ECE 471 – Embedded Systems
Lecture 6

Vince Weaver

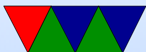
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

15 September 2016

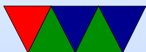
Announcements

- HW#3 will be posted today



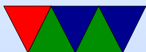
What the OS gives you at start

- Registers
- Instruction pointer at beginning
- Stack
- command line arguments, aux, environment variables
- Large contiguous VM space

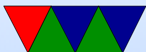


ARM Architecture

- 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)

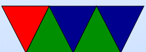


- Conditional execution
- Complicated addressing modes
- Many features optional (FPU [except in newer], PMU, Vector instructions, Java instructions, etc.)

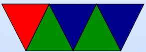


Registers

- Has 16 GP registers (more available in supervisor mode)
- r0 - r12 are general purpose
- r11 is sometimes the frame pointer (fp) [iOS uses r7]
- 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)

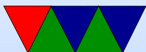


Low-Level ARM Linux Assembly

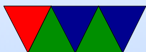


Linux C (ABI)

- Application Binary Interface
- The rules an executable needs to follow in order to talk to other code/libraries on the system
- A software agreement, this is not enforced at all by hardware
- r0-r3 are first 4 arguments/scratch (extra go on stack) (caller saved)

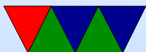


- r0-r1 are return value
- r4-r11 are general purpose, callee saved
- r12-r15 are special
- Things are more complex than this. Passing arrays and structs? 64-bit values? Floating point values? etc.



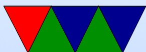
Kernel Programming ABIs

- 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 ARMv7 and VFP (vector floating point) support (armhf). Raspberry Pi (raspbian) is compiled for ARMv6 armhf.



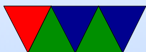
System Calls (EABI)

- System call number in r7
- Arguments in r0 - r6
- Return value in r0 (-1 if error, errno in -4096 - 0)
- 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.

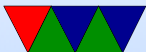


Manpage

The easiest place to get system call documentation.

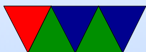
```
man open 2
```

Finds the documentation for "open". The 2 means look for system call documentation (which is type 2).



A first ARM assembly program: `hello_exit`

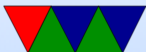
```
.equ SYSCALL_EXIT,     1

        .globl _start
_start:

        #================================
        # Exit
        #================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number (1) in r7
        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
```

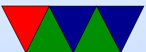


Assembly

- @ 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 $



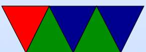
Let's look at our executable

- `ls -la ./hello_exit_arm`
  Check the size

- `readelf -a ./hello_exit_arm`
  Look at the ELF executable layout

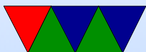
- `objdump --disassemble-all ./hello_exit_arm`
  See the machine code we generated

- `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"
```

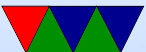


New things to note in `hello_world`

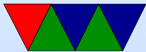
- The fixed-length 32-bit ARM cannot hold a full 32-bit immediate

- 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.



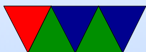
ARM Assembly Review



Arithmetic Instructions

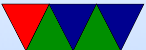
Operate on 32-bit integers. Most of these take optional s to set status flag

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



Register Manipulation

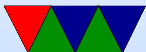
mov, movs	v1	move register
mvn, mvns	v1	move inverted



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`

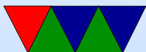
- You can specify you want the assembler to try to make the immediate for you: `ldr r0,=0xff`
  `ldr r0,=label`
  If it can't make the immediate value, it will store in nearby in a `literal pool` and do a memory read.



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)

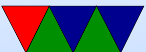


- Why no ASL?
- For example:

```
add r1, r2, r3, lsr #4
r1 = r2 + (r3>>4)
```

- Another example (what does this do):

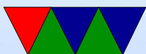
```
add r1, r2, r2, lsl #2
```



Shift Instructions

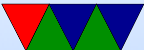
Pseudo operations. Implemented via `mov` with shift on arm32.

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



Logic Instructions

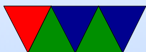
and	v1	bitwise and
bfc	??	bitfield clear, clear bits in reg
bfi	??	bitfield insert
bic	v1	bitfield clear: and with negated value
clz	v7	count leading zeros
eor	v1	exclusive or (name shows 6502 heritage)
orn	v6	or not
orr	v1	bitwise or



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)

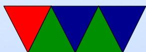


Control-Flow Instructions

Can use all of the condition code prefixes.

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

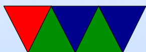
b	v1	branch
bl	v1	branch and link (return value stored in lr)
bx	v4t	branch to offset or reg, possible THUMB switch
blx	v5	branch and link to register, with possible THUMB switch
mov pc,lr	v1	return from a link



Prefixed instructions

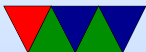
Most instructions can be prefixed with condition codes:

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

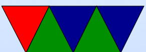
- `add r1,r2,r3`
- `adds r1,r2,r3` – set condition flag
- `addeqs r1,r2,r3` – set condition flag and prefix compiler and disassembler like `addseq`, GNU as doesn't?



Conditional Execution

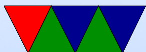
```
if (x == 1 )
    a+=2;
else
    b-=2;
```

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



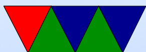
Load/Store Instructions

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

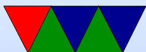


Addressing Modes

- `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



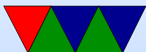
- `ldrb r1, [r2, #20]!` @ pre-index. Load from r2+20 then write back
- `ldrb r1, [r2, r3]!` @ pre-index. register
- `ldrb r1, [r2, r3, LSL #4]!` @ pre-index. shift
- `ldrb r1, [r2],#+1` @ post-index. load, then add value to r2
- `ldrb r1, [r2],r3` @ post-index register
- `ldrb r1, [r2],r3, LSL #4` @ post-index shift



Why some of these?

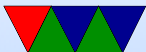
- `ldrb r1, [r2,#20]` @ register/offset
  Useful for structs in C (i.e. something.else=4;)

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



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 *does* have THUMB2
- THUMB-EE – some extensions for running in JIT runtime
- AARCH64 – 64 bit. Relatively new.

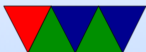


Recall the ARM32 encoding

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

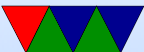
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
cond				0	0	0	0	1 Opcode	0	0	S	Rn			

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

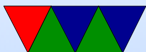


THUMB

- 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)



Calling THUMB

- We'll get more into this next time, but for the HW just know that to call into a THUMB function you need to use `blx` rather than just the plain `bl` (branch and link) instruction