SPARC V8
FPU (Floating-Point Unit) 실습

인하대학교 컴퓨터공학과

12211568 김관

12223682 정유진

12224422 오세훈

INDEX

OVERVIEW

본 내용은 「**SPARC 환경 실시간 운영체제 RTEMS 실습**」 교재를 기반으로,
GDB를 활용하여 SPARC 아키텍처를 보다 심층적으로 이해하고자 하는 목적으로 작성되었습니다.

RTEMS 실습 과정에서 실질적으로 도움이 되는 주요 내용만을 중심으로 구성하였으며,
실습에 직접 활용하기에 적합한 핵심 사항 위주로 다루고 있습니다.
따라서 SPARC 아키텍처 및 RTEMS 전반에 대한 보다 상세한 설명과 이론적 배경은
해당 교재를 참고해 주시기 바랍니다.

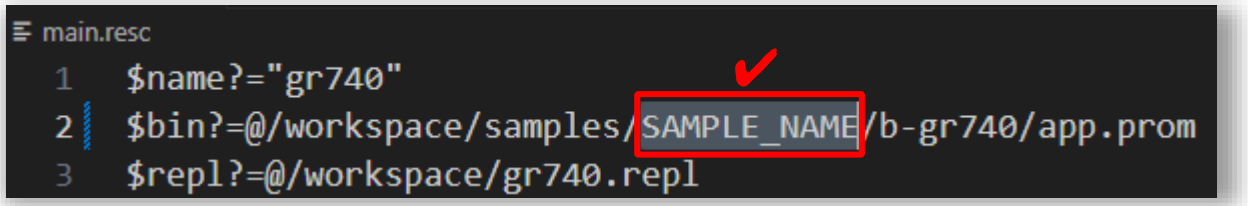
본 자료가 SPARC 기반 시스템의 동작 원리를 이해하고
디버깅 역량을 향상시키는 데 유용한 참고 자료가 되기를 바랍니다.

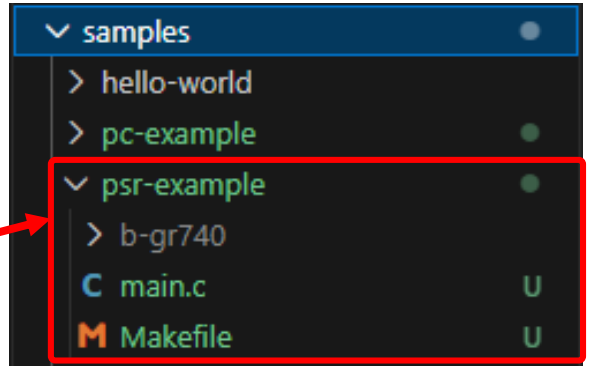
OVERVIEW

- **샘플 코드 적용하는 방법**
  - 샘플 코드 다운로드 (원하는 위치에서 cmd 또는 powershell 열기)

```
$ git clone https://github.com/kjude22/SPARC-V8-Samples.git
```

  - /workspace/samples 폴더에 샘플 폴더를 복사
    - 개별 샘플 폴더를 **main.resc 경로에 맞게 /samples 상위**에 넣도록
  - /workspace/main.resc 에서 경로 변경



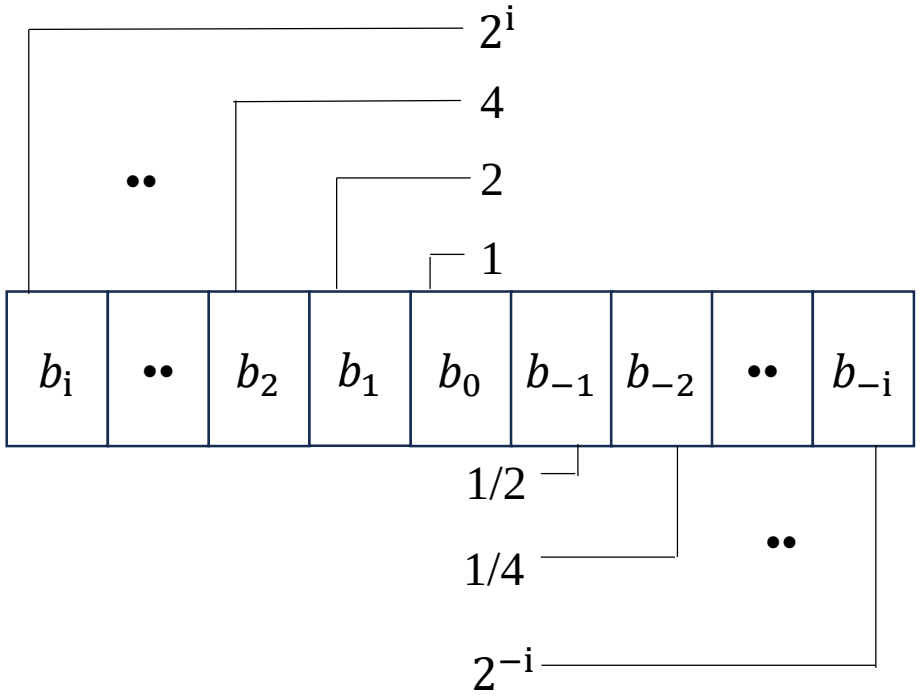


"Drag here to copy"

※ 자세한 내용 「**Renode GR740 설치 및 디버깅 환경설정**」 참고

IEEE 754

- **Fixed Point Representation**
  - 정수부와 소수부의 경계인 소수점의 위치를 특정 비트 지점에 고정하여 숫자를 표현하는 방식
  - Ex) $d_i d_{i-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-i}$



IEEE 754

- **Fixed Point Representation**
  - **Limitation #1**
    - $\frac{x}{2^k}$ 형태의 숫자만 정확하게 표현 가능 (x,k는 정수)
  - **Limitation #2**
    - 매우 크거나 작은 소수를 표현할 때 메모리 사용이 **비효율적**임
    - Ex) $\frac{1}{3}$ =0.0101010101010101 , $2^{16}-1$ =1111 1111 1111 1111

IEEE 754

- **IEEE Floating Point**
  - 지수와 가수를 사용하여 소수점의 위치를 유동적으로 변화시킴으로써, 매우 넓은 범위의 실수를 효율적으로 표현하는 표준 방식
  - **S**: 부호비트로, 0은 양수, 1은 음수를 나타낸다.
  - **M**: 가수로, 소수 부분을 의미하며, 정규화된 값이 되도록 이진법으로 표현된 소수를 시프트하여 [1.0,2.0)의 범위를 갖도록 한다.
  - **E**: 지수로, 2의 거듭제곱을 나타내며 숫자의 크기를 조정한다.

