Introduction

In this report, I will present the optimization of five different string library functions written in ARM assembly. The goal of optimization is to reduce the total number of cycles of runtime, including removing stalls and making use of more efficient approaches to solving the problem. I did not consider code length or registers used in my optimizations, except when those changes can further reduce the number of cycles needed to run.

strlen

Original

```
.global my_strlen

my_strlen:
       src .req r0
       cnt .req r1
       t   .req r2
       ret .req r0

       mov cnt, #0
      ; Unoptimized loop needs 8 cycles
       loop:
              add cnt, cnt, #1
              ldrb t, [src], #1
              cmp t, #0
              bne loop

       sub ret, cnt, #1
       bx lr
```

Optimized

```
.global my_strlen

my_strlen:
       src .req r0
       cnt .req r1
       t   .req r2
       ret .req r0

       mov cnt, #0

       ; t is preloaded
       ldrb t, [src], #1

       ; Optimized loop needs 6 cycles
       loop:
              add cnt, cnt, #1
              cmp t, #0

              ;t is preloaded
              ;This load is also made
              ;conditional.
              ldrbne t, [src], #1
              bne loop

       sub ret, cnt, #1
       bx lr
```

The existing code has one main inefficiency that needs to be fixed. There is a two cycle stall after the `ldrb` instruction because of the dependency on `t`.

To fix the stall, pre-loading of t is used. This means that there will be a 2 cycle stall for the first iteration of the loop, but future iterations will not have a stall. For most loop runs, this will result in a 2 cycle savings. The original loop requires 8 cycles in the normal case, and the optimized code requires only 6 cycles.

I decided to apply this change because it is the most straightforward approach to optimizing this code. I also investigated the possibility simply moving the add cnt, cnt, #1 instruction to be between the load of t and its usage, but this would only remove 1 stall cycle. Partial loop unrolling could also be used here, potentially for more performance gain. If I wanted to maximize my points, I should probably have done that.

strcpy

Original

```
.global my_strcpy

my_strcpy:
	dst .req r0
	src .req r1
	t   .req r2

	; Unoptimized loop needs 8 cycles
	strcpyloop:
		ldrb t, [src], #1
		strb t, [dst], #1
		cmp t, #0
		bne strcpyloop

	bx lr
```

Optimized

```
.global my_strcpy

my_strcpy:
	dst .req r0
	src .req r1
	t   .req r2

	; t is preloaded
	ldrb t, [src], #1

	; Optimized loop needs 6 cycles
	strcpyloop:
		strb t, [dst], #1
		cmp t, #0

		;t is preloaded
		;This load is also made
		;conditional.
		ldrbne t, [src], #1
		bne strcpyloop

	bx lr
```

The existing code has one main inefficiency that needs to be fixed. There is a stall after the `ldrb` instruction because of the dependency on `t`. How many cycles this stall is, however, is unclear. It is likely two cycles, but because the store instruction doesn't need t until the fourth stage in the pipeline, it might only be one cycle. To determine this I would need a cycle accurate simulator, which I don't have access to.

To fix the stall, pre-loading of t is used. This means that there will be a stall for the first iteration of the loop, but future iterations will not have a stall. Assuming the stall is 2 cycles, this will result in a 2 cycle savings. The original loop requires 8 cycles in the normal case, and the optimized code requires only 6 cycles.

I decided to apply this change because it is the most straightforward approach to optimizing this code. Partial loop unrolling could also be used here, potentially for more performance gain. If I wanted to maximize my points, I should probably have done that.

strtolower

Original

```
.global my_strtolower

my_strtolower:
	src .req r0
	t   .req r1

	;Number of cycles in the
	;loop is between 12 and 14
	;13 cycles if neither branch (blt
	;  and bgt) are executed,
	; 14 cycles if first branch (blt) is
	;  not executed, but second branch
	;  (bgt) is executed,
	; 12 cycles if first branch (blt) is
	;  executed
	loop:

		ldrb t, [src]
		cmp t, #'A'
		blt no_match
		cmp t, #'Z'
		bgt no_match
		add t, t, #'a'-'A'
	no_match:
		strb t, [src], #1
		cmp t, #0
		bne loop

	bx lr
```

Optimized

```
.global my_strtolower

my_strtolower:
	src .req r0
	t   .req r1
	tmp .req r2  ; Scratch variable

	;t is preloaded
	ldrb t, [src]

	; Optimized loop needs 9 cycles
	loop:
		;Branch instructions are
		;removed

		 ; Compare to test if the
		 ;letter is uppercase
		 sub tmp, t, #'A'
		 cmp temp, #'Z'-'A'

		 ;Conditionally make lower
		 addls t, t, #'a'-'A'

		 strb t, [src], #1
		 cmp t, #0

		 ;t is preloaded
		 ;This load is also made
		 ;conditional
		 ldrbne t, [src]
		 bne loop

	bx lr
```

The existing code has a number of inefficiencies that need to be fixed. First, there is a two cycle stall after the `ldrb` instruction because of the dependency on `t`. Second, there are multiple unnecessary branches related to determining if the character is uppercase or not.

