ECE 471 – Embedded Systems
Lecture 6

Vince Weaver

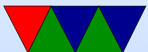
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

17 September 2015

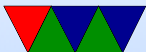
Announcements

- HW#3 will be posted today



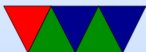
What the OS gives you at start

- Registers
- Instruction pointer at beginning
- Stack
- command line arguments, aux, environment variables
- Large contiguous VM space

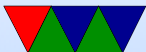


ARM Architecture

- 32-bit
- Load/Store
- Can be Big-Endian or Little-Endian (usually little)
- Fixed instruction width (32-bit, 16-bit THUMB) (Thumb2 is variable)
- arm32 opcodes typically take three arguments (Destination, Source, Source)
- Cannot access unaligned memory (optional newer chips)
- Status flag (many instructions can optionally set)

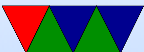


- Conditional execution
- Complicated addressing modes
- Many features optional (FPU [except in newer], PMU, Vector instructions, Java instructions, etc.)

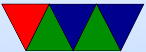


Registers

- Has 16 GP registers (more available in supervisor mode)
- r0 - r12 are general purpose
- r11 is sometimes the frame pointer (fp) [iOS uses r7]
- r13 is stack pointer (sp)
- r14 is link register (lr)
- r15 is program counter (pc)
  reading r15 usually gives PC+8
- 1 status register (more in system mode).
  NZCVQ (Negative, Zero, Carry, oVerflow, Saturate)

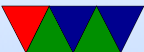


Low-Level ARM Linux Assembly



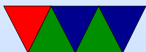
Kernel Programming ABIs

- OABI – "old" original ABI (arm). Being phased out. slightly different syscall mechanism, different alignment restrictions
- EABI – new "embedded" ABI (armel)
- hard float – EABI compiled with ARMv7 and VFP (vector floating point) support (armhf). Raspberry Pi (raspbian) is compiled for ARMv6 armhf.



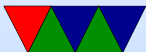
System Calls (EABI)

- System call number in r7
- Arguments in r0 - r6
- Call `swi 0x0`
- System call numbers can be found in `/usr/include/arm-linux-gnueabihf/asm/unistd.h` They are similar to the 32-bit x86 ones.



System Calls (OABI)

The previous implementation had the same system call numbers, but instead of r7 the number was the argument to `swi`. This was very slow, as there is no way to determine that value without having the kernel backtrace the callstack and disassemble the instruction.

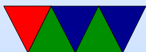


Manpage

The easiest place to get system call documentation.

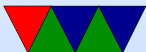
`man open 2`

Finds the documentation for "open". The 2 means look for system call documentation (which is type 2).



A first ARM assembly program: hello_exit

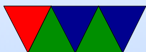
```
.equ SYSCALL_EXIT,     1

        .globl _start
_start:

        #================================
        # Exit
        #================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number (1) in eax
        swi     0x0                     @ and exit
```



hello_exit example

Assembling/Linking using make, running, and checking the output.

```
lecture6$ make hello_exit_arm
as -o hello_exit_arm.o hello_exit_arm.s
ld -o hello_exit_arm hello_exit_arm.o
lecture6$ ./hello_exit_arm
lecture6$ echo $?
5
```

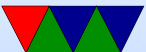


Assembly

- @ is the comment character. # can be used on line by itself but will confuse assembler if on line with code. Can also use /* */

- Order is source, destination

- Constant value indicated by # or $



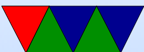
Let's look at our executable

- `ls -la ./hello_exit_arm`
  Check the size

- `readelf -a ./hello_exit_arm`
  Look at the ELF executable layout

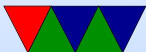
- `objdump --disassemble-all ./hello_exit_arm`
  See the machine code we generated

- `strace ./hello_exit_arm`
  Trace the system calls as they happen.



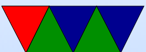
hello_world example

```
.equ SYSCALL_EXIT,      1
.equ SYSCALL_WRITE,     4
.equ STDOUT,            1

        .globl _start
_start:
        mov     r0,#STDOUT              /* stdout */
        ldr     r1,=hello
        mov     r2,#13                  @ length
        mov     r7,#SYSCALL_WRITE
        swi     0x0

        # Exit
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number in r7
        swi     0x0                     @ and exit

.data
hello:          .ascii "Hello World!\n"
```

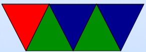


New things to note in `hello_world`

- The fixed-length 32-bit ARM cannot hold a full 32-bit immediate
- Therefore a 32-bit address cannot be loaded in a single instruction
- In this case the "=" is used to request the address be stored in a "literal" pool which can be reached by PC-offset, with an extra layer of indirection.



