Исследование избыточного сохранения регистров при вызове подпрограммы обработчика прерывания (MIK32)

Киндеркнехт Виктор, 30 апреля 2025

Оглавление

Введение....3
Дизассемблирование raw_trap_handler (код из примера)....4
Использование в качестве функции обработчика прерываний raw_trap_handler с атрибутом interrupt....5
Дизассемблирование raw_trap_handler (модифицированный код)....5
Использование raw_trap_handler без атрибута interrupt....7
Дизассемблирование raw_trap_handler без атрибута interrupt....8
Различия в инструкциях RET и MRET....9
Сохранение регистров при обработке прерывания....9
Вывод....10

Введение

В исследовании рассматривается пример HAL_Blink_IRQ с официального репозитория mik32-examples на GitFlic (https://gitflic.ru/project/mikron-mik32/mik32-examples).

Данный пример раз в определенный промежуток времени изменяет состояние вывода PORT2_5, который должен быть соединен пользователем с выводом PORT2_6. По нарастающему фронту на PORT2_6 вызывается прерывание GPIO, в котором изменяется флаг. В основном же цикле программы, состояние GPIO, управляющих индикаторными светодиодами, меняется сообразно состоянию флага. Таким образом, метрикой корректности работы программы служит факт мигания светодиодов.

Далее приложен исходный код обработчика прерывания из примера.

```
void trap_handler()
{
	if (EPIC_CHECK_GPIO_IRQ())
	{
			if (HAL_GPIO_LineInterruptState(GPIO_LINE_2))
		{
			flag = !flag;
		}
		HAL_GPIO_ClearInterrupts();
	}
	/* Сброс прерываний */
	HAL_EPIC_Clear(0xFFFFFFFF);
}
```

Обработка прерываний в MIK32 в настоящее время (30.04.2025) ведется следующим образом:

1. Процессор переходит на вектор MTVEC (0xC0)
2. С вектора MTVEC выполняется переход к подпрограмме raw_trap_handler, расположенной в секции .trap_text
3. В подпрограмме raw_trap_handler выполняется загрузка в стек всех 32 регистров общего назначения, затем выполняется переход к выполнению функции пользовательской обработки прерывания trap_handler
4. По возврату из trap_handler, содержимое РОН выгружается из стека, затем выполняется возврат к прерванной программе.

Цель: исследовать возможности оптимизации скорости обработки прерывания.

Дизассемблирование raw_trap_handler (код из примера)

```
020000c4 <raw_trap_handler>:

raw_trap_handler:
    // Save registers
    addi    sp, sp, -(EXCEPTION_STACK_SPACE)
 20000c4:    f8410113           add   sp,sp,-124 # 2003f84 <__global_pointer$
+0x364c>
    .irp index, EXCEPTION_SAVED_REGISTERS
        sw      x\index, 4*(index-1)(sp)
 20000c8:    c006               sw    ra,0(sp)
 20000ca:    c20a               sw    sp,4(sp)
 20000cc:    c40e               sw    gp,8(sp)
 20000ce:    c612               sw    tp,12(sp)
 20000d0:    c816               sw    t0,16(sp)
 20000d2:    ca1a               sw    t1,20(sp)
 20000d4:    cc1e               sw    t2,24(sp)
 20000d6:    ce22               sw    s0,28(sp)
 20000d8:    d026               sw    s1,32(sp)
 20000da:    d22a               sw    a0,36(sp)
 20000dc:    d42e               sw    a1,40(sp)
 20000de:    d632               sw    a2,44(sp)
 20000e0:    d836               sw    a3,48(sp)
 20000e2:    da3a               sw    a4,52(sp)
 20000e4:    dc3e               sw    a5,56(sp)
 20000e6:    de42               sw    a6,60(sp)
 20000e8:    c0c6               sw    a7,64(sp)
 20000ea:    c2ca               sw    s2,68(sp)
 20000ec:    c4ce               sw    s3,72(sp)
 20000ee:    c6d2               sw    s4,76(sp)
 20000f0:    c8d6               sw    s5,80(sp)
 20000f2:    cada               sw    s6,84(sp)
 20000f4:    ccde               sw    s7,88(sp)
 20000f6:    cee2               sw    s8,92(sp)
 20000f8:    d0e6               sw    s9,96(sp)
 20000fa:    d2ea               sw    s10,100(sp)
 20000fc:    d4ee               sw    s11,104(sp)
 20000fe:    d6f2               sw    t3,108(sp)
 2000100:    d8f6               sw    t4,112(sp)
 2000102:    dafa               sw    t5,116(sp)
 2000104:    dcfe               sw    t6,120(sp)
    .endr

    // Call handler
    la      ra, trap_handler
 2000106:    ff000097           auipc ra,0xff000
 200010a:    23008093           add   ra,ra,560 # 1000336 <trap_handler>
    jalr    ra
 200010e:    9082               jalr  ra

    // restore registers
    .irp index, EXCEPTION_SAVED_REGISTERS
        lw      x\index, 4*(index-1)(sp)
<...>
    .endr
    addi    sp, sp, EXCEPTION_STACK_SPACE
 200014e:    07c10113           add   sp,sp,124
    mret
 2000152:    30200073           mret
```

