Overview

- Compiler
- Assembler
- Linker
- Loader
- Example

Review: What is Subject about?

- Coordination of many levels of abstraction

Review: Programming Levels of Representation

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
ldr r0 , [r2, #0]
ldr r1 , [r2, #4]
str r1 , [r2, #0]
str r0 , [r2, #4]
```

1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0100
1111 0101 1000 0010 0000 0000 0000 0100
1111 0100 1000 0010 0000 0000 0000 0100
Review: Stored Program Concept

- **Stored Program Concept**: Both data and actual code (instructions) are stored in the same memory.
- **Type is not associated with data**: Bits have no meaning unless given in context.

---

Review: ARM Instruction Set Format

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Format Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data processing / PSR transfer</td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
</tr>
<tr>
<td>Long Multiply (v3M / v4 only)</td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td></td>
</tr>
<tr>
<td>Load/Store Byte/Word</td>
<td></td>
</tr>
<tr>
<td>Load/Store Multiple</td>
<td></td>
</tr>
<tr>
<td>Halfword transfer: immediate offset (v4 only)</td>
<td></td>
</tr>
<tr>
<td>Halfword transfer: Register offset (v4 only)</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td></td>
</tr>
<tr>
<td>Branch Exchange (v4T only)</td>
<td></td>
</tr>
<tr>
<td>Coprocessor data transfer</td>
<td></td>
</tr>
<tr>
<td>Coprocessor data operation</td>
<td></td>
</tr>
<tr>
<td>Coprocessor register transfer</td>
<td></td>
</tr>
<tr>
<td>Software interrupt</td>
<td></td>
</tr>
</tbody>
</table>

**All Instruction 32 bits**

---

Review: Example Assembly

```
sub r2, r3, #1          #=> 0Xe2432001
sub r2, r3, r4          #=> 0Xe0432004
b    foo               #=> 0Xea &foo-------
```

```
1110 001 0010 0011 0010 00000000001
1110 000 0010 0011 0010 000000000100
1110 101 0

14  1 2 0 3 2 0 1
14  0 2 0 3 2 0 0 4
14  5 0

?= ((pc + 8) - &foo) >>2
```
Compiler

- Input: High-Level Language Code (e.g., C, Java)
- Output: Assembly Language Code (e.g., ARM)
- Most Compiler can generate Object code (Machine language) directly

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

extern int posmul(int mlfer, int mcand);
int main (void)
{
    char *MESEG = "Multiplication";
    static int a=20, b=18, c;
    c = posmul(a, b);
    return c;
}

Where Are We Now?

C program: foo.c
Assembly program: foo.s
Object(mach lang module): foo.o
Executable(mach lang pgm): a.out

Compiler
Assembler
Linker
Loader
Memory

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run (#1/3)
.data
.align 2
a.0: .word 20

.align 2
b.1: .word 18

.align 2
c.2: .space 4

.section .rodata
.align 2
.LC0: .ascii "Multiplication\000"
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run (#2/3)

```c
.text
.align 2
.global main

main:
    stmfd sp!, {r4,lr}
    ldr     r4, .L2 ;  MESG
    ldr     r3, .L2+4
    ldr     r2, .L2+8
    ldr     r0, [r3, #0] ;  a
    ldr     r1, [r2, #0] ;  b
    bl      posmul
    mov     r2, r0
    ldr     r3, .L2+12
    str     r2, [r3, #0] ;  c
    mov     r0, r3
    ldmfd   sp!, {r4, pc}
```

Addresses of MESG, a, b & c are stored at label .L2

Indirect access to a, b & c

What is Assembler?

- Program that translates symbolic machine instructions into binary representation
- Encodes code and data as blocks of bits from symbolic instruction, declarations, and directives
- It builds the code words and the static data words
  - loaded into memory when program is run
- What must it do
  - map opcodes, regs, literals into bit fields
  - map labels into addresses

How does Assembler work?

- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File (*.o files)

Literal Pool for storing addresses of MESG, a, b & c

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run (#3/3)