$$(-1)^S \text{x } M \text{ x } 2^E$$

IEEE 754

- **IEEE Floating Point**
  - **Single Precision** Floating Point Numbers (32 bits)

  - **Double Precision** Floating Point Numbers (64 bits)

EXAMPLE: IEEE 754

```
rtems_task Init(rtems_task_argument ignored)
{
  (void) ignored;
  TEST_BEGIN();
  volatile float f1 = 1.0f;
  volatile float f2 = 1.5f;
  volatile double d1 = 2.75;
  volatile double d2 = 5.25;
  TEST_END();
  exit(0);
}
```

```
(gdb) x /wt &f1
  0x2a0d0:      00111111100000000000000000000000

(gdb) x/wt &f2
  0x2a0d4:      00111111110000000000000000000000

(gdb) x/gt &d1
  0x2a0d8:      0100000000000110000000000000000
                0000000000000000000000000000000
(gdb) x/gt &d2
  0x2a0e0:      0100000000010101000000000000000
                0000000000000000000000000000000
```

normalized-example/main.c

IEEE 754

- **IEEE Floating Point**
  - Denormalized form : When the exponent field is all zeros

  - Infinity

  - NaN

EXAMPLE: IEEE 754

```
rtems_task Init(rtems_task_argument ignored)
{
  (void) ignored;
  TEST_BEGIN();
  volatile double d = 0.0;
  volatile double d1 = INFINITY;
  volatile double d2 = -INFINITY;
  volatile double d3 = NAN;
  volatile double d4 = -NAN;
  volatile double d5 = 1.0/0.0;
  volatile double d6 = sqrt(-1);
  TEST_END();
  exit(0);
}
```

denormalized-example/main.c

```
(gdb) x /gt &d
  0x269d0:      0000000000000000000000000000000000000000000000000000000000000000
(gdb) x /gt &d1
  0x269d8:      0111111111110000000000000000000000000000000000000000000000000000
(gdb) x /gt &d2
  0x269e0:      1111111111110000000000000000000000000000000000000000000000000000
(gdb) x /gt &d3
  0x269e8:      0111111111111000000000000000000000000000000000000000000000000000
(gdb) x /gt &d4
  0x269f0:      1111111111111000000000000000000000000000000000000000000000000000
(gdb) x /gt &d5
  0x269f8:      0111111111110000000000000000000000000000000000000000000000000000
(gdb) x /gt &d6
  0x26a00:      0111111111111111111111111111111111111111111111111111111111111111
```

f Registers

- SPARC FPU는 32개의 32-bit f 레지스터(f0 ~ f31)를 가진다.
- SPARC FPU는 기본 단위가 32비트이며, 큰 정밀도의 자료형은 여러 레지스터를 묶어서 표현한다.

자료형	크기	사용 레지스터 수	IEEE 754 비트 구성
Single	32-bit	1개	1 + 8 + 23
Double	64-bit	2개	1 + 11 + 52
Quad	128-bit	4개	1 + 15 + 112

- 정확한 해석과 메모리 접근 효율을 위해 시작 레지스터 번호가 제한된다.
  - Double Precision: 반드시 짝수 번호 레지스터에서 시작
  - Quad Precision: 반드시 4의 배수 번호 레지스터에서 시작

EXERCISE 1: IEEE 754

- 1. 다음 빈칸에 들어갈 값을 작성하시오.

```
rtems_task Init(rtems_task_argument
ignored)
{
  (void) ignored;
  TEST_BEGIN();
  volatile float f = 2.5f;
  volatile double d = (1)      ;
  TEST_END();
  exit(0);
}
```

```
(gdb) x/wt &f
  0x2667c:
  (2)
(gdb) x/gt &d
  0x26680:
  0100000000000110000000000000000000
  0000000000000000000000000000000000
```

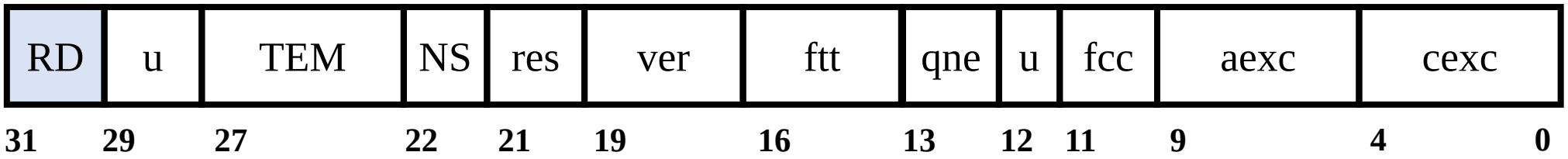
EXERCISE 1: IEEE 754

- **문제 해설**
  - (1) 2.75
    - 부호 (Sign): 맨 앞 비트가 0이므로 양수.
    - 지수 (Exponent): 10000000000 (1024_{10}) 실제 지수 = 1024 - 1023(Bias) =1
    - 가수 (Mantissa): 비트 패턴 상위가 011이다. 정규화된 형태는 1.011_2
    - 최종 계산: 1.011_2 x 2^1 = 10.11_2
  - (2) 01000000001000000000000000000000
    - 정규화 1.01_2 x 2^1
    - 부호 (Sign): 양수이므로 0
    - 지수 (Exponent): 1 + 127(Bias) = 128 ->10000000
    - 가수 (Mantissa): 소수점 아래 01을 적고 나머지는 0으로 채움 -> 01000000001000000000000000000000

FLOATING-POINT STATE REGISTER (FSR)

- **FSR (Floating-Point State Register)**

  FSR은 FPU의 상태와 모드 정보를 담고 있는 32비트 핵심 레지스터이다.



