Apex Runtime

This document was generated on 13-August-2018.

Build #530, based on clang 4.0.1

The following are runtime routines which contains implementation that are specific to APEX. They are available as part of the runtime library librt.a

```
/* runtime abort */
void compilerrt_abort_impl(const char *file, int line, const char *function);
```

This is called by other runtime routines to handle abort. For APEX, this calls the _exit_fsl() routine which flush the stdout and halt the program gracefully.

```
/* scalar memcpy */
void* __smemcpy(char *dst, const char *src, unsigned long n);
```

The compiler converts standard memcpy() calls to __smemcpy if the input pointers points to scalar type. It does a byte-wide copy of size n from src address to dst address. It returns dst.

```
/* scalar memset */
void* __smemset(char *dst, int val, unsigned long n);
```

The compiler converts standard memset() calls to __smemset if the input pointers points to scalar type. It does a byte-wide copy of size n from val to dst address. It returns dst.

```
/* vector memcpy */
void __vmemcpy(char *dst, const char *src, unsigned long n);
```

The compiler converts standard memcpy() calls to __vmemcpy if the input pointers points to vector type. It copies n vec08u vector from dst to src.

```
/* vector memset */
void* __vmemset(char *dst, int val, unsigned long n);
```

The compiler converts standard memset() calls to __vmemset if the input pointers points to vector type. It copies value val as vector vec08 from dst to dst + n

```
/* optimized vector memcpy */
void __vmemcpy_aligned_min4b_mult2b(vec08u *dst, const vec08u *src, unsigned long n);
```

This is an optimized asm implementation of vector memcpy to hide vector load store latency. It copies a two vec08 for each hardware loop iteration

```
/* optimized vector memset */
void __vmemset_aligned_min2b_mult2b(vec08u *dst, int val, unsigned long n);
```

This is an optimized asm implementation of vector memset to hide vector load store latency. It set two vec08 to lower 8-bit of val for each hardware loop iteration

```
/* optimized scalar memset */
void __smemset_aligned_mult4b(char *dst, int val, unsigned long n);
```

This is an optimized asm implementation of scalar memset. It set four chars to lower 8-bit of val for each hardware loop iteration

```
/* vec16s division */
vec16s __divi16v(vec16s a, vec16s b);
```

```
/* vec16u division */
vec16u __udivi16v(vec16u a, vec16u b);
```

```
/* vec16s modulo */
vec16s __remi16v(vec16s a, vec16s b);
```

```
/* vec16u modulo */
vec16u __uremi16v(vec16u a, vec16u b);
```

The 16-bit vector division and modulo operations all share the same underlying unsigned 16-bit division algorithm in hand-optimized assembly. The signed variant calculates the sign-bit result which then calls the unsigned version. The modulo function is simply (a - a/b * b)

```
/* vec32s division */
vec32s __divi32v(vec32s a, vec32s b);
```

```
/* vec32u division */
vec32u __udivi32v(vec32u a, vec32u b);
```

```
/* vec32s modulo */
vec32s __remi32v(vec32s a, vec32s b);
```

```
/* vec32s modulo */
vec32u __uremi32v(vec32u a, vec32u b);
```

The 32-bit vector division and modulo operations all share the same underlying unsigned 32-bit division algorithm. It performs fast reciprocal approximation using table lookup for 1/b and result is then multiplied to a. The signed variant calculate the sign-bit and then calls the unsigned version. The modulo function is simply (a - a/b * b)

```
/* vec32s mutiply */
vec32s __muli32v(vec32s a, vec32s b);
```

32-bit vector multiplication returning the lower 32-bit result in vec32s

```
/* jum_buf setup */
int setjmp(jmp_buf env);
```

Save current context to jmp_buf for later use by longjmp to restore execution to right after setjmp() call. On Apex, LR, SP, and VSP are saved.

```
/* restore execution context from jmp_buf */
void longjmp(jmp_buf env, int value);
```

Restore context from jmp_buf. On Apex, LR, SP, VSP are restored. Execution continues right after where setjmp() is last called.