ECE 471 – Embedded Systems Lecture 9

Vince Weaver

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

vincent.weaver@maine.edu

23 September 2024

Announcements

- HW#3 was assigned, sorry for the delay
- Most of the issue was making sure you can do it on 64-bit operating system

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 r7
        swi     0x0                     // and exit
```

Some GNU assembler notes

- Code comments
  - @ is the traditional comment character
  - # can be used on line by itself but will confuse assembler if on line with code.
  - Can also use /* */ and //
  - *Cannot* use ;
- Instruction opcode operand order is destination, source
- Constant value indicated by # or $
- .equ is equivalent to a C #define

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
```

Let's look at our executable

- `ls -la ./hello_exit_arm`
  Check the size
- `strip ./hello_exit_arm`
  Strip off the debugging info (makes smaller)
- `hexdump -C ./hello_exit_arm`
  See the raw binary (well, hex) values
- `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.
- Data can be declared with .ascii, .word, .byte
- BSS can be declared with .lcomm

Using gdb with `hello_world`

- Run `gdb ./hello_world`
- Type *run* to run program, will exit normally
- Can set breakpoint `break exit`
- Can single-step
- Can *info regis* to see registers
- Cam *disassem* to see disassembly

simple loop example

```
        # for(i=0;i<10;i++) do_something();

        mov     r0,#0           # set loop index to zero
loop:
        push    {r0}            # save r0 on stack
        bl      do_something    # branch to subroutine, saving
                                # return address in link register
        pop     {r0}            # restore r0 from stack

        add     r0,r0,#1        # increment loop counter
        cmp     r0,#10          # have we reached 10 yet?
        bne     loop            # if not, loop
```

string count example

Count the number of chars in a string until we hit a space.

```
        ldr     r1,=hello       # load pointer to hello string into r1
        mov     r2,#0           # initialize count to zero
loop:
        ldrb    r0,[r1]         # load byte pointed by r1 into r0
        cmp     r0,#' '         # compare r0 to space character
                                # this updates the status flags
        beq     done            # if it was equal, we are done
        add     r2,r2,#1        # increment our count
        add     r1,r1,#1        # increment our pointer
        b       loop            # branch (unconditionally) to loop
done:
```

HW3 Notes – Getting Weird Errors

- If the code won't assemble with errors about comment char it's often because you are compiling on a non-ARM32 machine
- This will happen if you run "make" on x86
- If on 64-bit system be sure you use the 64-bit code. I provide pre-disassembled files so you can still do the homework on 64-bit without a cross-compiler installed

HW3 Notes – Printing an Integer

- Writing int to string conversion is a complex task
  There are lots of ways to do it.
- When would you ever need code like this?
  In extreme embedded systems cases you might not have a printf() but still want to debug

HW3 Notes – Integer to String Algorithm

- Take integer
- 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')

HW3 Notes – ASCII

- American Standard Code for Information Interchange
- Old (late 1960s) standard for mapping text characters to numbers
- 7-bits (top bit either 0 or used for other purposes)
- Below 32 are control chars (like linefeed)
- 32 is space
- 48-57 is 0-1
- 65-90 is A-Z, 97-122 is a-z (bit 5 flipped)

HW3 Notes – Unicode

- what about other languages?
- Unicode, in theory 32 bit should hold all possible
- Windows and Java used 16-bit chars, but turned out not to be enough
- UTF-8 is interesting hack where bottom 127 chars map to ASCII, but when top bit set starts a complicated escape sequence that allows encoding any unicode value in 1 to 5 bytes
- still gives benefit to American English

HW3 Notes – Division if 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 `umull` instruction that will do a 32x32 multiply and give you the top half of the 64-bit result.

HW3 Notes – Corner cases?

- Leading zero removal
- Signed numbers (put a '-' in front?)

Really Brief Overview of ARM32 Assembly

- There's an Appendix at the end of these notes which covers ARM32 Assembly in more detail
- You have memory, registers, ALU, Program Counter, and flags (Negative, Zero, Carry, oVerflow): how do you turn this into a functioning program?

Why code in Assembly?

- Small binaries
  still useful on small embedded boards
- Optimal performance
  still good all systems, but be careful as newer chips might change the optimization parameters

ARM32 encoding

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

Data Processing (bits 27–26) | **Immediate** (bit 25) | **ADD opcode** (bits 23–21)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
cond				**0**	**0**	**0**	**0**	**1**	**0**	**0**	**S**	**Rn**			
								Opcode							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				**Shift imm5**					**Shift typ**		**Sh Reg**	**Rm**			

Immediate value (if immediate) (bits 11–0)

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 vs zero extend (lsrsb)
- Complex addressing modes. register, r1+r2, r1+r2+offset, auto-increment, etc

ARM32 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

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
  `ldmia sp!, {r0, r1, r2, r3, ip, pc}^`
- 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
```

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*value).

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

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)

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)