PPC E500 SPE Support#

Reference#

The requirements for this feature are governed by SRM7058.018 SDP v6.4.1 shall support Freescale E500 SPE.

Abbreviations#

ABI
: Application Binary Interface

ACC
: Floating point Accumulator register

API
: Application Programming Interface

APU
: Auxilliary Processing Unit

EFP
: Embedded Floating Point

GPR
: General Purpose Register

SIMD
: Single Instruction, Multiple Data

SPE
: Signal Processing Engine

SPEFSCR
: SPE Floating point Status and Control Register

Background#

Broadly speaking, adding full support for the PPC E500v2 SPE and EFP APUs required the addition of two major features in QNX Neutrino 6.4.1. On the one hand, Freescale specified an ABI for the E500 SPE which is not backwardly compatible with the existing PPC ABI, which necessitated creating a new Neutrino variant which was built with a version of the toolchain which uses the new ABI. Additionally, since the E500v2 EFP APU does not provide full support for IEEE 754 in hardware, software support was needed to provide full compliance.

Design Review Status#

No general design review was performed for this work because:

- the build environment changes were the standard changes required for introducing any new variant
- the `fpassist' changes were a straightforward implementation based on the mechanisms used by the existing `fpemu' library
- the `libm' changes were PPC E500 SPE specific and involved a trivial redirection of the get/set requests through the `fpassist' library

- the procnto/kernel changes were PPC E500 SPE specific and were based on the existing PPC FPU exception handing code

---

Design#

ABI Variant#

The E500v2 SPE introduces three major changes to the standard BookE register set:

- All general purpose registers have 64 bits;
- A dedicated floating point status and control register, SPEFSCR, is added; and
- A dedicated floating point accumulator register, ACC, is added.

The unusual characteristic of the 64 bit GPRs is that all of the non-SPE/EFP instructions continue to behave as though the GPRs had only 32 bits: they never modify the upper 32 bits (the high word) of any register. Nor can the SPEFSRC or ACC be modified by any non-SPE/EFP instruction.

Support for managing the high register words, the ACC, and the SPEFSCR have already been added to the QNX Neutrino kernel. (The usual mechanism is used: the EFP is disabled and the EFP unavailable exception is used to arrange to swap this "auxiliary" register context.)

E500v2 SPE EFP operations fall into three broad categories:

- single precision vector;
- single precision scalar; and
- double precision scalar.

Within each of those categories, there are usually the following sub-categories:

- simple unary and binary arithmetic operations (e.g. neg, abs, add, sub, mul, div)
- compare and test operations (e.g. gt, eq, lt)
- conversion operations (both between floating formats and integer formats).

The single precision vector operations, are SIMD operations which allow a single precision operation to be performed on two sets of operands simultaneously. One set of operands is taken from the low word of the GPRs given as arguments to the opcode; the other set from the high words. Neither, either, or both operations may result in exceptions being raised (and separate SPEFSCR flags allow the handler to distinguish which).

The Freescale E500 ABI dictates that floating point arguments be passed in some contexts (notably varargs functions) in ways which are not compatible with the original PPC ABI. As a result, the existing QNX Neutrino ppcbe variant is not, in general, compatible with the E500 SPE or EFP instructions. This requires that the entire user mode portion of QNX Neutrino have a variant, ppcbe-spe, built using the E500 ABI. Note that since the kernel, startup, and other bootfiles do not make any generalized use of SPE or EFP instructions, there is no need for there to be a variant for them. (This means that the ppcbe-spe variant uses precisely the same procnto-booke kernels as the ppcbe, a fact which unfortunately often leads to deployment confusion.)

'fpassist' Library#

The fpassist library must support the same basic API as the fpemu (floating point emulation) library. The main entry point must be the function: `int _math_emulator(const int sigcode, void ** const pdata, PPC_CPU_REGISTERS * const regs);`

This function will be called in two different circumstances:

- as a result of the kernel handling EFP data or rounding exceptions (see procnto/kernel, below); and

- by the user application through the fenv and fp_status APIs (see libm, below).

What follows here is a brief discussion of the major data structures and algorithms required. For detailed documentation of the data structures and functions, see E500_SPE_support/fpassist-doxygen.tar.gz.