Как видно из листинга, подпрограмма raw_trap_handler по-умолчанию сохраняет содержимое всех РОН процессора, что отнимает много времени при вызове прерывания.

Использование в качестве функции обработчика прерываний raw_trap_handler с атрибутом interrupt

Был выполнен тест, заключающийся в использовании в качестве обработчика прерывания не trap_handler, но подпрограммы raw_trap_handler, помеченной атрибутом interrupt.

Был использован дефайн INT_ATTR из mik32_hal.h

```
#define INT_ATTR __attribute__((noinline, interrupt, section(".trap_text")))
```

Таким образом, код обработчика прерывания стал выглядеть:

```
void INT_ATTR raw_trap_handler()
{
	if (EPIC_CHECK_GPIO_IRQ())
	{
			if (HAL_GPIO_LineInterruptState(GPIO_LINE_2))
			{
			flag = !flag;
			}
			HAL_GPIO_ClearInterrupts();
	}
	/* Сброс прерываний */
	HAL_EPIC_Clear(0xFFFFFFFF);
}
```

Дизассемблирование raw_trap_handler (модифицированный код)

```
void INT_ATTR raw_trap_handler()
{
 2000158:	7139                	add	sp,sp,-64
 200015a:	ce3a                	sw	a4,28(sp)
    if (EPIC_CHECK_GPIO_IRQ())
 200015c:	00050737            	lui	a4,0x50
{
 2000160:	cc3e                	sw	a5,24(sp)
    if (EPIC_CHECK_GPIO_IRQ())
 2000162:	42072783            	lw	a5,1056(a4) # 50420 <__stack_size+0x50020>
{
 2000166:	de06                	sw	ra,60(sp)
 2000168:	dc16                	sw	t0,56(sp)
 200016a:	da1a                	sw	t1,52(sp)
 200016c:	d81e                	sw	t2,48(sp)
 200016e:	d62a                	sw	a0,44(sp)
 2000170:	d42e                	sw	a1,40(sp)
 2000172:	d232                	sw	a2,36(sp)
 2000174:	d036                	sw	a3,32(sp)
 2000176:	ca42                	sw	a6,20(sp)
 2000178:	c846                	sw	a7,16(sp)
 200017a:	c672                	sw	t3,12(sp)
 200017c:	c476                	sw	t4,8(sp)
 200017e:	c27a                	sw	t5,4(sp)
 2000180:	c07e                	sw	t6,0(sp)
```

```
      if (EPIC_CHECK_GPIO_IRQ())
 2000182:    0207f793           and    a5,a5,32
 2000186:    eb8d               bnez   a5,20001b8 <raw_trap_handler+0x60>
 * Returns:
 * void.
 */
static inline __attribute__((always_inline)) void HAL_EPIC_Clear(uint32_t
InterruptMask)
{
    EPIC->CLEAR = InterruptMask;
 2000188:    000507b7           lui    a5,0x50
 200018c:    577d               li     a4,-1
 200018e:    40e7ac23           sw     a4,1048(a5) # 50418 <__stack_size+0x50018>
        }
        HAL_GPIO_ClearInterrupts();
    }
    /* Сброс прерываний */
    HAL_EPIC_Clear(0xFFFFFFFF);
 2000192:    50f2               lw     ra,60(sp)
 2000194:    52e2               lw     t0,56(sp)
 2000196:    5352               lw     t1,52(sp)
 2000198:    53c2               lw     t2,48(sp)