- **RD** (Rounding Direction, 2 bits)
  - ANSI/IEEE 754-1985 표준에 따라 **반올림 방향**을 결정함
  - 기본적으로 **Round-to-Nearest** 모드 (중간이면 **Even**)

RD	Mode
0	Round-to-Nearest (Even, if tie)
1	Round-toward-zero
2	Round-down
3	Round-up

FLOATING-POINT STATE REGISTER (FSR)

- Floating Point Operations: Basic Idea
  - $x +_f y = Round(x + y)$
  - $x \times_f y = Round(x \times y)$

  - **Round-to-Nearest**
    - 가장 가까운 값으로 반올림하는 방식
    - 정확히 중간일 때는 **Round-to-Even,** 가장 낮은 자리수가 짝수가 되도록 반올림

Example)

	Decimal	Binary	Rounded Binary	Condition	Rounded Decimal
	2 + 3/32	10.00011_2	10.00_2	(< ½) Down	2
	2 + 3/16	10.00110_2	10.01_2	(< ½) Up	2 + ¼
Half-way cases	2 + 7/8	10.11100_2	11.00_2	(½) Up	3
	2 + 5/8	10.10100_2	10.10_2	(½) Down	2 + ½

EXAMPLE: ROUNDING DIRECTION (1)

```
rtems_task Init(
  rtems_task_argument ignored
)
{
   /* Round-to-nearest even (default) */

   volatile float a = 1.5;
   volatile float b = -1.5;
   volatile float r1 = nearbyint(a);
   volatile float r2 = nearbyint(b);

   ...
}
```

fsr-example-1/main.c

```
(gdb) break Init
(gdb) c                                              // continue
  ⋮
(gdb) x /wf &a                                       // examine
  0x268b0:      1.5          1
(gdb) x /wt &a
  0x268b0:      00111111110000000000000000000000

(gdb) x /wf &b
  0x268b4:      -2.5
(gdb) x /wt &b
  0x268b4:      11000000001000000000000000000000

(gdb) x /wt &r1
  0x268b8:      01000000000000000000000000000000     // 2
(gdb) x /wt &r2
  0x268bc:      11000000000000000000000000000000     // -2
```

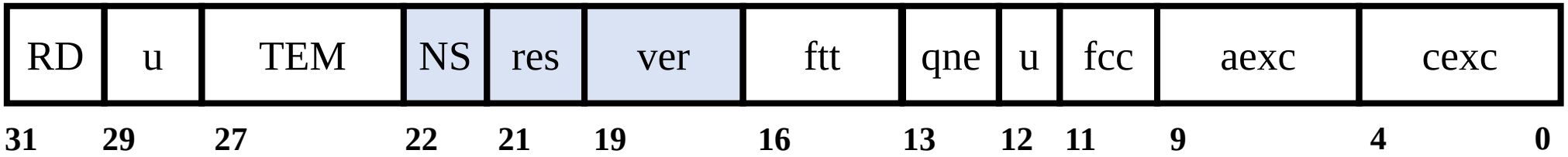
EXAMPLE: ROUNDING DIRECTION (2)

```
rtems_task Init(
  rtems_task_argument ignored
)
{
    /* Round-to-nearest even (default) */
    volatile float a = 1.5;
    volatile float b = -2.5;
    ...

    /* Round-toward-zero */
    sparc_set_rounding_mode(1);
    volatile float r3 = nearbyint(a);
    volatile float r4 = nearbyint(b);
    ...
}
```

fsr-example-1/main.c

```
  ⋮

(gdb) info register fsr
fsr            0x80000          [ ]                    // RD : 0 (Round-to-Nearset)

  ⋮

(gdb) info register fsr
fsr            0x40080020       [ NXC ]                // RD : 1 (Round-to-zero)

(gdb) x /wt &r3
0x268c0:       00111111100000000000000000000000       //  1.0

(gdb) x /wt &r4
0x268c4:       11000000000000000000000000000000       // -2.0
```

FLOATING-POINT STATE REGISTER (FSR)

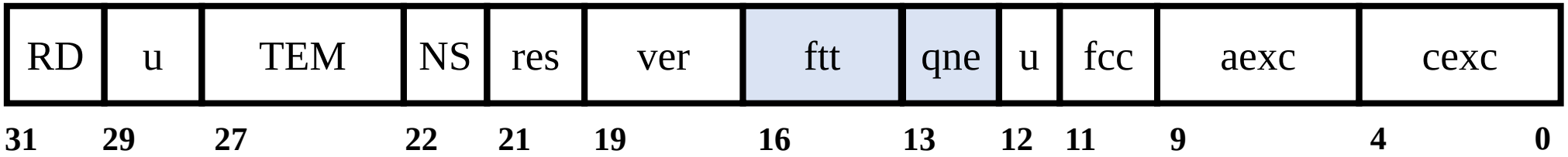


- **NS** (Non-Standard, 1 bits)
  - ANSI/IEEE 규격을 따르는지 나타냄
  - 1일 경우 이를 따르지 않음

- **res** (Reserved, 2 bits)
  - 현재까지 예약되어 사용되지 않음
  - 향후 호환성을 위해 이 값은 0으로 초기화되어 있어야 함

- **ver** (Version, 3 bits)
  - FPU 구현을 정의하는 필드로, 하드웨어 의존적으로 제작사에서 값을 고정적으로 입력함

FLOATING-POINT STATE REGISTER (FSR)



- **ftt** (Floating-Point Trap Type, 3 bits)
  - 트랩 유형을 식별하는 필드
  - 예외 발생 시, 예외 유형을 인코딩함

- **qne** (FQ Not Empty, 1 bit)
  - 현재 Renode에 구현된 가상 하드웨어 상에는
    FQFloating-Point Deffered-Trap Queue가 구현되어 있지않아 항상 0임

FLOATING-POINT STATE REGISTER (FSR)

- **ftt** (Floating-Point Trap Type, 3 bits)
  - IEEE_754_exception
    - ANSI/IEEE 표준에 준수하는 예외 발생
    - 예외 유형은 **cexec**에 인코딩됨
  - Unfinished_FPop
    - FPU가 ANSI/IEEE 표준의 정의대로 결과 혹은 예외를 생성할 수 없음
  - Unimplemented_FPop
    - FPU에 구현되지 않은 부동소수점 연산이 디코딩됨
  - Sequence_error
    - FQ가 없거나 비어있는 구현에서 STDFQ(큐 읽기) 사용 시도할 때 발생
    - FPU가 명령을 받아들일 준비가 되지 않았을 때 명령 실행을 시도할 때 발생

