ARM64 Assembly Language Notes

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

Integer registers

There are 32 main registers, x0–x30 (64-bit versions) or w0–w30 (32-bit versions). x31 is a special case register used as the stack pointer.

- x0–x7: function arguments, scratch (x0 is also function return value)
- x8–x18: scratch (x8 is syscall number, x16–x18 sometimes reserved)
- x19–x28: callee-saved registers (save to stack at beginning of function, restore from stack before returning)
- x29: frame pointer
- x30: link register (save to stack for non-leaf functions)
- sp: stack pointer

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 x0, x1, x2`: x0 = x1 + x2
- `add x0, x1, #3`: x0 = x1 + 3, only a limited range of constants
- `add x0, x1, x2, lsl #3`: x0 = x1 + x2×8 (x2 shifted left 3 times)
- `add x0, x1, x2, lsr #2`: x0 = x1 + x2÷4 (x2 shifted right 2 times)

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

- `sub x0, x1, x2`: x0 = x1 - x2
- `neg x0, x1`: x0 = -x1
- `and x0, x1, x2`: x0 = x1 AND x2 (bitwise AND)
- `orr x0, x1, x2`: x0 = x1 OR x2 (bitwise OR)
- `eor x0, x1, x2`: x0 = x1 XOR x2 (bitwise exclusive OR)
- `bic x0, x1, x2`: x0 = x1 AND NOT x2 (bit clear)
- `mov x0, x2`: x0 = x2
- `mvn x0, x2`: x0 = NOT x2 (flip all the bits)

Shifting instructions include:

- `lsl x0, x1, x2`: x0 = x1 shifted left x2 times
- `lsl x0, x1, #2`: x0 = x1 shifted left twice
- `lsr x0, x1, x2`: same but shifted right

The main compare instructions are:

- `cmp x1, x2`: compare x1 with x2 (subtract and set flags, but do not store result)
- `tst x1, x2`: test x1 against x2 (AND and set flags, but do not sture result)

The basic forms of multiply and divide are:

- `mul x0, x1, x2`: x0 = x1 × x2
- `sdiv x0, x1, x2`: x0 = x1 ÷ x2 (for signed division)
- `udiv x0, x1, x2`: x0 = x1 ÷ x2 (for unsigned division)

Memory access

To load a word (64 bits) from memory there are a few variations:

- `ldr x0, [x1]`: load the value at address x1 into x0
- `ldr x0, [x1, x2]`: load the value at address x1 + x2 into x0
- `ldr x0, [x1, #8]`: load the value at address x1 + 8 into x0
- `ldr x0, [x1, x2, lsl #3]`: load the value at address x1 + x2×8

Everything inside the square brackets is added together to compute an address, and the value at that address is loaded in the register (x0 in these examples). x1 (or whatever register goes in that position) is normally the beginning of an array or struct, and x2 (in the examples) is used to index into that array or struct. The x2 part of it can optionally be shifted left (multiplied by a power of two), convenient when x2 in an index but the size of the individual elements is 2, 4, or 8 bytes (an array of integers, for example).

There are also variations for loading bytes:

- `ldrb w0, [x1]`: load the byte at address x1 into x0
- ...

Note the use of w0 instead of x0. w0 refers to the 32-bit version of the register, and is required for byte accesses.

There are similar instructions for storing to memory:

- `str x0, [x1]`: store the value in x0 to the address in x1
- ...

and for storing bytes:

- `strb w0, [x1]`: store the byte in x0 to the address in x1
- ...

If you need to load an address into a register, you cannot normally use `mov x0, #address`, because there is not room in the instruction to hold such a large constant. Instead, there is a special instruction for loading addresses:

- `adr x0, =label`: load the address of label into x0

Finally, there is a special form of `ldr` for loading an arbitrary constant into a register:

- `ldr x0, =12345678`: load the constant 12345678 into x0

This is a pseudo-instruction provided by the assembler. If possible, it is converted into a `mov` instruction to load a constant. If the constant will not fit in a `mov` instruction, the assembler stores it in memory nearby (usually after the current `.text` segment block ends) and generates an instruction to load it from that location. When this happens it is actually an `ldr` instruction, but the form of the instruction is not quite what was written.