- **Exception Handling API and Protocol**

If the `sigcode` argument represents a `SIGFPE`, it is assumed that the kernel is handling an EFP data (`FPE_NOFPU`) or rounding (`FPE_FLTRES`) exception. The `pdata` argument will be a pointer to storage for the pointer to the thread's EFP context. The `regs` argument points to `PPC_CPU_REGISTERS` context reprenting the processor state st the time of the exception; it is assumed that the high words of the general purpose registers (GPRs) and the EFP accumulator are all unchanged since the point of the exception. (That the SPE/EFP registers are unchanged is guaranteed by the existing kernel mechanism for faulting in the auxiliary register context when an SPE or EFP instruction is issued.)

A thread initially has no EFP context associated with it: the `pdata` argument points to storage containing a `NULL` pointer. If this is the case, the first task of the fpassist library is to allocate a thread EFP context. For compatibility with `kerext_debug` and with the fpemu library, the thread context shall begin with an `PPC_FPU_REGISTER` structure followed by a `run_options` structure, all initialised with zeroes. (Neither of these areas is actually needed by the fpassist library but it facilitates interoperation with the debugger and with any future need to coexist with the fpemu library.) The context required by the fpassist library itself may follow these. The fpassist context must minimally contain enough storage to maintain a software version of the SPEFSCR.

The algorithm for handling the exception(s) is:

1. Retrieve the hardware SPEFSCR value at the point of exception from `regs`
2. Retrieve the hardware exception(s) and rounding mode from the SPEFSCR value
3. Examine the faulting instruction at the IAR in `regs`
4. Retry each operation which reported exceptions (up to two) using an IEEE 754 compliant software implementation, using the same rounding mode as the hardware
5. Update the software SPEFSCR sticky bits in the thread EFP context
    - For each operation which was retried (if any), update using the exceptions (if any) raised during the retry
    - If no operations were retried, update using the hardware sticky bits from the hardware SPEFSCR
6. Copy the sticky bits from the software SPEFSCR to the hardware SPEFSCR value in `regs`
7. Compute the ultimate result
    - Zero iff all of the exception(s) (if any) which were raised during retried operation are masked by the software SPEFSCR mask bits
    - Otherwise, the original `sigcode` value passed in
8. If the result will be zero, or if this was a rounding fault:
    a. Store the high word results directly in the high words of the GPRs and the low word results into `regs` (possibly including the CR register)
    b. Advance the IAR in `regs` to the next instruction
9. Return the previously computed sigcode result to the caller

For retried operations, the basic IEEE operations for signed 32 and 64 bit integers are provided by the third party softfloat library, which is also used by the fpemu library. For symmetry with that library, this module defines a large number of related IEEE helper functions, primarily to provide support for two data types used by the PPC SPE instruction set but for which there is no softfloat library analogue: unsigned 32 and 64 bit integers, as well as both signed and unsigned 32 bit fractional types. The names of these utility functions and types have largely been chosen for symmetry with existing functions in the softfloat library.

The final group of utility functions abstracts load and store operations for the thread's general purpose registers. The lower 32 bit words of the General Purpose Registers are stored in a register save area, which will be used to restore those registers when the thread resumes after the exception has been handled. The upper 32 bit

words of the General Purpose Registers (which are invisible and are unalterted by all instructions except SPE instructions) remain stored in their hardware registers. (Since this is user-mode, the usual kernel mechanics are at play in arbitrating access to this alt context, all of which is entirely transparent to the fpassist library.)

- **fp_status API and Protocol**

The fenv and fp_status APIs in libm attempt to operate cooperatively with the fpassist library's use of the SPEFSCR. Rather than accessing the register directly, the libm APIs first attempt to send read and write requests to fpassist `_math_emulator()` via the libc `_emulator_callout` mechanism, using a characteristic set of parameters to distinguish this usage from that of the kernel.

The fpassist library must arrange for the calling thread to be able to set and get the rounding mode, the sticky exception flags, and the exception mask flags. It must also arrange for the thread visible behaviour of the floating point operations to correspond to those settings. Specifically, it must arrange for `SIGFPE` faults to be delivered for any unmasked exceptions which occur during the execution of hardware floating point instructions. This is accomplished as outlined above.

