ILP32 for AArch64 Whitepaper

Application Note 490

ARM®

ARM DAI 0490A (ID060915)

ILP32 for AArch64 Whitepaper
Application Note 490

Release Information

Table 1 Change history

Date	Issue	Confidentiality	Change
May 2015	A	Non-Confidential	First release

Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

1 What is ILP32?

The ARMv8 architecture supports 32-bit and 64-bit instruction sets (AArch32 and AArch64 respectively), both use 32-bit instruction encodings but with different register lengths and data type sizes.

ILP32 is intended for use where, for whatever reason, it is beneficial to use int, long and pointer represented as 32-bit values. This could be for performance or legacy 32-bit compatibility reasons.

Linux on AArch64 uses LP64 as its standard data model, where Long and Pointer are 64-bit, Ints are 32-bit.

AArch64-ILP32 uses the AArch64 instruction set coupled with a data model where Int, Long and Pointer are 32-bit.

Table 2

C/C++ base type	LP64 (Unix/Linux)	LLP64 (Windows)	ILP32
int, short, char	≤32	≤32	≤32
Long	64	32	32
void * (pointer)	64	64	32
long long	64	64	64
SP Float	32	32	32
DP Float	64	64	64

- 64-bit types are manipulated in 64-bit registers.
- Stack pushing and popping by 64-bit register.
- Data structures will contain 32-bit pointers and longs.

2 Why do we need ILP32?

There are potential performance differences when moving to a 64-bit ISA:

- 64-bit ISAs tend to have more registers.
- Clean up of historical 32-bit ISA features, resulting in a more efficient instruction set.
- 32-bit data models tend to have lower runtime memory requirements than 64-bit models. Primarily due to the smaller size of pointers within data structures, resulting in better utilization of CPU caches.

Not all legacy code can be easily ported to 64-bit (LP64). ILP32 enables a recompilation of 32-bit source code to a 64-bit ISA but using the 32-bit data model removing the need for the port and enabling the legacy code to run on a 64-bit only CPU.

AArch64 was designed to remove known implementation challenges of AArch32 cores, such as predicated execution, ordered Load Store multiples, complex privileged states, and deprecated functionality.

64-bit only cores are a reality in the ARMv8 ecosystem. There are some advantages to this type of design:

- Simplified pipeline.
- Significant reduction in core validation space (75% reduction).
- Better power/performance/area compared to a 32+64 equivalent core.

ILP32 is one of the solutions to 32-bit source legacy, and a legitimate alternative ABI to LP64 for AArch64.

3 Toolchain

The compiler changes required for GCC have been accepted upstream in gcc-4.9. A single set of compiler binaries either compile LP64 or ILP32 (using -mabi=ilp32) using the AArch64 instruction set.

The glibc code changes are currently under review.

4 Linux Kernel

Support for AArch64 and AArch32 is upstream and ready for use.

The AArch64 kernel runs, and can only run, in LP64 mode. ILP32 uses 32-bit longs and pointers meaning there is an ABI mismatch between the kernel and userspace that is similar, but not identical, to compat mode used for AArch32 userspace.

ILP32 uses the compat (32-bit) ELF loader that:

- Sets the ELF class to EM_AARCH64 (instead of EM_ARM).
- Sets the special thread flag to TIF_ILP32 to indicate this is the ILP32 ABI.
- Sets up the stack, memory limits to 4GB and VDSO (Virtual dynamically linked shared objects).
- Invokes the application code or dynamic loader.

ILP32 uses the LP64 system calls wherever possible. However, it cannot share system calls where structures contain pointers (currently 31 such system calls), in which case ILP32 uses the compat (32-bit) version.

5 What are the challenges?

The main challenge for wide adoption of ILP32 is the availability of userspace libraries. Linux distributions are likely to ship with support for AArch32, AArch64 or both in a multilib arrangement. The adoption of ILP32 requires supporting distributions to provide a further set of userspace binaries.

In the enterprise space it is likely that distributions are going to support only AArch64 to prevent the creation of an AArch32 binary legacy.The challenge for ILP32 is to persuade the distributions to add a further userspace build for ILP32. The advantages in the mobile space are less clear because we already have an AArch32 binary legacy.

