ECE 498 – Linux Assembly Language Lecture 4

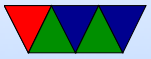
Vince Weaver

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

vincent.weaver@maine.edu

27 November 2012

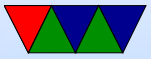
The ARM Architecture



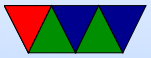
Models

Architecture vs Family

- ARMv1 : ARM1
- ARMv2 : ARM2, ARM3 (26-bit, status in PC register)
- ARMv3 : ARM6, ARM7
- ARMv4 : StrongARM
- ARMv5 : XScale

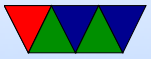


- ARMv6 : ARM11, ARM Cortex-M0 (Raspberry Pi)
- ARMv7 : Cortex A8,A9,Cortex-M3 (iPad, iPhone, Pandaboard)
- ARMv8 : Cortex A-50 (64-bit)

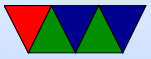


ARM Architecture

- 32-bit
- Load/Store
- Can be Big-Endian or Little-Endian
- Fixed instruction width (32-bit, 16-bit THUMB)
- Opcodes take three arguments
- Cannot access unaligned memory (optional newer chips)

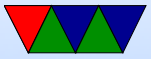


- Conditional execution
- Status flag (many instructions can optionally set)
- Complicated addressing mode
- Most features optional (FPU [except in newer], PMU, Vector instructions, Java instructions, etc.)

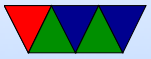


Registers

- Has 16 GP registers (more available in supervisor mode)
- r0 - r12 are general purpose
- r13 is stack pointer (sp)
- r14 is link register (lr)
- r15 is program counter (pc) (reading r15 usually gives PC+8)

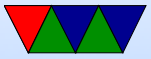


- 1 status register (more in system mode).
  NZCVQ (Negative, Zero, Carry, oVerflow, Saturate)



Orthogonality of PC

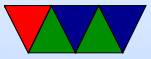
- Can branch with a “move” instruction
- Can do PC relative addressing easily
- Neat feature but complicates modern chip design



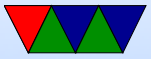
Prefixed instructions

Most instructions can be prefixed with condition codes:

- EQ, NE (equal)
- MI, PL (minus/plus)
- HI, LS (unsigned higher/lower)
- GE, LT (greaterequal/lessthan)
- GT, LE (graeterthan, lessthan)

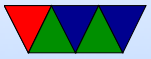


- CS, CC (carry set/clear)
- VS, VC (overflow set / clear)
- AL (always)



Setting Flags

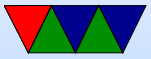
- `add r1,r2,r3`
- `adds r1,r2,r3` – set condition flag
- `addeqs r1,r2,r3` – set condition flag and prefix compiler and disassembler like `addseq`, GNU as doesn't?



Conditional Execution

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

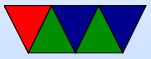
```
cmp     r1, #5
addeq   r2,r2,#2
subne   r3,r3,#2
```



Extra Shift in ALU instructions

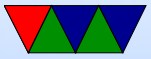
If second source is a register, can optionally shift:

- LSL – Logical shift left
- LSR – Logical shift right
- ASR – Arithmetic shift right
- ROR – Rotate Right
- RRX – Rotate Right with Extend



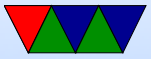
- For example:

```
add r1, r2, r3, lsl #4
r1 = r2 + (r3<<4)
```

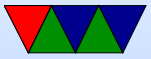


Addressing Modes

- `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

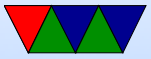


- `ldrb r1, [r2, #20]!` @ pre-index. Load from r2+20 then write back
- `ldrb r1, [r2, r3]!` @ pre-index. register
- `ldrb r1, [r2, r3, LSL #4]!` @ pre-index. shift
- `ldrb r1, [r2],#+1` @ post-index. load, then add value to r2
- `ldrb r1, [r2],r3` @ post-index register
- `ldrb r1, [r2],r3, LSL #4` @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



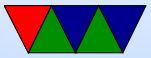
ABIs

- 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)



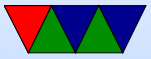
System Calls (EABI)

- 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.



A first ARM assembly program: hello_exit

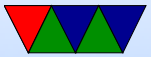
```
.equ SYSCALL_EXIT,     1

        .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

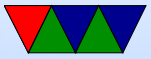
Assembling/Linking using `make`, running, and checking the output.

```
lecture4$ make hello_exit_arm
as -o hello_exit_arm.o hello_exit_arm.s
ld -o hello_exit_arm hello_exit_arm.o
lecture4$ ./hello_exit_arm
lecture4$ echo $?
5
```



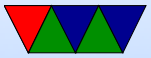
Assembly

- @ 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 $



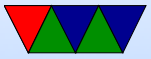
Let's look at our executable

- `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.