SMART ARM-based Microcontrollers

AT12200: SAM C Divide and Square Root Accelerator (DIVAS) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Divide and Square Root Accelerator functionality.

The following peripherals are used by this module:

- DIVAS (Divide and Square Root Accelerator)

The following devices can use this module:

- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

Table of Contents

1. Software License

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

This driver provides an interface for the Divide and Square Root Accelerator on the device.

The DIVAS is a programmable 32-bit signed or unsigned hardware divider and a 32-bit unsigned square root hardware engine. When running signed division, both the input and the result will be in two's complement format. The result of signed division is that the remainder has the same sign as the dividend and the quotient is negative if the dividend and divisor have opposite signs. When the square root input register is programmed, the square root function starts and the result will be stored in the Remainder register.

There are two ways to calculate the results:

- Call the DIVAS API
- Overload "/" and "%" operation
  Note: Square root operation can't implement overload operation.

3.1. Overload Operation

The operation is implemented automatically by EABI (Enhanced Application Binary Interface). EABI is a standard calling convention, which is defined by ARM. The four functions interface can implement division and mod operation in EABI.

The following prototypes for EABI division operation in ICCARM tool chain:

```
int __aeabi_idiv(int numerator, int denominator);
unsigned __aeabi_uidiv(unsigned numerator, unsigned denominator);
__value_in_regs idiv_return __aeabi_idivmod( int numerator, int
denominator);
__value_in_regs uidiv_return __aeabi_uidivmod( unsigned numerator,
                                               unsigned denominator);
```

The following prototypes for EABI division operation in GNUC tool chain:

```
int __aeabi_idiv(int numerator, int denominator);
unsigned __aeabi_uidiv(unsigned numerator, unsigned denominator);
uint64_t __aeabi_idivmod( int numerator, int denominator);
uint64_t uidiv_return __aeabi_uidivmod( unsigned numerator,
                                        unsigned denominator);
```

No matter what kind of tool chain, by using DIVAS module in the four functions body, the user can transparently access the DIVAS module when writing normal C code. For example:

```
void division(int32_t b, int32_t c)
{
    int32_t a;
    a = b / c;
    return a;
}
```

Similarly, the user can use the "a = b % c;" symbol to implement the operation with DIVAS, and needn't to care about the internal operation process.

3.2. Operand Size

- Divide: The DIVAS can perform 32-bit signed and unsigned division.
- Square Root: The DIVAS can perform 32-bit unsigned division.

3.3. Signed Division

When the signed flag is one, both the input and the result will be in two's complement format. The result of signed division is that the remainder has the same sign as the dividend and the quotient is negative if the dividend and divisor have opposite signs.

Note: When the maximum negative number is divided by the minimum negative number, the resulting quotient overflows the signed integer range and will return the maximum negative number with no indication of the overflow. This occurs for 0x80000000 / 0xFFFFFFFF in 32-bit operation and 0x8000 / 0xFFFF in 16-bit operation.

3.4. Divide By Zero

A divide by zero will cause a fault if the DIVISOR is programmed to zero. The result is that the quotient is zero and the reminder is equal to the dividend.

3.5. Unsigned Square Root

When the square root input register is programmed, the square root function starts and the result will be stored in the Result and Remainder registers.

Note: The square root function can't overload.

4. Special Considerations

There are no special considerations for this module.

5. Extra Information

For extra information, see Extra Information for DIVAS Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

For a list of examples related to this driver, see Examples for DIVAS Driver.

7. API Overview

7.1. Structure Definitions

7.1.1. Struct idiv_return

DIVAS signed division operator output data structure.

Table 7-1. Members

Type	Name	Description
int32_t	quotient	Signed division operator result: quotient
int32_t	remainder	Signed division operator result: remainder

7.1.2. Struct uidiv_return

DIVAS unsigned division operator output data structure.

Table 7-2. Members

Type	Name	Description
uint32_t	quotient	Unsigned division operator result: quotient
uint32_t	remainder	Unsigned division operator result: remainder

7.2. Function Definitions

7.2.1. Call the DIVAS API Operation

In this mode, the way that directly call the DIVAS API implement division or mod operation.

7.2.1.1. Function divas_idiv()

Signed division operation.

```
int32_t divas_idiv(
        int32_t numerator,
        int32_t denominator)
```

Run the signed division operation and return the quotient.

Table 7-3. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the signed division operation
[in]	denominator	The divisor of the signed division operation

Returns
The quotient of the DIVAS signed division operation.