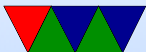
ARM Assembly Review



Floating Point

ARM floating point varies and is often optional.

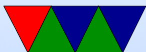
- various versions of vector floating point unit
- vfp3 has 16 or 32 64-bit registers
- Advanced SIMD – reuses vfp registers
  Can see as 16 128-bit regs q0-q15 or 32 64-bit d0-d31 and 32 32-bit s0-s31
- SIMD supports integer, also 16-bit?
- Polynomial?
- FPSCR register (flags)



Arithmetic Instructions

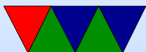
Most of these take optional s to set status flag

adc	v1	add with carry
add	v1	add
rsb	v1	reverse subtract (immediate - rX)
rsc	v1	reverse subtract with carry
sbc	v1	subtract with carry
sub	v1	subtract



Register Manipulation

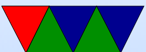
mov, movs	v1	move register
mvn, mvns	v1	move inverted



Loading Constants

- In general you can get a 12-bit immediate which is 8 bits of unsigned and 4-bits of even rotate (rotate by 2*value). `mov r0, #45`

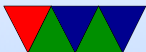
- You can specify you want the assembler to try to make the immediate for you: `ldr r0,=0xff`
  `ldr r0,=label`
  If it can't make the immediate value, it will store in nearby in a `literal pool` and do a memory read.



Extra Shift in ALU instructions

If second source is a register, can optionally shift:

- LSL – Logical shift left
- LSR – Logical shift right
- ASR – Arithmetic shift right
- ROR – Rotate Right (last bit into carry)
- RRX – Rotate Right with Extend
  bit zero into C, C into bit 31 (33-bit rotate)

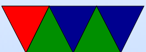


- Why no ASL?
- For example:

```
add r1, r2, r3, lsr #4
r1 = r2 + (r3>>4)
```

- Another example (what does this do):

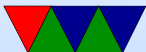
```
add r1, r2, r2, lsl #2
```



Shift Instructions

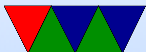
Implemented via `mov` with shift on arm32.

asr		arith shift right
lsl		logical shift left
lsr		logical shift right
ror		rors – rotate right
rorx		rotate right extend: bit 0 into C, C into bit 31



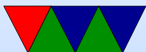
Rotate instructions

- Looked in my code, as well as in *Hacker's Delight*
- Often used when reversing bits (say, for endian conversion)
- Often used because shift instructions typically don't go through the carry glad, but rotates often do
- Used on x86 to use a 32-bit register as two 16-bit registers (can quickly swap top and bottom)



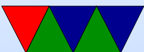
Shift Example

- Shift example (what does this do):
  `add r1, r2, r2, lsl #2`
- `teq` vs `cmp` – teq in general doesn't change carry flag
- Constant is only 8-bits unsigned, with 4 bits of even rotate



Logic Instructions

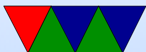
and	v1	bitwise and
bfc	??	bitfield clear, clear bits in reg
bfi	??	bitfield insert
bic	v1	bitfield clear: and with negated value
clz	v7	count leading zeros
eor	v1	exclusive or (name shows 6502 heritage)
orn	v6	or not
orr	v1	bitwise or



Comparison Instructions

Updates status flag, no need for s

cmp	v1	compare (subtract but discard result)
cmn	v1	compare negative (add)
teq	v1	tests if two values equal (xor) (preserves carry)
tst	v1	test (and)

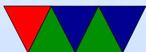


Multiply Instructions

Fast multipliers are optional

For 64-bit results,

mla	v2	multiply two registers, add in a third (4 arguments)
mul	v2	multiply two registers, only least sig 32bit saved
smlal	v3M	32x32+64 = 64-bit (result and add source, reg pair rdhi,rdlo)
smull	v3M	32x32 = 64-bit
umlal	v3M	unsigned 32x32+64 = 64-bit
umull	v3M	unsigned 32x32=64-bit

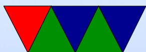


Control-Flow Instructions

Can use all of the condition code prefixes.

