SPARC V8
IU (Integer 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.

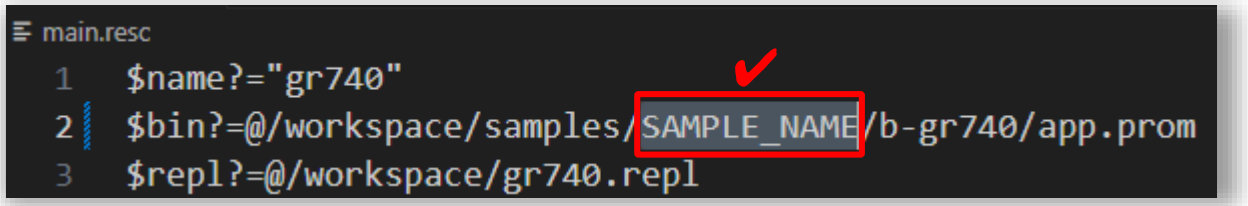
본 자료가 SPARC 기반 시스템의 동작 원리를 이해하고
디버깅 역량을 향상시키는 데 유용한 참고 자료가 되기를 바랍니다.

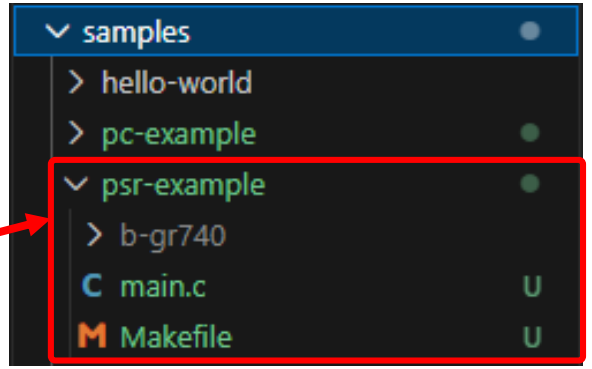
OVERVIEW

- **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

BASIC STRUCTURE

SPARC 프로세서는 크게 **IU**, **FPU**, **CP**로 구성된다.

IU **Interger Unit** 는 SPARC V8에서 **정수 연산**을 담당하는 유닛으로 가장 핵심적인 역할을 수행한다.

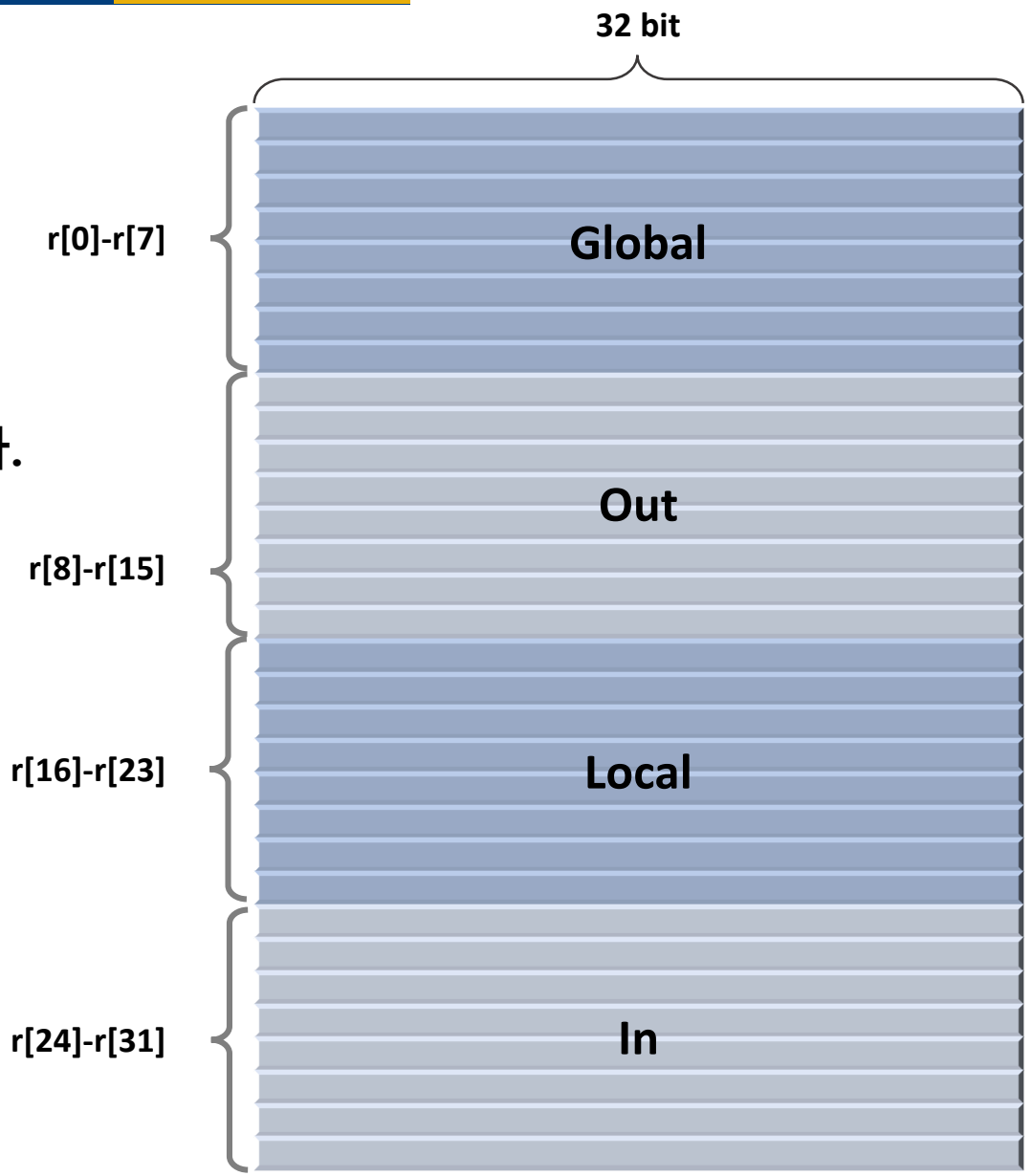
IU의 **범용 레지스터**는 **r 레지스터** **r Registers** 이다.