7.2.1.2. Function divas_uidiv()

Unsigned division operation.

```
uint32_t divas_uidiv(
        uint32_t numerator,
        uint32_t denominator)
```

Run the unsigned division operation and return the results.

Table 7-4. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the unsigned division operation
[in]	denominator	The divisor of the unsigned division operation

Returns
The quotient of the DIVAS unsigned division operation.

7.2.1.3. Function divas_idivmod()

Signed division remainder operation.

```
int32_t divas_idivmod(
        int32_t numerator,
        int32_t denominator)
```

Run the signed division operation and return the remainder.

Table 7-5. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the signed division operation
[in]	denominator	The divisor of the signed division operation

Returns
The remainder of the DIVAS signed division operation.

7.2.1.4. Function divas_uidivmod()

Unsigned division remainder operation.

```
uint32_t divas_uidivmod(
        uint32_t numerator,
        uint32_t denominator)
```

Run the unsigned division operation and return the remainder.

Table 7-6. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the unsigned division operation
[in]	denominator	The divisor of the unsigned division operation

Returns
The remainder of the DIVAS unsigned division operation.

7.2.1.5. Function divas_sqrt()

Square root operation.

```
uint32_t divas_sqrt(
        uint32_t radicand)
```

Run the square root operation and return the results.

Table 7-7. Parameters

Data direction	Parameter name	Description
[in]	radicand	The radicand of the square root operation

Returns
The result of the DIVAS square root operation.

7.2.2. DIVAS Overload '/' and '%' Operation

In this mode, the user can transparently access the DIVAS module when writing normal C code. E.g. "a = b / c;" or "a = b % c;" will be translated to a subroutine call, which uses the DIVAS.

7.2.2.1. Function __aeabi_idiv()

Signed division operation overload.

```
int32_t __aeabi_idiv(
        int32_t numerator,
        int32_t denominator)
```

Run the signed division operation and return the results.

Table 7-8. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the signed division operation
[in]	denominator	The divisor of the signed division operation

Returns
The quotient of the DIVAS signed division operation.

7.2.2.2. Function __aeabi_uidiv()

Unsigned division operation overload.

```
uint32_t __aeabi_uidiv(
        uint32_t numerator,
        uint32_t denominator)
```

Run the unsigned division operation and return the results.

Table 7-9. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the unsigned division operation
[in]	denominator	The divisor of the unsigned division operation

Returns
The quotient of the DIVAS unsigned division operation.

7.2.2.3. Function __aeabi_idivmod()

Signed division remainder operation overload.

```
uint64_t __aeabi_idivmod(
        int32_t numerator,
        int32_t denominator)
```

Run the signed division operation and return the remainder.

Table 7-10. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the signed division operation
[in]	denominator	The divisor of the signed division operation

Returns
The remainder of the DIVAS signed division operation.

7.2.2.4. Function __aeabi_uidivmod()

Unsigned division remainder operation overload.

```
uint64_t __aeabi_uidivmod(
        uint32_t numerator,
        uint32_t denominator)
```

Run the unsigned division operation and return the remainder.

Table 7-11. Parameters

Data direction	Parameter name	Description
[in]	numerator	The dividend of the unsigned division operation
[in]	denominator	The divisor of the unsigned division operation

Returns
The remainder of the DIVAS unsigned division operation.

7.2.3. Function divas_disable_dlz()

Disables DIVAS leading zero optimization.

```
void divas_disable_dlz( void )
```

Disable leading zero optimization from the Divide and Square Root Accelerator module. When leading zero optimization is disable, 16-bit division completes in 8 cycles and 32-bit division completes in 16 cycles.

7.2.4. Function divas_enable_dlz()

Enables DIVAS leading zero optimization.

```
void divas_enable_dlz( void )
```

Enable leading zero optimization from the Divide and Square Root Accelerator module. When leading zero optimization is enable, 16-bit division completes in 2-8 cycles and 32-bit division completes in 2-16 cycles.

8. Extra Information for DIVAS Driver

8.1. Acronyms

Acronym	Description
DIVAS	Divide and Square Root Accelerator
EABI	Enhanced Application Binary Interface

8.2. Dependencies

This driver has no dependencies.

8.3. Errata

There are no errata related to this driver.

8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

9. Examples for DIVAS Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Divide and Square Root Accelerator (DIVAS) Driver. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for DIVAS - No Overload
- Quick Start Guide for DIVAS - Overload

9.1. Quick Start Guide for DIVAS - No Overload

In this use case, the Divide and Square Root Accelerator (DIVAS) module is used.

