Contents

1 Introduction

Intro

This paper covers the new arm architecture ABI (EABI), explaining what it is, what is new and different about it compared to the existing arm ABI, and why that matters.

It also covers Debian's response to this: the armel architecture, the current status of that port, and possible futures for it.

EABI

OK, so first let us be clear what we are talking about. The ABI is all about the *binary* interface, not the programming interface (API). So we are coving how values are aligned on the stack in a function call, not what the parameters to that call are.

The main concern of the ABI is the C function calling convention, the size and alignment of C types, and FP instructions. Lets deal with C function calling first.

An important thing to understand is that every library that is called by a program must be using the same ABI as the program otherwise the function calls will probably explode messily. In practice this means that you cannot mix ABIs on a system. (It is theoretically possible but you need to be really sure which program is going to call which library so that library function calls always match up).

The net result of this is that the old ABI is not comptible with the new EABI so the whole system needs to be using one or the other - this is an incompatible change.

Whilst we were breaking everything like like this, it seemed like a good time to change the kernel syscall mechanism too, and get all the pain over in one go.

Finally, the E in EABI stnads for 'Embedded'. This doesn't really mean much, except to indicate that arm is generally used in an embedded context. So far as we are concerned it means 'New' - there isn't a non-embdedded ABI to worry about.

History

First a bit of historical context.

The Arm kernel port was done in 1998. It used GCC's C calling convention for ARM. This was designed to pass 5 or more arguments efficiently by using registers. This is the same as the RISCOS conventions, except that it doesn't use the condition codes to indicate errors/return status.

Floating point was done with the standard FP instruction set of the time. These are exectued by the FPU if present, or emulated if it is not.

Supported machines

Debian-arm port started in 2000

- Netwinder: 2000
- RiscPC, Cats: 2001
- Lart, Bast: 2003
- Iyonix, Manga: 2004
- NSLU2, Thecus: 2005/6

There are numerous other machines on which Debian-arm is used, but which do not have debian-installer support and so are not officially supported machines.

2 EABI changes

Floating Point

Unfortunately, in the real world, ARM FPUs have turned out to be like hen's teeth. There were a couple back in the mid-1990s, then none for best part of a decade until recent devices, but all those in fact have different instruction sets from the original ARM FPU.

- CPUs with FPU:
  - ARM FPA11, VLSI 7500FE
- CPUs without FPU:
  - Cirrus Logic 711x, 720T, 72xx, 73xx
  - Dec/Intel SA110, 1100, 1110
  - Intel PXA250, 255, 270, IXP4xx, ixp2000
  - Samsung chips
  - Atmel AT91xx ...
- New FPUs with different instruction sets:
  - Cirrus EP93xx has Maverick crunch
  - Intel IWMMXT
  - TI Omap 2420 (Arm11 - VFP)
  - Philips lpc3180 (arm9)

FP solutions

Because almost all arm systems do not have a real FPU, various forms of emulation are needed instead. Originally there was only a binary module from Acorn Computers Ltd, then the equivalent Netwinder FP Emulator which replaced it with free software, and later a reduced-accuracy/increased speed version (FastFPE).

The FPU instruction emulation mechanism is very inefficient because each FP instruction is executed, causing an illegal instruction abort, which is trapped by the kernel and then delivered to the emulation module, with the result being returned as if it came from a real FPU. All of this has hundreds of instructions of overhead for each emulated instruction.

A much better solution is GCC's softfloat which short-circuits the trapping/return part of the above and just puts the instructions which would be executed to actually do the calculations into the program at compile time. This is much more efficient.

- Runtime Emulators ('hard-float'):
  - Acorn FPE (binary module)
  - NWFPE, FastFPE

- Compile-time functions ('soft-float')
    - GCC softfloat

However the two mechanisms have different calling conventions and so cannot be mixed in a system - everything has to be using soft-float or everything has to be using hard-float.

Real FPU/emulation: Calls use r0-r3 for 1st 4 arguments, stack for the rest. floats can fill multiple registers, and be split across registers and stack. Return value is put in Coprocessor register f0.

With softfloat, the return value is put in r0-r2 (depending on size)

Debian-arm uses hard-float, because it pre-dates the soft-float concept.

FP formats

An interesting wrinkle on arm is the format used to represent doubles. It is big endian within each word, but the two words are arranged little-endian (on little endian arms). This complies with IEEE754 on FP formats, but is not used by any other arch and thus can reasonably be deemed 'weird'.

Pi in double format looks like this:

```
x86:   18 2d 44 54 fb 21 09 40 (little endian)
Sparc: 40 09 21 fb 54 44 2d 18 (big endian)
arm:   fb 21 09 40 18 2d 44 54 (mixed endian)
```

(Arms running in big-endian mode have big-endian word arrangement as well as being big-endian within the words, so are the same layout as other big-endian devices.)

Debian-arm is little-endian, because that was normal when it was created and in fact most arm CPUs are still run in little-endian mode.