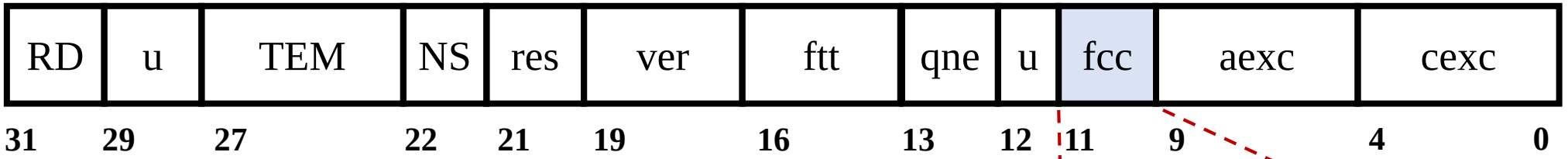
ftt	Trap Type
0	None
1	IEEE_754_exception
2	Unfinished_FPop
3	Unimplemented_FPop
4	Seqeuence_error
5	Hardware_error
6	Invalid_fp_register
7	Reserved

Floating-point State Register (FSR)

- Hardware_error
  - FPU가 Illegal State 또는 f 레지스터 접근 시 패리티 오류와 같은 치명적인 내부 오류를 감지할 때 발생
- Invalid_fp_register
  - 하나 이상의 피연산자operand가 잘못 정렬됨
  - 구현에 따라 트랩을 발생시키지 않을 수 있음

ftt	Trap Type
0	None
1	IEEE_754_exception
2	Unfinished_FPop
3	Unimplemented_FPop
4	Seqeuence_error
5	Hardware_error
6	Invalid_fp_register
7	*Reserved*

FLOATING-POINT STATE REGISTER (FSR)



- **fcc** (Floating-Point Condition Codes, 2 bits)
  - 부동소수점 비교 연산의 상태를 나타내는 플래그
  - FBfcc와 같은 분기Branch 명령어는 이 필드에 기반하여 제어 전달을 수행함

fcc	Relation
0	$f_{rs1} = f_{rs2}$
1	$f_{rs1} < f_{rs2}$
2	$f_{rs1} > f_{rs2}$
3	f_{rs1} ? f_{rs2} (unordered)

EXAMPLE: BRANCH BASED ON THE FCC FLAG

```
rtems_task Init(
  rtems_task_argument ignored
)
{
    volatile float a = 1.0f;
    volatile float b = 2.0f;
    volatile int taken = 0;

    if (a >= b) {
        taken = 0;
    } else {
        taken = 1;
    }
   …
}
```

fsr-example-2/main.c

```
(gdb) break Init
(gdb) c
(gdb) disassemble
  ⋮
  0x00001270 <+24>:    ld  [ %g1 + 0x268 ], %f8
  0x00001274 <+28>:    st  %f8, [ %fp + -12 ]          // a 로드/저장
  0x00001278 <+32>:    sethi  %hi(0x20400), %g1
  0x0000127c <+36>:    ld  [ %g1 + 0x26c ], %f8
  0x00001280 <+40>:    st  %f8, [ %fp + -8 ]           // b 로드/저장
  0x00001284 <+44>:    clr  [ %fp + -4 ]
  0x00001288 <+48>:    ld  [ %fp + -12 ], %f9
  0x0000128c <+52>:    ld  [ %fp + -8 ], %f8
  0x00001290 <+56>:    fcmpes  %f9, %f8                // a와 b 비교 후 fcc 설정
  0x00001294 <+60>:    nop
  0x00001298 <+64>:    fbul  0x12cc <Init+116>         // fcc 기반하여 분기
  ⋮
  0x000012cc <+116>:   mov  1, %g1                     // else
  0x000012d0 <+120>:   st  %g1, [ %fp + -4 ]
  0x000012d4 <+124>:   b  0x12a8 <Init+80>
  ⋮
```

EXERCISE 2: ROUNDING DIRECTION

- **2. 다음 세 개의 빈칸에 들어갈 값을 작성하시오.**

```
rtems_task Init(
  rtems_task_argument ignored
)
{
   volatile float a = 3.5;
   volatile float b = -1.5;
   volatile float r1 = nearbyint(a);
   volatile float r2 = nearbyint(b);

   sparc_set_rounding_mode(2);
   volatile float r3 = nearbyint(a);

   sparc_set_rounding_mode(3) ___or___ );
   volatile float r4 = nearbyint(b);
}
```

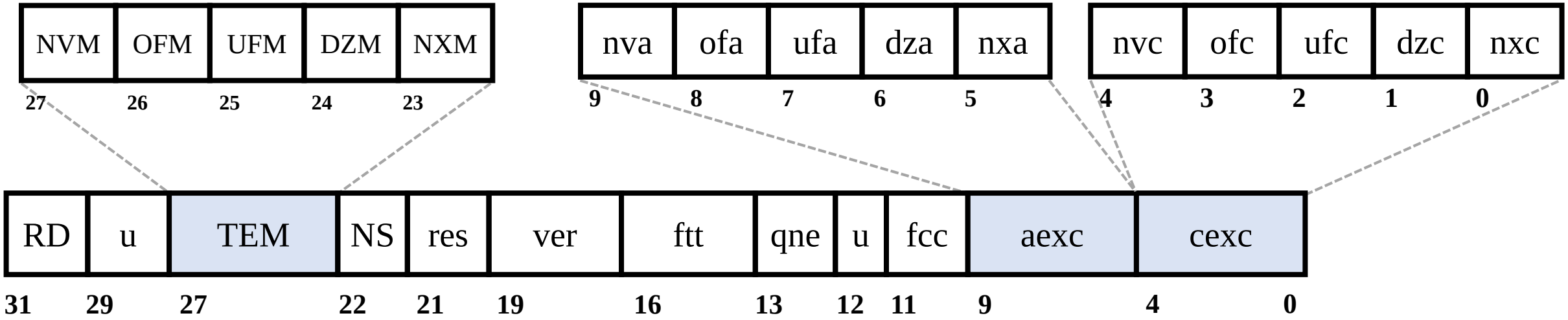
```
(gdb) break Init
   ⋮
(gdb) x /wf &r1
0x268b0:        4
(gdb) x /wf &r2
0x268b4:  1)_____

(gdb) x /wf &r3
0x268c0:  2)_____

(gdb) x /wf &r4
0x268c4:       -1
```

EXERCISE 2: ROUNDING DIRECTION

- **문제 해설**
  - 1) -2
    라운딩 모드의 디폴트 값는 0 Round-to-Nearest 이다.
    따라서 -1.5에 이를 적용하면 -2로 반올림된다.
  - 2) 2
    라운딩 모드가 2이면 Round-down이다.
  - 3) 1 or 3
    라운딩 모드가 1일때는 소수점 아래가 제거되고,
    라운딩 모드가 3일때는 Round-up이다.

