Programming the ARM

Computer Design 2002, Lecture 4
Robert Mullins
Quick Recap

- The Control Flow Model
  - Ordered list of instructions, fetch/execute, PC

- Instruction Set Architectures
  - Types of internal storage
    - Accumulator, stack and general purpose register machines
  - Addressing modes
    - Register indirect, displacement, memory indirect etc.
  - Branch conditions
    - Condition codes, condition registers, branch and compare
  - Encoding Instructions
    - Fixed length or variable length encodings
    - Representing immediates, opcode and operands
Quick Recap

- Making the common case fast and Amdahl’s Law
  - See H&P (chapters 1 and 2)

- The Big Picture - Interaction between the compiler, architecture and instruction set
  - E.g. a poorly designed instruction set may make high-performance implementation difficult and restrict the effectiveness of an optimizing compiler.
Lecture 4

This Lecture

- Implementing functions and procedures
- The ARM Procedure Call Standard (APCS)
- Development tools
- Practical 4 of the ECAD labs
Functions and Procedures

- Structure program (abstraction/hierarchy)
- Package useful code so it can be reused
  - Must use a well defined interface

Questions
- How do we pass arguments to the function?
- How do we obtain the result?
Procedure Calls

- Register values are preserved by saving and restoring them from a stack in memory
- Agree on a standard, define how registers are used (part of ABI – Application Binary Interface)
  - Which registers hold arguments
  - Which register holds the result (or pointer to)
  - Some registers may need to be saved (preserved) by caller (**caller-saved**)
  - Some may need to be saved by callee (**callee-saved**)
ARM Procedure Call Standard

- There are in fact many variants, let’s look at base standard

- Four argument registers (r0-r3)
  - Not preserved by routine/function (callee)
  - May be saved by caller if necessary
  - May be used to return results to caller

- Registers r4-r11 are typically used to hold the routine’s local variables
  - The value of these registers remains unchanged after a subroutine call
  - Registers may need to be saved and restored by callee

- Registers r12-r15 have special dedicated roles
  - e.g. the link register holds the return address
# APCS Register Usage Convention

<table>
<thead>
<tr>
<th>Register</th>
<th>APCS Name</th>
<th>APCS Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a1</td>
<td>argument 1 / result / scratch register for function</td>
</tr>
<tr>
<td>1</td>
<td>a2</td>
<td>argument 2 / result / scratch register for function</td>
</tr>
<tr>
<td>2</td>
<td>a3</td>
<td>argument 3 / result / scratch register for function</td>
</tr>
<tr>
<td>3</td>
<td>a4</td>
<td>argument 4 / result / scratch register for function</td>
</tr>
<tr>
<td>4</td>
<td>v1</td>
<td>variable register 1</td>
</tr>
<tr>
<td>5</td>
<td>v2</td>
<td>variable register 2</td>
</tr>
<tr>
<td>6</td>
<td>v3</td>
<td>variable register 3</td>
</tr>
<tr>
<td>7</td>
<td>v4</td>
<td>variable register 4</td>
</tr>
<tr>
<td>8</td>
<td>v5</td>
<td>variable register 5</td>
</tr>
<tr>
<td>9</td>
<td>sb or v6</td>
<td>static base or variable register 6</td>
</tr>
<tr>
<td>10</td>
<td>sl or v7</td>
<td>stack limit or variable register 7</td>
</tr>
<tr>
<td>11</td>
<td>fp</td>
<td>frame pointer (pointer to start of current stack frame) or variable register 8</td>
</tr>
<tr>
<td>12</td>
<td>ip</td>
<td>the intra-procedure-call scratch register</td>
</tr>
<tr>
<td>13</td>
<td>sp</td>
<td>lower end of current stack frame (stack pointer)</td>
</tr>
<tr>
<td>14</td>
<td>lr</td>
<td>the link register (return address)</td>
</tr>
<tr>
<td>15</td>
<td>pc</td>
<td>the program counter</td>
</tr>
</tbody>
</table>
Parameter Passing and result return

- a1-a4 (r0-r3) may be used to hold parameters
- Additional parameters may be passed on the stack
- Results may be returned in a1-a4 as a value or indirectly as an address
## Simple function call

<table>
<thead>
<tr>
<th>Instruction</th>
<th>registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r0, #10</td>
<td>r0</td>
<td>Move 10 into register r0</td>
</tr>
<tr>
<td>MOV r1, #5</td>
<td>r1</td>
<td>Move 5 into register r1</td>
</tr>
<tr>
<td>BL max</td>
<td></td>
<td>Call function max</td>
</tr>
<tr>
<td>CMP r0, r1</td>
<td>r0, r1</td>
<td>Compare r0 and r1, set flags</td>
</tr>
<tr>
<td>MOVLT r0, r1</td>
<td>r0, r1</td>
<td>If r0 &lt; r1, move value of r1 to r0</td>
</tr>
<tr>
<td>MOV pc, lr</td>
<td></td>
<td>Move program counter to link register</td>
</tr>
</tbody>
</table>

