ECE 471 – Embedded Systems
Lecture 6

Vince Weaver

http://www.eece.maine.edu/∼vweaver

vincent.weaver@maine.edu

19 September 2013

Announcements

- HW#1 is due today
- For next class, at least skim book Chapter 4

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)

Stuff I missed last time

- 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

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 equievlent

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.

Other Instructions

- swp – atomic swap value between register and memory (deprecated armv7)
- ldrex/strex – atomic load/store (armv6)
- pli etc – preload instructions

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

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 VFP (vector floating point) support (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.

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