Branches

The most common conditions are presented here, and are used for conditional branches. These are described as though you just ran `cmp x0, x1`:

- `eq`: if x0 = x1
- `ne`: if x0 ≠ x1
- `lt`: if x0 < x1
- `le`: if x0 ≤ x1
- `gt`: if x0 > x1
- `ge`: if x0 ≥ x1

For example, `b.lt rejoin` branches to the label `rejoin`, but only if `x0 < x1` (from the `cmp` instruction given above)

To branch:

- `b.<condition>`, e.g., `b`, `b.eq`, `b.ne`, etc. This form just branches to a new location, which is normally a program label.
- `bl`: this form is branch-and-link, which puts a return address in `x30` (the link register). This is normally how you would call a function.
- `blr x0`: this is a variation of `bl` that jumps to the address in a register (`x0` in this example). This is how you would call a function when the function's address is given to you as a pointer instead of being a regular label.
- `ret x0`: branch to the address in x0. `ret` with no argument is shorthand for `ret x30`. This is normally

how you would return from a function, using the value that a previous `bl` provided.

There are a few specialized branch instructions:

- `cbz x0, label`: branch to label if x0 contains a zero
- `cbnz x0, label`: branch to label if x0 is NOT a zero

Making function calls

To make a function call, put the parameters in the appropriate registers (`x0`, `x1`, etc.) then issue a branch-with-link instruction: `bl funcname`. This jumps to the given label, but also puts the return address into the link register.

When the function call returns, you should assume that all registers labeled as “scratch” have lost their former values. Values stored in the callee-saved 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 `x2`, use a non-scratch register like `x20` 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, and I recommend avoiding this. 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

For leaf function (one that does *not* call any other functions), you can typically use only scratch registers and ignore the stack. For a non-leaf function (or one where the scratch registers are insufficient), start each function with these two instructions:

```
stp    x29, x30, [sp, #-16]!
mov    x29, sp
```

The first one stores a pair of registers, `x29` (the frame pointer), and `x30` (the link register) onto the stack. The memory location where the two values are stored is computed as `sp - 16`, but it also has the side-effect of updating `sp` to the computed value (the `!` symbol at the end causes this side-effect). The effect is to change the stack from this:

```
     |                      |
sp-->| caller's stack frame |
     +----------------------+
```

where the stack pointer identifies the last used location on the stack (which belongs to the caller), to:

```
     |                      |
     | caller's stack frame |
     |                      |
     +----------------------+
     | saved return address |  +8
     +----------------------+
fp-->| saved frame pointer  |   0
     +----------------------+
```

i.e., the return address and old frame pointer have been saved on the stack (the `stp` instruction does this), the stack pointer moved down to make room for the two values (also done by the `stp` instruction), and the new frame pointer is set as indicated (the `mov` instruction does this). Note at this point the stack pointer and the frame pointer are referring to the same memory address, but the stack pointer is omitted from the diagram.

The right side of the diagram also shows the location of each value on the stack, relative to the frame pointer. For example, the saved return address is stored eight bytes past the location the frame pointer refers to. This is one of the main functions of the frame pointer—it provides an anchor at the top of the stack frame and all other addresses on the stack frame are computed relative to it.

Next, 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. You reserve stack space by simply moving the stack pointer:

```
sub     sp, sp, #32         // reserve 32 bytes (four 64-bit words)
```

This would result in the following stack frame:

```
     |                      |
     | caller's stack frame |
     |                      |
     +----------------------+
     | saved return address |  +8
     +----------------------+
fp-->| saved frame pointer  |   0
     +----------------------+
     |                      |  -8
     +----------------------+
     |                      |  -16
     +----------------------+
     |                      |  -24
     +----------------------+
sp-->|                      |  -32
     +----------------------+
```

The next step is to save any callee-saved registers that you intend to use:

```
str    x19, [x29, #-32]
str    x20, [x29, #-24]
str    x21, [x29, #-16]
str    x22, [x29, #-8]
```

resulting in:

```
     |                      |
     | caller's stack frame |
     |                      |
     +----------------------+
     | saved return address |  +8
     +----------------------+
fp-->| saved frame pointer  |   0
     +----------------------+
     | saved x22            |  -8
     +----------------------+
     | saved x21            |  -16
     +----------------------+
     | saved x20            |  -24
     +----------------------+
sp-->| saved x19            |  -32
     +----------------------+
```

Adjust accordingly if you need more or fewer slots on your stack frame. I recommend creating a diagram like this and putting it in the comments at the top of your function so it is easy to keep track of your stack usage. This stack setup and storage of callee-saved registers is often called the *function prelude*.

