Lecture Topics

- Today: The Translation Process (H&H 6.4-6.5)
- Next: continued

Announcements

- Self-study Module #10
- Project #10 (due no later than 11/21)
- Project #11 (due no later than 11/30)
The Translation Process

Three main steps in the translation process:

- Compilation
- Assembly
- Linking

Compilation broken down into smaller steps (preprocessing, lexical analysis, semantic analysis, etc).

Example #29

Three distinct modules:

- example29.driver.c
- example29.support.s
- standard_library.o

The standard library object code is actually in several different .o files.
Typical command to translate and link:

```bash
gcc example29.driver.c example29.support.s
```

The "gcc" package is configured to automatically link certain object code modules for the standard libraries.

Executable program in "a.out" (default).

The "verbose" option displays each step:

```bash
gcc --verbose example29.driver.c \ 
    example29.support.s
```

Course website:

```
~cse320/Examples/example29.pdf
```
ARM Memory Model

Memory is viewed as a linear sequence of bytes (flat memory model).

Addresses are 32 bits wide, so there are $2^{32}$ bytes (4 Gigabytes).

Addresses range from 00000000 to FFFFFFFF.

Manage as “segments” of 64 Kilobytes:
(00000000 to 0000FFFF, 00010000 to 0001FFFF, 00020000 to 0002FFFF, etc.)
Assembler Processing

Read assembly language, produce equivalent machine language:

- Convert symbolic machine language instructions into machine language instructions
- Process assembler directives
Assembler Processing

Two passes over the source code:

Pass 1 – determine number of bytes used by text section and data section, build symbol table

Pass 2 – emit object code (machine language instructions in text section, initial values in data section)

Pass 1 Processing

Assembler must determine number of bytes needed for the text section and data section: read each line, determine number of bytes for that line.

Labels (symbolic addresses) are offsets into the text section or the data section, so process them (put them into symbol table with the correct offset).

Symbol table also used to handle symbolic constants.
Pass 2 Processing

Assembler must emit the correct object code: read each line, emit machine language or data value for that line.

Examples:

```
add     r0, r0, r3       ==>  E0800003
.word 21               ==>  00000015
```

Why two passes?

Assembler must handle forward references:

```
loop:
    cmp r1, #NUM
    bge endloop  # forward reference
    .
    .
    .
    b loop
endloop:
```
Why two passes?

Pass 1: build symbol table so that symbolic addresses such as “loop” and “endloop” have meaning (offsets within text section).

Pass 2: use symbol table to emit correct machine language instructions (including forward refs).

Example symbol table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>A/R</th>
<th>L/G</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM</td>
<td>00000006</td>
<td>ABS</td>
<td>L</td>
<td>line 6</td>
</tr>
<tr>
<td>main</td>
<td>.text+00000000</td>
<td>REL</td>
<td>G</td>
<td>line 11</td>
</tr>
<tr>
<td>loop</td>
<td>.text+00000010</td>
<td>REL</td>
<td>L</td>
<td>line 17</td>
</tr>
<tr>
<td>endloop</td>
<td>.text+0000002c</td>
<td>REL</td>
<td>L</td>
<td>line 27</td>
</tr>
<tr>
<td>vector</td>
<td>.data+00000000</td>
<td>REL</td>
<td>L</td>
<td>line 40</td>
</tr>
<tr>
<td>total</td>
<td>.data+00000018</td>
<td>REL</td>
<td>L</td>
<td>line 42</td>
</tr>
</tbody>
</table>
Attributes of a symbol (label):

- Symbol (key)
- Value
- Absolute or Relative
- Local or Global
- Definition (source code line number)

Pass 1 processing:

```
location counter <-- 0
while not end-of-file
    read a line
    if line begins with a label
        insert label into symbol table
        (value is location counter)
    endif
    determine number of bytes to add to
    location counter
endwhile
```
Pass 2 processing:

while not end-of-file
  read a line
  emit corresponding object code
    (machine language or data value)
endwhile

Example #30

Source code contains a mix of symbolic machine instructions and assembler directives.

Symbols in source code:
- symbolic constant (“NUM”)
- symbolic addresses in text section
- symbolic addresses in data section
Example #30

First page:

- source code
- translation and execution

Second page:

- symbol table (built during Pass 1)
- assembly listing (object code)

Pass 1 Processing

- No change to location counter:
  - Blank line
  - Comment line
  - Line with label, but no operation
- Symbolic machine instructions – 4 bytes
- Assembler directives – variable number of bytes
Assembler Directives

Flag `main` as global in symbol table:

```
.global main
```

Switch into text section:

```
.text
```

Add 0, 1, 2 or 3 bytes to the location counter:

```
.balign 4
```

Assembler Directives

Switch into data section;
Add 0, 1, 2 or 3 bytes to the location counter:

```
.data
.balign 4
```

