Subroutine Examples

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

(a - 2*b) + (c - d) subroutine

- "expr.s" (lec 20) program can be rewritten as a subroutine
- r_expr.S
  - 1 bit left shift multiplication
  - Subtraction using sub
  - Addition with add
  - mov pc, lr return to caller
- Let's run a table test:
  - r0=2, r1=1, r2=4, r3=1

```
// expression for (a - 2*b) + (c-d)
// a is r0, b is r1, c is r2, d is r3
r_expr:
        lsl r1, r1, #1 // b = 2 * b
        sub r0, r0, r1 // a = a - b
        sub r2, r2, r3 // c = c - d
        add r0, r0, r2 // a = a + c
        mov pc, lr
```


MEMORIAL UNIVERSITY
www.mun.ca

(a - 2*b) + (c - d) subroutine

- a, b, c, and d are passed into the subroutine with r0, r1, r2, and r3
- The result is returned in r0
- The r_expr is a leaf routine
- It does not call any other routines
    - No need to save lr since it is not modified
    - lr is not pushed

```
// expression for (a - 2*b) + (c-d)
// a is r0, b is r1, c is r2, d is r3
r_expr:
        lsl r1, r1, #1 // b = 2 * b
        sub r0, r0, r1 // a = a - b
        sub r2, r2, r3 // c = c - d
        add r0, r0, r2 // a = a + c
        mov pc, lr
```

(a - 2*b) + (c - d) subroutine

- A C program can test the subroutine r_expr
- expr_main.c
- The declaration tells the C compiler the number and types of arguments for the r_expr function
- r_expr
    - r0=2
    - r1=1
    - r2=4
    - r3=1

```c
#include <stdio.h>

// extern int r_expr(int a, int b, int c, int d );

extern int r_expr(int, int, int, int);

int main() {
    int r;
    r = r_expr( 2, 1, 4, 1);
    printf("r_expr( 2, 1, 4, 1) = %d\n", r );
    r = r_expr( 14, 3, 22, 10);
    printf("r_expr( 14, 3, 22, 10) = %d\n", r );
    return 0;
}
```

(a - 2*b) + (c - d) subroutine

The C and assembler files are combined and tested with:

```
arm-linux-gnueabi-gcc --static expr_main.c r_expr.S

qemu-arm ./a.out
```

The assembly code for the C program can be generate and shown with:

```
arm-linux-gnueabi-gcc -march=armv4 -Os -S expr_main.c

cat expr_main.s
```

(a - 2*b) + (c - d) subroutine

- In expr_main.s we can see the r_expr function call
    - r0=1
    - r1=2
    - r2=3
    - r3=4
- `r_expr( 1, 2, 3, 4);`

```
main:
        @ ...
        @ ...
        push {r4, lr}
        mov  r3, #4
        mov  r2, #3
        mov  r1, #2
        mov  r0, #1
        bl   r_expr
```


MEMORIAL UNIVERSITY
www.mun.ca

maximum subroutine

- The maximum.s program (lec 22) rewritten as a subroutine
    - compare r0 and r1, r0 and r2
    - conditional mov
- r0 will have the largest number
- mov pc, lr return to caller

```
maximum:
    // find maximum of r0, r1, r2, r3
    cmp r0, r1
    movlt r0, r1 // if (r0 < r1) r0 = r1
    cmp r0, r2
    movlt r0, r2 // if (r0 < r2) r0 = r2
    cmp r0, r3
    movlt r0, r3 // if (r0 < r3) r0 = r3
    mov pc, lr // return value in r0
    .size maximum, .-maximum
```


MEMORIAL UNIVERSITY
www.mun.ca

maximum subroutine

- The variables, a, b, c, and d are passed into the subroutine with r0, r1, r2, and r3
- The result is returned in r0
- maximum_main.c

```
arm-linux-gnueabi-gcc --static maximum_main.c
maximun.S

qemu-arm ./a.out
```

```
maximum:
    // find maximum of r0, r1, r2, r3
    cmp r0, r1
    movlt r0, r1 // if (r0 < r1) r0 = r1
    cmp r0, r2
    movlt r0, r2 // if (r0 < r2) r0 = r2
    cmp r0, r3
    movlt r0, r3 // if (r0 < r3) r0 = r3
    mov pc, lr // return value in r0
    .size maximum, .-maximum
```


