

ECE 471 – Embedded Systems

Lecture 9

Vince Weaver

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

`vincent.weaver@maine.edu`

21 September 2020

Announcements

- HW#3 was assigned, sorry for the delay
- Don't forget to pick up parts
- If you need parts mailed, let me know as soon as possible



HW3 Notes

- Writing int to string conversion is a complex task
There are lots of ways to do it.
- Corner cases: leading zero suppression?
- Went over algorithm from homework example.
 - Divide by 10, put remainder into array backwards
 - Take quotient as next source and repeat until zero
 - Also need to convert to ASCII. (by adding 0x30 or '0')
- Dividing by 10 on system that has no divide?
 - Original Pi-1B had ARM1176 without a divide



instruction

- To be backwards compatible even new Pis are compiled w/o divide even though new chips have support
- Various ways in software. Iterative subtraction. Shift and subtract.
- For constant values you can divide by instead multiplying by the reciprocal
- gcc will do this. It use 32.32 fixed point multiply by $1/10$. (429496730). ARM has `umul` instruction that will do a 32x32 multiply and give you the top half of the 64-bit result.



Really Brief Overview of ARM32 Assembly

- There's an Appendix at the end of these notes which covers ARM32 Assembly in more detail



ARM32 Assembly – Register moves

- Moving register values around
- `mov r0,r1` – r0 is destination
- `mov r0,#0` – move immediate value
- There are also `msr` and `mrs` to move into special system variables



ARM32 Assembly – Load/Stores

- ARM32 is load/store meaning have to load into register before using values
- `ldrb r0, [address]` – load byte into r0 from pointer
- `strb r0, [address]` – store byte from r0 to memory at pointer
- can support different widths (`ldr`, `ldrb`, `ldrh`, etc)
- sign and zero extend
- Complex addressing modes. register, $r1+r2$, $r1+r2+offset$, auto-increment, etc



ARM32 Assembly – Arithmetic

- add, sub, ...
- `add r0,r1,r2`
- `add r0,r1,#0`
- Barrel shifter allows complex stuff like `add r0,r1,r2`
`LSL #4` to optionally shift/rotate



ARM32 Assembly – Logic

- and,orr, eor
- and r0,r1,r2
- eor r0,r1,#0
- Barrel shifter too



ARM32 Assembly – Comparison

- `cmp r0,r1` – sets flags
- Same as a subtract but doesn't update destination
- Can do same thing with arithmetic if you add 'S' adds `r0,r1,r2`



ARM32 Assembly – Branches

- Branch based on previous comparison
- beq, blt, bgt, etc
- b – unconditional
- bl – branch and link, calls a function and puts return value in special LR (link register)



ARM32 Assembly – Stack Manipulation

- Old “store multiple” instructions, really powerful, can use any arbitrary reg as stack, arbitrary number of registers to push/pop, can change direction and post or pre-increment
- Modern also supports `push {r0, r1}` and `pop {r0, r1}`
- On ARM32 Program Counter (PC) is a regular register. Code will often `push {r0, LR}` at beginning of function to save return, then `pop {r0, PC}` at end which puts



LR back into PC to return without an extra bl LR instruction



Conditional Execution

Why are branches bad?

```
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 :
```

