Procedure/Subroutine Calls For ARMv4

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

ARMv4 Call and Return Instructions

- The “bl target” (branch to target and link) instruction is used to transfer control to the target subroutine
- The PC is set to the target and the LR (link register) is set to the instruction following the bl instruction
- The LR is then used to return from the subroutine
- The “mov pc, lr” set the PC to the LR, transferring control back from the called routine

```
0004 bl 000A // LR = PC + 4 = 0008; PC = 000A (move to my_func)
0008 some_instr
     ...
000A my_func
000B other_instr
     ...
000F mov pc, lr // PC = 0008
```

ARMv4 register convention

The ARMv4 register usage convention is:

- R0 - argument, return value, temporary
- R1-R3 - arguments, temporaries
- R4-R11 - callee-save
- R12 - temporary
- R13 (SP) - Stack Pointer
- R14 (LR) - Link Register
- R15 (PC) - Program Counter

ARMv4 register convention

- R0–R3 and R12 are used to hold temporary results
  - Non-preserved
  - These calculations typically complete before a function call is made
  - They are not preserved, and it is rare that the caller needs to save them
- R0–R3 are often overwritten in the process of calling a function
  - They must be saved by the caller
  - if the caller depends on any of its own arguments after a called function returns
- LR, R4 - R11 must be saved in the stack and restored later by the callee
  - callee-save registers

ARMv4 register convention

- These conventions means that:
  - After any routine call, the register r0, r1, r2, r3, and r12 could be changed
  - The LR (link register) must be saved if any subroutine is called in the current routine
  - The SP (stack pointer) must be preserved across subroutine calls
    - The number of pushes must match the number of pops
  - If registers R4 to R11 are changed in a routine, they first must be saved, usually with a stack push
  - Compiler writers must follow this conventions
  - Software libraries must follow these conventions

abc in ARMv4 (abc.s)

The ARMv4 assembly language version of the ABC program we saw in python

- PC = 0

- "_start" routine is called first
- "_start" then calls "a"
- What is in the LR?
  - LR = 4
- What is the PC?
  - PC = a

```
_start:
        0: bl  a
        4: mov r0, #0 // exit sys call
        8: mov r7, #1
        16: swi #0
                .size   _start, .-_start
```

abc in ARMv4 (abc.s)

- Inside function a
  - LR = 4
- push LR to the stack
  - Stack = [4]
- Prepare message for print “start a”
- “a” calls “print” (see later)
- “a” calls “b”
- LR = 26
- PC = b

```
a:

    10: push  {lr}

    14: ldr   r0, =a_start_msg

    18: bl    print

    22: bl    b

    26: ldr   r0, =a_end_msg

    30: bl    print

    34: pop   {lr}

    38: mov   pc, lr
```

abc in ARMv4 (abc.s)

- Inside function b
  - LR = 26
- push LR to the stack
  - Stack = [26, 4]
- Prepare message for print “start b”
- “b” calls “print” (see later)
- “b” calls “c”
- LR = 116
- PC = c

```
b:
    100: push  {lr}
    104: ldr   r0, =b_start_msg
    108: bl    print
    112: bl    c
    116: ldr   r0, =b_end_msg
    120: bl    print
    124: pop   {lr}
    128: mov   pc, lr
```

abc in ARMv4 (abc.s)

- Inside function c
  - LR = 116
- push LR to the stack
  - Stack = [116, 26, 4]
- Prepare message for print “start/end c”
- “c” calls “print” (see later)
- pop LR from the stack
  - LR = 116
  - Stack = [26, 4]
- mov PC <= LR (go back to “b”)
  - PC = 116

```
c:
    200: push  {lr}
    204: ldr   r0, =c_msg
    208: bl    print
    212: pop   {lr}
    216: mov   pc, lr
```

abc in ARMv4 (abc.s)

Returning from “c”

- PC = 116
- LR = 116
- Stack = [26, 4]
- Prepare message for print “end b”
- “b” calls “print” again
- pop LR from the stack
  - LR = 26
  - Stack = [4]
- mov PC <= LR (go back to “a”)
  - PC = 26

```
b:
    100: push  {lr}
    104: ldr   r0, =b_start_msg
    108: bl    print
    112: bl    c
    116: ldr   r0, =b_end_msg
    120: bl    print
    124: pop   {lr}
    128: mov   pc, lr
```

abc in ARMv4 (abc.s)

After returning from “b”

- PC = 26
- LR = 26
- Stack = [4]
- Prepare message for print “end a”
- “a” calls “print”
- pop LR from the stack
  - LR = 4
  - Stack = []
- mov PC <= LR (go back to “_start”)0
  - PC = 4

```
a:
    10: push   {lr}
    14: ldr    r0, =a_start_msg
    18: bl     print
    22: bl     b
    26: ldr    r0, =a_end_msg
    30: bl     print
    34: pop    {lr}
    38: mov    pc, lr
```

abc in ARMv4 (abc.s)

- PC = 4
- Prepare return system call value “0”
- R7, holds syscall number identification
- swi, interruption/call
- Program ends
- OS receives “0” and believes everything was fine
- Different return values
  - Different errors/messages to the OS

```
_start:

        0: bl  a

        4: mov r0, #0 // exit sys call

        8: mov r7, #1

        12: swi #0

        .size   _start, .-_start
```

abc in ARMv4 (abc.s)

Print subroutine (extra)

- r0 is NUL terminated string
- count to '\0'

```
print:
        push   {r7}
        mov    r1, #0
print_cnt:
        ldrb r2,[r0,r1]
        cmp   r2, #0
        beq   print_pr
        add   r1, r1, #1
        b     print_cnt
```

abc in ARMv4 (abc.s)

- setup and write syscall
- byte count
- bytes to write
- selecting stdout
- write syscall
  - 4 = print_string
- raise an interruption
- pop lr from the stack
- set pc <= lr (go back)

- Read only section with string ending in zero

```
print_pr:
        mov   r2, r1
        mov   r1, r0
        mov   r0, #1
        mov   r7, #4
        swi   #0
        pop   {r7}
        mov   pc, lr
c_msg:
        .asciz   "start/end\n"
```

Next steps

Arm subroutines examples