To fix the stalls, pre-loading of t is used. This means that there will be a 2 cycle stall for the first iteration of the loop, but future iterations will not have a stall. For most loop runs, this will result in a 2 cycle savings. To fix the unnecessary branches, the branch structure is removed and conditional execution is added. The number of cycles saved depends on the character being tested. See the comments in the code for more detail. In the best case, 5 cycles are saved per loop. In the worst case, 3 cycles are saved.

I decided to apply these changes because they are most straightforward approach to optimizing this code. I also investigated the possibility of applying partial loop unrolling, but I think it makes this code needlessly complex.

strchr

Original

```
.global my_strchr

my_strchr:
	src .req r0
	c   .req r1
	i   .req r2
	t   .req r3
	ret .req r0

	mov i, #0

	; Unoptimized loop needs 11 cycles
	loop:
		ldrb t, [src, +i]
		cmp t, #0
		beq loop_end
		cmp t, c
		beq found
		add i, i, #1
		b loop
	found:
		mov ret, i
		b end
	loop_end:
		mov ret, #-1
		b end
	end:
		bx lr
```

Optimized

```
.global my_strchr

my_strchr:
	src .req r0
	c   .req r1
	i   .req r2
	t   .req r3
	ret .req r0

	mov i, #0

	; t is preloaded
	ldrb t, [src, +i]

	; Optimized loop needs 9 cycles
	loop:
		cmp t, #0
		beq loop_end
		cmp t, c
		beq found
		add i, i, #1

		; t is preloaded
		ldrb t, [src, +i]
		b loop
	;Removed two useless branches below
	found:
		mov ret, i
		bx lr
	loop_end:
		mov ret, #-1
		bx lr
```

The existing code has a number of inefficiencies that need to be fixed. First, there is a two cycle stall after the `ldrb` instruction because of the dependency on `t`. Second, there are unnecessary `b end` instructions involved in the end of the routine that can be removed.

To fix the stalls, pre-loading of t is used. This means that there will be a 2 cycle stall for the first iteration of the loop, but future iterations will not have a stall. For most loop runs, this will result in a 2 cycle savings. The original loop requires 11 cycles in the normal case, and the optimized code requires only 9 cycles. To fix the unnecessary `b end` instructions, the end label is removed and `bx lr` is repeated (called inlining) in both places where `b end` was previously. This will save 3 cycles as the subroutine finishes, and also removes one instruction from the code.

I decided to apply these changes because they are most straightforward approach to optimizing this code. I also investigated the possibility of applying partial loop unrolling (for example, checking 4 characters per loop iteration) but if you do that then you need to add extra code to the loop to properly calculate i, which removes the efficiency gain.

strcmp

Original

```
.global my_strcmp

my_strcmp:
	str1 .req r0
	str2 .req r1
	i    .req r2
	t1   .req r3
	t2   .req r4
	ret .req r0

	stmfd sp!,{r4}

	mov i, #0
	; Unoptimized loop needs 12 cycles
	loop:
		ldrb t1, [str1, +i]
		ldrb t2, [str2, +i]
		cmp t1, t2
		bne fail
		cmp t1, #0
		beq loop_end
		add i, i, #1
		b loop
	fail:
		mov ret, i
		b end
	loop_end:
		mov ret, #0
		b end
	end:
		ldmfd sp!, {r4}
		bx lr
```

Optimized

```
.global my_strcmp

my_strcmp:
	str1 .req r0
	str2 .req r1
	i    .req r2
	t1   .req r3
	t2   .req r4
	ret .req r0

	stmfd sp!,{r4}

	mov i, #0

	; t1 and t2 are preloaded
	ldrb t1, [str1, +i]
	ldrb t2, [str2, +i]

	; Optimized loop needs 10 cycles
	loop:
		cmp t1, t2
		bne fail
		cmp t1, #0
		beq loop_end
		add i, i, #1

		; t1 and t2 are preloaded
		ldrb t1, [str1, +i]
		ldrb t2, [str2, +i]

		b loop
	;Removed two useless branches below
	fail:
		mov ret, i
		ldmfd sp!, {r4}
		bx lr
	loop_end:
		mov ret, #0
		ldmfd sp!, {r4}
		bx lr
```

The existing code has a number of inefficiencies that need to be fixed. First, there is a one cycle stall after the `ldrb t1` instruction and another two cycle stall from the `ldrb t2` instruction. (However, the one cycle stall becomes part of the two cycle stall because of parallelism in the pipeline, so the net stall is two cycles.) Second, there are multiple unnecessary branches at the end of the program.

To fix the stalls, pre-loading of both t1 and t2 is used. This means that there will be a 2 cycle stall for the first iteration of the loop, but future iterations will not have a stall. For most loop runs, this will result in a 2 cycle savings. To fix the unnecessary branches, the branch structure is removed and conditional execution is added. This will save 3 cycles as the subroutine finishes, as only one of those branches is taken at a time.

I decided to apply these changes because they are most straightforward approach to optimizing this code. I also investigated the possibility of applying partial loop unrolling, but if you do that then you will quickly need lots of extra registers.

Conclusion

In this report, I presented the optimization of five different string library functions written in ARM assembly. The primary optimizations employed were preloading to remove stalls and removing branches by using conditional instructions. I could have done more optimization to some of the functions by applying loop unrolling, but in some cases it wasn't feasible.