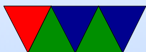
Branch to a label, which is +/- 32MB from PC

b	v1	branch
bl	v1	branch and link (return value stored in lr)
bx	v4t	branch to offset or reg, possible THUMB switch
blx	v5	branch and link to register, with possible THUMB switch
mov pc,lr	v1	return from a link



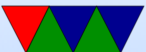
Load/Store Instructions

ldr	v1	load register
ldrb	v1	load register byte
ldrd	v5	load double, into consecutive registers (Rd even)
ldrh	v1	load register halfword, zero extends
ldrsb	v1	load register signed byte, sign-extends
ldrsh	v1	load register halfword, sign-extends
str	v1	store register
strb	v1	store byte
strd	v5	store double
strh	v1	store halfword

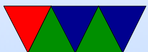


Addressing Modes

- `ldrb r1, [r2]` @ register
- `ldrb r1, [r2,#20]` @ register/offset
- `ldrb r1, [r2,+r3]` @ register + register
- `ldrb r1, [r2,-r3]` @ register - register
- `ldrb r1, [r2,r3, LSL #2]` @ register +/- register, shift



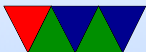
- `ldrb r1, [r2, #20]!` @ pre-index. Load from r2+20 then write back
- `ldrb r1, [r2, r3]!` @ pre-index. register
- `ldrb r1, [r2, r3, LSL #4]!` @ pre-index. shift
- `ldrb r1, [r2],#+1` @ post-index. load, then add value to r2
- `ldrb r1, [r2],r3` @ post-index register
- `ldrb r1, [r2],r3, LSL #4` @ post-index shift



Load/Store multiple (stack?)

In general, no interrupt during instruction so long instruction can be bad in embedded
Some of these have been deprecated on newer processors

- ldm – load multiple memory locations into consecutive registers
- stm – store multiple, can be used like a PUSH instruction
- push and pop are thumb equivelent

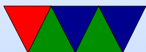


Can have address mode and ! (update source):

- IA – increment after (start at Rn)
- IB – increment before (start at Rn+4)
- DA – decrement after
- DB – decrement before

Can have empty/full. Full means SP points to a used location, Empty means it is empty:

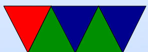
- FA – Full ascending



- FD – Full descending
- EA – Empty ascending
- ED – Empty descending

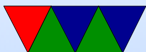
Recent machines use the "ARM-Thumb Proc Call Standard" which says a stack is Full/Descending, so use LDMFD/STMFD.

What does `stm SP!, {r0,lr}` then `ldm SP!, {r0,PC,pc}` do?

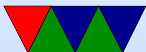


System Instructions

- svc, swi – software interrupt
  takes immediate, but ignored.
- mrs, msr – copy to/from status register. use to clear interrupts? Can only set flags from userspace
- cdp – perform coprocessor operation
- mrc, mcr – move data to/from coprocessor
- ldc, stc – load/store to coprocessor from memory

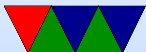


Co-processor 15 is the *system control coprocessor* and is used to configure the processor. Co-processor 14 is the debugger 11 is double-precision floating point 10 is single-precision fp as well as VFP/SIMD control 0-7 vendor specific



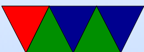
Other Instructions

- swp – atomic swap value between register and memory (deprecated armv7)
- ldrex/strex – atomic load/store (armv6)
- wfe/sev – armv7 low-power spinlocks
- pli/pld – preload instructions/data
- dmb/dsb – memory barriers



Pseudo-Instructions

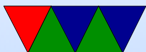
adr		add immediate to PC, store address in reg
nop		no-operation



Prefixed instructions

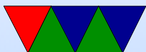
Most instructions can be prefixed with condition codes:

EQ, NE	(equal)	Z==1/Z==0
MI, PL	(minus/plus)	N==1/N==0
HI, LS	(unsigned higher/lower)	C==1&Z==0/C==0|Z==1
GE, LT	(greaterequal/lessthan)	N==V/N!=V
GT, LE	(greaterthan, lessthan)	N==V&Z==0/N!=V|Z==1
CS,HS, CC,LO	(carry set,higher or same/clear)	C==1,C==0
VS, VC	(overflow set / clear)	V==1,V==0
AL	(always)	(this is the default)



Setting Flags

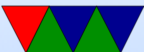
- `add r1,r2,r3`
- `adds r1,r2,r3` – set condition flag
- `addeqs r1,r2,r3` – set condition flag and prefix compiler and disassembler like `addseq`, GNU as doesn't?