RD	Mode
0	Round-to-Nearest (Even, if tie)
1	Round-toward-zero
2	Round-down
3	Round-up

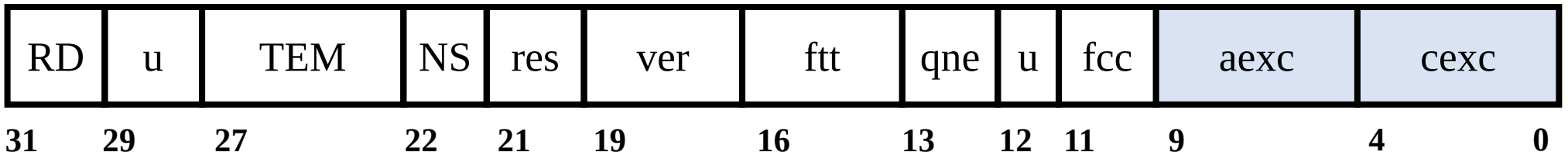
FLOATING-POINT EXCEPTION FIELDS



- **트랩 활성화 마스크(TEM), 누적 예외 필드(aexc), 현재 예외 필드(cexc)**
  - ANSI/IEEE 표준 754-1985에 따른 5가지 예외를 인코딩함
  - 각 비트는 FSR_invalid, FSR_overflow , FSR_underflow , FSR_division-by-zero, FSR_inexact 예외를 의미

FLOATING-POINT EXCEPTION FIELDS

FSR

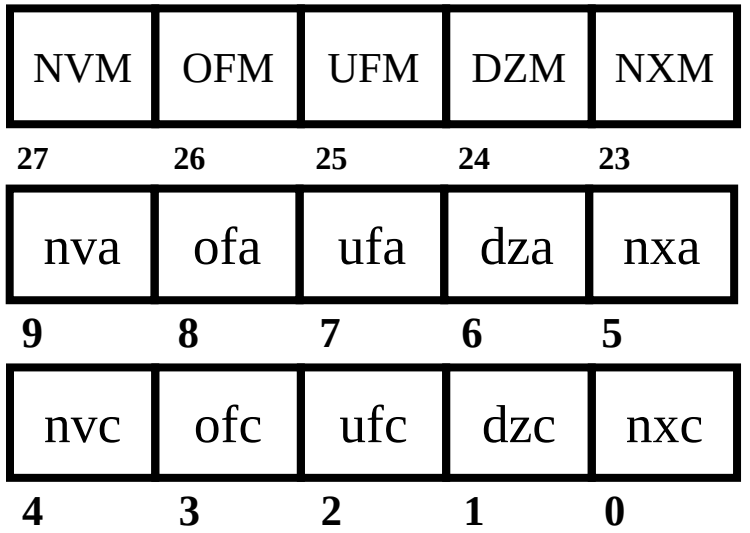


- **FSR_accured_exception(aexc)**
  - 누적 예외 필드(9~5번째 비트)
  - **fp_exception_trap이 비활성화된 동안** 예외를 누적한다
    → TEM에서 해당 예외 비트가 0이라서, 예외가 발생해도 트랩(fp_exception_trap)이 걸리지 않는 상태

- **FSR_current_exception(cexc)**
  - 현재 예외 필드(4~0번째 비트)
  - 가장 최근에 실행된 FPop 명령어에 의해 하나 이상의 예외가 생성되었음을 나타냄

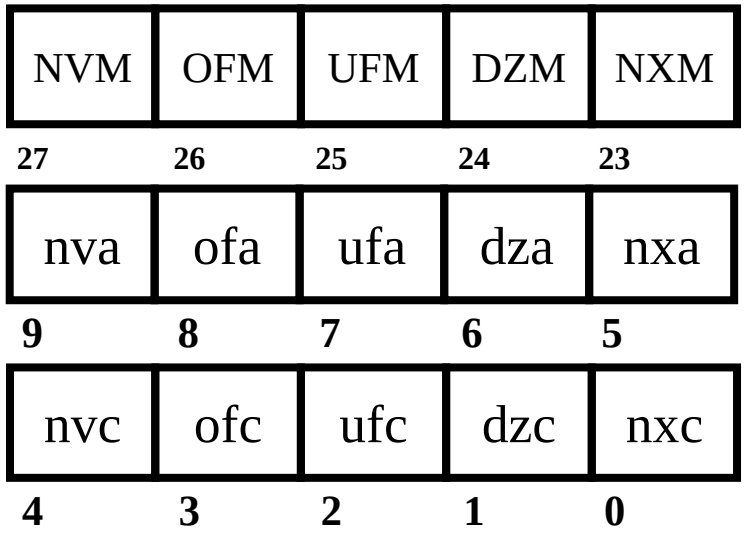
IEEE-754 Five Floating-Point Exceptions

- **FSR_invalid** (NVA, NVC)
  - $0/0$, $\infty - \infty$ 와 같이 피연산자가 연산에 적합하지 않은 경우
  - 1: invalid, 0: valid
- **FSR_overflow** (OFA, OFC)
  - 반올림 결과가 가장 큰 정규화 수보다 큰 경우
  - 1: overflow, 0: X
- **FSR_underflow** (UFA, UFC)
  - 반올림 결과가 정확하지 않으며, 가장 작은 정규화 수보다 작은 경우
  - 1: underflow, 0: X



IEEE-754 Five Floating-Point Exceptions

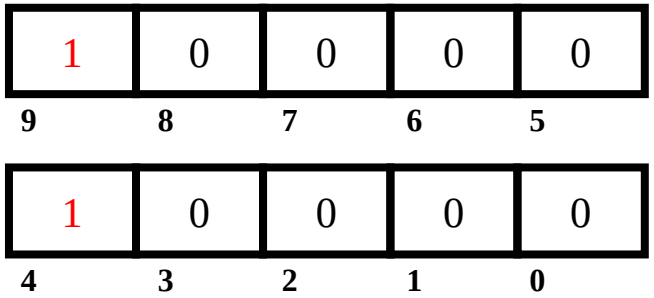
- **FSR_division-by-zero** (DZA, DZC)
  - $X/0$의 형태의 연산을 하는 경우 (x는 subnormal 혹은 정규화)
  - $0/0$은 제외한다
- **FSR_inexact** (NXA, NXC)
  - 반올림 결과가 무한히 정확한 올바른 결과와 다른 경우
  - 2진 부동소수점으로 정확히 표현이 어려운 경우
  - 0.1, $\sqrt{2.0}$