At the end of the function, you need a corresponding *function postlude*. After loading the return value into `x0`, clean up and return using code like this:

```
// restore the callee-saved registers that we used
ldr    x19, [x29, #-32]
ldr    x20, [x29, #-24]
ldr    x21, [x29, #-16]
ldr    x22, [x29, #-8]

// free the stack space that we reserved
// note: this should exactly match the subtraction at the top
add    sp, sp, #32

// load the old frame pointer and return address,
// and also move the stack pointer back up 16 bytes
ldp     x29, x30, [sp], 16
```

```
// return: the return value should have been loaded into x0 earlier
ret
```

Note that the stack pointer must always be a multiple of 16, so if you only need an odd number of 8-byte slots you should round it up and reserve a multiple of 16 bytes.

Returning from a function call

There are two common ways to return from a function:

1. For a leaf function (one that does *not* call any other functions): at the beginning of the function, do nothing, and then at the end issue:

   `ret`

   When the function begins, the return address is in the `x30` 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 `x30` can remain undisturbed through your entire function. Note especially that calling another function overwrites the value of `x30`, so any function that calls another function cannot use this approach.

2. At the beginning of the function, push the value of `x30` onto the stack (the stack code listed above does this). 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 back into a register, usually back into `x30`. Isssue a `ret` instruction.

Passing parameters on the stack

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

```
        sub     sp, sp, #16
        mov     x0, #90
        str     x0, [sp, #0]
        mov     x0, #10
        mov     x1, #20
        mov     x2, #30
        mov     x3, #40
        mov     x4, #50
        mov     x5, #60
        mov     x6, #70
        mov     x7, #80
        bl      sum
        add     sp, sp, #16
```

In this example, we put the ninth parameter (#90) onto the stack first, then load the other eight values into x0–x7. Because the stack must always have an even number of elements, we reserve 16 bytes even though we only need 8 of them.

After the function call completes, we throw away the values on the stack by adding 16 to the stack pointer.

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

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

```
sub     sp, #32             // always keep sp a multiple of 16
mov     x0, #90
str     x0, [sp, #0]
mov     x0, #100
str     x0, [sp, #8]
mov     x0, #110
str     x0, [sp, #16]
```

and then load the first eight parameters in x0–x7 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: sub sp, sp, #16 // make sure sp remains a multiple of 16 str x30, [sp, #0] add x0, x0, x1 // get the sum of x0 through x7 add x2, x2, x3 add x4, x4, x5 add x6, x6, x7 add x0, x0, x2 add x4, x4, x6 add x0, x0, x4 ldr x1, [sp, #16] // parameter number 9 ldr x2, [sp, #24] // parameter number 10 ldr x3, [sp, #32] // parameter number 11 add x1, x1, x2 add x1, x1, x3 add x0, x0, x1 // ... ldr x30, [sp, #0] add sp, sp, #16 ret

Since `sum` starts by reserving 16 bytes on the stack (two registers worth of space), it must compensate when calculating where its parameters are relative to the stack pointer. Since its ninth parameter was the first item on the stack when it was called, that ninth parameter is now 16 bytes past the beginning of the stack. Likewise, the tenth parameter is 24 bytes past the beginning, and the eleventh parameter is 32 bytes past the beginning. After reserving those initial 16 bytes, the stack looks like:

```
     | ...                  |
     +----------------------+
     | parameter 11         |
     +----------------------+
     | parameter 10         |
     +----------------------+
     | parameter 9          |
     +----------------------+
     | padding              |
     +----------------------+
sp-->| saved x30 reg        |
     +----------------------+
```

where each box is 8 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 x8 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     x0, #1                  // stdout
adr     x1, buffer              // where to find the data to write
mov     x2, #20                 // number of bytes to write
mov     x8, #sys_write          // sys_write is 64, defined elsewhere
svc     #0
```

The result of the call is in x0 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). This chart comes from `/usr/include/asm-generic/errno-base.h`:

Name	Number	Error message
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Argument list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No child processes
EAGAIN	11	Try again
ENOMEM	12	Out of memory
EACCES	13	Permission denied
EFAULT	14	Bad address

EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Device or resource busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of func
ERANGE	34	Math result not representable

The syscalls you will need in this course are as follows (taken from `/usr/include/asm-generic/unistd.h`):

Number	Example call	Return value
93	`sys_exit(status)`	does not return
63	`sys_read(fd, pointer, size)`	returns the number of bytes read (negative for error)
64	`sys_write(fd, pointer, size)`	returns the number of bytes written (negative for error)
1024	`sys_open(name, flags, mode)`	returns the new file descriptor (negative for error)
57	`sys_close(fd)`	no return value (negative for error)