6 Benchmark Data

The following tables and graph show the preliminary benchmark data courtesy of Cavium:

Table 3

	20150421T192	20150422T070	20150422T183	20150423T181619
benchmark	Linaro LP64	Linaro ILP32	Linaro AArch32	Linaro AArch32 #2
400.perlbench	12.3	12.3	12.7	12.7
401.bzip2	8.91	8.23	8.82	8.83
403.gcc	12	13.2	12.8	12.8
429.mcf	8.47	10.8	10.9	10.9
445.gobmk	13.7	13	13.8	13.7
456.hmmer	11.2	10.4	8.51	8.51
458.sjeng	13.5	12.1	13	13.1
462.libquantum	28	27.7	27.6	27.6
464.h264ref	20.7	21.8	21.6	21.7
471.omnetpp	9.24	10.6	10.7	10.7
473.astar	9.57	9.13	10.1	10.1
483.xalancbmk	11.9	12.6	13.4	13.4

Table 4

benchmark	Linaro ILP32/LP	Cavium ILP32/L	Linaro AArch32	Cavium AArch32/LP64
400.perlbench	0.00%	4.00%	3.25%	5.58%
401.bzip2	-7.63%	-12.00%	-1.01%	-9.28%
429.mcf	27.51%	28.00%	28.69%	15.32%
445.gobmk	-5.11%	-4.40%	0.73%	-5.63%
456.hmmer	-7.14%	-18.00%	-24.02%	-19.21%
458.sjeng	-10.37%	-0.22%	-3.70%	-7.01%
462.libquantum	-1.07%	19.00%	-1.43%	6.25%
464.h264ref	5.31%	15.00%	4.35%	-6.50%
471.omnetpp	14.72%	12.00%	15.80%	10.45%
473.astar	-4.60%	6.00%	5.54%	3.41%
483.xalancbmk	5.88%	6.10%	12.61%	4.33%

Raw data (if viewing in a PDF, copy and paste the URL into a web browser)
https://drive.google.com/a/linaro.org/folderview?id=0B7GHsMqIDrjIfklaNFRtNWtHeURJdnZMRmhTQlRMWHl1QUQ3cnVKdmlna01SM2dSaXRrNzQ&usp=sharing

Note

Linaro results were obtained by running on four Cortex®-A53 cores of Juno; we do not know on which cores Cavium results were obtained.

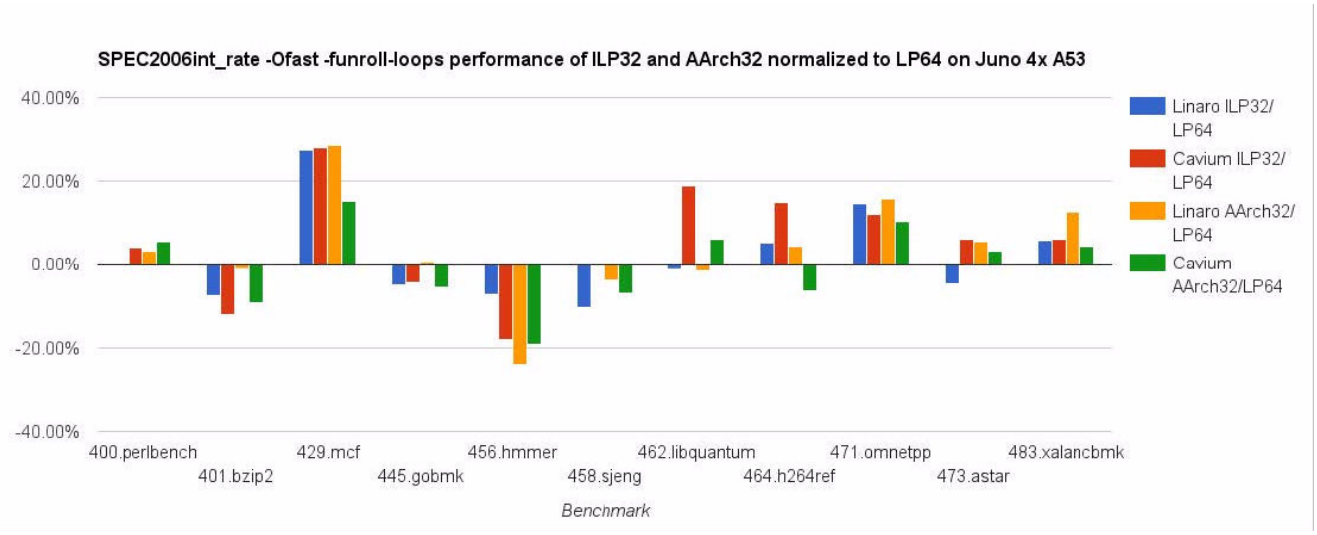

Figure 1 Graph of preliminary benchmark data

The following diagram and table shows the data collected for SPECINT and SPECINT-rate. Data is presented for a single instance of SPECINT and 48 instances of SPECINT (using SPECint2006).

A couple of things to note here:

- Overall the % improvement in SPECINT performance may seem modest, but if you look at individual components, we see benefits of up to 22%.
  - So there will be classes of applications that benefit enormously from ILP32.
  - Compiler optimization for ILP32 can potentially make this improvement greater as it becomes more mature.
- The benefit of ILP32 goes up with increasing number of cores as it helps reduce cache pressure.

