SPARC V8
Procedure Call and Stack 실습

인하대학교 디자인테크놀로지학과

12223682 정유진

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.

- **본 챕터는 IU 레지스터 및 Register Window 구조에 대한 기본 이해를 전제로 진행됩니다.**

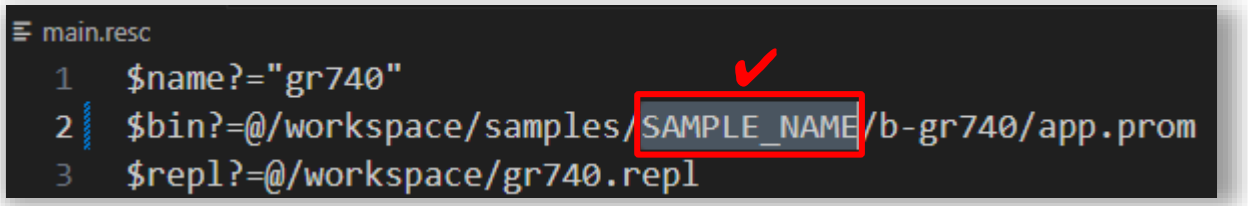
관련 내용은 SPARC_V8_ISA_IU_실습_v1.0.pptx를 통해 사전 학습하시기를 권장합니다.

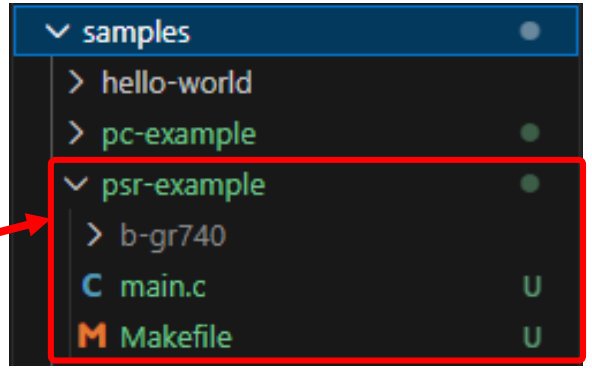
OVERVIEW

- **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



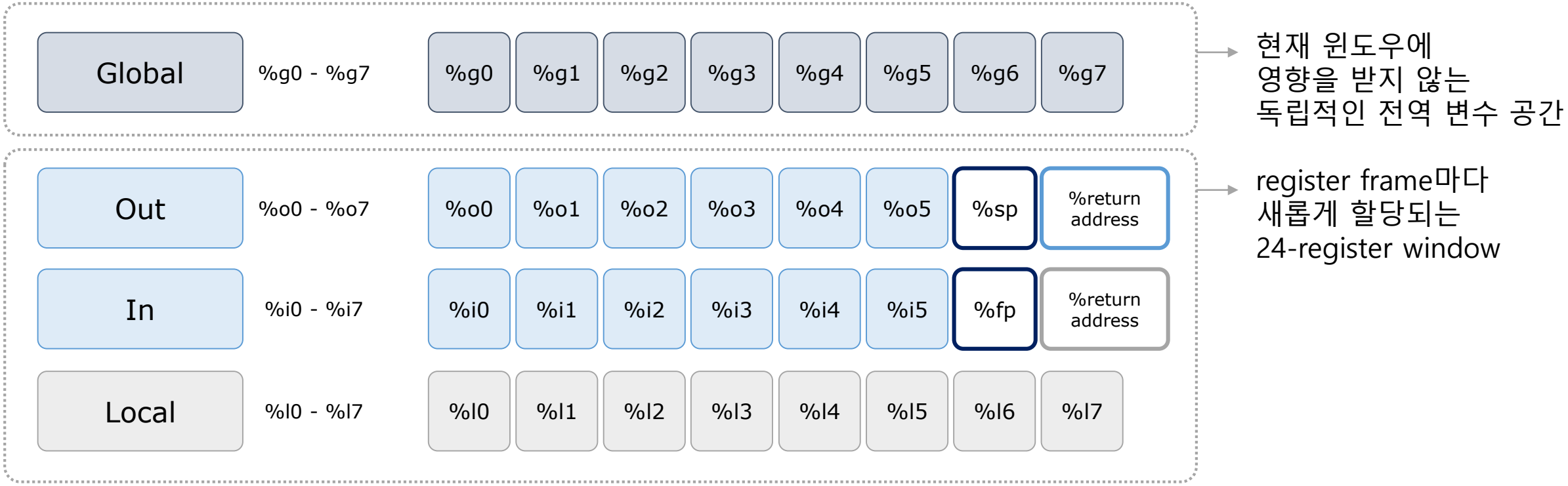


"Drag here to copy"