EXAMPLE: CHECK CHANGES IN CEXC AND AEXC FIELDS

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  ⋮
  volatile double a = 0.0;
  volatile double b = 1.0;
  volatile double c = 0.0;

  volatile double res_invalid = a / c;
  volatile double res_zero = b / c;
}
```

fsr-example-3/main.c



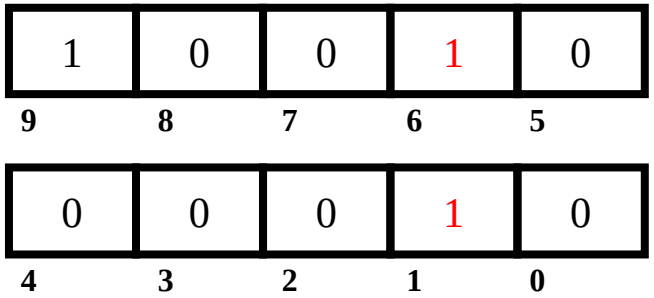

```
(gdb) info register fsr
fsr            0x80000                                    // 초기상태

(gdb) disassemble Init
Dump of assembler code for function Init:
   ⋮
   0x00001288 <+48>:    ldd  [ %fp + -40 ], %f8
   0x0000128c <+52>:    ldd  [ %fp + -24 ], %f10
   0x00001290 <+56>:    fdivd  %f8, %f10, %f8
```

```
(gdb) info register fsr
fsr            0x80210
(gdb) p/t (($fsr >> 23) & 0x1f)                         // TEM 필드 확인(27~23)
$1 = 0
(gdb) p/t (($fsr >> 5) & 0x1f)                          // aexc 필드 확인(9~5)
$2 = 10000
(gdb) p/t ($fsr & 0x1f)                                 // cexc 필드 확인(4~0)
$3 = 10000

// 0.0/0.0을 수행한 결과 TEM 비트는 설정하지 않은 채 FSR_invalid 에러를 반영한다.
```

EXAMPLE: CHECK CHANGES IN CEXC AND AEXC FIELDS

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  ⋮
  volatile double a = 0.0;
  volatile double b = 1.0;
  volatile double c = 0.0;

  volatile double res_invalid = a / c;
  volatile double res_zero = b / c;
}
```

fsr-example-3/main.c

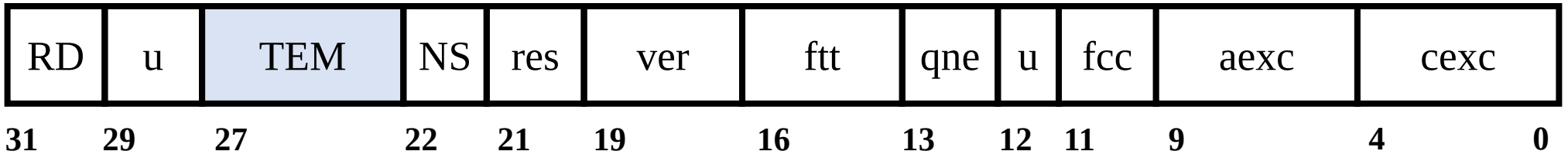



```
(gdb) disassemble Init
Dump of assembler code for function Init:
   ⋮
   0x000012ac <+84>:    ldd  [ %fp + -32 ], %f8
   0x000012b0 <+88>:    ldd  [ %fp + -24 ], %f10
   0x000012b4 <+92>:    fdivd  %f8, %f10, %f8
```

```
(gdb) info register fsr
fsr            0x80242
(gdb) p/t (($fsr >> 23) & 0x1f)                    // TEM 필드 확인(27~23)
$1 = 0
(gdb) p/t (($fsr >> 5) & 0x1f)                     // aexc 필드 확인(9~5)
$2 = 10010
(gdb) p/t ($fsr & 0x1f)                            // cexc 필드 확인(4~0)
$3 = 10

// FSR_invalid 예외가 발생한 후, FSR_division-by-zero가 발생했을 때
// aexc은 누적된 예외 비트를 저장하고 cexc는 가장 최신 명령어의 예외 비트를 담는다
```