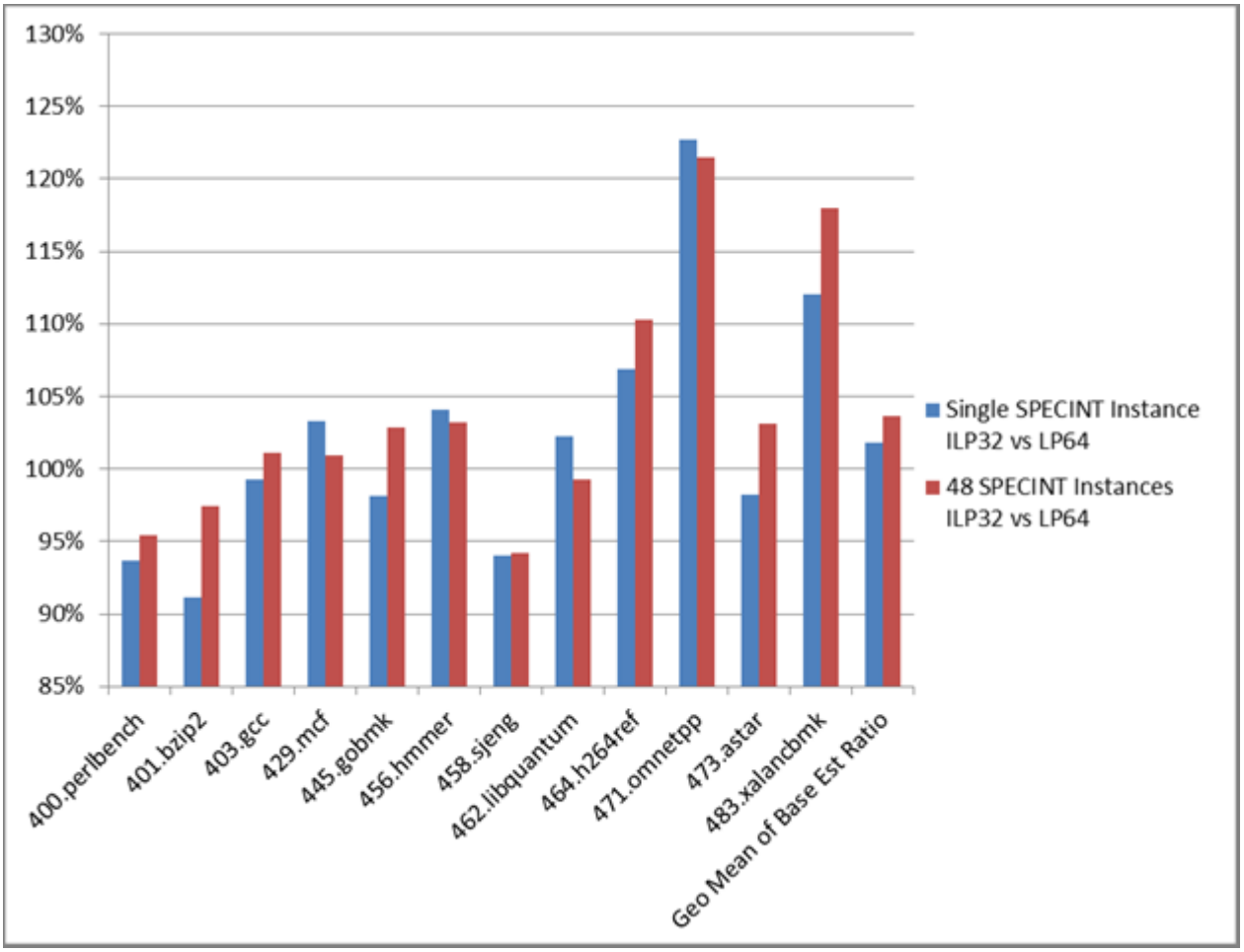

Figure 2 Graphic of benchmark data

LP64 is the baseline at 100%

Table 5 Benchmark data

Benchmark Component	Single SPECINT instance ILP32 vs LP64	48 SPECINT instances ILP32 vs LP64
400.perlbench	93.64%	95.45%
401.bzip2	91.13%	97.40%
403.gcc	99.29%	101.09%
429.mcf	103.30%	100.92%
445.gobmk	98.14%	102.81%
456.hmmer	104.06%	103.23%
458.sjeng	94.05%	94.21%
462.libquantum	102.25%	99.25%
464.h264ref	106.87%	**110.25%**
471.omnetpp	**122.72%**	**121.50%**

Table 5 Benchmark data (continued)

Benchmark Component	Single SPECINT instance ILP32 vs LP64	48 SPECINT instances ILP32 vs LP64
473.astar	98.18%	103.08%
483.xalancbmk	**112.03%**	**118.01%**
Geo Mean of Base Est Ratio	101.81%	103.63%

7 Documentation

Two documents have been prepared around ILP32. These documents are to be updated from Beta to published when the Linux Kernel changes and tools patches have been accepted upstream. Until now no significant issues have been found.

