Topic: ARM Procedures

CSE 30: Computer Organization and Systems Programming

Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego
C functions

```c
main() {
    int a, b, c;
    ...
    c = sum(a, b); /* a, b, c: r0, r1, r2 */
    ...
}

/* sum function */
int sum(int x, int y) {
    return x + y;
}
```

What information must compiler/programmer keep track of?

Calle\textsubscript{R}: the calling function
Calle\textsubscript{E}: the function being called

What instructions can accomplish this?
Steps needed for function call & return

1. Transfer control to the function being called (callee) (This is some location in memory different from the current address in pc)
2. Pass parameters to the function
3. Transfer control back to the caller once function execution is complete
4. Make return values available to caller function

Let’s focus on the transfer of control to and from the function called
In ARM, all instructions are stored in memory just like data. So here we show the addresses of where the programs are stored.
Using the branch instruction….

... sum(a,b);... /* a,b:r4,r5 */
}
int sum(int x, int y) {
    return x+y;
}

Is there something wrong with using the simple branch instruction?

address
1000 ...
1004 ...
1008 ...
1012 B sum ; branch to sum
1016 return_loc: ...
1020 ...
2000 sum: ADD r0,r0,r1
2004 B return_loc

A. Nothing is wrong
B. It is not sufficient for generic usage scenarios
Using the branch instruction....

... sum(a,b);... /* a,b:r4,r5 */
}

int sum(int x, int y) {
    return x+y;
}

Is there something wrong with using the simple branch instruction?

A. Nothing is wrong
B. It is not sufficient for generic usage scenarios

Reason: sum might be called by many functions, so we can’t return to a fixed place.

The calling proc to sum must be able to say “return back here” somehow.
Instruction Support for Functions

... sum(a,b);... /* a,b:r4,r5 */
}

int sum(int x, int y) {
    return x+y;
}

address
1000 ...
1004 ...

1008 MOV  lr,1016 ; lr = 1016
1012 B    sum ; branch to sum
1016 ...
1020 ...

2000 sum: ADD r0,r0,r1
2004 BX lr ; MOV pc,lr i.e., return
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}

Question: Why use BX here? Why not simply use B?

Answer: sum might be called by many functions, so we can’t return to a fixed place. The calling proc to sum must be able to say “return here” somehow.
Instruction Support for Functions

- Single instruction to jump and save return address: jump and link (BL)

- Before:
  1008  MOV lr, 1016 ; lr=1016
  1012  B sum ; goto sum

- After:
  1008  BL sum  # lr=1012, goto sum

- Why have a BL? Make the common case fast: function calls are very common. Also, you don’t have to know where the code is loaded into memory with BL.
Instruction Support for Functions

- Syntax for BL (branch and link) is same as for B (branc):
  \[
  \text{BL} \quad \text{label}
  \]

- BL functionality:
  - Step 1 (link): Save address of \textit{next} instruction into \texttt{lr} (Why next instruction? Why not current one?)
  - Step 2 (branch): Branch to the given label
Instruction Support for Functions

- Syntax for BX (branch and exchange):
  
  BX register

- Instead of providing a label to jump to, the BX instruction provides a register which contains an address to jump to

- Only useful if we know exact address to jump

- Very useful for function calls:
  - BL stores return address in register (lr)
  - BX lr jumps back to that address
Things to think on…

Following questions remain:

- How do we pass arguments?
  - Use registers?
- How do we share registers between caller and callee

1. Need to follow a standard on the usage of registers by caller and callee. We have a std: The ARM Application Procedure Call Std. (AAPCS)
2. Need to interact with memory (done via stack)
Steps for Making a Procedure Call

1. Save necessary values onto stack
2. Assign argument(s), if any
3. BL call
4. Restore values from stack
5. Must follow register conventions
Rules for Procedures

1. Called with a \texttt{BL} instruction, returns with a \texttt{BX lr} \texttt{(or MOV pc, lr)}
2. Accepts up to 4 arguments in r0, r1, r2 and r3
3. Return value is always in r0 (and if necessary in r1, r2, r3)
4. Must follow \texttt{register conventions} (even in functions that only you will call)! So what are they?
Register Conventions: A set of generally accepted rules as to which registers are guaranteed to be unchanged after a procedure call (BL) and which may be changed.
## Register Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>Arguments into function</td>
</tr>
<tr>
<td>r1</td>
<td>Result(s) from function</td>
</tr>
<tr>
<td>r2</td>
<td>otherwise corruptible</td>
</tr>
<tr>
<td>r3</td>
<td>(Additional parameters passed on stack)</td>
</tr>
<tr>
<td>r4</td>
<td>Register variables</td>
</tr>
<tr>
<td>r5</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>r6</td>
<td>- Stack base</td>
</tr>
<tr>
<td>r7</td>
<td>- Stack limit if software stack checking selected</td>
</tr>
<tr>
<td>r8</td>
<td>Scratch register (corruptible)</td>
</tr>
<tr>
<td>r9/sp</td>
<td>- SP should always be 8-byte (2 word) aligned</td>
</tr>
<tr>
<td>r10/sl</td>
<td>- R14 can be used as a temporary once value stacked</td>
</tr>
<tr>
<td>r11</td>
<td>Scratch register (corruptible)</td>
</tr>
<tr>
<td>r12</td>
<td>Scratch register (corruptible)</td>
</tr>
<tr>
<td>r13/sp</td>
<td>Scratch register (corruptible)</td>
</tr>
<tr>
<td>r14/lr</td>
<td>Scratch register (corruptible)</td>
</tr>
<tr>
<td>r15/pc</td>
<td>Scratch register (corruptible)</td>
</tr>
</tbody>
</table>

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see AAPCS).

