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
- `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
- `cmp r0, r1, r2`: compare r1 with r2 (subtract and set flags, but do not store result)
- `tst r0, r1, r2`: test r1 against r2 (AND and set flags, but do not store 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)
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 $r0 \neq r1$
- **lt**: if $r0 < r1$
- **le**: if $r0 \leq r1$
- **gt**: if $r0 > r1$
- **ge**: if $r0 \geq r1$

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`.

**Functions and stack operations**

- `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.