@ equivalent w/o branches



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



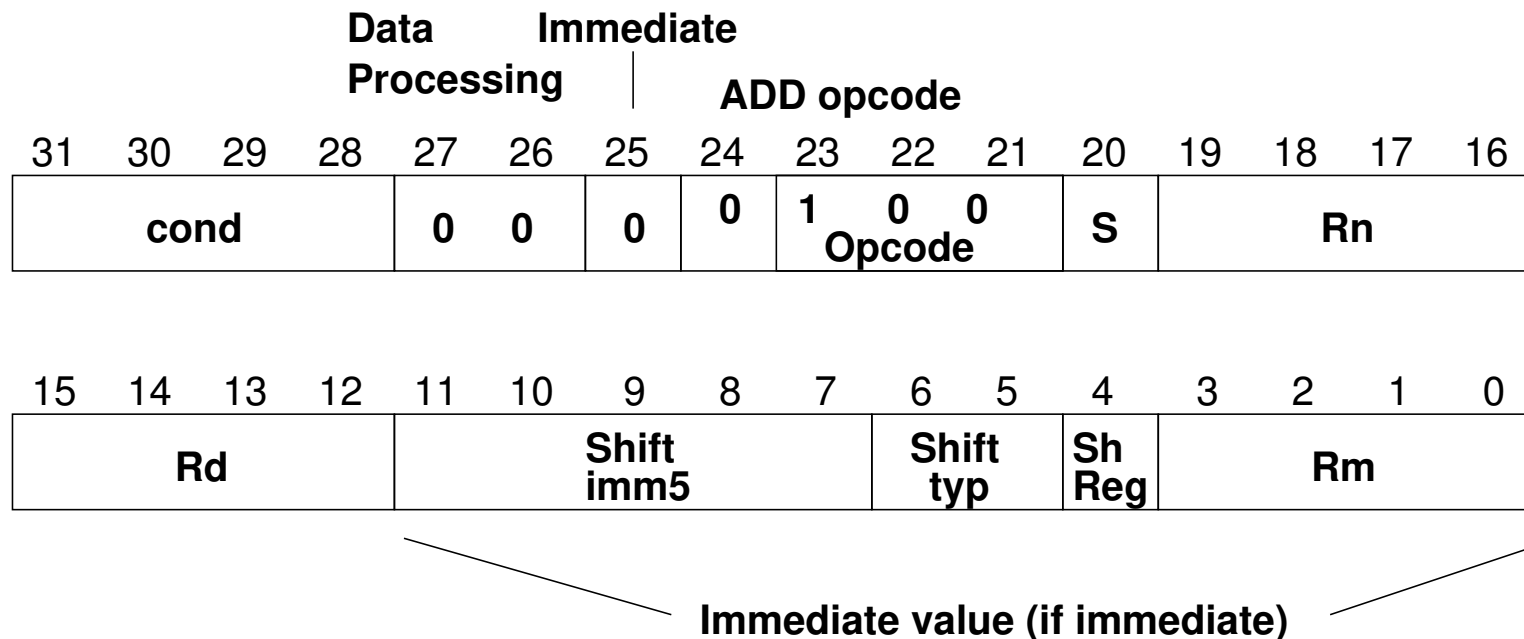
Why Code Density

- Smaller code can be better
- Lower resources: Cheaper? If you can fit more features into smaller RAM or disk you can save money
- Faster? It depends. Modern chips are really hard to predict, but if your processor has Caches and you can fit better in instruction cache it can potentially speed things up a lot



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.
Sets/clears the T (thumb) flag in status register
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
- The 32-bit instructions are *not* the standard 32-bit ARM instructions.
- 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
- Most cannot have PC as src/dest

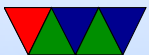


- Shifts in ALU instructions are by constant, cannot shift by register like in arm32
- THUMB-2 code can assemble to either ARM-32 or THUMB2

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
rotate bottom 7 bits|0x80 right by top 5 bits
              01000 -- 1bcdefgh 00000000 00000000 00000000
              ...
              11111 -- 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