This use case will calculate the data in No Overload mode. If all the calculation results are the same as the desired results, the board LED will be lighted. Otherwise, the board LED will be flashing. The variable "result" can indicate which calculation is wrong.

9.1.1. Setup

9.1.1.1. Prerequisites

There are no special setup requirements for this use-case.

9.1.1.2. Code

The following must be added to the user application source file, outside any function:

The signed and unsigned dividend:

```
#define BUF_LEN 8

const int32_t numerator_s[BUF_LEN] = {
    2046, 415, 26, 1, -1, -255, -3798, -65535};

const int32_t excepted_s[BUF_LEN] = {
    2046, 207, 8, 0, 0, -42, -542, -8191};

const int32_t excepted_s_m[BUF_LEN] = {
    0, 1, 2, 1, -1, -3, -4, -7};

const uint32_t numerator_u[BUF_LEN] = {
    0x00000001,
    0x0000005A,
    0x000007AB,
    0x00006ABC,
    0x0004567D,
    0x0093846E,
    0x20781945,
    0x7FFFFFFF,
};

const uint32_t excepted_u[BUF_LEN] = {
    0x00000001,
    0x0000002d,
    0x0000028E,
    0x00001AAF,
    0x0000DE19,
```

```
    0x00189612,
    0x04A37153,
    0x0FFFFFFF,
};

const uint32_t excepted_u_m[BUF_LEN] = {
    0, 0, 1, 0, 0, 2, 0, 7};

const uint32_t excepted_r[BUF_LEN] = {
    0x00000001,
    0x00000009,
    0x0000002C,
    0x000000A5,
    0x00000215,
    0x00000C25,
    0x00005B2B,
    0x0000B504,
};

static int32_t result_s[BUF_LEN], result_s_m[BUF_LEN];
static uint32_t result_u[BUF_LEN], result_u_m[BUF_LEN];
static uint32_t result_r[BUF_LEN];
static uint8_t result = 0;
```

Copy-paste the following function code to your user application:

```
static void signed_division(void)
{
    int32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_s[i];
        denominator = i + 1;
        result_s[i] = divas_idiv(numerator, denominator);
        if(result_s[i] != excepted_s[i]) {
            result |= 0x01;
        }
    }
}

static void unsigned_division(void)
{
    uint32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_u[i];
        denominator = i + 1;
        result_u[i] = divas_uidiv(numerator, denominator);
        if(result_u[i] != excepted_u[i]) {
            result |= 0x02;
        }
    }
}

static void signed_division_mod(void)
{
    int32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_s[i];
```

```
        denominator = i + 1;
        result_s_m[i] = divas_idivmod(numerator, denominator);
        if(result_s_m[i] != excepted_s_m[i]) {
            result |= 0x04;
        }
    }
}

static void unsigned_division_mod(void)
{
    uint32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_u[i];
        denominator = i + 1;
        result_u_m[i] = divas_uidivmod(numerator, denominator);
        if(result_u_m[i] != excepted_u_m[i]) {
            result |= 0x08;
        }
    }
}

static void squart_root(void)
{
    uint32_t operator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        operator = numerator_u[i];
        result_r[i] = divas_sqrt(operator);
        if(result_r[i] != excepted_r[i]) {
            result |= 0x10;
        }
    }
}
```

Add to user application initialization (typically the start of `main()`):

```
system_init();
```

9.1.2. Implementation

9.1.2.1. Code

Copy-paste the following code to your user application:

```
signed_division();
unsigned_division();
signed_division_mod();
unsigned_division_mod();
squart_root();

while (true) {
    if(result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 50000;
        while(delay--) {
        }
    } else {
        port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
```

```
    }
}
```

9.1.2.2. Workflow

1. Signed division calculation.
```
signed_division();
```
2. Unsigned division calculation.
```
unsigned_division();
```
3. Signed reminder calculation.
```
signed_division_mod();
```
4. Unsigned reminder calculation.
```
unsigned_division_mod();
```
5. Square root calculation.
```
squart_root();
```
6. Infinite loop.
```
while (true) {
    if(result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 50000;
        while(delay--) {
        }
    } else {
        port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
    }
}
```

9.2. Quick Start Guide for DIVAS - Overload

In this use case, the Divide and Square Root Accelerator (DIVAS) module is used.