IU의 **제어/상태 레지스터** **IU Control/Status Registers** 에는 다음 레지스터들이 포함된다.

- **PSR** (Processor State Register)
- **WIM** (Window Invalid Mask Register)
- **TBR** (Trap Base Register)
- **Y** (Multiply/Divide Register)
- **PC**, **nPC** (Program Counters)
- **ASR** (Ancillary State Registers)

IU R REGISTER

- **IU r Registers**
  - IU r 레지스터는 구현에 따라 **40개에서 520개 사이**로 개수가 결정되는 **범용 32비트 레지스터**이다.
  - SPARC 아키텍처에서 실행 중인 프로그램은 **어느 시점에서든** 총 32개의 r 레지스터에 **접근할 수 있다.**
  - 32개의 레지스터들은 **윈도우 구조**를 형성하기 위해 각각 8개씩 **global, in, local, out** 레지스터로 분류된다.



IU R REGISTER

- **IU r Registers**
  - **global 레지스터** `%g0-%g7`

    현재 윈도우에 영향을 받지 않는 독립적인 전역 변수 공간
  - **out 레지스터** `%o0-%o7`

    호출자(Caller)가 피호출자(Callee)에게 인자를 전달하고 리턴 주소를 저장하는 공간
  - **in 레지스터** `%i0-%i7`

    피호출자(Callee)가 전달받은 인자를 읽는 공간
  - **local 레지스터** `%l0-%l7`

    현재 함수(윈도우) 내부에서만 사용하는 고유한 지역 변수 공간

REGISTER WINDOW

레지스터 윈도우 Register Window 는 **함수 호출 및 복귀**에 드는
비용을 줄이기 위해 고안된 **하드웨어 매커니즘**이다.

함수가 사용할 지연변수나 매개변수를 **메모리 스택이 아닌**
프로세서 내부 레지스터에 유지함으로써 **함수 호출 오버헤드**를 **최소화**한다.

프로그램은 24-레지스터 윈도우에 8개의 global 레지스터를 더하여
총 32개의 레지스터에 접근할 수 있다.

16-register set = **8-in register** + **8-local register**

24-register window = **16-register set** + **8-out register**

32-register visible to program = **24-register window** + **8-global register**

EXAMPLE: R REGISTERS

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  volatile int a = 10;
  volatile int b = 3;
  volatile int c = 4;
  volatile int result =
              compute_add_sub(a, b, c);
  printf("IU add/sub result = %d\n", result);
  exit(0);
}
```

```
static volatile int compute_add_sub
(int a, int b, int c)
{
  volatile int r1 = a + b;
  volatile int r2 = r1 - c;
  volatile int r3 = r2 + 7;
  volatile int r4 = r3 - 5;
  volatile int r5 = r4 + a;
  return r5;
}
```

r-register_example/main.c

```
(gdb) break Init
(gdb) continue                                  // c
(gdb) disassemble Init
  ⋮
  0x00001260 <+8>:    mov  0xa, %g1             // Init 내부의 모든 함수에서 사용되는
  ⋮                                             // 전역 변수는 %g에 저장됨
  0x00001268 <+16>:   mov  3, %g1
  ⋮
  0x00001270 <+24>:   mov  4, %g1
  ⋮
  0x00001284 <+44>:   mov  %g3, %o2             // 피호출자 함수에 넘겨줄 인자는
  0x00001288 <+48>:   mov  %g2, %o1             // %o을 통해 윈도우로 전달됨
  0x0000128c <+52>:   mov  %g1, %o0
  0x00001290 <+56>:   call  0x12c8 <compute_add_sub>
  ⋮
```