`libm' Library#

User applications have access to both the C99 fenv API and the older QNX fp_status API, which serves a similar purpose. Through these APIs, user applications can select which (if any) floating point exceptions should be signalled using `SIGFPE`. When a full hardware floating point implementation is available, the implementation of these functions can simply enable the underlying hardware exceptions directly and expect that the existing kernel mechanisms will result in the appropriate signal being delivered.

When emulation or IEEE assitance is needed, however, this is not viable. In the case of the EFP APU, four of the five hardware exceptions must be enabled at all times in order that the fpassist library can be invoked at the appropriate time to provide IEEE compliant results, independently of whether the user application wishes to receive `SIGFPE` for any of those exceptions. Accordingly, the fp_status API (and the equivalent fenv API) must cooperate with the fpassist library to correctly provide this feature.

This is accomplished by having the fp_status (and fenv) API attempt to communicate with the fpassist library through the `_emulator_callout`, using a characteristic set of parameters to differentiate this type of invocation from that of the kernel servicing an IEEE exception. If the fpassist library is present, it serves as a proxy for the exception sticky flags required by the fp_status and fenv APIs; otherwise, it will fall back to accessing the SPEFSCR directly.

The fpassist library will arrange to have the hardware exceptions permanently enabled. When it processes an exception, it checks to see if the calling thread has enabled the related exception bit using the fp_status or fenv APIs. If so, the fpassist library will arrange to deliver a `SIGFPE` with the appropriate sigcode to the thread. Otherwise, it merely sets the sticky flag for the thread (if appropriate).

procnto/kernel#

Kernel modifications are required to install two new exception handlers for the E500v2 EFP APU: handlers for the floating point data and round exceptions. These handlers will both use the same basic design as the existing floating point emulation handler.

Each of the handlers will perform the following steps:

1. Determine the sigcode appropriate to the exception
2. Acquire the kernel
3. Get the fpassist context (if any) from the TLS
4. Get the dispatch function address from the PLS
5. Unlock the kernel (to take any user stack faults)
6. Set up a register context on the user stack

7. Lock the kernel again
8. Save and disable single stepping
9. Exit the kernel, having arranged for the user thread to call the dispatch function when it resumes

The remainder of the execution path follows the existing design and implementation for the floating point emulator (i.e. no changes to existing code are necessary):

1. The dispatch function from the PLS is executed which, in practice, is always `_math_emu_stub()` in libc
2. libc `_math_emu_stub()` calls the function stored in the `_emulator_callout` variable in libc
3. Initially `_emulator_callout` points to the `_math_emulator()` in libc (N.B. not in fpassist)
4. libc `_math_emulator()` makes a `pthreads_once()` call to `_math_emu_load()` in libc
5. libc `_math_emu_load()` attempts to `dlopen()` the fpassist library and to `dlsym()` the `_math_emulator()` in fpassist
   - if fpassist `_math_emulator()` is found, `_emulator_callout` is updated to point directly to it and it is called
   - if not, `_emulator_callout` will remain pointing to libc `_math_emulator()`, which will now just return the sigcode with which it was called
6. The return value of the `_emulator_callout` (the `_math_emulator()` either in fpassist or in libc) is passed as a sigcode to `SignalFault()`
   - A non-zero sigcode causes a fault signal to be dropped on the user thread
   - A zero sigcode causes control to pass back to the IAR saved in the register context on the stack, and the stack is cleaned up to the state it was in before the exception.

Implementation#

ABI Variant#

In addition to the required variant directories, various adjustments were made to the build system. The main changes were:

- the addition of a `CPUVARDIR` makevar which contains the variant qualified directory name (e.g. `ppcbe-spe`);
- reworking of build-cfg and related collateral to handle autoconf builds of SPE variants (and general improvements for all platforms); and
- modification of search rules in `mkifs` to more transparently handle the fact that there are no SPE-specific bootfiles.

'fpassist' Library#

Minor modifications were required to libc `_math_emu_load()` to load fpassist instead of fpemu for the SPE variant.

- **Exception Handling API and Protocol**

The entry point into fpassist is the `_math_emulator()` function, which is usually called via the `_emulator_callout` mechanism in libc. When this routine is called as part of the handling of an embedded floating point exception, the code byte of the `SIGFPE` sigcode describes the exception (`FPE_NOFPU` for data, `FPE_FLTRES` for rounding); the IAR pointing at the instruction causing the exception; and the remainder of the register context filled in appropriately. The high words of the GPRs and the accumulator are assumed to be in the state they were in at the time of the exception (since they can only be modified by embedded floating point instructions).