 200019a:    5532               lw     a0,44(sp)
 200019c:    55a2               lw     a1,40(sp)
 200019e:    5612               lw     a2,36(sp)
 20001a0:    5682               lw     a3,32(sp)
 20001a2:    4772               lw     a4,28(sp)
 20001a4:    47e2               lw     a5,24(sp)
 20001a6:    4852               lw     a6,20(sp)
 20001a8:    48c2               lw     a7,16(sp)
 20001aa:    4e32               lw     t3,12(sp)
 20001ac:    4ea2               lw     t4,8(sp)
 20001ae:    4f12               lw     t5,4(sp)
 20001b0:    4f82               lw     t6,0(sp)
 20001b2:    6121               add    sp,sp,64
 20001b4:    30200073           mret
        if (HAL_GPIO_LineInterruptState(GPIO_LINE_2))
 20001b8:    02000513           li     a0,32
 20001bc:    ff000097           auipc  ra,0xff000
 20001c0:    44e080e7           jalr   1102(ra) # 100060a
<HAL_GPIO_LineInterruptState>
 20001c4:    c519               beqz   a0,20001d2 <raw_trap_handler+0x7a>
            flag = !flag;
 20001c6:    8201a783           lw     a5,-2016(gp) # 20001dc <flag>
 20001ca:    0017b793           seqz   a5,a5
 20001ce:    82f1a023           sw     a5,-2016(gp) # 20001dc <flag>
        HAL_GPIO_ClearInterrupts();
 20001d2:    ff000097           auipc  ra,0xff000
 20001d6:    452080e7           jalr   1106(ra) # 1000624 <HAL_GPIO_ClearInterrupts>
 20001da:    b77d               j      2000188 <raw_trap_handler+0x30>
```

Как видно из ассемблерного кода, компилятор при входе в подпрограмму, помеченную атрибутом interrupt, сперва выделяет область стека для сохранения данных регистров (выделено желтым).

Далее, перед загрузкой новых данных в регистр a4 компилятор сохраняет его значение в выделенной области стека (помечено персиковым).

После сохраняет содержимое всех необходимых регистров в стек, а перед возвратом из обработчика прерывания выгружает значения из стека (помечено зеленым).

Также следует отметить, что для возврата используется инструкция mret заместо стандартной ret.

Код примера с измененным обработчиком прерываний, как было выяснено в ходе теста, работает корректно.

Использование raw_trap_handler без атрибута interrupt

Ради теста была проведена проверка работоспособности программы, использующей в качестве обработчика прерываний raw_trap_handler без атрибута interrupt. Код стал выглядеть так:

```
void __attribute__((noinline, section(".trap_text"))) raw_trap_handler()
{
	if (EPIC_CHECK_GPIO_IRQ())
	{
			if (HAL_GPIO_LineInterruptState(GPIO_LINE_2))
			{
			flag = !flag;
			}
			HAL_GPIO_ClearInterrupts();
	}
	/* Сброс прерываний */
	HAL_EPIC_Clear(0xFFFFFFFF);
}
```

Код с таким обработчиком прерывания работать не стал.