MEMORIAL UNIVERSITY
www.mun.ca

ifelse subroutine

- The ifelse.s program (Lec 22) can be rewritten as a subroutine
  - comparison r1 - r0
  - conditional branch
- The variables, a and b are passed into the subroutine with r0 and r1
- The result is returned in r0
- ifelse_main.c
  - if r0 < r1
    - r0 = 24
  - if r1 > r0
    - r0 = 12

```
ifelse:
        cmp r0, r1 // if test
        ble skip_if
        mov r1, #5 // if body
        mov r2, #7
        b skip_else
skip_if:
        mov r1, #10 // else body
        mov r2, #14
skip_else:
        add r0, r1, r2
        mov pc, lr // return value in r0
        .size ifelse, .-ifelse
```

count_bits subroutine

- The count_bits.s program can be rewritten as a subroutine
  - test and branch to stop
  - loop back with branch
- The variables, bits is passed into the subroutine with r0
- The result is returned in r0
- count_bits_main.c
- Notice that the number of 1's in -1 is 32.

```
count_bits:
     mov r1, r0
     mov r0, #0
top:
     teq r1, #0      // while test
     beq done        // done when r1 == 0
     and r2, r1, #1 // extract bit0
     add r0, r0, r2 // count bit
     lsr r1, r1, #1 // logic shift right
     b    top        // branch to top
done:
     mov pc, lr      // return value in r0
     .size count_bits, .-count_bits
```

Recursive Functions

Table test with “recur_sum.c”
What does the “recur_sum.c” program print?

```
gcc -Wall -O --static recur_sum.c
./a.out
```

How many stack levels are created?

```c
#include <stdio.h>

static int rsum( int n ) {
   printf("start level %d\n", n );
   if ( n == 0 ) return 0;
   int v =  n + rsum( n-1 );
   printf("end level %d\n", n );
   return v;
}

int main(int ac, char *av[] ) {
   printf("sum is %d\n", rsum(4) );
   return 0;
}
```

Recursive Functions

What does the “recur_sum.c” program print?

```
gcc -Wall -O --static recur_sum.c
./a.out
```

How many stack levels are created?

rsum(0) // return 0

rsum(1) // return 1

rsum(2) // return 3

rsum(3) // return 6

rsum(4) // return 10

main // print 10

C program with two objects files

- The function rsum and main functions can be separated into two C source files
- The “rsum0_n.c” contains:

```
int rsum( int n ) {

    if ( n == 0 ) return 0;

    return n + rsum( n-1 );

}
```

C program with two objects files

- These files can be separately compiled, linked together, and the executable run with

```
gcc -c -O rsum0_n.c
```

- creates rsum0_n.o

```
gcc -c -O main_rsum0_n.c
```

- creates main_rsum0_n.o
- Compile them together. Note the end ".o"

```
gcc --static rsum0_n.o main_rsum0_n.o
./a.out
```

Translation of rsum0_n.c

- The compiler can generate the assembly version of the rsum function with the -S options
- The -O options requests that the code be optimized.

```
arm-linux-gnueabi-gcc -march=armv4 -O -S rsum0_n.c
```

- Creates a rsum0_n.S

```
cat rsum0_n.s
```

- Notice that the push {r4, lr} instruction is used to save the link register and r4
- The pop {r4, lr} instruction restores the saved values.
- The function is a non-leaf that calls itself, so it must save preserved and non-preserved registers

Translation of rsum0_n.c

`cat rsum0_n.s`

- Notice that the push {r4, lr} instruction is used to save the link register and r4
- The pop {r4, lr} instruction restores the saved values.
- The function is a non-leaf that calls itself, so it must save preserved and non-preserved registers
- n == 0?
  - if not 0 -> branch to L4
  - if 0 -> continue to L2
- recursive call

```
rsum:
        push {r4, lr}
        subs r4, r0, #0
        bne  .L4
.L2:
        mov  r0, r4 // Return v
        pop  {r4, pc}
.L4:
        sub  r0, r4, #1
        bl   rsum
        add  r4, r4, r0
        b    .L2
        .size    rsum, .-rsum
```


MEMORIAL UNIVERSITY
www.mun.ca

Next Steps

Memory Access

Unit 6

Questions?