ECE 471 – Embedded Systems
Lecture 9

Vince Weaver

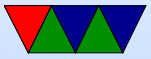
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

18 September 2017

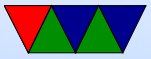
Announcements

- How is HW#3 going?

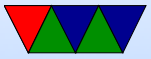


HW3 Notes

- Writing int to string conversion is a complex test.
- Good reverse engineering experience. Block of code from one of my older projects when I wasn't quite as good at ARM assembly.
- What does .lcomm do? Reserves region in the BSS. `.lcomm buffer,20` is similar to C `char buffer[20]`
- Went over algorithm. Need to divide by 10, put remainder into array backwards, then keep dividing the quotient. Also need to convert to ASCII.



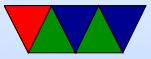
ARM32 Assembly Review



Arithmetic Instructions

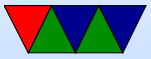
Operate on 32-bit integers. Most of these take optional `s` to set status flag

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



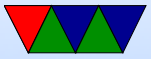
Logic Instructions

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



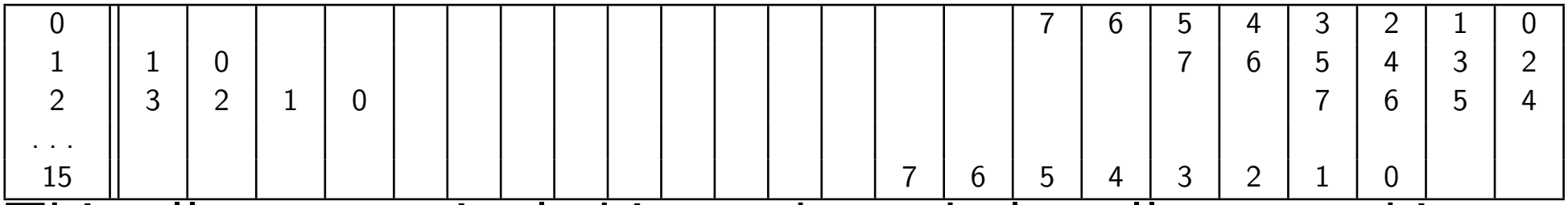
Register Manipulation

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



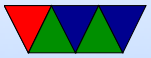
Loading Constants

- 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).



This allows any single bit mask, and also allows masking of any four sub-bytes.

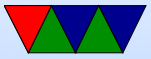
- You can specify you want the assembler to try to make



the immediate for you: `ldr r0,=0xff`

`ldr r0,=label`

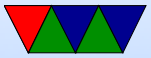
If it can't make the immediate value, it will store in nearby in a `literal pool` and do a memory read.



Barrel 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
- RRX – Rotate Right with Extend
  bit zero into C, C into bit 31 (33-bit rotate)
- Why no ASL?
- Adding s `lsls`, `lsrs` puts shifted out bit into C.

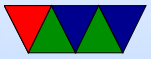


- shift pseudo instructions

  `lsr r0, #3` is same as `mov r0,r0 LSR #3`
- For example:

  ```
  add r1, r2, r3, lsr #4
  r1 = r2 + (r3>>4)
  ```
- Another example (what does this do):

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

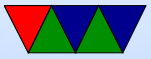


Multiply Instructions

Fast multipliers are optional

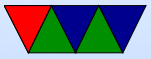
For 64-bit results,

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



Divide Instructions

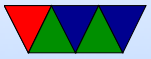
- On some machines it's just not there. Original Pi. Why?
- What do you do if you want to divide?
- Shift and subtract (long division)
- Multiply by reciprocal.



Prefixed instructions

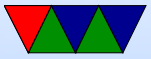
Most instructions can be prefixed with condition codes:

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

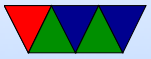
- `add r1,r2,r3`
- `adds r1,r2,r3` – set condition flag
- `addeqs r1,r2,r3` – set condition flag and prefix compiler and disassembler like `addseq`, GNU as doesn't?