This has caused problems in Mozilla, Perl, Gnumeric, maths libraries and more: anything that tries to manipulate floats on its own rather than using glibc.

This 'feature' goes away with EABI.

Why do we care?

So, what are the pros and cons of this new ABI?

Pros

- Standardisation across toolchains, debuggers
- Most arm wierdness removed (FP formats, packing, C++ exceptions)
- Hard/soft float interworking (soft-float *much* faster)
- Thumb interworking
- Interchangeable binaries (PalmOS, GNU/Linux, Symbian OS)
- More efficient syscall convention

Cons

- Almost total incompatibility with existing port

One feature of the new ABI is that it has been designed to be used across numerous OSes, so the same code can run on SymbianOS and Linux. This is really only useful to proprietary code vendors, but is nevertheless pleasing standardisation, which makes code analysis tools more widely applicable, as does the use of standards, such as the DWARF3 debug format.

So there are lots of good reasons why it is technically better, but the fact that it is incompatible with the last decade of arm work is a big deal.

New kernel syscall convention

An example is the easiest way to show the differnce between the legacy ABI and EABI kernel syscall mechanisms:

Example: long ftruncate64(unsigned int fd, loff_t length):

- legacy ABI:
  - - put fd into r0
  - - put length into r1-r2
  - - use "`swi #(0x900000 + 194)`" to call the kernel
- EABI:
  - - put fd into r0
  - - put length into r2-r3 (skipping over r1)
  - - put 194 into r7
  - - use "`swi 0`" to call the kernel

Originally ARMs had a von Neumman arch (combined data and instruction cache) so having the syscall encoded in the SWI instruction made sense (already in cache, saves store/load).

In a Harvard arch (separate data/instruction cache,as found on modern arm CPUs) having the syscall number in the SWI instruction is wasteful (pollutes data cache) and the extra register store/load is much more efficient.

The syscall mechanism was changed in kernel 2.6.15, and reached mainline in 2.6.16. A compatibility mechanism for the old-style calls is in the kernel so that both EABI and legacy programs can be run by one kernel.

The speed gain that can be realised by the new mechanism is only actually realised if the compatibilty mechanism is disabled.

There was a transitional period represented by glibc 2.3.6 and the 2005q3 Codesourcery toolchain which used shims in glibc to allow old syscalls with new EABI. This was removed in glibc 2.3.7/2.4 and later.

EABI genesis

The EABI was created by taking existing external standards wherever possible, merging this with some ARM Corp. internal standards, and a few things had to be defined specially.

- external stuff:
  - ELF
  - DWARF-3

  - generic C++ ABI

- internal stuff:
  - Procedure call standard (AAPCS = simplified, clarified ATPCS)
  - ELF processor supplement
  - anticipates thumb-2 and arm v6 (new-style BE8)

- plus new stuff:
  - C++ exception handling, Clibrary, run-time helper functions

Timeline

- New ABI published Dec 2003
- Code sourcery 1st cross-tools q3 2005 GCC v3.4.4
- 2005: Early Linux adopters (montavista, nokia) - shimmed glibc
- Kernel syscalls changed during 2.6.15
- Debian port started q1 2006 - all new
- Aleph One and Code sourcery gcc4.1 cross-tools q1 2006
- Angstrom OE EABI Aug 2006
- ADS/Lennert Buytenhek working port Jan 2007 (v4t build)
- DD-signed (Riku Voipio) buildd announced April 2007 (v4t build)

Main EABI changes
These are the main changes that the EABI causes relative to the legacy ABI. Primarily alignment/packing is now natural making arm a more 'conventional' architecture.

- Structure packing
  - Old ABI had min structure packing size of 4 bytes
  - EABI has no minimum - packing is determined by type sizes

- Argument alignment
  - 8-byte stack alignment at public function entry points (Old ABI was 4-bytes)
  - 64-bit data types (e.g. long long) are 8-byte aligned (Old ABI was 4-bytes)

- Enums
  - EABI allows enums to have variable type size (-mabi=aapcs)
  - Not used on GNU/Linux - they remain as 4-bytes. (-mabi=aapcs-linux)

- Floating point
  - Mixed-endian LE format goes away
  - Can mix GCC softfloat and FPU hardfloat/emulation

Tools

The toolchain work has been done by Codesourcery on behalf of ARM Corp. EABI support was originally added in gcc3.4.4 with the -mabi=aapcs-linux option.

From gcc 4.1 the mode was given a new gnu triplet: linux-arm-gnueabi. The legacy ABI is is linux-arm-gnu. i.e. gcc treats it as a whole new architecture, which technically, it is.

Glibc had shims added in 2.3.6 and new syscalls in 2.3.7/2.4. In fact 2.4 has serious shortcomings on arm, so glibc 2.5 is really needed.

Kernel has new syscalls added in 2.6.16

Crosstool from v0.42 has the necessary juju to build working crosstools with EABI support.

3 EABI and Debian-arm

Debian port

The debian arm port has been working on incorporating these changes into Debian. We could just ignore them and carry on with our existing port, but it is clear that there are significant technical advantages to changing, and it seems likely that the legacy arm ABI will eventually become obsolete.