```
(gdb) info registers
g0          0x0             0          // %g, %o, %l, %i 각각 8개씩 존재
g1          0x1258          4696
g2          0xa010001       167837697
  ⋮
```

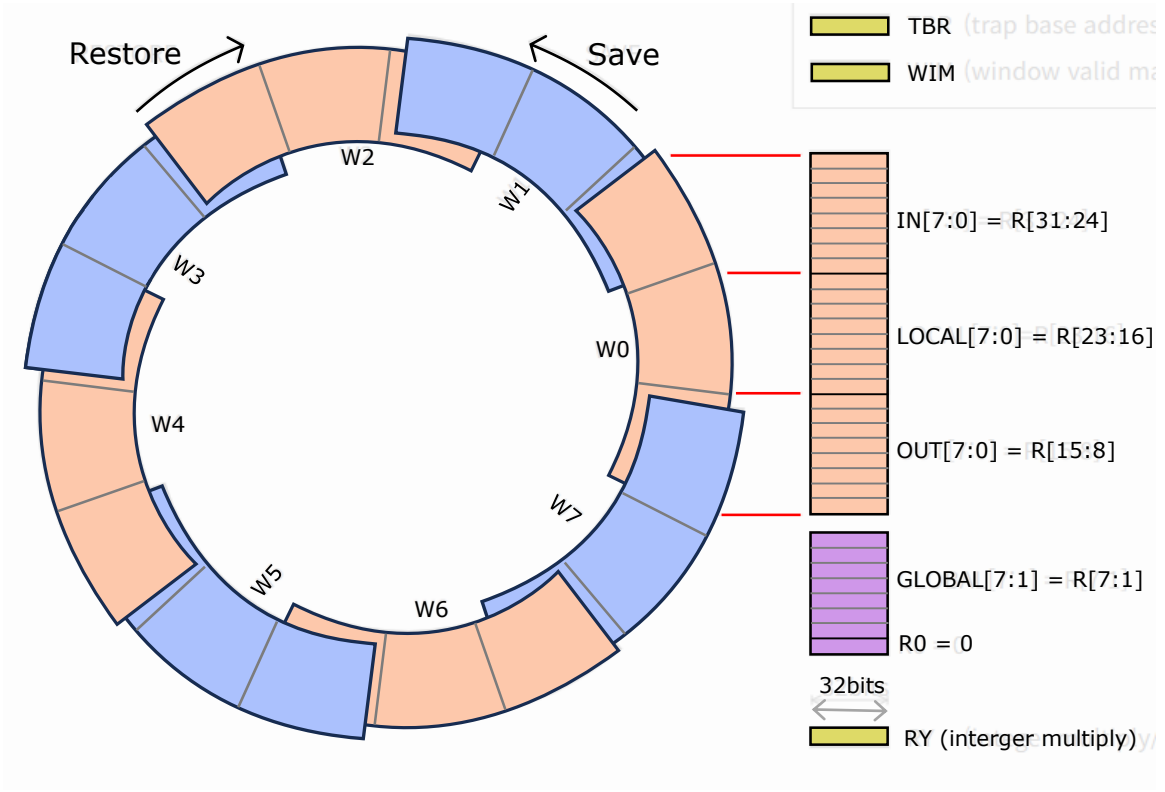
REGISTER WINDOW

- **구현 가능한 윈도우의 개수 NWINDOWS**
  - 하드웨어 설계에 따라 2개에서 32개
  - **r 레지스터 개수**에 따라 결정
    - r 레지스터의 총 개수는 40개에서 520개
    - 윈도우의 논리적 크기는 24개이지만, 인접 윈도우와 8개씩 공유하므로, 물리적으로는 16개씩만 추가되는 구조

총 r 레지스터 개수 = **8개의 global 레지스터** + **윈도우 개수만큼의 16-레지스터 세트**

REGISTER WINDOW

- **레지스터 윈도우 구조**
  - 레지스터 윈도우는 **원형 버퍼** Circular Buffer 형태로 구현됨
    - 가장 낮은 번호의 윈도우와
      가장 높은 번호의 윈도우가 서로 연결됨
  - 그림과 같이 **두 함수(Caller와 Callee)**
    각각의 in 레지스터와 out 레지스터를 **물리적으로 공유**



REGISTER WINDOW

- **CWP** **Current Window Pointer**
  - 프로세서의 상태를 나타내는 PSR 레지스터의 일부
  - 현재 프로세서가 사용 중인 레지스터 윈도우 위치를 가리키는 포인터
  - 함수를 호출하거나 복귀할 때, CWP 값을 1만큼 증감하여 새로운 레지스터 세트로 전환

- **파라미터 전달**
  - 이전 그림에서와 같이 main 함수(Caller)가 func1(Callee)을 호출
  - Caller가 자신의 out 레지스터에 값을 기록

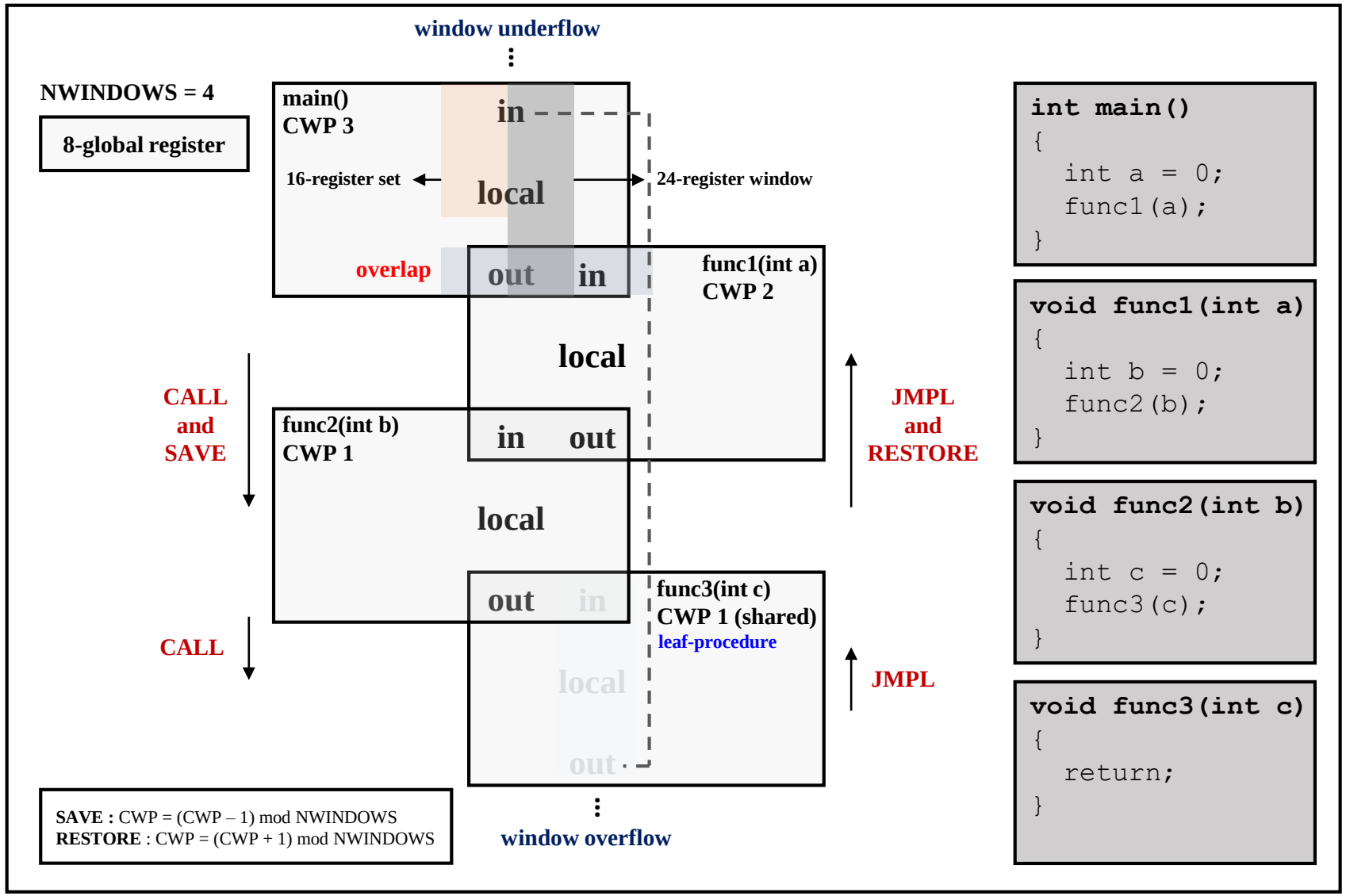
    → Callee는 자신의 in 레지스터를 통해
    메모리 접근 없이 즉시 읽음

REGISTER WINDOW

- **명령어와 윈도우 이동**
  - **SAVE 명령어를 통해 CWP 감소시켜 새로운 윈도우 할당**
    - CALL 명령어 자체는 분기만 수행할 뿐, 윈도우를 할당하지 않음
  - **RESTORE 명령어를 통해 CWP 감소시켜 이전 윈도우로 복귀**
    - JMPL 명령어 자체는 Caller 함수로 복귀만 수행할 뿐, 윈도우를 반환하진 않음
  - **복귀할 윈도우가 없는 상태**에서 **RESTORE**를 수행하면, **언더플로우 트랩**
  - **사용 가능한 윈도우가 없는 상태**에서 **SAVE**를 수행하면, **오버플로우 트랩**

- **리프 프로시저 최적화**
  - 리프 프로시저 : 더 이상 다른 함수를 호출하지 않는 말단 함수
  - 오버플로우 트랩 비용을 절약하기 위해, 윈도우를 새로 할당하지 않는 기법을 권장

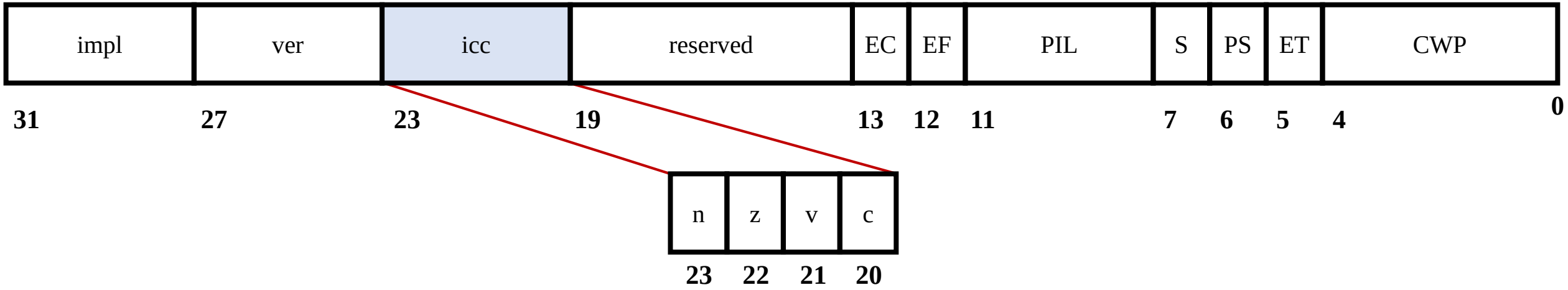
REGISTER WINDOW



