Review: What is Subject about?

° Coordination of many levels of abstraction

Review: Programming Levels of Representation

High Level Language Program (e.g., C)

<table>
<thead>
<tr>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r0, [r2, #0]</td>
</tr>
<tr>
<td>ldr r1, [r2, #4]</td>
</tr>
<tr>
<td>str r1, [r2, #0]</td>
</tr>
<tr>
<td>str r0, [r2, #4]</td>
</tr>
</tbody>
</table>

Assembly Language Program (e.g., ARM)

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

Machine Language Program (ARM)

1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0100
1110 0101 1000 0100 0001 0000 0000 0000
1110 0101 1000 0010 0001 0000 0000 0100


Compiler
Assembler
Linker
Loader
Example
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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data processing / PSR transfer</td>
<td>Multiply</td>
</tr>
<tr>
<td>Multiply</td>
<td>Long Multiply (v3M / v4 only)</td>
</tr>
<tr>
<td>Swap</td>
<td>Load/Store Byte/Word</td>
</tr>
<tr>
<td>Load/Store Multiple</td>
<td>Halfword transfer: immediate offset (v4 only)</td>
</tr>
<tr>
<td>Halfword transfer: Register offset (v4 only)</td>
<td>Branch</td>
</tr>
<tr>
<td>Branch Exchange (v4T only)</td>
<td>Coprocessor data transfer</td>
</tr>
<tr>
<td>Coprocessor data operation</td>
<td>Coprocessor register transfer</td>
</tr>
<tr>
<td>Coprocessor register transfer</td>
<td>Software interrupt</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------
```

Steps to Starting a Program

```
C program: foo.c

Assembly program: foo.s

Object(mach lang module): foo.o

Executable(mach lang pgm): a.out
```

Memory
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 mlier, int mcand);
int main (void)
{
    char *srcstr = "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

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)

```
.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

  str     r2, [r3, #0] ;  c

  mov     r0, r3

  ldmsdf sp!, {r4, lr}
```

Text Segment

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

Indirect access to a, b & c

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

```
.L3:  .align 2

.L2: 
```

Text Segment (continued)

Literal Pool for storing addresses of MESG, 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)
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, 0xffffffff => movn 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 ± 212 (±4 Kbytes)
  • adr rdest, address => 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, address => same thing for 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 01 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 11 | 98 | 76 | 54 | 32 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 11 |
Pseudo-instruction Replacement (#4/6)
° Solution: Use Pseudo Instruction
  • Replace \texttt{ldr \ rdest, \#imm} by \texttt{ldr \ rdest, =imm} => 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

```assembly
.text
.align 2
.global main
main:    
  ldr r2, =4118633130
end:    
```
° Changes to

```assembly
.text
.align 2
.global main
main:    
  ldr r2, [pc, #offset]
end:    
.word 4118633130
```

Offset = end - (pc + 8)

Pseudo-instruction Replacement (#6/6)
° Recall: \texttt{ldr/str \ rdest, \ label} => load/stores a value at label (address) in the same segment. Converts to
  - \texttt{ldr \ rdest, [pc, \#offset]} instruction, where \texttt{offset} is computed by (\texttt{address@label} - \texttt{pc + 8}).
  - Offset range \(\pm 2^{12}\) (\(\pm 4\) Kbytes)

° \texttt{ldr \ rdest, =label} => load address of ANY label

° PC relative LDR instruction will be generated to load the address of the label from literal pool inserted at the end of the text segment.

Handling Addresses by Assembler (#1/2)
° Branches: \texttt{b, \& bl} (branch and link)

```assembly
b/bl  
label
```
° 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 absolute
Handling Addresses by Assembler (#2/2)

° Loads and stores to variables in static area
  `ldr/str Rdest, [pc, offset]`
  • Such addresses are stored in the literal pool by the compiler/Assembler
  • The reference to the literal pool is via PC relative addressing
  • 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.