CSE 30: Computer Organization and Systems Programming

Lecture 15: ARM Loops and ARM Procedure Call Standard

Diba Mirza
University of California, San Diego
While loops

\[ a : ro \]

\[
\text{while (a<0)} \]
\[ a++; \]

\[
\text{do} \]
\{
  \[ a++; \]
\}\text{while (a<0);}
For loops

```c
for (i=0; i<10; i++) {
    a++;
    b--;
}
```
C functions

main() {
    int a=10, b=20, c;
    c = sum(a, b); /* a, b, c: r0, r1, r2*/
    ...
}

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

main() {
    int a=10, b=20, c;
    c = sum(a, b);
    ...
}

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

Steps in function call

1. Pass parameters (using registers)
2. Call function (sum)
3. Compute return value (put it in the right register)
4. Transfer control back to caller

Steps in function call
Making the function call

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

address
1000 ... Is there something wrong with using the simple branch instruction?  A. Yes
1004 ... B. No
1008 ...
1012 B sum ; branch to sum
1016 return_loc:... 
1020 ...
2000 sum: ADD r0,r0,r1
2004 B return_loc ×

First instruction of sum
Using the branch instruction….

c=sum(a,b)
c=c+1;
.....
}
int sum(int x, int y) {
    return x+y;
}

address Is there something wrong with using the simple branch instruction?
1000 ... A. Yes
1004 ...
1008 ...
1012 B sum ;branch to sum
1016 return_loc:...
1020 ...
2000 sum: ADD r0,r0,r1
2004 B return_loc X

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.
Making a function call

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

address
1000   ...
1004   ...
1008   MOV  lr, 1016
1012   B    sum
1016   ...
1020   ...
2000   sum: ADD  r0,r0,r1
2004   BX  lr ; MOV pc,lr i.e., return
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 ; go to 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.
Passing arguments & return values

main() {
    int a=10, b=20, c;
    c = sum(a, b);
    c = c + 1;
    ...
}

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

- **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

## Arguments into function
- r0
- r1
- r2
- r3

## Result(s) from function
- otherwise corruptible

(Additional parameters passed on stack)
- r4
- r5
- r6
- r7
- r8
- r9/sb
- r10/s1
- r11

## Register variables
- Must be preserved

## Scratch register
- (corruptible)
- r12

## Stack Pointer
- r13/sp

## Link Register
- r14/lr

## Program Counter
- r15/pc

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.

Callee (sum) has to make sure that the values in r4–r11 are unchanged by it.

- Stack base
- Stack limit if software stack checking selected

- SP should always be 8-byte (2 word) aligned
- R14 can be used as a temporary once value stacked
int mult (int mcand, int mlier){
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}

How to save the “saved registers”?

mult:    push $s8 - s11
        MOV r4, #0 ; prod=0
Loop:    CMP r1, #0 ; mlier == 0?
        BLE Fin       ; if mlier <= 0 goto Fin
        ADD r4, r4, r1 ; product += mcand
        ADD r1, r1, #1 ; mlier -= 1
        B Loop        ; goto Loop
Fin:      BX lr       ; return
Example: Caller Assembly code

```c
main() {
    int i, j, k, m; /* i-m:r4-r7 */
    i = mult(j, k);
    m = mult(i, i) + j;
}
```

```
main:
    MOV r0, r5 ; arg1 = j
    MOV r1, r6 ; arg2 = k
    BL mult ; call mult
    MOV r4, r0 ; i=mult()
    MOV r1, r0 ; arg2 = i
    BL mult ; call mult
    ADD r7, r0, r5 ; m=mult(i,i)+j
```
Example: Caller Assembly code

```assembly
main() {
    int i, j, k, m; /* i-m:r4-r7 */
    i = mult(j, k);
    m = mult(i, i) + j;
}

main:
    MOV r0, r5 ; arg1 = j
    MOV r1, r6 ; arg2 = k
    BL mult ; call mult
    MOV r4, r0 ; i=mult()
    MOV r1, r0 ; arg2 = i
    BL mult ; call mult
    ADD r7, r0, r5 ; m = mult(i, i) + j
```
int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}

How many errors are there in the ARM code given below?

A. Zero
B. One
C. Two
D. Three
E. More than three

Mult: push {r4, r3}

MOV r4, #0; prod=0

Loop: CMP r1, #0; mlier == 0?

BGT Fin ; if mlier > 0 goto Fin

ADD r4, r4, r1 ; product += mcand

ADD r1, r1, #1-1; mlier -= 1

B Loop ; goto Loop

Fin: BX lr ; return

Fin: pop r3, r4, r0; more final value in r0
node *list_add(node* head, int new_value)
{
    node *new_node=(node*) malloc(sizeof(node));
    if (!new_node)
        return NULL;
    new_node->value = new_value;
    //Line 5
}

What should Line 5 be to achieve the pointer diagram (below)?

A. new_node->next = head;
B. next=head;
C. head=new_node;
D. new_node->next = *head;
node *list_add(node* head, int new_value) {
    node *new_node=(node*) malloc(sizeof(node));
    if (new_node==NULL)
        return NULL;
    new_node->value = new_value;
    new_node->next = head;
    return new_node;
}