IU CONTROL/STATUS REGISTERS

- **PSR (Processor State Register)**

  PSR은 프로세서의 제어 및 상태 정보를 담고 있는 32비트 핵심 레지스터이다.



  - **icc** (Integer Condition Codes, 4 bits)
    - IU 연산 상태를 나타내는 플래그
    - 주로 이름이 cc로 끝나는 산술 및 논리 명령어(예: ADDcc. ANDcc)가 실행될때 결과에 따라 갱신됨
    - 분기 명령어(Bicc)는 이 icc 필드를 참조하여 분기를 결정함

PROCESSOR STATE REGISTER (PSR)

PSR

- **S** (Superavisor, 1 bit)
  - 프로세서의 실행 모드
  - **1** : 수퍼바이저 모드 (모든 명령어 및 특권 레지스터 접근 가능)
  - **0** : 유저 모드 (특권 명령어 실행 시 **privileged_instruction** 트랩 발생)

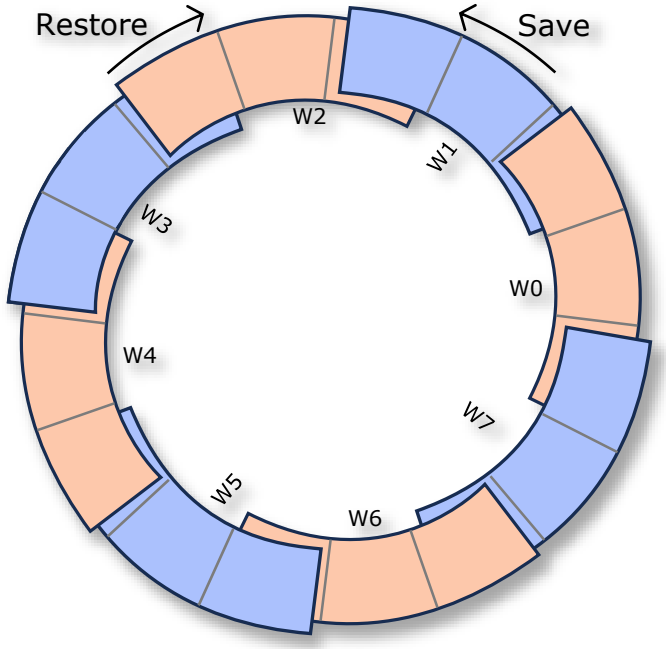
- **P** (Previous Supervisor, 1 bit)
  - 트랩이 발생하기 직전의 S 비트 값을 보존 (트랩 발생시, 자동으로 S=1으로 전환)

- **ET** (Enable Traps, 1 bit)
  - 트랩의 전역적인 활성화 여부를 제어
  - ET=0일 때 인터럽트 요청은 자동으로 무시
  - 트랩 발생시, 하드웨어가 자동으로 0으로 설정하여 중첩 트랩을 방지

PROCESSOR STATE REGISTER (PSR)

PSR

- **CWP** (Current Window Pointer, 5 bit)
  - 현재 사용 중인 레지스터 윈도우를 가리키는 포인터
  - **SAVE** 명령어 수행 시 감소하고, **RESTORE** 명령어 수행 시 증가
  - 이 연산은 **NWINDOWS**를 기준으로 **모듈로 연산**으로 수행됨 (**원형 버퍼**)



WINDOW INVALID MASK REGISTER (WIM)

- **WIM (Window Invalid Mask Register)**

  WIM 레지스터WIM Register는 **SAVE**나 **RESTORE** 명령어 실행 시,

  윈도우 오버플로우/언더플로우 트랩을 발생시킬지 여부를 결정하는 마스크 레지스터

  이 레지스터의 각 비트는 레지스터 윈도우와 1대1로 매핑됨

  WIM[n] 비트는 **CWP 값이 n인 윈도우**에 대응됨

  - 비트 값 1: Invalid, 데이터가 존재하여 보호 중이므로 덮어쓸 수 없음
  - 비트 값 0: Valid, 사용 가능함

EXAMPLE: PSR ICC FLAG

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  volatile int a = 10;
  volatile int b = 3;
  volatile int c = 4;
  volatile int result =
              compute_add_sub(a, b, c);
  printf("IU add/sub result = %d\n", result);
  exit(0);
}
```