The main advantages are that we get:-

- Much better FP performance by default (approx 20 times better)
- The ability to mix hard-fp and soft-float on the same system
- More conventional packing rules so stuff that never built probably will now

There are other less signifcant reasons such as binary compatibility with other OSes, wider debug tool support, and the aforementioned avoiding obsolesence.

The main *disadvantage* is the fact that the new port is completely incompatible with the old one.

So - How do we make the change? There are a number of possible approaches.

Rename all library packages

Pros

- Can do apt-get dist-upgrade

Cons

- Every single library package needs to be renamed
- Will take a long time, during which unstable will be broken for all arches (6months for C++, so 2yrs?)
- Not popular due to large hassle for other arches
- Will lose v3, may lose v4 support.

New architecture

Pros

- Fits with gcc approach
- Does not affect non-arm arches
- Can keep 'arm' for v3 and maybe v4 machines
- Can be done relatively quickly as no interaction with other arches/releases

Cons

- Current arm users don't have easy upgrade path
- Need archive space for new arch

ABI: field in control file

Suggested as part of multiarch proposal

Pros

- Reflects ABI correctly, would help other transitions too

Cons

- No existing implementation
- No consensus on including it yet
- Questions over resolving dependencies and how it fits into archive

Conflicting libc packages

Make a libc6-eabi-dev depending on eabi and ld-linux.so.3, that conflicts with libc6.

Pros

- Only have to change glibc (and rebuild everything)
- Does not affect other arches

Cons

- Most of port will be uninstallable for a very long time
- apt-get dist-upgrade still won't work due to huge number of conflicts
- Will lose v3, may lose v4 support

'New Arch' won

Largely because it could be done in an independent manner (not affecting the existing arm port, or the rest of Debian), and matched the toolchain view, the arm porters decided that a new architecture was the best way to go.

There was some debate what to call it; as the best name 'arm' had already been taken by the existing port. Mailing list discussion failed to acheive consensus, so when we had most of the people involved at the Embedded Debian work session in Extramadura in April 2005 we thrashed it out and decided to go with "armel", following the nomenclature arm themelves use for naming cpu types.

This allows for a corresponding 'armeb' big-endian port, although there was already an unoffical one called that which was using the legacy ABI, so whilst it is OK for Debian, there is an issue for existing users changing over.

CPU choice

The next significant issue is which CPUs to support with which port. This is a summary of the instruction set versions, and which machines use them.

- v3 (RiscPC)
- v4 (Strongarm)
- v4t (most arm 7)
- v5 (xscale, arm9, 10, 11)
- v6 coming soon - new BE8 mode

So all current arm CPUs are v4t or v5.

Thumb interworking

The big issue here is thumb interworking. Thumb is ARMs reduced-space instruction set. It has 16-bit instructions, rather than the standard 32-bit instructions, generally resulting in a 30% space saving.

The EABI provides for the facility to swap back and forth between thumb code on every function-call boundary. However to do this (on v4t) it uses the BX instruction to set the core to the correct state on call and return. This instruction is not present prior to v4t. This means that code built for EABI does not run on v4 processors.

Currently all debian's arm buildds are v4 machines (CATS, Netwinder). Although new build machines are likely to be installed soon as these are slow, and certainly the armel port will be using newer hardware. Nevertheless the machines are still reasonably well-used and it is not clear at what point support could/should be dropped. v3 machines such as RISCOS we can definately give up on at this stage - the remaining userbase is tiny.

Thumb is not actually used at all in Debian, and may not be in the future. We could simply drop the interworking support in order to be v4 compatible, but that would mean breaking the ABI rules, and it seems like a bad idea to remove the possibility of using thumb in Debian at all if we can help it.

It is fairly simple to modify the function call code so that it works with v4, by having a test for thumb capability to skip the BX:

```
tst lr, \#1
moveq pc, lr
bx lr
```

However this is an extra couple of instructions of overhead on every function call.

A patch to support this now exists, but the speed differential has not yet been tested.

Remaining Issues

The port is now working and useable, with about half the archive built at the time of writing (may 2007). Bugs and issues are being worked through.

We need to have enough buildds, and get them added to the buildd monitoring infrastructure so that people can find and fix problems easily.

Then clearly the target is to get enough packages built to be eligible for official port status. The ARM team intends armel to be a lenny release architecture.

A decision needs to be taken on supporting v4 or not in armel, and if it is to rebuilt stuff already built for v4t. This is currently waiting on toolchain testing.

Related is the issue of how long to support the legacy arm port - With v4 in armel is could be ditched after Lenny. (i.e lenny is the only release containing both arm and armel). If armel only supports v4t or higher then we may need to support arm and armel for longer in order to have something that v4 machines can run (how long before all the strongarm machines are dead? - they are just about ceasing to be sold around now).

Thanx for reading this far. See http://wiki.debian.org/ArmEabiPort for mode complete info.