This use case will calculate the data in overload mode. If all the calculation results are the same as the desired results, the board LED will be lighted. Otherwise, the board LED will be flashing. The variable "result" can indicate which calculation is wrong.

9.2.1. Setup

9.2.1.1. Prerequisites

There are no special setup requirements for this use-case.

9.2.1.2. Code

The following must be added to the user application source file, outside any function:

The signed and unsigned dividend:

```
#define BUF_LEN 8

const int32_t numerator_s[BUF_LEN] = {
    2046, 415, 26, 1, -1, -255, -3798, -65535};
```

```
const int32_t excepted_s[BUF_LEN] = {
    2046, 207, 8, 0, 0, -42, -542, -8191};

const int32_t excepted_s_m[BUF_LEN] = {
    0, 1, 2, 1, -1, -3, -4, -7};

const uint32_t numerator_u[BUF_LEN] = {
    0x00000001,
    0x0000005A,
    0x000007AB,
    0x00006ABC,
    0x0004567D,
    0x0093846E,
    0x20781945,
    0x7FFFFFFF,
};

const uint32_t excepted_u[BUF_LEN] = {
    0x00000001,
    0x0000002d,
    0x0000028E,
    0x00001AAF,
    0x0000DE19,
    0x00189612,
    0x04A37153,
    0x0FFFFFFF,
};

const uint32_t excepted_u_m[BUF_LEN] = {
    0, 0, 1, 0, 0, 2, 0, 7};

const uint32_t excepted_r[BUF_LEN] = {
    0x00000001,
    0x00000009,
    0x0000002C,
    0x000000A5,
    0x00000215,
    0x00000C25,
    0x00005B2B,
    0x0000B504,
};

static int32_t result_s[BUF_LEN], result_s_m[BUF_LEN];
static uint32_t result_u[BUF_LEN], result_u_m[BUF_LEN];
static uint32_t result_r[BUF_LEN];
static uint8_t result = 0;
```

Copy-paste the following function code to your user application:

```
static void signed_division(void)
{
    int32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_s[i];
        denominator = i + 1;
        result_s[i] = numerator / denominator;
        if(result_s[i] != excepted_s[i]) {
            result |= 0x01;
        }
    }
```

```
}

static void unsigned_division(void)
{
    uint32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_u[i];
        denominator = i + 1;
        result_u[i] = numerator / denominator;
        if(result_u[i] != excepted_u[i]) {
            result |= 0x02;
        }
    }
}

static void signed_division_mod(void)
{
    int32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_s[i];
        denominator = i + 1;
        result_s_m[i] = numerator % denominator;
        if(result_s_m[i] != excepted_s_m[i]) {
            result |= 0x04;
        }
    }
}

static void unsigned_division_mod(void)
{
    uint32_t numerator, denominator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        numerator = numerator_u[i];
        denominator = i + 1;
        result_u_m[i] = numerator % denominator;
        if(result_u_m[i] != excepted_u_m[i]) {
            result |= 0x08;
        }
    }
}

static void squart_root(void)
{
    uint32_t operator;
    uint8_t i;

    for (i = 0; i < BUF_LEN; i++) {
        operator = numerator_u[i];
        result_r[i] = divas_sqrt(operator);
        if(result_r[i] != excepted_r[i]) {
            result |= 0x10;
        }
    }
}
```

Add to user application initialization (typically the start of `main()`):

```
system_init();
```

9.2.2. Implementation

9.2.2.1. Code

Copy-paste the following code to your user application:

```
signed_division();
unsigned_division();
signed_division_mod();
unsigned_division_mod();
squart_root();

while (true) {
    if(result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 50000;
        while(delay--) {
        }
    } else {
        port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
    }
}
```

9.2.2.2. Workflow

1. Signed division calculation.

   ```
   signed_division();
   ```

2. Unsigned division calculation.

   ```
   unsigned_division();
   ```

3. Signed reminder calculation.

   ```
   signed_division_mod();
   ```

4. Unsigned reminder calculation.

   ```
   unsigned_division_mod();
   ```

5. Square root calculation.

   ```
   squart_root();
   ```

6. Infinite loop.

   ```
   while (true) {
       if(result) {
           port_pin_toggle_output_level(LED_0_PIN);
           /* Add a short delay to see LED toggle */
           volatile uint32_t delay = 50000;
           while(delay--) {
           }
       } else {
           port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
       }
   }
   ```

10. Document Revision History

Doc. Rev.	Date	Comments
42644A	01/2016	Initial document release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**