Add 4 bytes for each value in list:

```
.word   21, 45, 96, 72, 53, 39
```

Add 4 bytes to the location counter:

```
.skip    4
```
Symbol table at end of Pass 1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>A/R</th>
<th>L/G</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM</td>
<td>00000006</td>
<td>ABS</td>
<td>L</td>
<td>line 6</td>
</tr>
<tr>
<td>main</td>
<td>.text+00000000</td>
<td>REL</td>
<td>G</td>
<td>line 11</td>
</tr>
<tr>
<td>loop</td>
<td>.text+00000010</td>
<td>REL</td>
<td>L</td>
<td>line 17</td>
</tr>
<tr>
<td>endloop</td>
<td>.text+0000002c</td>
<td>REL</td>
<td>L</td>
<td>line 27</td>
</tr>
<tr>
<td>vector</td>
<td>.data+00000000</td>
<td>REL</td>
<td>L</td>
<td>line 40</td>
</tr>
<tr>
<td>total</td>
<td>.data+00000018</td>
<td>REL</td>
<td>L</td>
<td>line 42</td>
</tr>
</tbody>
</table>

Pass 2 Processing

- No object code:
  - Blank line
  - Comment line
  - Line with label, but no operation
- Symbolic machine instructions – 4 bytes
- Assembler directives – variable number of bytes
Symbolic machine instructions

Look up operation, emit machine instruction:

E52DE004 push {lr}
E1A02101 lsl r2, r1, #2
E7943002 ldr r3, [r4, r2]
E5850000 str r0, [r5]

Assembler directives

Emit values(s) for data-generating directives:

00000015 .word 21, 45, 96, 72, 53, 39
0000002D
00000060
00000048
00000048
00000035
Instructions with symbols

Substitute value for absolute symbols:

E3510006    cmp    r1, #NUM

Look up NUM in symbol table, use its value (which is 00000006) in the “immediate” field.

---

Instructions with symbols

References to undefined symbols:

EBFFFFFE    bl    display4

Use place holder in “bl” instructions (linker will have to fix later). If not defined in some other object code module, linker will flag as “undefined reference”.
Instructions with symbols

References to relative symbols in the text section:

AA000004  bge  endloop
EAFFFFF8  b  loop

Look up symbol, use its value and the current location to compute the 24-bit "offset" field.

Processing branch instructions

The assembler has to calculate the correct 24-bit "offset" value so that the appropriate actions occur during execution:

\[ \text{PC} + 8 + (\text{offset} \times 4) \rightarrow \text{PC} \]

Use location counter during assembly processing as surrogate for PC.
Branch Instructions

Machine language format:

<table>
<thead>
<tr>
<th>31:28</th>
<th>27:26</th>
<th>25:24</th>
<th>23:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>op 10</td>
<td>1L</td>
<td>imm24</td>
</tr>
</tbody>
</table>

- Bits 31:28 – condition (AL, EQ, VS, etc)
- Bits 27:26 – opcode (10 for branches)
- Bits 25:24 – function (10 for B, 11 for BL)
- Bits 23:0 – 24-bit immediate value (offset)

Execution: check NZCV bits; if condition is true, branch to a different location in the program (continue sequentially if condition is false).

PC-relative branching: new location is some displacement (+ or -) from the current location:

\[
PC + 8 + (imm24 \times 4) \rightarrow PC
\]
text+14: bge endloop

During execution:

PC + 8 + (imm24 * 4) \rightarrow PC

During assembly:

\[(\text{text+14}) + 8 + \text{imm24} \times 4 = (\text{text+2c})\]
\[8 + \text{imm24} \times 4 = (\text{text+2c}) - (\text{text+14})\]
\[8 + \text{imm24} \times 4 = 18\]
\[\text{imm24} \times 4 = 10\]
\[\text{imm24} = 10 / 4\]
\[\text{imm24} = 4\] (hex values!)

---

text+14: bge endloop

cond: 1010 (GE)

op: 10 (Branch)

funct: 10 (B)

imm24: 00000000000000000000000000100

IR: 101010100000000000000000000000100

IR: AA000004
text+28:   b     loop

During execution:

PC + 8 + (imm24 * 4) → PC

During assembly:

(text+28) + 8 + imm24 * 4 = (text+10)
8 + imm24 * 4 = (text+10) - (text+28)
8 + imm24 * 4 = -18
imm24 * 4 = -20
imm24 = -20 / 4
imm24 = -8 (hex values!)

cond:   1110 (AL -- always)
op:     10 (Branch)
funct:  10 (B)
imm24:  111111111111111111111000

IR: 11101010111111111111111111111000

IR: EAFFFFF8
The assembler outputs object code:

- Machine instructions (text section)
- Data values (data section)
- Info for linker

Linker needs to know which instructions could not be completely processed by assembler (references to addresses in “BL” instructions, typically).

Overview: linker processing

The linker reads multiple object code files, produces single text section and single data section (combined from input files).

The linker builds a master symbol table so that it can fix the references that the assembler could not completely process (ex: location of “display4” after all text sections combined).