Дизассемблирование raw_trap_handler без атрибута interrupt

```
void __attribute__((noinline, section(".trap_text"))) raw_trap_handler()
{
    if (EPIC_CHECK_GPIO_IRQ())
 2000158:    00050737           lui    a4,0x50
 200015c:    42072783           lw     a5,1056(a4) # 50420 <__stack_size+0x50020>
 2000160:    0207f793           and    a5,a5,32
 2000164:    e799                      bnez   a5,2000172 <raw_trap_handler+0x1a>
 * Returns:
 * void.
 */
static inline __attribute__((always_inline)) void HAL_EPIC_Clear(uint32_t
InterruptMask)
{
    EPIC->CLEAR = InterruptMask;
 2000166:    000507b7           lui    a5,0x50
 200016a:    577d                      li     a4,-1
 200016c:    40e7ac23           sw     a4,1048(a5) # 50418 <__stack_size+0x50018>
 2000170:    8082                      ret
{
 2000172:    1141                      add    sp,sp,-16
    {
        if (HAL_GPIO_LineInterruptState(GPIO_LINE_2))
 2000174:    02000513           li     a0,32
{
 2000178:    c606                      sw     ra,12(sp)
        if (HAL_GPIO_LineInterruptState(GPIO_LINE_2))
 200017a:    ff000097           auipc  ra,0xff000
 200017e:    45c080e7           jalr   1116(ra) # 10005d6
<HAL_GPIO_LineInterruptState>
 2000182:    c519                      beqz   a0,2000190 <raw_trap_handler+0x38>
        {
            flag = !flag;
 2000184:    8201a783           lw     a5,-2016(gp) # 20001a8 <flag>
 2000188:    0017b793           seqz   a5,a5
 200018c:    82f1a023           sw     a5,-2016(gp) # 20001a8 <flag>
        }
        HAL_GPIO_ClearInterrupts();
 2000190:    ff000097           auipc  ra,0xff000
 2000194:    460080e7           jalr   1120(ra) # 10005f0 <HAL_GPIO_ClearInterrupts>
    }
    /* Сброс прерываний */
    HAL_EPIC_Clear(0xFFFFFFFF);
 2000198:    40b2                      lw     ra,12(sp)
 200019a:    000507b7           lui    a5,0x50
 200019e:    577d                      li     a4,-1
 20001a0:    40e7ac23           sw     a4,1048(a5) # 50418 <__stack_size+0x50018>
 20001a4:    0141                      add    sp,sp,16
 20001a6:    8082                      ret
```

Листинг, получившись заметно меньше по длине, уже не содержит участков кода для сохранения большого количества регистров общего назначения. Для возврата используется обычная инструкция ret. Также, рассматривая вход в подпрограмму, можно заметить, что компилятор не сохраняет содержимое регистра a4 перед загрузкой в него нового значения.

Различия в инструкциях RET и MRET

При входе в прерывание trap процессор сохраняет адрес возврата не в регистре x1 и не в стеке, как обычно, а в специальном регистре MEPC, описанном в документе The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10. Псевдоинструкция RET раскрывается как JALR x0,0(ra), где ra – регистр, содержащий адрес возврата. Она задает регистру-указателю адреса значение регистра ra. Инструкция же MRET задает регистру-указателю адреса значение MEPC. Таким образом, для выхода из подпрограммы обработчика прерывания trap следует использовать именно MRET.

Сохранение регистров при обработке прерывания

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Данная картинка взята из документа, посвященному описанию вызова подпрограмм в RISC-V (https://riscv.org/wp-content/uploads/2024/12/riscv-calling.pdf). Здесь: Caller – вызывающая подпрограмма; Callee – вызываемая подпрограмма (функция).

Как можно видеть, содержимое temporary-регистров t0-t6, а также регистров аргументов a0-a7 в случае нормального вызова подпрограммы сохраняются вызывающим. Однако как можно видеть из листинга обработчика прерываний с использованием raw_trap_handler с атрибутом interrupt, значения всех используемых регистров, даже тех, которые должен сохранять вызывающий в случае стандартного вызова подпрограммы, сохраняются при вызове прерывания. Напротив, такого не наблюдается, если обработчик прерывания не помечен атрибутом interrupt.

Вывод

Скорость обработки прерываний можно существенно увеличить, если в качестве функции обработчика прерывания использовать raw_trap_handler с атрибутом interrupt. Ввиду того, что raw_trap_handler, согласно linker-скрипту, должен лежать в секции .trap_text, данная функция должна быть помечена также атрибутом section(“.trap_text”). Для удобства использования рекомендуется пользоваться готовым макросом INT_ATTR из mik32_hal.h.

При использовании raw_trap_handler с макросом INT_ATTR компилятор сам сохраняет все используемые им регистры, соответственно, нужда в принудительном сохранении всех РОН отпадает.