ARMv6 Assembly Language Notes

This document is a quick reference for material that we talk about in class.

Integer registers

There are 16 main registers, r0–r15:

- r0: 1st function argument, scratch, function return value
- r1: 2nd function argument, scratch
- r2: 3rd function argument, scratch
- r3: 4th function argument, scratch
- r4: callee-saved
- r5: callee-saved
- r6: callee-saved
- r7: callee-saved, system call number
- r8: callee-saved
- r9: callee-saved
- r10: callee-saved
- r11: callee-saved
- r12 (ip): scratch
- r13 (sp): stack pointer, callee-saved
- r14 (lr): link register, scratch
- r15 (pc): program counter

You do not need to save the values in scratch registers, but you also cannot assume they have been saved when you call another function.

You must save callee-saved registers if you plan to use them, and restore the value before you return. This is most often done using the stack.

Basic instructions

Here are some of the most common assembly language instructions we will use. Code is normally part of the text segment.

Basic integer operations

- `add r0, r1, r2`: r0 = r1 + r2
- `add r0, r1, #3`: r0 = r1 + 3, only a limited range of constants
- `add r0, r1, r2, lsl #3`: r0 = r1 + r2×8 (r2 shifted left 3 times)
- `add r0, r1, r2, lsr #2`: r0 = r1 + r2÷4 (r2 shifted right 2 times)

The same variations for the second input argument are available for:

- `sub r0, r1, r2`: r0 = r1 - r2
- `rsb r0, r1, r2`: r0 = r2 - r1
- `rsb r0, r0, #0`: r0 = -r0
- `and r0, r1, r2`: r0 = r1 AND r2 (logical AND)
- `orr r0, r1, r2`: r0 = r1 OR r2 (logical OR)
- `eor r0, r1, r2`: r0 = r1 XOR r2 (exclusive OR)
- `bic r0, r1, r2`: r0 = r1 AND NOT r2 (bit clear)
- `mov r0, r2`: r0 = r2
- `mvn r0, r2`: r0 = NOT r2 (flip all the bits)
- `cmp r1, r2`: compare r1 with r2 (subtract and set flags, but do not store result)
- `tst r1, r2`: test r1 against r2 (AND and set flags, but do not sture result)

Any of these (except `cmp` and `tst`) can be suffixed with `s` to set the condition flags, e.g., `adds r0, r1, r2`.

The basic form for multiply is:

- `mul r0, r1, r2`: r0 = r1 × r2

(note: `mul r0, r0, r1` is NOT valid, destination must not be same as first argument)

Branches

All instructions can be run conditionally. However, the most common use is conditional branches, so the most common conditions are presented here. These are described as though you just ran `cmp r0, r1`:

- `eq`: if r0 = r1
- `ne`: if r0 ≠ r1
- `lt`: if r0 < r1
- `le`: if r0 ≤ r1
- `gt`: if r0 > r1
- `ge`: if r0 ≥ r1

For example, `addlt r5, r6, r7` gives r5 = r6 + r7, but only if the r0 < r1 (from the `cmp` instruction given above)

To branch:

- `b<condition>`, e.g., `b`, `beq`, `bne`, etc. This form just branches to a new location, which is normally a program label.
- `bl<condition>`, e.g., `bleq` or just `bl` This form is branch-and-link, which puts a return address in `lr`.
- `bx<condition>`, e.g., `bx lr`. This form branches to an address in a register (instead of a program label).
- `blx<condition>`. Branch to an address in a register and store the return address in lr.

Making function calls

To make a function call, put the parameters in the appropriate registers (`r0`, `r1`, etc.) then issue a branch-with-link instruction: `bl funcname`. This jumps to the given label, but also puts the return address into the `lr` register.

When the function call returns, you should assume that all registers labeled as “scratch” have lost their former values. Values stored in the other registers should have the same value as before the function call.

If you have a value that you care about in a scratch register and you need to make a function call, you have two choices:

1. Move the value into a non-scratch register. You may want to plan for this earlier. For example, instead of storing an important value in `r2`, use a non-scratch register like `r8` instead. You may need to rewrite earlier code to make this work.
2. Push the value onto the stack right before making the call, then pop it back off after the call finishes.

The disadvantage to option 2 is that you have to do it before every function call. I recommend using option 1 until you run out of registers. Only start spilling values onto the stack when there are no registers available.

