{PREVBT} \quad \text{EQU} \quad \text{BUFFER-1}

4) 1034 \quad \text{BUFFER} \quad \text{RESB} \quad 4096

5) 2034 \quad \text{BUFEND} \quad \text{EQU} \quad *