The documents are described in the following sections:

- *PCS - Procedure Calling Standard*.
- *ELF - Executable Linker Format* on page 14.

7.1 PCS - Procedure Calling Standard

Procedure Call Standard for the ARM 64-bit Architecture (AArch64)
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055c/IHI0055C_beta_aapcs64.pdf

This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the ARM 64-bit architecture.

- Defines fundamental types
- Function calling method
  - Argument marshaling
  - Register and Stack usage
- 32-bit Code and Data pointers
- No tagged pointers
- Stack contents are 64-bit

The text highlighted in yellow in the following figures shows all significant ILP32 changes.

C/C++ Type	Machine Type			Notes
	ILP32	LP64	LLP64	
`[signed] long`	signed word	signed double-word	signed word	
`unsigned long`	unsigned word	unsigned double-word	unsigned word	
`__fp16`	IEEE754-2008 half-precision format	IEEE754-2008 half-precision format	Alternative Format	**TBC:** LLP64 Alternate format?
`wchar_t`	unsigned word	unsigned word	unsigned halfword	
`T *`	32-bit data pointer	64-bit data pointer	64-bit data pointer	Any data type `T`
`T (*F)()`	32-bit code pointer	64-bit code pointer	64-bit code pointer	Any function type `F`
`T&`	32-bit data pointer	64-bit data pointer	64-bit data pointer	C++ reference

Table 4, C/C++ type variants by data model

Figure 3 C/C++ type variants by data model

Typedef	ILP32	LP64	LLP64
size_t	unsigned long	unsigned long	unsigned long long
ptrdiff_t	signed long	signed long	signed long long

Table 5, Additional data types

Figure 4 Additional data types

7.2 ELF - Executable Linker Format

ELF for the ARM 64-bit Architecture (AArch64) beta
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0056c/IHI0056C_beta_aaelf64.pdf

This document describes the use of the ELF binary file format in the Application Binary Interface (ABI) for the ARM 64-bit architecture.

- Relocations define code linkage
- Renumbered
  - 32-bit ELF uses 8-bit ranges
  - 64-bit ELF uses 16-bit ranges
  - AArch64 relocs started at 256
- Re-use of most AArch64 relocations
  - 93 ELF32 codes vs 131 for ELF64
  - No large or huge memory models
- Extra care at boundary conditions and large offsets.

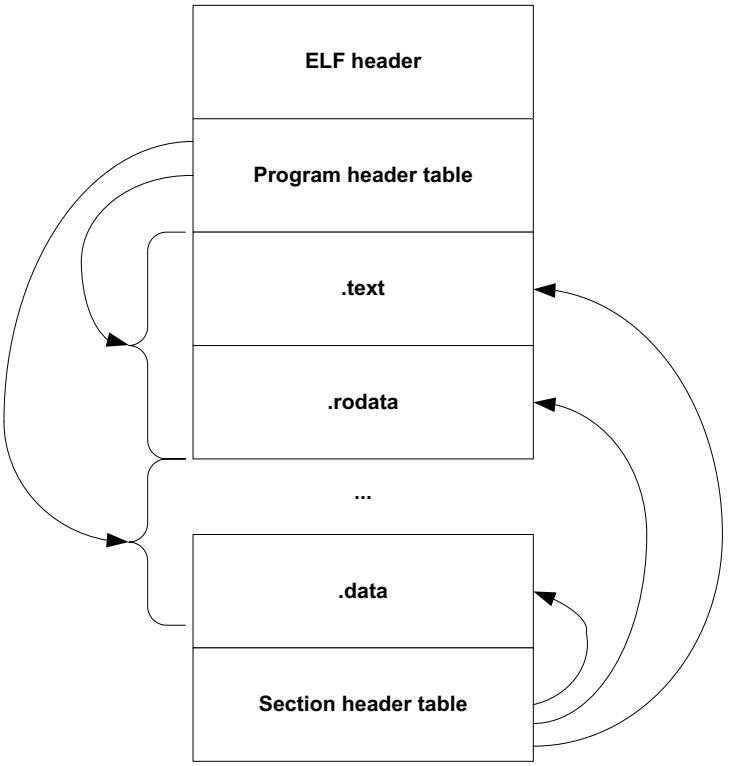

Figure 5 ELF file structure