Conditional Execution

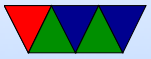
```
if (x == 1 )
    a+=2;
else
    b-=2;
```

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



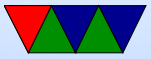
Load/Store Instructions

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

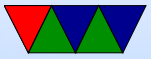
- Regular
  - `ldrb r1, [r2]` @ register
  - `ldrb r1, [r2,#20]` @ register/offset
  - `ldrb r1, [r2,+r3]` @ register + register
  - `ldrb r1, [r2,-r3]` @ register - register
  - `ldrb r1, [r2,r3, LSL #2]` @ register +/- register, shift
- Pre-index. Calculate address, load, then store back
  - `ldrb r1, [r2, #20]!` @ pre-index. Load from



r2+20 then write back into r2

  - `ldrb r1, [r2, r3]! @` pre-index. register
  - `ldrb r1, [r2, r3, LSL #4]! @` pre-index. shift

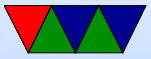
- Post-index: load from base, then add in and write new value to base
  - `ldrb r1, [r2],#+1 @` post-index. load, then add value to r2
  - `ldrb r1, [r2],r3 @` post-index register
  - `ldrb r1, [r2],r3, LSL #4 @` post-index shift



Why some of these?

- `ldrb r1, [r2,#20]` @ register/offset
  Useful for structs in C (i.e. something.else=4;)

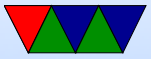
- `ldrb r1, [r2,r3, LSL #2]` @ register +/- register, shift
  Useful for indexing arrays of integers (a[5]=4;)



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)

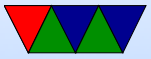


Control-Flow Instructions

Can use all of the condition code prefixes.

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

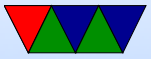
b	v1	branch
bl	v1	branch and link (return value stored in lr)
bx	v4t	branch to offset or reg, possible THUMB switch
blx	v5	branch and link to register, with possible THUMB switch
mov pc,lr	v1	return from a link



Load/Store 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

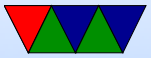


Can have address mode and ! (update source):

- IA – increment after (start at 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:

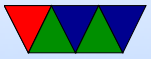
- FA – Full ascending



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

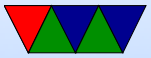
Recent machines use the "ARM-Thumb Proc Call Standard" which says a stack is Full/Descending, so use LDMFD/STMFD.

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

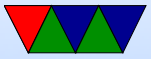


System Instructions

- 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

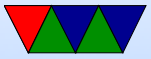


Co-processor 15 is the *system control coprocessor* and is used to configure the processor. Co-processor 14 is the debugger 11 is double-precision floating point 10 is single-precision fp as well as VFP/SIMD control 0-7 vendor specific



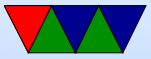
Other Instructions

- 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



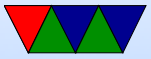
Pseudo-Instructions

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

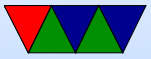


Fancy ARMv6

- 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)



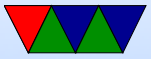
- sasx – (armv6)
- sbfx – signed bit field extract (armv6)
- sdiv – signed divide (only armv7-R)
- udiv – unsigned divide (armv7-R only)
- sel – select bytes based on flag (armv6)
- sm* – signed multiply/accumulate
- setend – set endianess (armv6)
- sxtb – sign extend byte (armv6)
- tbb – table branch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



Floating Point

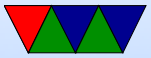
ARM floating point varies and is often optional.

- 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)



Setting Flags

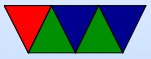
- `add r1,r2,r3`
- `adds r1,r2,r3` – set condition flag
- `addeqs r1,r2,r3` – set condition flag and prefix compiler and disassembler like `addseq`, GNU as doesn't?



