hbr

hint for Branch (r-form)

Month: February 2011

Assembly Primer Part 4 — Hello World — ARM

On to Assembly Primer — Part 4. This is where we start writing a small assembly program for the platform. In this case, I don't know the language and I don't know the ABI. Learning these from scratch ranges from interesting to tedious :)

Regarding the language (available instructions, mnemonics and assembly syntax): I'm using the ARM Architecture Reference Manual as my reference for the architecture (odd, I know). It's very long and the documentation for each instruction is extensive — which is good because there are a lot of instructions, and many of them do a lot of things at once.

Regarding the ABI (particularly things like argument passing, return values and system calls): there's the Procedure Call Standard for the ARM Architecture, and there are a few other references I've found, such as the Debian ARM EABI Port wiki page.

> *"EABI is the new "Embedded" ABI by ARM ltd. EABI is actually a family of ABI's and one of the "subABIs" is GNU EABI, for Linux."*
>
> *– from Debian ARM EABI Port*

System Calls

To perform a system call using the GNU EABI:

- put the system call number in r7
- put the arguments in r0-r6 (64bit arguments must be aligned to an even numbered register i.e. in r0+r1, r2+r3, or r4+r5)
- issue the **Supervisor Call** instruction with a zero operand — **svc #0**

(**Supervisor Call** was previously named **Software Interrupt** — **swi**)

Just Exit

Based on the above, it's not difficult to reimplement **JustExit.s** (original) for ARM.

```
.text

.globl _start

_start:
        mov r7, #1
        mov r0, #0
        svc #0
```

mov here is **Move (Immediate)** which puts the **#**-prefixed literal into the named register.

Hello World

Likewise, the conversion of **HelloWorldProgram.s** (original) is not difficult:

```
.data

HelloWorldString:
      .ascii "Hello World\n"

.text
```

```
.globl _start

_start:
      # Load all the arguments for write ()

      mov r7, #4
      mov r0, #1
      ldr r1,=HelloWorldString
      mov r2, #12
      svc #0

      # Need to exit the program

      mov r7, #1
      mov r0, #0
      svc #0
```

This includes the **load register** pseudo-instruction, **ldr** — the compiler stores the address of **HelloWorldString** into the literal pool, a portion of memory located in the program text, and the 32bit address is loaded from the literal pool (more details).

When compiling a similar C program with **-mcpu=cortex-a8**, I notice that the compiler generates **Move (immediate)** and **Move Top** — **movw** and **movt** — instructions to load the address directly from the instruction stream, which is presumably more efficient on that architecture.

February 14, 2011 / general, programming / ARM, Assembly / 3 Comments

Assembly Primer Parts 1, 2 and 3 — ARM

I had started a series of posts on assembly programming for the Cell BE PPU and SPU, based on the assembly primer video series from securitytube.net. I have recently acquired a Nokia N900, and so thought I might take the opportunity to continue the series with a look at the ARM processor as well.

Wikipedia lists the N900's processor as a Texas Instruments OMAP3430, 600MHz ARMv7 Cortex-A8. I'm not at all familiar with the processor family, so I'll be attempting to find out what all of this means as I go :P

I've set up a cross compiler on my desktop machine using Gentoo's neat crossdev tool (built using **crossdev -t arm-linux-gnueabi**). The toolchain builds a functional Hello, World!

(I note that scratchbox appears to be the standard tool/environment used to build apps for Maemo — I may take a closer look at that at a later date)

I have whatever the latest public 'stable' Maemo 5 release is on the N900 (PR 1.3, I think), with an **apt-get install openssh gdb** — thus far, enough to "debug" a functional Hello, World!

What follows are some details of the Cortex-A8 architecture present in the N900, particularly in how it differs from IA32, as presented in the videos Part 1 — System Organisation, Part 2 — Virtual Memory Organization and Part 3 — GDB Usage Primer. I've packed them all into this post because gdb usage and Linux system usage are largely the same on ARM as they are on PPC and IA32.

