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`: $r0 = r1 + r2 \times 8$ (r2 shifted left 3 times)
- `add r0, r1, r2, lsr #2`: $r0 = r1 + r2 \div 4$ (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`: $r0 = r1 \text{ AND } r2$ (logical AND)
- `orr r0, r1, r2`: $r0 = r1 \text{ OR } r2$ (logical OR)
- `eor r0, r1, r2`: $r0 = r1 \text{ XOR } r2$ (exclusive OR)
- `bic r0, r1, r2`: $r0 = r1 \text{ AND NOT } r2$ (bit clear)
- `mov r0, r2`: $r0 = r2$
- `mvn r0, r2`: $r0 = \text{ NOT } r2$ (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 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`: $r0 = r1 \times r2$

(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 $r0 \neq r1$
- `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`.

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.

**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
- `fadd d0, d1, d2`: add d1 and d2, store result in d0
- `fsub d0, d1, d2`: subtract d2 from d1, store result in d0
- `fmul d0, d1, d2`: multiply d1 and d2, store result in d0
- `fdiv 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:

```markdown
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:

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

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

```markdown
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).

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

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

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