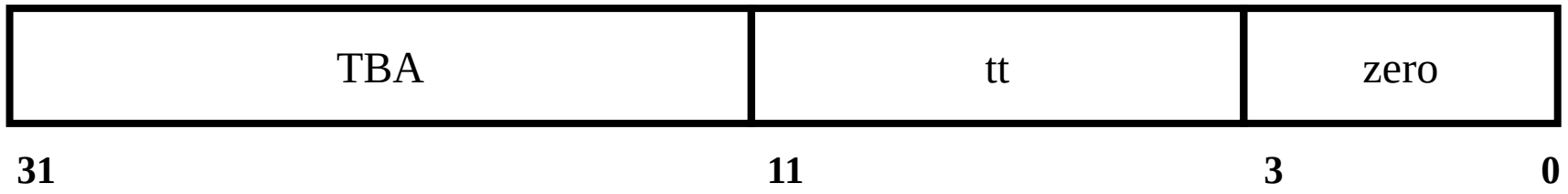
```
static volatile int compute_add_sub(int a,
int b, int c)
{
  volatile int r1 = a - 10;
  if (r1 == 0) {
    r1 = 0;
  }
  volatile int r2 = b - c;
  if (r2 < 0) {
    r2 = 0;
  }
  return r2;
}
```

psr_example/main.c

```
(gdb) break Init
(gdb) continue
        :
(gdb) step
        :
(gdb) info register psr
psr         0x400000        [ ]     // icc flag : 0100 (Zero Flag)
        :
(gdb) info register psr
psr         0x800000        [ ]     // icc flag : 1000 (Negative Flag)

                                    // if 문이 존재할 때, branch 여부를 결정하는
                                    // icc flag가 바뀌는 것을 확인할 수 있음
```

Trap Base Register (TBR)

- **역할**: 프로세서가 해당 트랩에 맞는 핸들러로 정확하게 이동할 수 있도록 돕는 레지스터
- **주소 생성 원리**: 트랩 테이블의 **베이스 주소와 발생한 트랩의 타입을 조합**하여 목적지 주소를 생성
- **구성요소**
  - TBA(Trap Base Address, 20 bits)
  - tt (Trap Type, 8 bits)
  - zero (4 bits)



Trap Base Register (TBR)

- **TBA(Trap Base Address, 20 bits)**
  - 트랩 테이블의 시작 주소 (기준점)
  - 운영체제가 WRTBR(Write TBR) 명령어로 1회 설정
  - 트랩 발생 시에도 변하지 않는 **고정값**
- **tt (Trap Type, 8 bits)**
  - 발생한 **트랩의 종류를 구분**
  - 트랩 발생 순간 하드웨어가 자동으로 기록
- **zero (4 bits)**
  - 항상 **0000으로 고정**
  - 주소 정렬 목적 ex)0001 0000, 0010 0000,0011 0000
  - 최대 4개의 명령어를 배치할 수 있는 공간을 제공

EXAMPLE: COMPARISON OF TBA AND TRAP TABLE ADDRESSES

```
static unsigned int read_tbr(void)
{
  unsigned int result;
  __asm__("rd %%tbr, %0" : "=r"(result));
  return result;
}
```

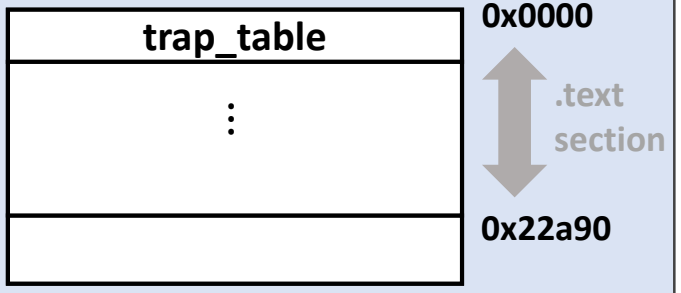
```
rtems_task Init(
  rtems_task_argument ignored
)
{
  tbr = read_tbr();
  tba = tbr & 0xFFFFF000;
  trap_table_addr
  = (unsigned int)trap_table;

  if (tba == trap_table_addr) {
    printf("Trap Table Address, Correct!\n");
  } else {
    printf("Mismatch!\n");
  }
}
```

tbr_example-1/main.c

(gdb) **maintenance info sections**
[0] 0x0000 → 0x22a90 at 0x0010000: .text // 0x0은 text section이다

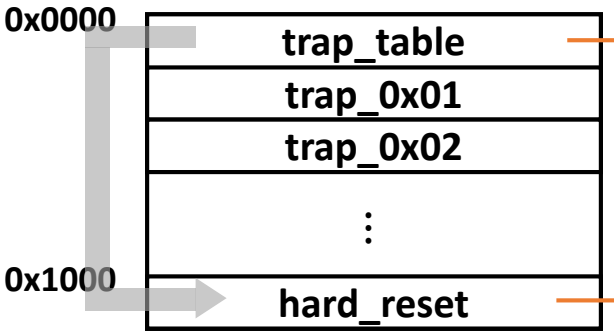
(gdb) **info address trap_table**
(gdb) **info symbol 0x0**
Symbol "trap_table" is static storage at address 0x0.
trap_table in section .text // trap_table은 0x0에 위치

