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.
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**

```assembly
.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

.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\nWorld!\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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adc v1</td>
<td>add with carry</td>
</tr>
<tr>
<td>add v1</td>
<td>add</td>
</tr>
<tr>
<td>rsb v1</td>
<td>reverse subtract (immediate - rX)</td>
</tr>
<tr>
<td>rsc v1</td>
<td>reverse subtract with carry</td>
</tr>
<tr>
<td>sbc v1</td>
<td>subtract with carry</td>
</tr>
<tr>
<td>sub v1</td>
<td>subtract</td>
</tr>
</tbody>
</table>
# Register Manipulation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Dest.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov, movs</td>
<td>v1</td>
<td>move register</td>
</tr>
<tr>
<td>mvn, mvns</td>
<td>v1</td>
<td>move inverted</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asr</td>
<td>arith shift right</td>
</tr>
<tr>
<td>lsl</td>
<td>logical shift left</td>
</tr>
<tr>
<td>lsr</td>
<td>logical shift right</td>
</tr>
<tr>
<td>ror</td>
<td>rors – rotate right</td>
</tr>
<tr>
<td>rorx</td>
<td>rotate right extend: bit 0 into C, C into bit 31</td>
</tr>
</tbody>
</table>
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 flag, 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and v1</td>
<td>bitwise and</td>
</tr>
<tr>
<td>bfc ??</td>
<td>bitfield clear, clear bits in reg</td>
</tr>
<tr>
<td>bfi ??</td>
<td>bitfield insert</td>
</tr>
<tr>
<td>bic v1</td>
<td>bitfield clear: and with negated value</td>
</tr>
<tr>
<td>clz v7</td>
<td>count leading zeros</td>
</tr>
<tr>
<td>eor v1</td>
<td>exclusive or (name shows 6502 heritage)</td>
</tr>
<tr>
<td>orn v6</td>
<td>or not</td>
</tr>
<tr>
<td>orr v1</td>
<td>bitwise or</td>
</tr>
</tbody>
</table>
Comparison Instructions

Updates status flag, no need for $s$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>v1</td>
<td>compare (subtract but discard result)</td>
</tr>
<tr>
<td>cmn</td>
<td>v1</td>
<td>compare negative (add)</td>
</tr>
<tr>
<td>teq</td>
<td>v1</td>
<td>tests if two values equal (xor) (preserves carry)</td>
</tr>
<tr>
<td>tst</td>
<td>v1</td>
<td>test (and)</td>
</tr>
</tbody>
</table>
Multiply Instructions

Fast multipliers are optional
For 64-bit results,

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mla v2</td>
<td>multiply two registers, add in a third (4 arguments)</td>
</tr>
<tr>
<td>mul v2</td>
<td>multiply two registers, only least sig 32bit saved</td>
</tr>
<tr>
<td>smlal v3M</td>
<td>32x32+64 = 64-bit (result and add source, reg pair rdhi,rdlo)</td>
</tr>
<tr>
<td>smull v3M</td>
<td>32x32 = 64-bit</td>
</tr>
<tr>
<td>umlal v3M</td>
<td>unsigned 32x32+64 = 64-bit</td>
</tr>
<tr>
<td>umull v3M</td>
<td>unsigned 32x32=64-bit</td>
</tr>
</tbody>
</table>
Control-Flow Instructions

Can use all of the condition code prefixes.