```c
.L3:
    .align 2

.L2:
    .word   .LC0
    .word   a.0
    .word   b.1
    .word   c.2
```

Text Segment (continued)
**Assembler Directives**

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.globl` `sym`: declares `sym` global and can be referenced from other files
  - `.asciz str`: Store the string `str` in memory and null-terminate it
  - `.word w1...wn`: Store the `n` 32-bit quantities in successive memory words

**Pseudo-instruction Replacement (#1/6)**

- Assembler provide many convenient shorthand special cases of real instructions
  - `nop` ⇒ `mov, r0, r0`
  - `mov r0, #0xffffffff0` ⇒ `mvn r0, 0xf`
  - `ldr/str rdest, label` ⇒ load/stores a value at label (address) in the same segment. Converts to
    - `ldr rdest, [pc, #offset]` instruction, where `offset` is computed by `(address@label - [pc + 8])`
    - Offset range ±2^12 (±4 Kbytes)
  - `adr rdest, label` ⇒ load address of a label (in the same segment) computed as an offset from PC. Converts to
    - `sub rdest, pc,#imm ; #imm: 8 bit number`
    - `add rdest, pc,#imm ; or rotated version`
    - `#imm` is computed as `-(address@label - [pc + 8])`

**Pseudo-instruction Replacement (#2/6)**

- `ladr rdest, label` ⇒ same thing as `adr` when offset value cannot fit in 8 bit rotated format
  - Converts to sequence of two `sub` & `add` instructions
  - If second `sub/add` not needed is replaced by `nop`

- **Especially important Pseudo Instructions are for building literals**
  - `ldr rdest, =imm32` ⇒ load (move) ANY immediate to `rdest`

**Pseudo-instruction Replacement (#3/6)**

- Limitation on `mov rdest, #imm` Instruction
  - Any 8-bit value in the range 0 – 255 (0x0 – 0xff)
  - Any 8 bit value in the range 0 – 255 (0x0 – 0xff) rotated to the right two bits at a time.
    - Max rotation = 30 bits
  - Example:
    - 0x000000FE, 0x8000003f (rot. 2 bits), 0xe000000f (rot. 4 bits), ... all are valid values

\[
\begin{array}{cccc}
31 & 30 & 29 & 28 \\
15 & 14 & 13 & 12 \\
11 & 10 & 9 & 8 \\
7 & 6 & 5 & 4 \\
3 & 2 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
98 & 76 & 54 & 32 \\
10 & 11 & 11 & 11 \\
11 & 11 & 11 & 11 \\
11 & 11 & 11 & 11 \\
\end{array}
\]
Pseudo-instruction Replacement (#4/6)

° Solution: Use Pseudo Instruction
  • Replace \texttt{mov rdest, \#imm} by \texttt{ldr rdest, =imm} \Rightarrow \text{load (move) ANY immediate}
  • Converts to \texttt{mov} or \texttt{mvn} instruction, if the constant can be generated by either of these instructions.
  • PC relative LDR instruction will be generated to load the \texttt{imm} from literal pool inserted at the end of the text segment.

Pseudo-instruction Replacement (#5/6)

° Example: Use Pseudo Instruction

\begin{verbatim}
.text
.align 2
.global main
main:
: ldr r2, =4118633130
: end:
\end{verbatim}

° Changes to

\begin{verbatim}
.text
.align 2
.global main
main:
: ldr r2, [pc, \#offset]
: end:
\end{verbatim}

\texttt{Offset} = \text{end} - (\texttt{pc} + 8)

Pseudo-instruction Replacement (#6/6)

° Recall: \texttt{ldr/str rdest, label} \Rightarrow \text{load/stores a value at label (address) in the same segment. Converts to}
  
  \begin{itemize}
  \item \texttt{ldr/str rdest, [pc, \#offset]} instruction, where \texttt{offset} is computed by (address@label - [pc + 8]).
  \item \texttt{offset} range \(2^{12}\) (\(\pm 4\) Kbytes)
  \end{itemize}

° \texttt{ldr rdest1, =label} \Rightarrow \text{load address of ANY label}

° PC relative LDR instruction will be generated to load the address of the \texttt{label} from literal pool inserted at the end of the text segment.
  
  \begin{itemize}
  \item \texttt{ldr rdest1, [pc, \#offset]}
  \end{itemize}

° Next load the \texttt{value} at the label by one additional instruction:
  
  \begin{itemize}
  \item \texttt{ldr/str rdest2, [rdest1]}
  \end{itemize}

Handling Addresses by Assembler (#1/2)

° Branches: \texttt{b, & bl} (branch and link)

\begin{verbatim}
\texttt{b/bl} \hspace{1cm} \texttt{label}
\end{verbatim}

° Such branches are normally taken to a label (address) labels at fixed locations, in the same module (file) or other modules (eg. Call to functions in other modules)

° The address of the label is \text{absolute}
Handling Addresses by Assembler (#2/2)
° Loads and stores to variables in static area
  • Such addresses are stored in the literal pool by the compiler/Assembler
  • The reference to the literal pool is via PC relative addressing
    ldr Rdest1, [pc, #offset]
    ldr/str Rdest2, [Rdest1]
  • Sometimes they are referenced via Static Base Pointer (SB)
    ldr/str Rdest, [sb, #offset]
° Loads and stores to local variables
  • Such variables are put direct on registers or on stack and are referenced via sp or fp.

Reading Material
° Reading assignment:
  • Steve Furber: ARM System On-Chip; 2nd Ed, Addison-Wesley, 2000, ISBN: 0-201-67519-6. chapter 2, section 2.4

Things to Remember
° Compiler converts a single HLL file into a single assembly language file.
° Assembler removes pseudos, converts it to machine language. This changes each .s file into a .o file.
° Linker combines several .o files and resolves absolute addresses.
° Loader loads executable into memory and begins execution.