Conditional Execution

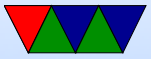
```
if (x == 5 )
    a+=2;
else
    b-=2;
```

```
        cmp     r1 , #5
        bne     else
        add     r2 , r2 ,#2
        b       done
else :
        sub     r3 , r3 ,#2
```



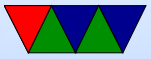
```
done:

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



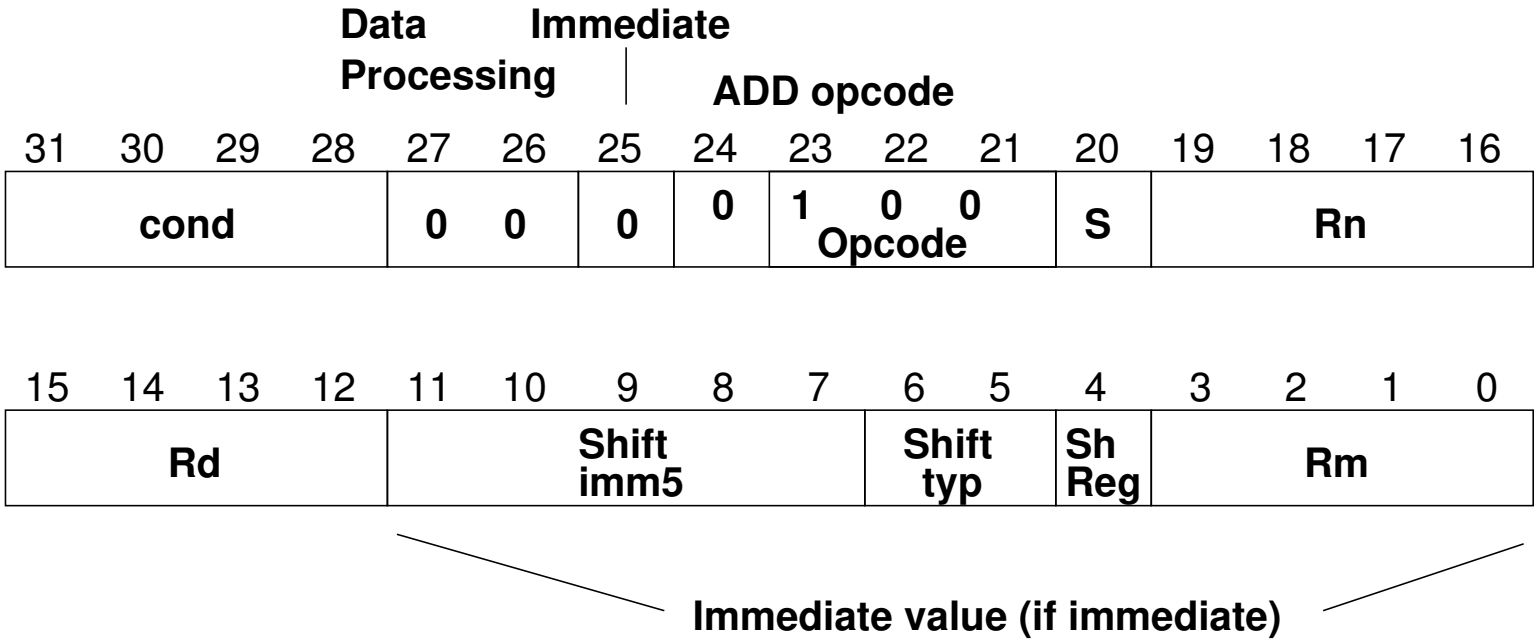
ARM Instruction Set Encodings

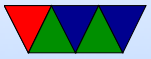
- ARM – 32 bit encoding
- THUMB – 16 bit encoding
- THUMB-2 – THUMB extended with 32-bit instructions
  - STM32L *only* has THUMB2
  - Original Raspberry Pis *do not* have THUMB2
  - Raspberry Pi 2/3 *does* have THUMB2
- THUMB-EE – extensions for running in JIT runtime
- AARCH64 – 64 bit. Relatively new. Completely different from ARM32



Recall the ARM32 encoding

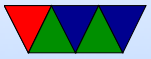
```
ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```



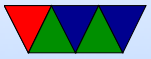


THUMB

- 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.