ECE 471 – Embedded Systems
Lecture 5

Vince Weaver

`http://www.eece.maine.edu/∼vweaver`

`vincent.weaver@maine.edu`

16 September 2014

Announcements

- HW#2 is due Thursday
  - People using BeagleBoneBlack
  - Comment your code!
  - I'm adopting Prof. Zhu's "Something Cool" part of the homeworks. If you want to do "something cool" beyond what I list as suggetions, let me know so I can clear it first.

Homework Review

- Follow directions! E-mail title, spell name wrong
- embedded, resource constrained, real-time, dedicated purpose, GUI interface?
- Be strong in your convictions!
- Make sure realize difference between the embedded system (the whole device) and the processor (SoC or microcontroller)

- Phone has real time constraints? Know I didn't explain real-time well, later. Can meet deadlines. Yes, but on which processor?
- Thermostat on grid power? Not necessarily. Mine AAA batteries.
- iPhone – most said was *not* embedded system
- thermostat – all said was embedded system
- ARM1176JZF-S: Java, TrustZone, Vector Floating, Synthesizable

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 as the frame pointer
- 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)

Floating Point

vector floating point vfp3– 32 or 16 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