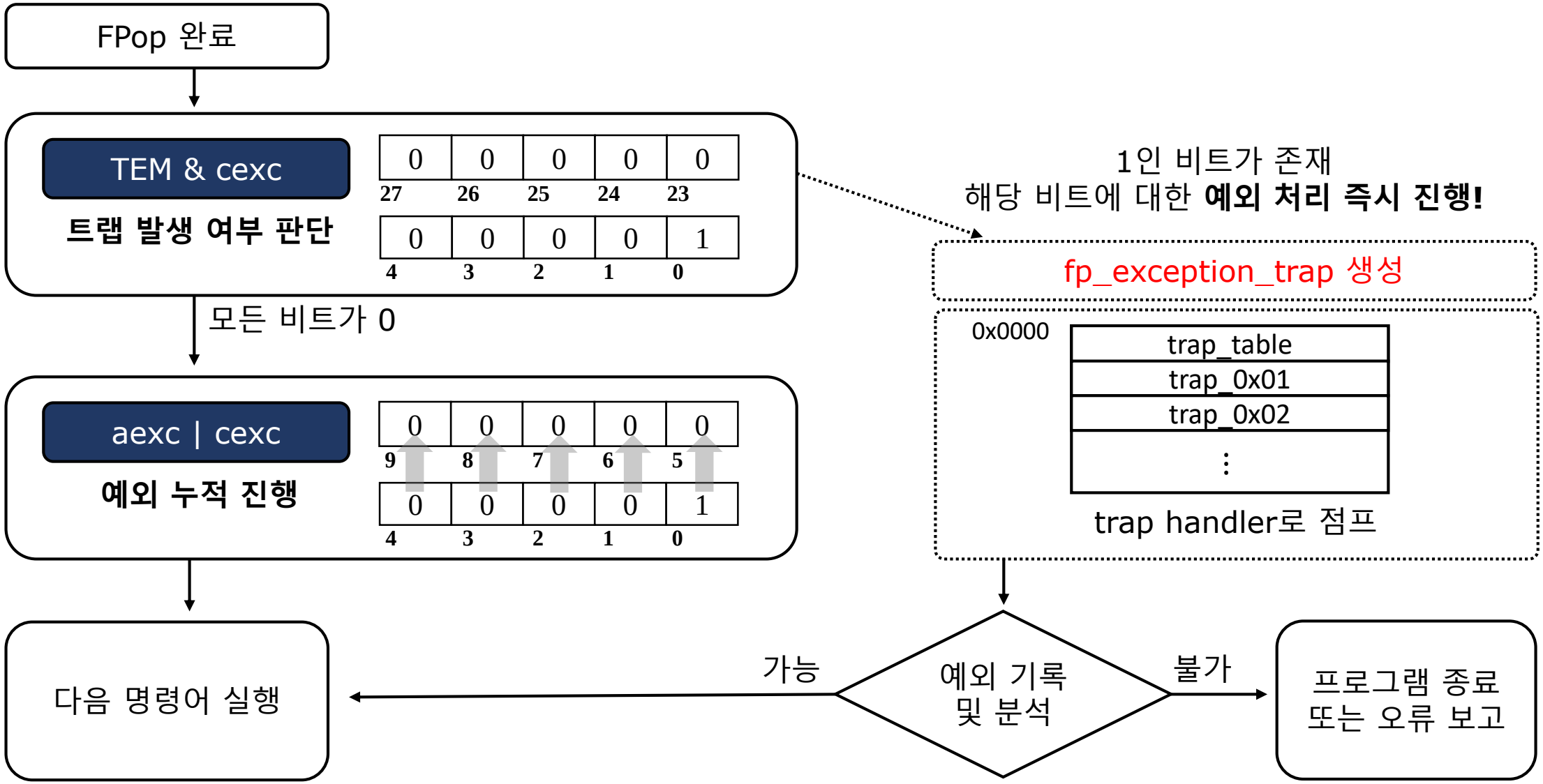
FLOATING POINT EXCEPTION FIELDS

FSR



- **FSR_trap_enable_mask (TEM)**
  - 트랩 활성화 마스크(27~ 23번째 비트)
  - cexc에 표현 가능한 5가지 부동 소수점 예외에 대한 활성화 비트
  - 하나 이상의 예외가 발생함과 동시에, 해당 TEM 비트가 1이면 fp_exception이 발생함
  - FPop이 완료된 이후, TEM와 cexc 필드는 and 연산을 진행하고 결과가 1이면 트랩을 생성한다
  - TEM && cexc의 결과가 0이라면 cexc 필드가 aexc필드로 or 연산되어 에러를 누적한다

EXCEPTION REFLECTION FLOW



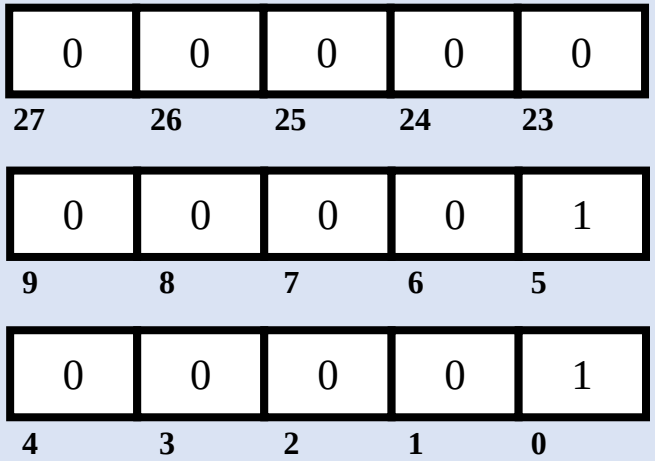
EXAMPLE: UNDERSTANDING THE ROLE OF TEM

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  ⋮
  volatile double x = 1.0;
  volatile double y = 3.0;

  volatile double res_no_trap = x / y;
  volatile double res_trap = x / y;
}
```

fsr-example-4/main.c

```
(gdb) next
volatile double res_trap = x / y;

(gdb) info register fsr
fsr            0x80021
(gdb) p/t (($fsr >> 23) & 0x1f)
$1 = 0
(gdb) p/t (($fsr >> 5) & 0x1f)
$2 = 1
(gdb) p/t ($fsr & 0x1f)
$3 = 1
```

// TEM 설정 없이 에러 반영할 경우




```
// res_trap 계산 전 fsr 레지스터 재설정: 초기화 + TEM의 NXM 비트 1로 세팅
(gdb) set $fsr = 0x80000
(gdb) set $fsr = $fsr | (1<<23)

(gdb) info register fsr
fsr            0x880000
```

EXAMPLE: UNDERSTANDING THE ROLE OF TEM

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  ⋮
  volatile double x = 1.0;
  volatile double y = 3.0;

  volatile double res_no_trap = x / y;
  volatile double res_trap = x / y;
}
```

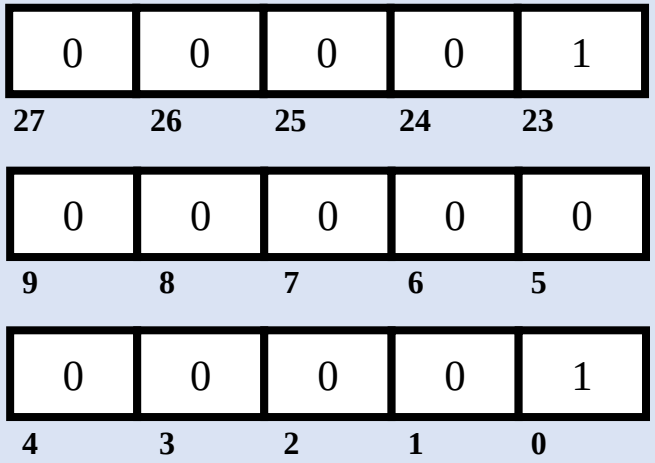
fsr-example-4/main.c

```
(gdb) disassemble Init
Dump of assembler code for function Init:
   ⋮
   0x00001298 <+64>:    ldd  [ %fp + -32 ], %f8
   0x0000129c <+68>:    ldd  [ %fp + -24 ], %f10
=> 0x000012a0 <+72>:    fdivd  %f8, %f10, %f8
```