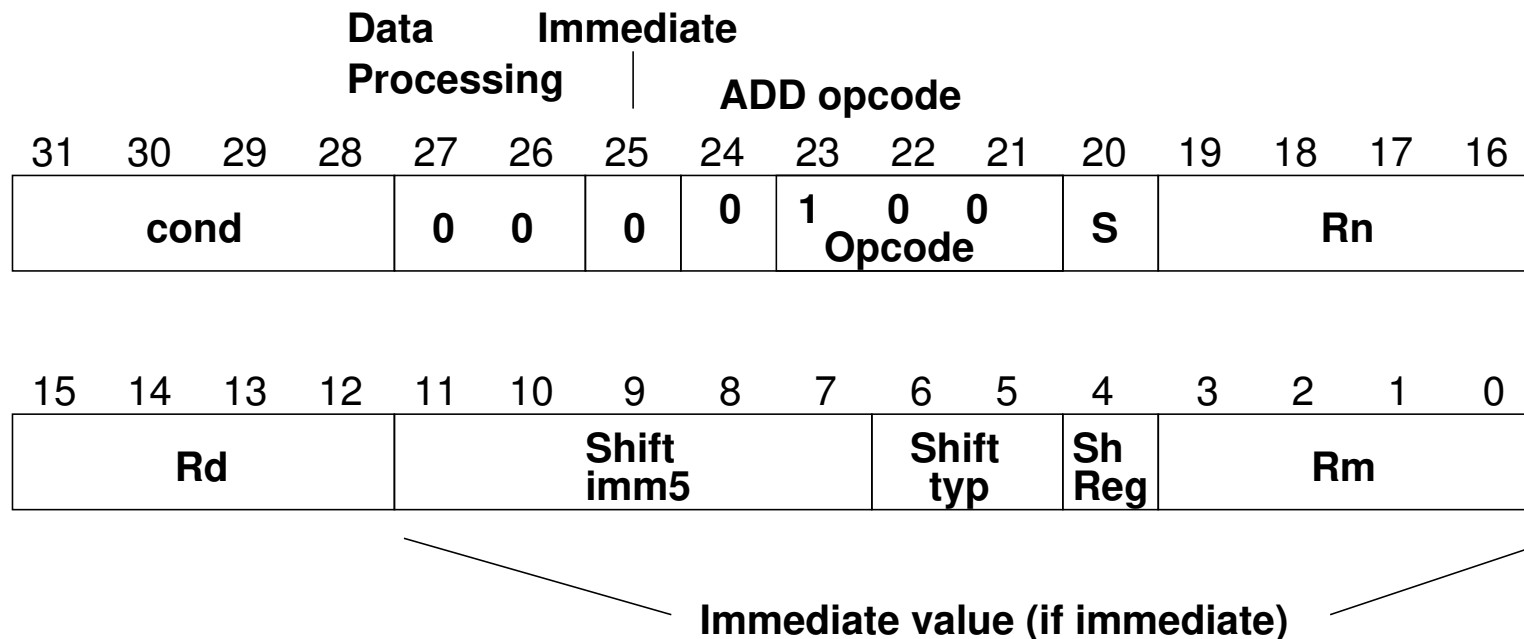
ARM Instruction Set Encodings

- 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>}



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
```

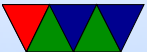
```
itete cc
```