The first action is to allocate a thread context, if one has not already been allocated. The thread context is required (by `kerext_debug`) to begin with a (true, non-embedded) floating point register save area, which is not otherwise used by the fpassist library and in which all registers are always set to zero. Following this is a softfloat environment, which is also unused by the fpassist library. (It is reserved for future extension and preserves an identical layout with the context used by the fpemu library. Although there is currently no functional need for compatibility, it may prove to be worthwhile if it is ever necessary to support both fpemu and fpassist simultaneously.) Next is the private context used by the fpassist library. This contains the current value of the application visible exception flags and mask.

The general processing of the exception then follows three main phases: operand extraction, operation emulation, and result storage.

Broadly speaking, operand extraction falls into one of three categories: vector, single precision, or double precision. Operations having single precision operands operate on only the low 32 bit word of those operands; those with double precision operands operate on the full 64 bits. Operands for vector operations are always fetched from the full 64 bits as well, but the operation itself is performed as two single precision SIMD operations. For vector operations, either (or both) the low and high word operands may be needed; if only the low or high computation faulted, only that computation is emulated.

The emulation step consists of computing the IEEE compliant result of the mathematical, logical, or conversion operations. A separate softfloat environment is maintained for each operation which is performed (i.e. up to two, when a vector operation is performed). Single precision computations always involve only the low word(s) of GPRs; double precision computations always involve the full 64 bits their operand(s). Vector operations perform simulataneous operations on pairs of operand(s), one value each in the low and high words.

Result storage consists of determining which, if any, destination registers need to be updated. In particular, the SPEFSCR sticky flags are updated from the softfloat environment(s) which were used. The application's exception mask is consulted to determine if a `SIGFPE` should still be delivered; typically the result is not stored if a signal will be delivered (just as it would not had the fpassist library not been present.)

- **fp_status API and Protocol**

The `fp_setenv()` implementation in libm for SPE PPC uses a simple protocol to communicate with the fpassist library in order to cooperatively use the SPEFSCR register. It calls the `_emulator_callout()` function directly, passing `SIGFPE`; an IAR of `~0U`; and a USPRG0 of either `O_RDONLY`, for a read, or `O_WRONLY`, for a write of the SPEFSCR. If the fpassist library is missing, this will return non-zero (`SIGFPE`) and `fp_setenv()` will fall back to accessing the hardware directly. (In this case, unmasked exceptions will result in `SIGFPE` delivery.) Otherwise, the value manipulated will in fact be the one from the application context allocated above.

The saved value is used only for its exception flags and mask; the remaining bits are always read directly from the hardware.

When the SPEFSCR is written by the application, the value supplied is stored in the application context. The hardware SPEFSCR is updated unconditionally to enable all of the exceptions required for proper IEEE 754 emulation by the fpassist library (FINVE, FDBZE, FOVFE, and FUNFE; FINXE is set only as the application requests).

This procedure allows the actual hardware sticky exception flags and the hardware exception mask to vary independently from the application exception flags and mask. This is necessary because four of the five exceptions must be enabled at all times to provide IEEE 754 compliant results, regardless of whether the application has masked those exceptions to avoid `SIGFPE` delivery. Likewise, the application exception flags must never be cleared by the emulator

This latter situation would be extremely rare anyway in practice. It could only arise if the hardware sticky flags were used directly to represent the application exception flags and the application had directly set one or more

of them. In that case, if the corresponding exception were actually to arise during hardware execution, but be negated during emulation, it would not be possible to know if it were correct to clear the flag (which would be the proper course of action had the application not set the flag). A practical example of this would be the "invalid" flag raised by the hardware for a subnormal operand. Subsequent emulation will probably result in the operand no longer being considered "invalid", so although the hardware sticky flag is raised, the application should see it as clear - unless it had earlier set the flag itself directly. This situation will be exceedingly rare because most applications will never set a sticky flag; they will only clear them.

`libm' Library#

For the PPC, there are two variants relevant to the implementation of the fp_status API in libm: cores which support the full PPC hardware FPU; and those which support the EFP APU.