(gdb) **break Init**
(gdb) **continue**
(gdb) **print/x tbr**
$1 = 0x50
(gdb) **print/x tba**
$2 = 0x0
(gdb) **print/x trap_table_addr**
$3 = 0x0



// 실행 결과 "Trap Table Address, Correct!"가 나온다
// TBA는 Trap Table 주소를 가리킨다는 사실을 확인할 수 있음

EXAMPLE: UNDERSTANDING TRAP TABLE OPERATIONS



tt=0으로 트랩이 걸려
0x00 엔트리를 실행하면
0x1000로 점프한다

```
(gdb) disassemble trap_table
Dump of assembler code for function trap_table:
   0x00000000 <+0>:     mov  %g0, %l0                  // l0 register 0으로 초기화
   0x00000004 <+4>:     sethi  %hi(0x1000), %l4        // %l4상위 비트 ← 0x1000
   0x00000008 <+8>:     jmp  %l4                       // Jump 수행
   0x0000000c <+12>:    clr  %l3
End of assembler dump.
```

```
(gdb) info symbol 0x1000
hard_reset in section .text
(gdb) disassemble 0x1000
Dump of assembler code for function hard_reset:
   0x00001000 <+0>:     sethi  %hi(0), %g1
   0x00001004 <+4>:     mov  %g1, %g1   ! 0x0 <trap_table>
   0x00001008 <+8>:     wr  %g1, %tbr
   ⋮
   0x000010d4 <+212>:   mov  %g6, %o0

// CPU가 리셋 직후 제일 먼저 돌리는 초기화 루틴
// 즉 tt=0 트랩이 발생한 경우, hard_reset 루틴이 호출되어 초기화를 진행한다
```

EXAMPLE: Trap Base Register (TBR)

```
rtems_task Init(rtems_task_argument ignored){
  (void) ignored;
  TEST_BEGIN();
  volatile int a = 3;
  volatile int zero = 0;
  volatile int result;
  result = a / zero;
  TEST_END();
  exit(0);
}
```

tbr_example-2/main.c

```
(gdb) disassemble Init
Dump of assembler code for function Init:
                    ⋮
   0x00001290 <+56>:    nop
   0x00001294 <+60>:    sdiv  %g1, %g2, %g1
   0x00001298 <+64>:    st  %g1, [ %fp + -4 ]
                    ⋮
```

TBR: tbr
TBA: tbr >> 12
tt: (tbr >> 4) & 0xFF
zero: tbr & 0xF

```
(gdb) break *0x00001294
(gdb) continue
(gdb) print /x $pc
$1 = 0x1294
(gdb) print /x $tbr
$2 = 0x50
(gdb) print /x $tbr >> 12
$3 = 0x0
(gdb) print /x ($tbr >> 4) & 0xFF
$4 = 0x5
(gdb) print /x $tbr & 0xF
$5 = 0x0
```

Trap 발생 후

```
(gdb) stepi
^C
(gdb) print /x $pc
$6 = 0x13740
(gdb) print /x $tbr
$7 = 0x800
(gdb) print /x $tbr >> 12
$8 = 0x0
(gdb) print /x ($tbr >> 4) & 0xFF
$9 = 0x80
(gdb) print /x $tbr & 0xF
$10 = 0x0
```

EXERCISE 1: UNDERSTANDING THE MEANING OF TBA VALUES

- 1. 다음 빈칸에 들어갈 값을 작성하시오.

```
/* C 코드 */
rtems_task Init(
  rtems_task_argument ignored
)
{
  tbr = read_tbr();
  tba = tbr & 0xFFFFF000;
  trap_table_addr
  = (unsigned int)trap_table;

  if (tba == trap_table_addr) {
    printf("Trap Table Address,
Correct!\n");
  } else {
    printf("Mismatch!\n");
  }
}
```

```
/* SPARC 어셈블리 코드 */
(gdb) print/x tbr
$1 = 0x50
(gdb) print/x tba

_____

(gdb) print/x trap_table_addr
$3 = 0x0
```

EXERCISE 1: UNDERSTANDING THE MEANING OF TBA VALUES

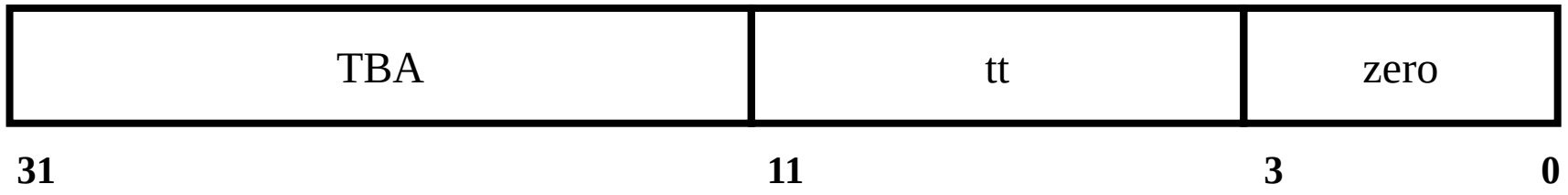
- **문제 해설**
  - 정답: 0x0
  - TBA는 TBR의 상위 20비트를 의미한다.
  - TBA는 메모리상에 존재하는 Trap Table의 시작 주소를 담고 있는 비트이므로, trap_table_addr 변수와 같은 값을 가지고 있을 것이다. 따라서 동일한 값인 0x0을 가진다.