Conditional Execution

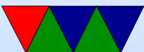
```
if (x == 1 )
    a+=2;
else
    b-=2;
```

```
cmp      r1, #5
addeq    r2,r2,#2
subne    r3,r3,#2
```

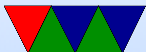


Fancy ARMv6

- mla – multiply/accumulate (armv6)
- mls – multiply and subtract
- pkh – pack halfword (armv6)
- qadd, qsub, etc. – saturating add/sub (armv6)
- rbit – reverse bit order (armv6)
- rbyte – reverse byte order (armv6)
- rev16, revsh – reverse halfwords (armv6)
- sadd16 – do two 16-bit signed adds (armv6)
- sadd8 – do 4 8-bit signed adds (armv6)

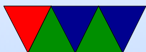


- sasx – (armv6)
- sbfx – signed bit field extract (armv6)
- sdiv – signed divide (only armv7-R)
- udiv – unsigned divide (armv7-R only)
- sel – select bytes based on flag (armv6)
- sm* – signed multiply/accumulate
- setend – set endianess (armv6)
- sxtb – sign extend byte (armv6)
- tbb – table branch byte, jump table (armv6)
- teq – test equivalence (armv6)
- u* – unsigned partial word instructions



ARM Instruction Set Encodings

- ARM – 32 bit encoding
- THUMB – 16 bit encoding
- THUMB-2 – THUMB extended with 32-bit instructions
  - STM32L *only* has THUMB2
  - Original Raspberry Pis *do not* have THUMB2
  - Raspberry Pi 2 *does* have THUMB2
- THUMB-EE – some extensions for running in JIT runtime
- AARCH64 – 64 bit. Relatively new.

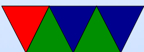


Recall the ARM32 encoding

```
ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

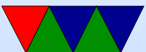
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
cond				0	0	0	0	1	0	0	S	Rn			
								Opcode							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Shift imm5					Shift typ		Sh Reg	Rm			

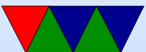


THUMB

- Most instructions length 16-bit (a few 32-bit)
- Only r0-r7 accessible normally
  `add`, `cmp`, `mov` can access high regs
- Some operands (sp, lr, pc) implicit
  Can't always update sp or pc anymore.
- No prefix/conditional execution
- Only two arguments to opcodes
  (some exceptions for small constants: `add r0,r1,#1`)
- 8-bit constants rather than 12-bit

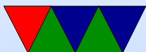


- Limited addressing modes: [rn,rm], [rn,#imm], [pc|sp,#imm]
- No shift parameter ALU instructions
- Makes assumptions about “S” setting flags
  (gas doesn’t let you superfluously set it, causing problems if you naively move code to THUMB-2)
- new push/pop instructions (subset of ldm/stm), neg (to negate), asr,lsl,lsr,ror, bic (logic bit clear)



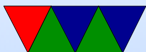
THUMB/ARM interworking

- See `print_string_armthumb.s`
- BX/BLX instruction to switch mode.
  If target is a label, *always* switchmode
  If target is a register, low bit of 1 means THUMB, 0 means ARM
- Can also switch modes with `ldrm`, `ldm`, or `pop` with PC as a destination
  (on armv7 can enter with ALU op with PC destination)
- Can use `.thumb` directive, `.arm` for 32-bit.

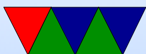


THUMB-2

- Extension of THUMB to have both 16-bit and 32-bit instructions
- 32-bit instructions *not* standard 32-bit ARM instructions. It's a new encoding that allows an instruction to be 32-bit if needed.
- Most 32-bit ARM instructions have 32-bit THUMB-2 equivalents *except* ones that use conditional execution. The `it` instruction was added to handle this.
- `rsc` (reverse subtract with carry) removed

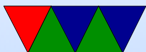


- Shifts in ALU instructions are by constant, cannot shift by register like in arm32
- THUMB-2 code can assemble to either ARM-32 or THUMB2
  The assembly language is compatible.
  Common code can be written and output changed at time of assembly.
- Instructions have "wide" and "narrow" encoding.
  Can force this (`add.w` vs `add.n`).
- Need to properly indicate "s" (set flags).
  On regular THUMB this is assumed.



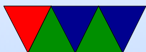
THUMB-2 Coding

- See `test_thumb2.s`
- Use `.syntax unified` at beginning of code
- Use `.arm` or `.thumb` to specify mode



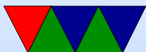
New THUMB-2 Instructions

- BFI – bit field insert
- RBIT – reverse bits
- movw/movt – 16 bit immediate loads
- TB – table branch
- IT (if/then)
- cbz – compare and branch if zero; only jumps forward



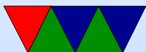
Thumb-2 12-bit immediates