Most of the following information comes from the ARM Architecture Reference Manual.

(The number of possible configurations of ARM hardware makes it interesting at times to work out exactly which features are present in my particular processor. From what I can tell, the N900's Cortex-A8 is ARMv7-A and includes VFPv3 half, single and double precision float support, and NEON (aka Advanced SIMD). I expect I'll find out more when I actually start to try and program the thing. As to which gcc -march, -mcpu or -mfpu options are most correct for the N900 — I have no idea.)

1. Registers

Integer

There are sixteen 32bit ARM core registers, R0 to R15, where R0–R12 are for general use. R13 contains the stack pointer (SP), R14 is the link register (LR), and R15 is the program counter (PC).

The current program status register (CSPR) contains various status and control bits.

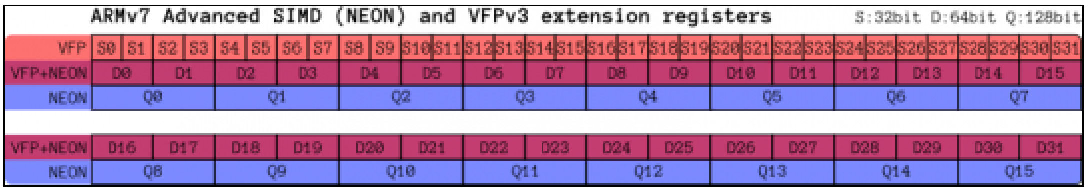
VFPv3 (Floating point) & NEON (Advanced SIMD)

There are thrirty two doubleword (64bit) registers, that can be referenced in a number of ways.

NEON instructions can access these as thirty two doubleword registers (D0–D31) or as sixteen quadword registers (Q0–Q15), able to be used interchangeably.

VFP instructions can view the same registers as 32 doubleword registers (again, D0–D31) or as 32 single word registers (S0–S31). The single word view is packed into the first 16 doubleword registers.

Something like this pic (click to embiggen):


ARMv7 Advanced SIMD (NEON) and VFPv3 extension registers
S:32bit D:64bit Q:128bit
VFP
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20 S21 S22 S23 S24 S25 S26 S27 S28 S29 S30 S31
VFP+NEON
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15
NEON
Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7
VFP+NEON
D16 D17 D18 D19 D20 D21 D22 D23 D24 D25 D26 D27 D28 D29 D30 D31
NEON
Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15

VFP in this core (apparently) supports single and double precision floating point data types and arithmetic, as well as half precision (possibly in two different formats…).

NEON instructions support accessing values in extension registers as

- 8, 16, 32 or 64bit integer, signed or unsigned,
- 16 or 32bit floating point values, and
- 8 or 16bit polynomial values.

There's also a floating point status and control register (FPSCR).

2. Virtual Memory Organisation

On this platform, program text appears to be loaded at **0x8000**.

After an **echo 0 > /proc/sys/kernel/randomize_va_space**, the top of the stack appears to be **0xbf000000**.

3. SimpleDemo

Compared to the video, there are only a couple of small differences when running SimpleDemo in gdb on ARM.

Obviously, the disassembly is not the same as for IA32. Rather than the **call** instructions noted in the video, you'll see **bl** (Branch with Link) for the various functions called.

Where the return address is stored on the stack for IA32, the link register (**lr** in **info registers** output) stores the return address for the current function, although **lr** will be pushed to the stack before another function is called.

(From a cursory googling, it seems that to correctly displaying all VFP/NEON registers requires gdb-7.2 — I'm running the 6.8-based build from the maemo repo. crossdev will build me a gdb I can run on my desktop PC — **crossdev -t arm-linux-gnueabi –ex-gdb** — but I believe I still need to build a newer **gdbserver** to run on the N900.)

Other assembly primer notes are linked here.

February 12, 2011 / general, programming / ARM, Assembly / Leave a comment

hbr / Proudly powered by WordPress