CSE 30: Computer Organization and Systems Programming

Lecture 16:
Nested Procedures
ARM Assembler directives
Data alignment

Diba Mirza
University of California, San Diego
Nested Procedures

```c
int sumSquare(int x, int y){
    return (mult(x,x)+y);
}
```

Which function violates the ARM Procedure call standard?

A. sumSquare
B. mult
C. main
D. None of the above
int sumSquare(int x, int y) {
    return (mult(x, x) + y);
}

sumSquare: push {r4-r11}

MOV r4, r1
MOV r1, r0
BL mult
ADD r0, r0, r4
pop {r4-r11}

Which instruction is executed after BX lr?
A. MOV r3, r0
B. ADD r0, r0, r4
C. None of the above
Nested Procedures

int sumSquare(int x, int y) {
    return (mult(x, x) + y);
}

Which function should save the value of lr?
A. sumSquare
B. mult
C. Both
Storing constants in ARM

Convert the following C statement to ARM

```c
int x = 0xaabbcddd;
printf("%d\n", x);
```

%int printf(const char *format, ...)

- We need to store the following somewhere
  1. The string "\%d\n"
  2. The value 0xaabbcddd

- Where should we NOT store the characters of the string "\%d\n"?
  A. Register
  B. In memory: Data segment
  C. In memory: On the stack
  D. In memory: On the heap
  E. None of the above
Storing constants in data segment

\[ x=0xaabbccdd; \]
\[ \text{printf(“\%d\n”,} x); \]

.data

@Switch the destination of the following statements to the data section

Now specify your data in any of the following formats

- .byte ‘\%',`d','\n', 0
- .ascii “\%d\n\0”
- .asciz “\%d\n”
- .hword 0x6425, 0x000A
- .word 0x0A6425

Ascii values

<table>
<thead>
<tr>
<th>symbol</th>
<th>Dec</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>\n</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>\0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Which one is most convenient for storing the value of \( x \)?
- How about the string “\%d\n” asciiz “\%d\n”
Labeling constants in data segment

\texttt{x=0xaabbccdd;}\nnprintf(“%d\n”, \texttt{x});

\texttt{.data} \quad \text{@Switch the destination of the following }\n\text{@} \quad \text{statements to the data section}

\texttt{mystx: .asciz “%d\n”} \n\texttt{mynum: .word \textcolor{red}{0x\textbf{aabbccdd}}} \n
\textcolor{red}{\underline{What should be the first argument to \texttt{printf}?}}

\textcolor{red}{\textbf{A.} The memory address of ‘%’} \n\textbf{B.} The 4 byte binary pattern for: %d\n\textbf{C.} None of the above
Loading constants into registers

\[ x = 0xaabbccdd; \]
\[ \text{printf(“\%d\n”,} x); \]

.data
  @Switch the destination of the following statements to the data section
  mystr: .asciz “%d\n”
  mynum: .word 0xaabbccdd

print_fun:  LDR r0,=mystr
            LDR r1,=mynum

What is stored in r1 after the above statements execute?
A. The value: 0xaabbccdd
B. The memory address of 0xaabbccdd
C. None of the above
Complete the ARM code

x=0xaabbcddd;
printf("%d\n",x);

.data          @Switch the destination of the following
               @statements to the data section
myst: .asciz "%d\n"
mynum: .word 0xaabbcddd

print_fun:    LDR r0,=myst
              LDR r1,=mynum
              LDR r1, [r1]
              BL printf
              BX LR

Nor quite right yet.
Alignment

\[ x=0xaabbccdd; \]
\[ \text{printf}("\%d\n", x); \]

.data  @Switch the destination of the following statements to the data section
    @statements to the data section
.text
.align 2
mystr: .asciz \"%d\n\"
mynum: .word 0xaabbccdd

print_fun: push {lr}
    LDR r0,=mystr
    LDR r1,=mynum
    LDR r1,[r1]
    BL printf
pop {lr}
BX lr

What happens when this code is compiled?
A. Error: unaligned opcodes detected in executable segment
B. Code compiles without error
Alignment requirements

• Data objects should start at memory addresses that are divisible by the size of the data object

  • short (2 byte) at address divisible by 2 $0b_{ }_{ }_{ }_{ }_{ }_0$
  • int (4 byte) at address divisible by 4 $0b_{ }_{ }_{ }_{ }_00$
  • double (8 byte) at address divisible by 8 $0b_{ }_{ }_{ }_{ }_000$

• The stack pointer should be 8 byte aligned
• Instructions should be 4 byte aligned
struct p {
    char x;
    int y;
};

Struct p sp;

What is the size of sp?

Without data alignment = 5
With data alignment = 8
Data Alignment

- Processors do not always access memory in byte sized chunks, instead in 2, 4, 8, even 16 or 32 byte chunks.
- Boundaries at which data objects are stored affects the behavior of read/write operations into memory.

Programmer's view of memory

Processor's view of memory
Data Alignment

- Consider the task of reading 4 bytes from memory starting at 0x00 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?
Why data alignment?

- Now consider the task of reading 4 bytes from memory starting at 0x01 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?
  - Some processors just would not allow this scenario because it is extra work for the h/w and it affects the performance.