For simple leaf functions that only use r0-r3 (a1-a4) the function overhead is small as no registers need to be saved. In real programs around half the functions may be simple leaf functions like this.
Saving and restoring registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>myfunction</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BL</strong> myfunction</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>.....</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STMFD sp!, {r4-r10, lr} ; save registers</td>
</tr>
<tr>
<td></td>
<td><strong>.....</strong></td>
</tr>
<tr>
<td></td>
<td><strong>.....</strong></td>
</tr>
<tr>
<td></td>
<td>LDMFD sp!, {r4-r10, pc} ; restore and</td>
</tr>
<tr>
<td></td>
<td>; return</td>
</tr>
</tbody>
</table>
The Stack

Full-descending stack - ARM stack grows down, stack pointer points to last entry (not next free entry)
ARM Assembler Examples
Assembler Directives

- Information for the assembler
- Common uses
  - Define and name new section, code or data
  - Constants, aliases
  - Allocate bytes of memory as data and initialize contents
Allocating memory for data

- Allocate bytes of memory (DCB)
  C_String DCB "MyString", 0

- Allocate words of memory (DCD)
  Data DCD 1234, 1, 5, 20

- Reserve a ‘zeroed’ block of memory (SPACE or %)
  table % 1024 ; 1024 bytes of zeroed memory
### Writing Assembler, Hello World Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY</td>
<td>entry point into program</td>
</tr>
<tr>
<td>START ADR r1, TEXT</td>
<td>ADD r1, pc, offset of TEXT</td>
</tr>
<tr>
<td>LOOP LDRB r2, [r1]</td>
<td>load character</td>
</tr>
<tr>
<td>CMP r2, #0</td>
<td>check we are not at end of string</td>
</tr>
<tr>
<td>MOV r0, #SWI_WriteC</td>
<td>call putchar (r1 points to char)</td>
</tr>
<tr>
<td>SWINE 0x123456</td>
<td>do system call</td>
</tr>
<tr>
<td>ADD r1, r1, #1</td>
<td>increase our pointer</td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>loop if we are not finished</td>
</tr>
<tr>
<td>MOV r0, #SWI_Exit</td>
<td>standard exit system call</td>
</tr>
<tr>
<td>MOV r1, #0x20000</td>
<td></td>
</tr>
<tr>
<td>ADD r1, r1, #0x26</td>
<td></td>
</tr>
<tr>
<td>SWI 0x123456</td>
<td></td>
</tr>
</tbody>
</table>

#### Sections

- **SWI_WriteC**
- **SWI.Exit**
- **ENTRY**
- **START LOOP**
- **TEXT**
- **END**

#### Code

```assembly
; create new code area called ‘hello’
AREA hello, CODE, READONLY

SWI_WriteC EQU 0x3 ; symbolic name for constant
SWI.Exit EQU 0x18

ENTRY

START ADR r1, TEXT
ADD r1, pc, offset of TEXT

LOOP LDRB r2, [r1]
CMP r2, #0

MOV r0, #SWI_WriteC
SWINE 0x123456

ADD r1, r1, #1
BNE LOOP

MOV r0, #SWI.Exit
MOV r1, #0x20000
ADD r1, r1, #0x26
SWI 0x123456

TEXT DCB "Hello World\n", 0

END ; end of source file!
```
The program in memory....
int fib (int a) {
    if (a<=1) return a;
    else return fib (a-1)+fib(a-2);
}

fib:

    stmfd r13!, {r4, r5, r14} ; preserve registers
    mov r4, r0                  ; r4=a
    cmp r0, #1                  ; compare a with 1
    movle r0, r4                 ; (if a<=1) result=a
    ldmlefd r13!, {r4, r5, pc}    ; (if a<=1) return
    sub r0, r4, #1
    bl fib                       ; fib (a-1)
    mov r5, r0                    ; r5=fib(a-1)
    sub r0, r4, #2
    bl fib                       ; fib (a-2)
    add r0, r5, r0                ; result=fib(a-1)+fib(a-2)
    ldmfd r13!, {r4, r5, pc}      ; return
R13 = 0x07FFFFFFF8 (stack pointer)
R4 = A720, R5 = 8ED8
R14 = 80E8 (address after BL fib instruction in main)

STMFD r13!, {r4, r5, r14}
Fib(5)

R5
R4

Return Address to Main()

R4=5
Fib(R4-1)

R5
R4 (a=5)

Return address

Fib(3)….. etc.

Fib(R4-2)

Return sum
ECAD Workshop Four

Sieve of Eratosthenes

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 0 0 0 0 0 0 0 0 0 0 0 0

2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 1 0 1 0 1 0 1 0 1 0 1 0

2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 1 0 1 0 1 1 1 0 1 0 1 1
```
Later Workshops

- Make sure you use procedure call standard correctly
  - Use branch-and-link (BL) instruction to call procedure
  - Pass parameters in correct registers
  - Save registers when necessary
  - Restore registers on exit
Next Lecture

- OS Support and Memory Management
  - Virtual Memory
  - Interrupts/Exceptions
  - ARM specifics (operating modes, page table organisation etc.)
Self Study/Supervision Work

- Write a simple C program to sum the numbers 1 to n, compile it and examine the assembler produced
  - You might want to try executing the program in the debugging environment, this will allow you to single step through the program. You can look at the contents of the registers and memory as you go.

- Challenge
  - Write a program (in ARM assembly language) to reverse the bytes of a 32-bit register
  - It’s possible to do this using only one additional register to hold temporary results and 4 ARM instructions!