// NXM 비트를 1로 설정한 후 나눗셈을 다시 수행하면 trap이 발생한다

```
(gdb) info register pc
pc             0x136e0          0x136e0 <syscall>

(gdb) info register fsr
fsr            0x884001
(gdb) p/t (($fsr >> 23) & 0x1f)
$4 = 1
(gdb) p/t (($fsr >> 5) & 0x1f)
$5 = 0
(gdb) p/t ($fsr & 0x1f)
$6 = 1
```

// TEM 설정 후 에러 반영할 경우



EXERCISE 3: PREDICTING AEXC AND CEXC FIELDS

- 3. 다음 빈칸에 들어갈 값을 작성하시오.

```
/* C 코드 */
rtems_task Init(
  rtems_task_argument ignored
)
{
  ⋮
  volatile double a = 0.0;
  volatile double b = 1.0;
  volatile double c = 0.0;

  volatile double res_invalid = a / c;
  volatile double res_zero = b / c;

   printf("res_invalid: %f\n", res_invalid);
   printf("res_zero: %f\n", res_zero);
}
```

```
/* SPARC 어셈블리 코드 */
(gdb) next
volatile double res_zero = b / c;

(gdb) info register fsr
fsr            0x80210

(gdb) next
printf("res_invalid: %f\n", res_invalid);

(gdb) p/t (($fsr >> 5) & 0x1f)
$1 = _____
(gdb) p/t ($fsr & 0x1f)
$2 = _____
```

EXERCISE 3: PREDICTING AEXC AND CEXC FIELDS

- **문제 해설**
  - 정답: 10010, 10(00010)
  - volatile double res_invalid = a / c; 이 실행된 후, fsr은 invalid 에러에 대한 정보를 담고 있다.

기본적으로 TEM의 값은 전부 0이기 때문에, nva 비트와 nvc 비트가 1로 세팅된다

<aexc>

1	0	0	0	0
9	8	7	6	5

<cexc>

1	0	0	0	0
4	3	2	1	0

- volatile double res_zero = b / c; 를 실행한 후에는 division-by-zero 에러에 대한 정보를 반영한다.

aexc는 dza 비트를 누적하고, cexc는 가장 최신 명령어가 발생시킨 dzc 비트만을 1로 설정한다.

<aexc>

1	0	0	1	0
9	8	7	6	5

<cexc>

0	0	0	1	0
4	3	2	1	0

EXERCISE 4: PREDICTING EXCEPTION FIELDS

- **4. 예외 필드 판단 문제**
  - 트랩 활성화 마스크(TEM), 현재 예외 필드(cexc), 누적 예외 필드(aexc)는 5가지 예외를 인코딩한다.

다음의 SPARC 어셈블리 코드를 참고하여 예외 발생 시 필드의 변화 과정을 이해한다.

```
/* C 코드 */
rtems_task Init(
  rtems_task_argument ignored
)
{
    volatile double a = 1e308;
    volatile double b = 1e308;

    volatile double res = a * b;
}
```

```
/* SPARC 어셈블리 코드 */
(gdb) set $fsr = $fsr | (1<<26)  ..... ①
(gdb) set $fsr = $fsr | (1<<23)  ..... ①

(gdb) disassmeble Init
Dump of assembler code for function Init:
   :
   0x0000127c <+36>:    ldd  [ %fp + -24 ], %f8
   0x00001280 <+40>:    ldd  [ %fp + -16 ], %f10
=> 0x00001284 <+44>:    fmuld  %f8, %f10, %f8  ..... ②
```

fsr-example-5/main.c

EXERCISE 4: PREDICTING EXCEPTION FIELDS

- 4.1 ①번을 작성하지 않고 수행한다고 했을 때, FSR은 어떤 값을 가지고 있을지 예측하시오.

- 4.2 ②번 명령어를 실행한 직후 FSR을 살펴본 결과 다음과 같았다. TEM, aexc, cexc이 각각 저장하고 있는 값을 작성하고, 이 값을 통해 알 수 있는 점을 서술하시오.

```
(gdb) info register fsr
fsr            0x4884009
```

EXERCISE 4: PREDICTING EXCEPTION FIELDS

- **문제 해설(4.1)**
  - 정답: 0x80129
  - ①번은 fsr의 26번, 23번 비트를 1로 세팅하는 명령이다. 이는 예외에 대한 활성화 비트(TEM)이며, 값이 1로 세팅되었을 경우에만 fp_exception이 발생한다.
  - TEM의 값이 0인 경우, 즉 트랩이 비활성화 된 경우에만 aexc에 예외가 누적된다.
  - ①번 명령을 수행하지 않는다면 TEM은 전부 0의 값을 가질 것이고, aexc에 예외가 누적될 것이며 cexc에는 최근에 실행된 Fpop 명령어에 대한 모든 예외 비트가 켜질 것이다.

<TEM>

0	0	0	0	0
27	26	25	24	23

<aexc>

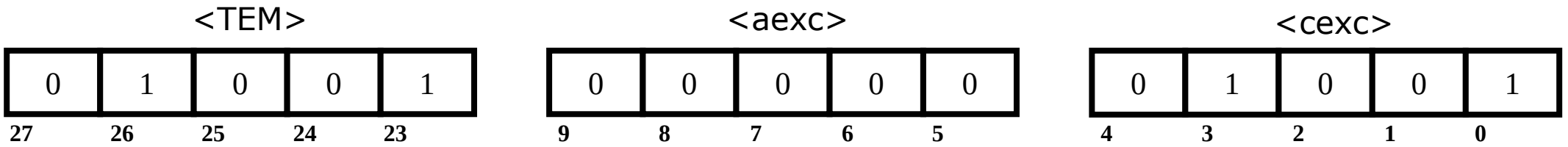
0	1	0	0	1
9	8	7	6	5

<cexc>

0	1	0	0	1
4	3	2	1	0

EXERCISE 4: PREDICTING EXCEPTION FIELDS

- **문제 해설(4.2)**
  - FSR이 0x4884009라는 값을 가진다. 이를 TEM, aexc, cexc에 해당하는 비트로 분류해 해석하면 다음과 같이 나올 것이다.



  - TEM은 0x9, aexc은 0x0, cexc는 0x9의 값을 가진다.
  - 즉 overflow, inexact 트랩이 활성화 되어있으며, 실제로 fmuld 연산 수행 후 overflow, inexact 예외가 함께 발생했다는 사실을 확인할 수 있다. TEM이 세팅 되어 있으므로 이후 fp_exception 상태로 진입할 것이다.