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);
}
```

sumSquare:

```assembly
MOV r4, r1
MOV r1, r0
BL mult
ADD r0, r0, r4
BX lr
```

main: MOV r0, #2
      MOV r1, #3
      BL sumSquare
      MOV r3, r0

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);
}

Which instruction is executed after BX lr?
A. MOV r3, r0
B. ADD r0, r0, r1
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

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

\[
\%\text{int printf} \left( \text{const char *format, ...} \right)
\]

• We need to store the following somewhere
  1. The string “%d\n”
  2. The value 0xaabbccdd

• 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
Storing constants in data segment

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

\texttt{.data \ Switch the destination of the following statements to the data section}

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
symbol & Dec & Hex \\
\hline
% & 37 & 25 \\
\hline
d & 100 & 64 \\
\hline
\n & 10 & A \\
\hline
\0 & 0 & 0 \\
\hline
\end{tabular}
\end{center}

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

```c
x=0xaabbccdd;
printf(“%d\n”,x);
```

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

mystr: .asciz “%d\n”
mynum: .word __________
```

What should be the first argument to `printf`?

A. The memory address of ‘%’
B. The 4 byte binary pattern for: %d\n\0
C. None of the above

<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>
Loading constants into registers

\[
x=0xaabbccdd;
\]

\[
printf("\%d\n",x);
\]

\[
.data
\]

\[
@Switch the destination of the following statements to the data section
\]

\[
.mys\text{\textbar} \text{str}:\text{.asciz "\%d\n"
\]

\[
\text{mynum}:\text{.word }0xaabbccdd
\]

\[
print\text{\textbar}fun:\text{LDR r0,=mys\text{\textbar}str}
\]

\[
\text{LDR r1,=mynum}
\]

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=0xaabbccdd;
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
Alignment

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

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

.text
.align 2
mystr: .asciz "%d\n"
mynum: .word  0xaabbccddd

print_fun:   push {lr}
     LDR r0,=myst
     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?
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:

```
0x00
0x01
0x02
0x03
0x04
0x05
```

Processor’s view of memory:

```
0x00
0x02
0x04
```
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?

1 byte reads
0x00
0x01
0x02
0x03
0x04
0x05
1 byte reads
0x00
0x02
0x04
2 byte reads
0x00
0x04
4 byte reads
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.