EXERCISE 2: BIT MANIPULATION OF TBR REGISTER

- **2. TBR(Trap Base Register) 레지스터에서 아래와 같은 비트 연산이 수행되는 이유를 설명하시오.**

```
TBR: tbr
TBA: tbr >> 12
tt: (tbr >> 4) & 0xFF
zero: tbr & 0xF
```

cf)



EXERCISE 2: BIT MANIPULATION OF TBR REGISTER

- **문제 해설**
  - 1. TBA: 상위 20비트만을 유효한 주소로 추출하기 위해 오른쪽으로 12비트 시프트 연산을 수행.
  - 2. tt: >>4: 하위 4비트를 제거하여 tt값을 최하위 비트 위치로 이동

    &0xFF: 상위 TBA 영역의 값을 마스킹하여 8비트 트랩 번호만을 추출
  - 3. zero: &0xF: 상위 TBA 영역의 값을 마스킹하여 4비트 데이터만을 추출

Multiply / Divide Register (Y)

- **SPARC V8 아키텍처**
  - 32비트 RISC 아키텍처로, 일반 레지스터의 폭과 주소 공간이 모두 32비트임
  - SPARC V8 에서는 정렬이 깨진 채로 메모리 접근을 요청하면 트랩이 발생함

32-bits (word)

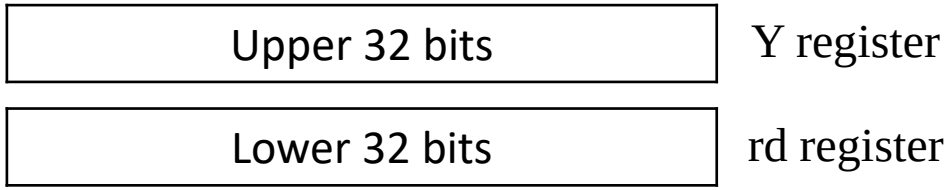
- **Multiply / Divide Register** **Y register**
  - 정수 곱셈과 나눗셈 연산에서 **상위 32비트를 저장하는 레지스터**
  - 큰 단위로의 정확한 부호확장을 위해 **Y 레지스터**가 필요함

Multiply / Divide Register (Y)

- **SPARC 곱셈 명령어**

UMUL	Unsigned 32 bit * 32 bit 곱셈
SMUL	Signed 32 bit * 32 bit 곱셈
UMULcc	Unsigned 32 bit * 32 bit 곱셈 (icc bit 갱신)
SMULcc	Signed 32 bit * 32 bit 곱셈 (icc bit 갱신)

- **곱셈 연산에서의 Y 레지스터 활용**
  - 상위 32 bit의 값은 **Y 레지스터**에 저장
  - 하위 32 bit의 값은 목적지 레지스터에 저장



Multiply / Divide Register (Y)

- **SPARC 나눗셈 명령어**

UDIV	Unsigned 64 bit를 32 bit로 나눔
SDIV	Signed 64 bit를 32 bit로 나눔
UDIVcc	Unsigned 64 bit를 32 bit로 나눔 (icc bit 갱신)
SDIVcc	Signed 64 bit를 32 bit로 나눔 (icc bit 갱신)

- **나눗셈 연산에서의 Y 레지스터 활용**
  - 나눗셈 연산은 피제수가 64 bit로 간주됨
  - 피제수의 상위 32 bit의 값은 Y 레지스터에 저장
  - 피제수의 하위 32 bit의 값은 소스 레지스터(rs1)에 저장

EXAMPLE: OBSERVING THE Y REGISTER CHANGE IN MULTIPLICATION

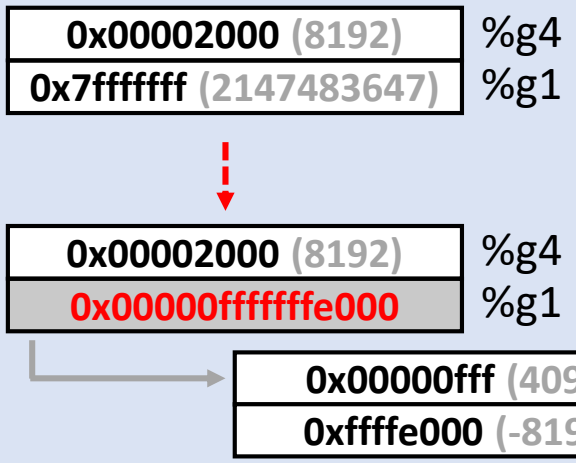
```
void Multiple( void )
{
  volatile int res_s = s1 * s2;
  volatile unsigned int res_u = u1 * u2;
  printf("%d*%d = %d\n", s1, s2, res_s);
  printf("%u*%u = %u\n", u1, u2, res_u);
}
```

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  u1 = 123u;
  u2 = 7u;
  s1 = 2147483647;
  s2 = 8192;

  Multiple();
  Divide();
}
```

y-register_example/main.c

```
(gdb) info register y
y              0x0             0                    // y 레지스터 초기값 = 0

(gdb) disassemble Multiple
Dump of assembler code for function Multiple:
   ⋮
   0x0000127c <+36>:    ld   [ %i3 + 0xe4 ], %g1     // %g1에 s1의 값 저장
   0x00001280 <+40>:    ld   [ %i4 + 0xe0 ], %g4     // %g4에 s2의 값 저장
   0x00001284 <+44>:    smul  %g1, %g4, %g1          // %g1, g4의 곱 수행
   0x00001288 <+48>:    st   %g1, [ %fp + -4 ]       // 스택 프레임에 %g1 저장
```

```
(gdb) info register g1 g4 y
g1             0xfffe000       -8192
g4             0x2000           8192
y              0xfff            4095
```