The implementation for the PPC FPU is very straightforward: the values from the hardware floating point status and control register are just proxied through the API. All that is done is to translate the hardware specific bit positions into the hardware independent values used by the fp_status API.

For the EFP, the behaviour is more complex. In broad terms, it behaves similarly to the FPU case: the values from the hardware SPE floating point status and control register are proxied. However, rather than accessing the hardware directly, an attempt is first made to read or write the desired hardware value using the fpassist library (via the `_emulator_callout` function pointer). This allows the fpassist library (which is required in order to provide IEEE compliance) to be kept informed about what values the application wishes to assign for the sticky bits and the exception mask. (It needs to know this because it must keep hardware exceptions enabled in order to carry out its own function, even if the application wishes the associated software exception to be masked.) Only if the call via `_emulator_callout` fails does this implementation resort to manipulating the hardware SPEFSCR register directly.

The system's use of `_emulator_callout` is somewhat subtle.

The storage for `_emulator_callout` itself is provided by libc and is process local. Initially it points to a helper function within libc. When the `_emulator_callout` is invoked for the first time (and only the first time), the helper function attempts to load the fpassist library. If it succeeds, the `_emulator_callout` is pointed directly at the newly loaded library - otherwise it continues to point at the helper function (in which case it subsequently just returns the sigcode it receives).

The `_emulator_callout` API requires three arguments: a sigcode (which must represent a `SIGFPE` signal); a pointer to a pointer which may be used as TLS (in which the emulator can store a pointer to its context); and a `PPC_CPU_REGISTERS` structure. All three are mandatory, and the API effectively requires that the TLS pointer used is the address of `__tls()->__fpuemu_data` (which will initially be `NULL`).

The `_emulator_callout` is normally invoked by the hardware exception handlers for the EFP hardware exceptions (data and round). The fpalloc library then uses the information passed to provide IEEE compliance for the faulting operation.

In this case, the `_emulator_callout` is being used to communicate between the application and the fpassist library. In order that the fpassist library can distinguish between these use cases, a special protocol is used. Most of the "registers" passed are uninitialized (and unused). The IAR (the PPC instruction pointer) is set to `~0U`, a value which is not otherwise possible as the address of a faulting instruction; and the USPRG0 register is set to either `O_RDONLY` (for a SPEFSCR read) or `O_WRONLY` (for a write). (The choice of `O_RDONLY` and `O_WRONLY` is entirely arbitary - any two distinct values would do. It was felt that some sort of self-documenting value should be used, but it was also felt that it was undesirable to add a new shared header and symbols for this one extremely specialized usage.) The value to be written would be stored in the SPEFSCR field, while the value to be read would be returned in the same fashion.

procnto/kernel#

The implementation of the floating point rounding and data exception handlers closely follows that of the floating point emulation handler. The EFP rounding and data exception handlers share almost all of their code, differing only in the sigcode which they ultimately pass on the the `_math_emulator()` function. The common portion, which starts at `__exc_spe_efp_assist`, performs the steps outlined above.

CI and Submissions#

The work was carried out under PRs 64398 and 66309.

The CI posts related to the changes can be found at:

- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6065
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6129
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6147
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6148
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6175
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6215
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6216
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6219
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6391
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6680
- http://community.qnx.com/sf/go/projects.core_os/discussion.osrev.topc6713
- http://community.qnx.com/sf/go/projects.internal_core_os/discussion.code_reviews.topc6681 (shiplist)

The subversion submissions related to the changes can be found at:

- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213032
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213033
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213035
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213115
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213124
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213127
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213331
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213332
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213334
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213335
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213338
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213366
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213368
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=213380
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=215092
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=215096
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=215100
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=215101
- http://websvn.ott.qnx.com/redir.cgi?repository=product&revision=215660
- http://community.qnx.com/integration/viewcvs/viewcvs.cgi/?root=core_networking&system=exsy1001&rev=805&view=rev
- http://community.qnx.com/integration/viewcvs/viewcvs.cgi/?root=core_networking&system=exsy1001&rev=806&view=rev
- http://community.qnx.com/integration/viewcvs/viewcvs.cgi/?root=core_networking&system=exsy1001&rev=807&view=rev