Branch to a label, which is +/- 32MB from PC

<table>
<thead>
<tr>
<th>b</th>
<th>v1</th>
<th>branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>bl</td>
<td>v1</td>
<td>branch and link (return value stored in lr)</td>
</tr>
<tr>
<td>bx</td>
<td>v4t</td>
<td>branch to offset or reg, possible THUMB switch</td>
</tr>
<tr>
<td>blx</td>
<td>v5</td>
<td>branch and link to register, with possible THUMB switch</td>
</tr>
<tr>
<td>mov pc,lr</td>
<td>v1</td>
<td>return from a link</td>
</tr>
</tbody>
</table>
## Load/Store Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr</td>
<td>v1</td>
<td>load register</td>
</tr>
<tr>
<td>ldrb</td>
<td>v1</td>
<td>load register byte</td>
</tr>
<tr>
<td>ldrd</td>
<td>v5</td>
<td>load double, into consecutive registers (Rd even)</td>
</tr>
<tr>
<td>ldrh</td>
<td>v1</td>
<td>load register halfword, zero extends</td>
</tr>
<tr>
<td>ldrsrb</td>
<td>v1</td>
<td>load register signed byte, sign-extends</td>
</tr>
<tr>
<td>ldrsh</td>
<td>v1</td>
<td>load register halfword, sign-extends</td>
</tr>
<tr>
<td>str</td>
<td>v1</td>
<td>store register</td>
</tr>
<tr>
<td>strb</td>
<td>v1</td>
<td>store byte</td>
</tr>
<tr>
<td>strd</td>
<td>v5</td>
<td>store double</td>
</tr>
<tr>
<td>strh</td>
<td>v1</td>
<td>store halfword</td>
</tr>
</tbody>
</table>
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 equivalent
Can have address mode and ! (update source):

- IA – increment after (start at \( R_n \))
- IB – increment before (start at \( R_n + 4 \))
- 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

<table>
<thead>
<tr>
<th>adr</th>
<th>add immediate to PC, store address in reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>no-operation</td>
</tr>
</tbody>
</table>
## Prefixed instructions

Most instructions can be prefixed with condition codes:

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ, NE</td>
<td>(equal)</td>
<td>Z==1/Z==0</td>
</tr>
<tr>
<td>MI, PL</td>
<td>(minus/plus)</td>
<td>N==1/N==0</td>
</tr>
<tr>
<td>HI, LS</td>
<td>(unsigned higher/lower)</td>
<td>C==1&amp;Z==0/C==0</td>
</tr>
<tr>
<td>GE, LT</td>
<td>(greaterequal/lessthan)</td>
<td>N==V/N!=V</td>
</tr>
<tr>
<td>GT, LE</td>
<td>(greaterthan, lessthan)</td>
<td>N==V&amp;Z==0/N!=V</td>
</tr>
<tr>
<td>CS, HS, CC, LO</td>
<td>(carry set, higher or same/clear)</td>
<td>C==1,C==0</td>
</tr>
<tr>
<td>VS, VC</td>
<td>(overflow set / clear)</td>
<td>V==1,V==0</td>
</tr>
<tr>
<td>AL</td>
<td>(always)</td>
<td>(this is the default)</td>
</tr>
</tbody>
</table>
Setting Flags

- `add r1,r2,r3`
- `adds r1,r2,r3` – set condition flag
- `addeq$s 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

\[ \text{ADD}\{S\}\langle c\rangle \ <Rd>,<Rn>,<Rm>\{,\langle \text{shift}\rangle\} \]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>Shift imm5</td>
<td>Shift typ</td>
<td>Sh Reg</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

47
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_armthumbs.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 Raspberry 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
```
### Example Code

```plaintext
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
bcn.s 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 11 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 – Izss

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Code Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC</td>
<td>320 bytes</td>
</tr>
<tr>
<td>ia64</td>
<td>192 bytes</td>
</tr>
<tr>
<td>alpha</td>
<td>128 bytes</td>
</tr>
<tr>
<td>parisc</td>
<td>96 bytes</td>
</tr>
<tr>
<td>mips</td>
<td>64 bytes</td>
</tr>
<tr>
<td>sparc</td>
<td>32 bytes</td>
</tr>
<tr>
<td>microblaze</td>
<td>16 bytes</td>
</tr>
<tr>
<td>6502</td>
<td>8 bytes</td>
</tr>
<tr>
<td>m68k</td>
<td>4 bytes</td>
</tr>
<tr>
<td>s390</td>
<td>2 bytes</td>
</tr>
<tr>
<td>arm.eabi</td>
<td>1 byte</td>
</tr>
<tr>
<td>PowerPC</td>
<td>1 byte</td>
</tr>
<tr>
<td>pdp-11</td>
<td>1 byte</td>
</tr>
<tr>
<td>88k</td>
<td>1 byte</td>
</tr>
<tr>
<td>arm64</td>
<td>1 byte</td>
</tr>
<tr>
<td>m68k</td>
<td>1 byte</td>
</tr>
<tr>
<td>avr32</td>
<td>1 byte</td>
</tr>
<tr>
<td>sh3</td>
<td>1 byte</td>
</tr>
<tr>
<td>Thumb</td>
<td>1 byte</td>
</tr>
<tr>
<td>Thumb-2</td>
<td>1 byte</td>
</tr>
<tr>
<td>vax</td>
<td>1 byte</td>
</tr>
<tr>
<td>x86</td>
<td>1 byte</td>
</tr>
<tr>
<td>x86_64</td>
<td>1 byte</td>
</tr>
<tr>
<td>x32</td>
<td>1 byte</td>
</tr>
<tr>
<td>crisv32</td>
<td>1 byte</td>
</tr>
<tr>
<td>i386</td>
<td>1 byte</td>
</tr>
<tr>
<td>8086</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

**Note:** Code density is measured in bytes. The chart compares different architectures in terms of code density.
Put string example

.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\nWorld!\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