CPSR flags may be corrupted by function call. Assembler code which links with compiled code must follow the AAPCS at external interfaces.

- Stack base
- Stack limit if software stack checking selected
## ARM Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
<th>Role in Procedure Call Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0-r1</td>
<td>a1-a2</td>
<td>Argument/Result/Scratch Register</td>
</tr>
<tr>
<td>r2-r3</td>
<td>a3-a4</td>
<td>Argument/Scratch Register</td>
</tr>
<tr>
<td>r4-r8</td>
<td>v1-v5</td>
<td>Variable Register</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
<td>Variable Register</td>
</tr>
<tr>
<td>r10-r11</td>
<td>v7-v8</td>
<td>Variable Register</td>
</tr>
<tr>
<td>r12</td>
<td>ip</td>
<td>Intra-Procedure Call Scratch Register</td>
</tr>
<tr>
<td>r13</td>
<td>sp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>r14</td>
<td>lr</td>
<td>Link Register</td>
</tr>
<tr>
<td>r15</td>
<td>pc</td>
<td>Program Counter</td>
</tr>
</tbody>
</table>
Saved Register Conventions

- **r4-r11 (v1-v8):** *Restore if you change.* Very important. If the **callee** changes these in any way, it must restore the original values before returning.

- **sp:** *Restore if you change.* The stack pointer must point to the same place before and after the **BL** call, or else the caller won’t be able to restore values from the stack.
**Volatile Register Conventions**

- *lr*: **Can Change.** The BX call itself will change this register. **Caller** needs to save on stack if nested call.

- *r0-r3 (a1-a4)*: **Can change.** These are volatile argument registers. **Caller** needs to save if they’ll need them after the call. E.g., r0 will change if there is a return value.

- *r12 (ip)* may be used by a linker as a scratch register between a routine and any subroutine it calls. It can also be used within a routine to hold intermediate values *between* subroutine calls.
Register Conventions

What do these conventions mean?

- If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a BL call.
- Function E must save any saved registers it intends to use before garbling up their values.
- Remember: Caller/callee need to save only volatile/saved registers they are using, not all registers.
main() {
    int i, j, k, m; /* i-m:v0-v3 */
    ...
    i = mult(j, k); ...
    m = mult(i, i); ...
}

int mult (int mcand, int mlier){
    int product;
    product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;}

Example: Caller Assembly code

```assembly
main() {
    int i,j,k,m; /* i-m:v0-v3 */
    ... 
    i = mult(j,k); ... 
    m = mult(i,i); ... }

_start: 
    MOV a1,v1 ; arg1 = j 
    MOV a2,v2 ; arg2 = k 
    BL mult ; call mult 
    MOV v0,r0 ; i = mult() 
    ... 
    MOV a1,v0 ; arg1 = i 
    MOV a2,v0 ; arg2 = i 
    BL mult ; call mult 
    MOV v3,r0 ; m = mult() 
    ... 

done
```
Example

Notes:

- `main function ends with done, not BX lr`, so there’s no need to save `lr` onto stack
- All variables used in `main function` are saved in registers, so there’s no need to save these onto stack
Example: Callee Code

```c
int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}

int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}
```

```
mult: MOV r3,#0   ; prod=0
Loop: CMP r2, #0   ; mlier == 0?
      BLE Fin   ; if mlier <= 0 goto Fin
      ADD r3,r3,r1 ; product += mcand
      ADD r2,r2,#-1 ; mlier -= 1
      B Loop    ; goto Loop

Fin:  MOV r0,r3   ; setup return value
      BX lr     ; return
```
Example

Notes:

- No **BL** calls are made from **mult** and we don’t use any saved registers, so we don’t need to save anything onto stack
- Temp registers are used for intermediate calculations (could have used saved registers, but would have to save the caller’s on the stack.)
- **r2** is modified directly (instead of copying into a temp register) since we are free to change it
- Result is put into **r0** before returning
Conclusion

- Functions are called with BL, and return with BX lr.
- The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.

**Register Conventions**: Each register has a purpose and limits to its usage. Learn these and follow them, even if you’re writing all the code yourself.
Instructions so far:

- Previously:
  - ADD, SUB, MUL, MULA, [U|S]MULL, [U|S]MLAL, RSB
  - AND, ORR, EOR, BIC
  - MOV, MVN
  - LSL, LSR, ASR, ROR
  - CMP, B{EQ,NE,LT,LE,GT,GE}
  - LDR, STR, LDRB, STRB, LDRH, STRH

- New:
  - BL, BX

Registers we know so far

- All of them!