```
addcc r1,r2
```

```
addcs r1,r2
```

```
addcc r1,r2
```

```
addcs r1,r2
```



11 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
- 32 64-bit GP registers
 - x0 - x7 = args
 - x8 - x18 = temp (x8=syscall num during syscall)
 - x19-x28 = callee saved
 - x29 = frame pointer
 - x30 = link register
 - x31 = zero register or stack pointer
- PC is not a GP register



- only branches conditional
- no load/store multiple
- No thumb

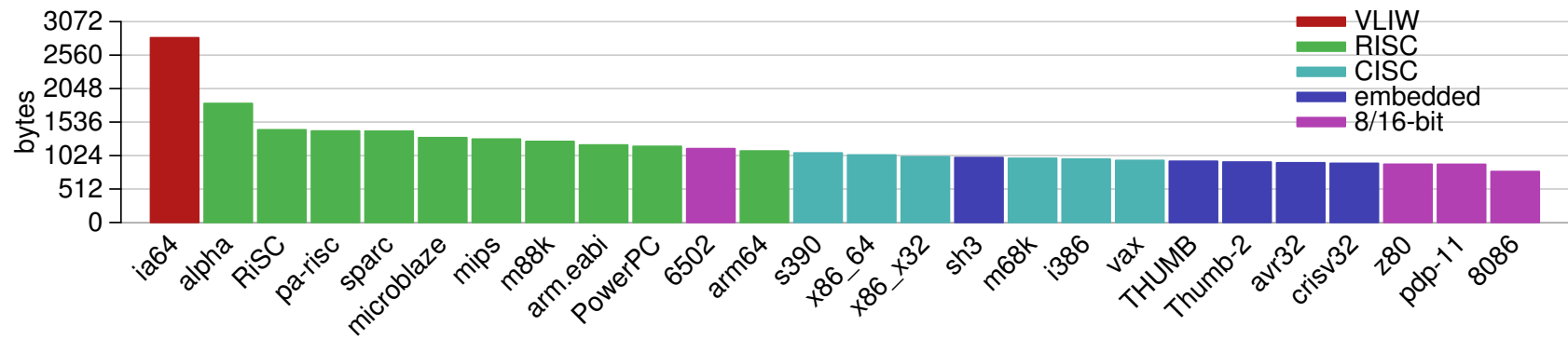


Code Density

- 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

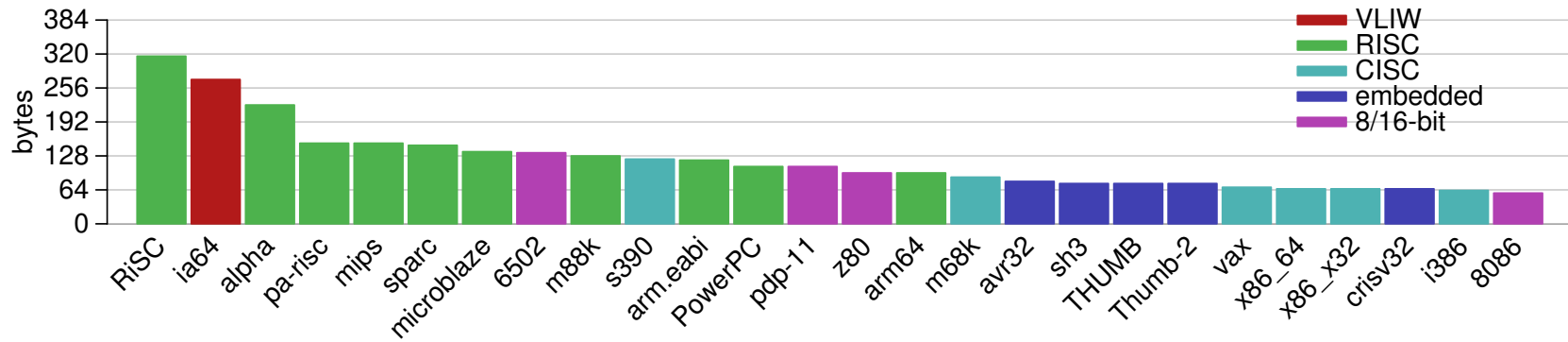


lzss compression

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



Code Density – lzss



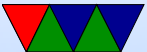
Put string example

```
.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
```



Appendix: Extra notes on ARM32 Assembly



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
```



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



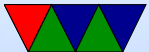
Logic Instructions

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 $2 \times \text{value}$).

0																7	6	5	4	3	2	1	0
1	1	0																7	6	5	4	3	2
2	3	2	1	0																7	6	5	4
...																							
15													7	6	5	4	3	2	1	0			

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`

$r1 = r2 + (r3 \gg 4)$

- 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	$32 \times 32 + 64 = 64$ -bit (result and add source, reg pair rdhi,rdlo)
smull	v3M	$32 \times 32 = 64$ -bit
umlal	v3M	unsigned $32 \times 32 + 64 = 64$ -bit
umull	v3M	unsigned $32 \times 32 = 64$ -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	(greater equal/less than)	$N == V / N != V$
GT, LE	(greater than, less than)	$N == V \& Z == 0 / N != V Z == 1$
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)



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 \pm 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 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:

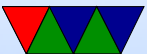
- FA – Full ascending



- 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

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 endianness (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)