```
top 4 bits 0000 -- 00000000 00000000 00000000 abcdefgh
           0001 -- 00000000 abcdefgh 00000000 abcdefgh
           0010 -- abcdefgh 00000000 abcdefgh 00000000
           0011 -- abcdefgh abcdefgh abcdefgh abcdefgh
           0100 -- 1bcdedfh 00000000 00000000 00000000
            ...
           1111 -- 00000000 00000000 00000001 bcdefgh0
```



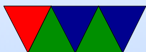
Compiler

- Original RASPBERRY PI DOES NOT SUPPORT THUMB2
- `gcc -S hello_world.c`
  By default is arm32
- `gcc -S -march=armv5t -mthumb hello_world.c`
  Creates THUMB (won't work on Rapberry Pi due to HARDFP arch)
- `-mthumb -march=armv7-a` Creates THUMB2



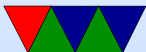
IT (If/Then) Instruction

- Allows limited conditional execution in THUMB-2 mode.
- The directive is optional (and ignored in ARM32) the assembler can (in-theory) auto-generate the IT instruction
- Limit of 4 instructions



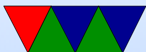
Example Code

```
it cc
addcc r1,r2

itete cc
addcc r1,r2
addcs r1,r2
addcc r1,r2
addcs r1,r2
```



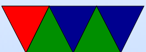
`ll` Example Code