Functions and stack operations

At the beginning of a function, work out which registers you will need, and save the old values by pushing them onto the stack. Then pop them off at the end of the function to restore them:

- `push {r7,r8,r9,lr}`: push the given registers onto the stack. Should be an even number of registers.
- `pop {r7,r7,r9,pc}`: pop the given registers off the stack. Normally matches an earlier `push`.

When you pop an address into `pc`, it has the side effect of forcing a branch. This is a common way to return from a function.

Returning from a function call

There are two common ways to return from a function:

1. At the beginning of the function, do nothing, and then at the end issue:

   `bx lr`

   When the function begins, the return address is in the `lr` register. If you can leave that register alone, it will be there when the function finishes so you can branch to it to return.

Note that this approach is only an option if `lr` can remain undisturbed through your entire function. Note especially that calling another function overwrites the value of `lr`, so any function that calls another function cannot use this approach.

2. At the beginning of the function, push the value of `lr` onto the stack. Within the function, you can use it as another scratch register, bearing in mind that it will be lost as soon as you make another function call (this is true of all scratch registers).

   At the end of the function, pop the value off the stack and directly into the `pc` register, which has the side effect of returning from the current function. This is typically combined with pushing and popping other registers. See the previous section for an example.

Passing parameters on the stack

When a function has more than four integer arguments, additional parameters must be passed in on the stack. For example, calling a function `sum` that takes five arguments might look like:

```
mov     r0, #50
push    {r0,r1}
mov     r0, #10
mov     r1, #20
mov     r2, #30
mov     r3, #40
bl      sum
add     sp, #8
```

In this example, we put the fifth parameter (#50) onto the stack first, then load the other four values into r0–r3. Because the stack must always have an even number of elements, we push an addition junk value after r0 (r1 in this case).

After the function call completes, we throw away the values on the stack by adding 8 to the stack pointer. We could pop them off instead, but then we would have to pop them into some registers. Since we do not actually care about those values any more, we just discard them instead.

The following would be another way of accomplishing the same thing:

```
sub     sp, #8
mov     r0, #50
str     r0, [sp]
mov     r0, #10
mov     r1, #20
mov     r2, #30
mov     r3, #40
bl      sum
add     sp, #8
```

If we had seven integer parameters so three of them needed to go on the stack, we could use:

```
sub     sp, #16         @ always keep sp a multiple of 8
mov     r0, #50
str     r0, [sp]
mov     r0, #60
str     r0, [sp,#4]
mov     r0, #70
str     r0, [sp,#8]
```

and then load the first four parameters in r0–r3 as usual.

The function that is being called can access those values by loading them directly from the stack. Note that it must also account for any changes it makes to the stack. For example:

```
sum:
        push    {ip,lr}         @ pushing 8 bytes onto the stack
        add     r0, r0, r1      @ get the sum of r0 through r3
        add     r2, r2, r3
        add     r0, r0, r2
        ldr     r1, [sp, #8]    @ parameter number 5
        ldr     r2, [sp, #12]   @ parameter number 6
```

```
        ldr     r3, [sp, #16]   @ parameter number 7
        add     r1, r1, r2
        add     r1, r1, r3
        add     r0, r0, r1
        pop     {ip,pc}
```

Since `sum` starts by pushing 8 bytes onto the stack (two registers), it has effectively subtracted 8 from the stack pointer it was given. Since its fifth parameter was the first item on the stack when it was called, that fifth parameter is now 8 bytes past the beginning of the stack. Likewise, the sixth parameter is 12 bytes past the beginning, and the seventh parameter is 16 bytes past the beginning. After the push instruction, the stack looks like:

```
| ...           |
+---------------+
| parameter 7   |
+---------------+
| parameter 6   |
+---------------+
| parameter 5   |
+---------------+
| saved lr reg  |
+---------------+
| saved ip reg  |
+---------------+ <--- sp points here
```

where each box is 4 bytes in size.

System calls

A system call is similar to a function call, except that the call is being made to the operating system instead of to more code within your program. The basic process is the same as for a function call, with parameters going in the same registers (at least for the parameters we will use). In addition, the system call number must be loaded into the r7 register, and then instead of issuing a “bl” instruction, you should issue a “svc #0” instruction.