```
// 곱셈 결과, 64 비트의 결과가 나옴
// y레지스터가 %g1의 상위 32 비트를 저장
// 기존의 %g1 레지스터는 하위 32 비트를 저장
```

EXAMPLE: OBSERVING THE Y REGISTER CHANGE IN DIVISION

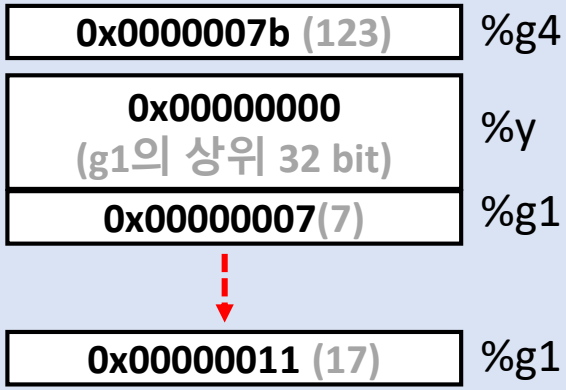
```
void Divide( void )
{
  volatile int res_s = s1 / s2;
  volatile unsigned int res_u = u1 / u2;
  printf("%d/%d = %d\n", s1, s2, res_s);
  printf("%u/%u = %u\n", u1, u2, res_u);
}
```

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  u1 = 123u;
  u2 = 7u;
  s1 = 2147483647;
  s2 = 8192;

  Multiple();
  Divide();
}
```

y-register_example/main.c

```
(gdb) disassemble Divide
Dump of assembler code for function Multiple:
   0x000012d4 <+8>:     ld   [ %g3 + 0xdc ], %g1 ! 0x274dc <s1>
   0x000012e4 <+24>:    ld   [ %g2 + 0xd8 ], %g4
   0x00001284 <+44>:    sdiv  %g1, %g4, %g1          //signed 나눗셈 명령어
   :
   0x000012fc <+48>:    ld   [ %i3 + 0xe4 ], %g4 ! 0x274e4 <u1>
   0x00001308 <+60>:    ld   [ %i4 + 0xe0 ], %g1
   0x00001314 <+72>:    udiv  %g4, %g1, %g1          //unsigned 나눗셈 명령어
```

```
(gdb) info register g1 g4 y
g1             0x11             17
g4             0x7b             123
y              0x0              0
```



// unsigned 나눗셈 결과, y 레지스터는
피제수의 상위 32 비트를 담기 때문에 0의 값을 가진다

EXERCISE 3: DETERMINING Y REGISTER

- **Y 레지스터 판단 문제**
  - SPARC의 Y 레지스터는 곱셈, 나눗셈 연산에 활용되며 상위 32 비트를 담는다. 다음의 SPARC 어셈블리 코드를 참고하여 Y 레지스터의 변화 과정을 이해한다.

```
/* C 코드 */
volatile int num = 2147483647;
volatile int a = 256;
volatile int b = -1024;

void Mul(void)
{
  volatile int res_a = num * a;
  volatile int res_b = num * b;
}
```

```
/* SPARC 어셈블리 코드 */
(gdb) disassemble Mul
Dump of assembler code for function Mul:
   0x00001368 <+12>:    ld   [ %i4 + 0x120 ], %g1
   0x0000136c <+16>:    ld   [ %g2 + 0x11c ], %g3
   0x00001370 <+20>:    smul  %g1, %g3, %g1   ..... ①
   :
   0x0000137c <+32>:    ld   [ %i4 + 0x120 ], %g1
   0x00001380 <+36>:    ld   [ %i3 + 0x118 ], %g3
   0x00001384 <+40>:    smul  %g1, %g3, %g1   ..... ②
```

EXERCISE 3: DETERMINING Y REGISTER

- 3.1 ①번에서 %g1, %g3의 결과가 다음과 같다고 할 때, Y 레지스터의 결과를 예상하시오.
  (이때, 2147483647의 값은 0x7fffffff이다)

0xffffff00	%g1
0x00000100 (256)	%g3

- 3.2 ②번까지의 코드를 실행해보고, 어떤 과정을 거쳐 Y 레지스터의 값이 도출되는지 서술하시오.

EXERCISE 3: DETERMINING Y REGISTER

- **문제 해설(3.1)**
  - 정답: 0x7f (=127)
  - 곱셈 연산 전에 각 레지스터가 가지고 있는 값은 다음과 같다.

0x7fffffff (2147483647)	%g1
0x00000100 (256)	%g3
0x0	%Y

  - 256은 2^8이므로 0x7fffffff와 곱 연산 시 0x7fffff00의 결과가 나온다.

(8bit가 밀려나므로 16비트 상 두 칸 left shift된다)

  - %g1에 곱셈의 결과를 저장하기 때문에, 하위 32비트의 값은 %g1으로 들어가며,

나머지 상위 32비트는 Y레지스터에 저장된다

0x7fffff00	%g1
0x7f	%Y

EXERCISE 3: DETERMINING Y REGISTER

- **문제 해설(3.2)**
  - num = 0x7fffffff (2147483647), b = 0xfffffc00 (-1024)값을 가진다.
  - 이 두 값을 곱하면 64비트의 결과가 나온다. (0xfffffe00_00000400)
  - disassemble로 확인해 본 결과, 곱셈 후 %g1에 저장하므로 %g1에는 하위 32비트가 보인다.

  상위 32비트는 Y 레지스터에 저장된다

0x00000400	%g1
0xfffffc00 (-1024)	%g3
0xfffffe00	%Y

Program Counters (PC, nPC)

- **PC (Program Counter)**

  현재 IU가 실행하고 있는 명령어의 주소를 담고 있다.

- **nPC (Next Program Counter)**

  트랩이 발생하지 않는다는 가정하에, 다음에 실행될 명령어의 주소를 담고 있다 (일반적으로 PC + 4)

EXAMPLE: Program Counters (PC, nPC)

```
rtems_task Init(rtems_task_argument ignored){
  (void) ignored;
  TEST_BEGIN();
   int a = 10;
   int b = 3;
   int sum  = a + b;
   int diff = a - b;
   printf("%d + %d = %d\n", a, b, sum );
   printf("%d - %d = %d\n", a, b, diff);
   TEST_END();
   exit(0);
}
```

pc_example/main.c

```
(gdb) disassemble Init
Dump of assembler code for function Init:
                    ⋮
      0x00001278 <+32>:    mov  0xd, %o4
      0x0000127c <+36>:    mov  3, %o3
      0x00001280 <+40>:    mov  0xa, %o2
                    ⋮
```

```
(gdb) break *0x0000127c
(gdb) continue
(gdb) print /x $pc
$1 = 0x127c
(gdb) print /x $npc
$2 = 0x1280
(gdb) stepi
(gdb) print /x $pc
$3 = 0x1280
(gdb) print /x $npc
$4 = 0x1284
```