```
        ittt cs @ If CS Then Next plus CS for next 3
discrete_char:
        ldrbcs  r4,[r3]      @ load a byte
        addcs   r3,#1        @ increment pointer
        movcs   r6,#1        @ we set r6 to one so byte
        bcs.n   store_byte   @ and store it
offset_length:
```



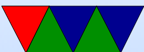
AARCH64

- 32-bit fixed instruction encoding
- 31 64-bit GP registers (x0-x30), zero register (x30)
- PC is not a GP register
- only branches conditional
- no load/store multiple
- No thumb

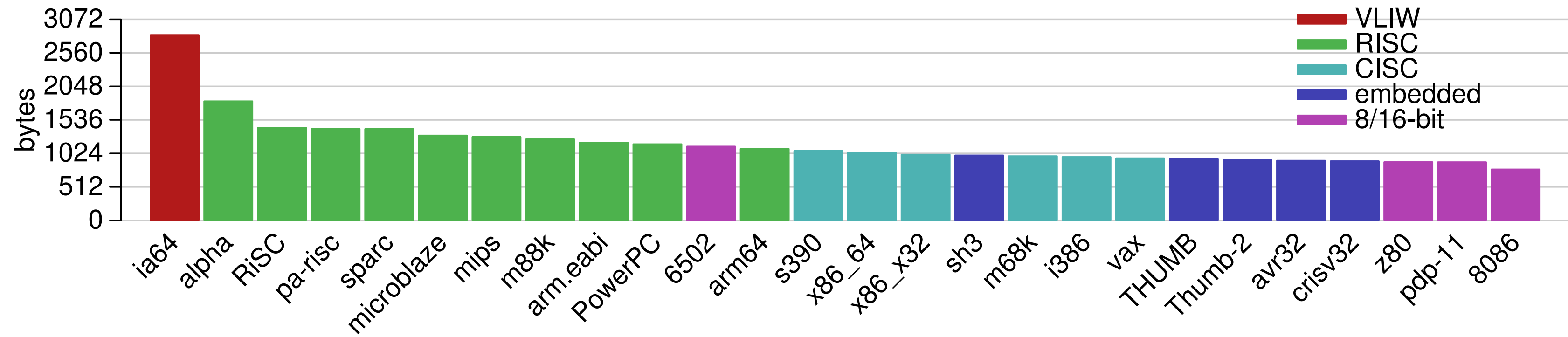


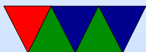
Code Density

- Overview from my `ll` ICCD'09 paper
- Show code density for variety of architectures, recently added Thumb-2 support.
- Shows overall size, though not a fair comparison due to operating system differences on non-Linux machines



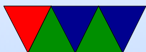
Code Density – overall



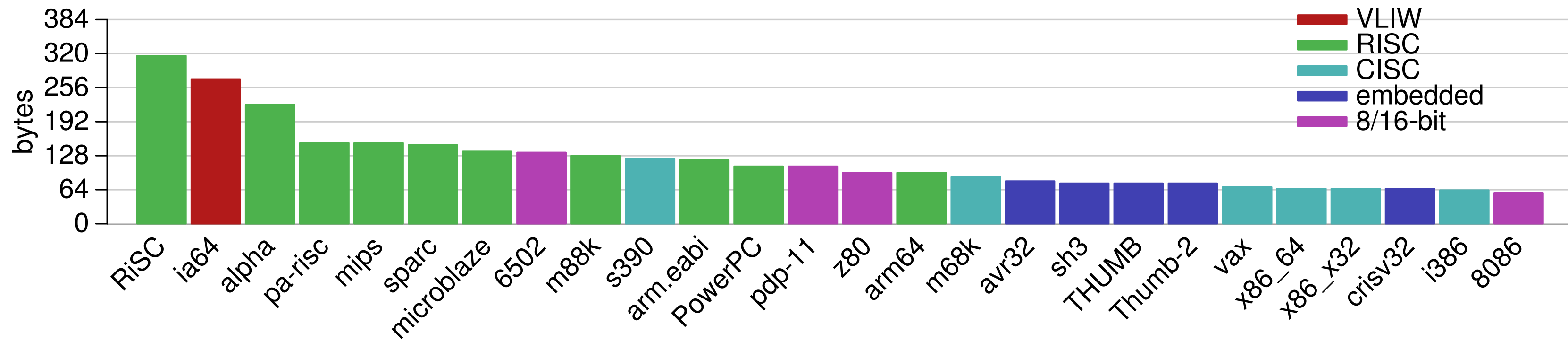


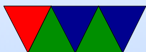
lzss compression

- Printing routine uses lzss compression
- Might be more representative of potential code density



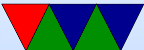
Code Density – lzss



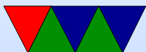


Put string example

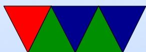
```
.equ SYSCALL_EXIT,      1
.equ SYSCALL_WRITE,     4
.equ STDOUT,            1

        .globl _start
_start:
        ldr     r1,=hello
        bl      print_string            @ Print Hello World
        ldr     r1,=mystery
        bl      print_string            @
        ldr     r1,=goodbye
        bl      print_string            /* Print Goodbye */

        #=================================
        # Exit
        #=================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number (1) in eax
        swi     0x0                     @ and exit
```



```
        #====================
        # print string
        #====================
        # Null-terminated string to print pointed to by r1
        # r1 is trashed by this routine

print_string:
        push    {r0,r2,r7,r10}          @ Save r0,r2,r7,r10 on stack

        mov     r2,#0                   @ Clear Count

count_loop:
        add     r2,r2,#1                @ increment count
        ldrb    r10,[r1,r2]             @ load byte from address r1+r2
        cmp     r10,#0                  @ Compare against 0
        bne     count_loop              @ if not 0, loop

        mov     r0,#STDOUT              @ Print to stdout
        mov     r7,#SYSCALL_WRITE       @ Load syscall number
        swi     0x0                     @ System call

        pop     {r0,r2,r7,r10}          @ pop r0,r2,r7,r10 from stack

        mov     pc,lr                   @ Return to address stored in
```



```
                                              @ Link register

.data
hello:            .string "Hello␣World!\n"   @ includes null at end
mystery:          .byte 63,0x3f,63,10,0      @ mystery string
goodbye:          .string "Goodbye!\n"       @ includes null at end
```



Clarification of Assembler Syntax

- @ is the comment character. # can be used on line by itself but will confuse assembler if on line with code. Can also use /* */
- Constant value indicated by # or $
- Optionally put % in front of register name