For example, to write a message to stdout (which always has the file descriptor 1):

```
mov r0, #1                      @ stdout
ldr r1, =buffer                 @ where to find the data to write
mov r2, #20                     @ number of bytes to write
mov r7, #sys_write              @ sys_write is 1, defined elsewhere
svc #0
```

The result of the call is in r0 after the call finishes. If this value is negative, it normally indicates an error.

While testing your code, it may be helpful to use an error status code as the value in a call to exit. If you do this, then you can look up the error code in this chart (the syscall result is the error code negated):

```
#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO          5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
```

```
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
#define EMFILE      24  /* Too many open files */
#define ENOTTY      25  /* Not a typewriter */
#define ETXTBSY     26  /* Text file busy */
#define EFBIG       27  /* File too large */
#define ENOSPC      28  /* No space left on device */
#define ESPIPE      29  /* Illegal seek */
#define EROFS       30  /* Read-only file system */
#define EMLINK      31  /* Too many links */
#define EPIPE       32  /* Broken pipe */
#define EDOM        33  /* Math argument out of domain of func */
#define ERANGE      34  /* Math result not representable */
```

The syscalls you will need in this course are as follows:

- 1: `sys_exit(status)` does not return.
- 2: `sys_fork()` returns 0 for the child process, the pid of the next child for the parent process, or negative to signal an error.
- 3: `sys_read(fd, buffer, count)` returns the number of bytes read, or negative to signal an error.
- 4: `sys_write(fd, buffer, count)` returns the number of bytes written, or negative to signal an error.
- 5: `sys_open(filename, flags, mode)` returns the file descriptor, or negative to signal an error.
- 6: `sys_close(fd)` returns negative to signal an error.
- 42: `sys_pipe(&fd_pair)` returns negative to signal an error. It creates a pipe and writes the file descriptor (fd) of the read end of the pipe to the address you pass in, and the fd of the write end of the pipe four bytes after that.

Floats

To use double-precision floats, you must work with the registers d0–d15 instead of the more familiar integer registers r0–r15. Here are the most common instructions you will need:

- `fcpyd d0, d1`: copy a double from d1 to d0
- `faddd d0, d1, d2`: add d1 and d2, store result in d0
- `fsubd d0, d1, d2`: subtract d2 from d1, store result in d0
- `fmuld d0, d1, d2`: multiply d1 and d2, store result in d0
- `fdivd d0, d1, d2`: divide d1 by d2, store result in d0

Comparing two doubles is also straightforward:

- `fcmpd d0, d1`: compare d0 with d1, set flags based on result

However, the results are stored in a different flag register than normal. To set the normal flag register and allow conditional branching based on the result, follow `fcmpd` with `fmstat`. For example:

```
fcmpd d5, d3        @ compare d5 with d3
fmstat              @ copy flags to integer status register
bge 1f              @ branch if d5 ≥ d3
```

To load values into a double register, there are a few options depending on where the value is coming from. To load a constant:

```
fldd d0, half       @ load the value stored at half into d0
…                   @ after the function, store the constant
half: .double 0.5   @ constant goes here
```

Note: this only works when the labeled value (half) is nearby, normally in the text segment right before or after the function. To load from an arbitrary address, first load the address into a regular register and then issue the `fldd` instruction:

```
ldr r8, =xcenter
fldd d3, [r8]
```

Similarly, to save a float value in memory, first load the address into a register and then issue the store instruction:

```
ldr r7, =somelabel
```

```
fstd d2, [r7]
```

If you need to load or store a value from a computed address, e.g., an element of an array or a location on the stack, do the math first:

```
add  ip, sp, #24
fldd d4, [ip]
```

The approach is the same for saving floats.

To convert a value from an integer register into a float requires two steps:

```
vmov s5, r0         @ copy the integer value from r0 into s5
fsitod d0, s5       @ convert the int in s5 into a double in d0
```

Note that the `s` registers and the `d` registers overlap each other, with two `s` registers for each `d` register. So the above example will overwrite any value in d2 (because d2 overlaps s4 and s5).

Similarly, to convert a value from a float register to an integer requires two steps:

```
ftosid s0, d0       @ round and convert the float in d0 to s0
vmov r3, s0         @ copy the integer value from s0 to r3
```

The same caveat about overlapping `s` and `d` registers applies.

To view one of these registers from within gdb, use:

```
(gdb) p $d0.f64
```

This will print (p) register d0, interpreted as a float with 64 bits.