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

REGISTER WINDOW (REVIEW)

- **IU r Registers**

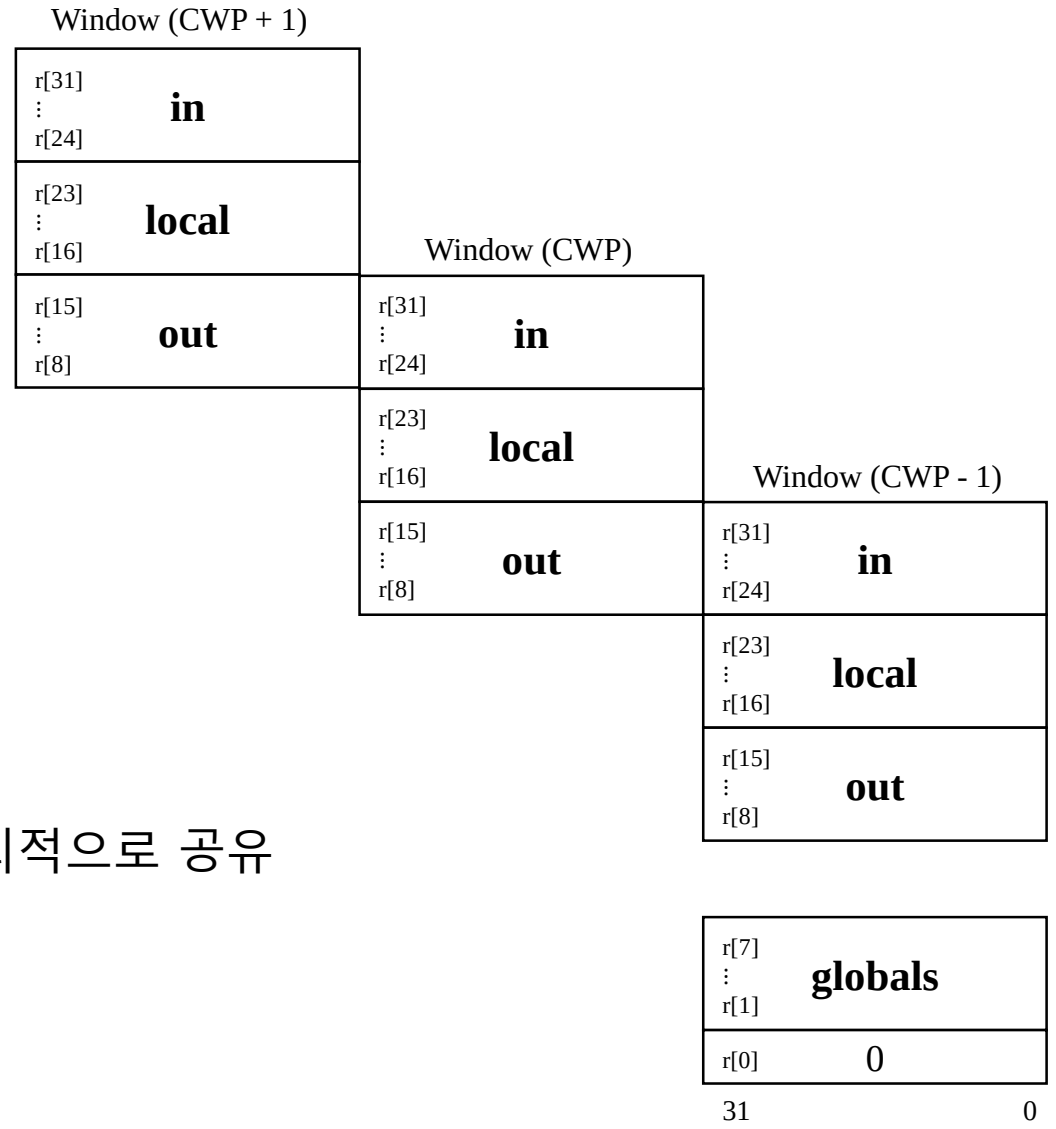


- %o7 : CALL 호출 시점의 주소 저장 (return address 계산에 사용)
- %i7 : 이전 함수가 저장해 둔 복귀 주소 (caller의 CALL 호출 시점)를 저장

REGISTER WINDOW (REVIEW)

- **레지스터 윈도우**
  - 함수 호출 및 복귀에 드는 비용을
    줄이기 위해 고안된 하드웨어 매커니즘

- **레지스터 윈도우 구조**
  - **원형 버퍼** Circular Buffer 형태로 구현됨
    - 가장 낮은 번호의 윈도우와
      가장 높은 번호의 윈도우가 서로 연결됨
  - Caller의 out 레지스터와 Callee의 in 레지스터를 물리적으로 공유



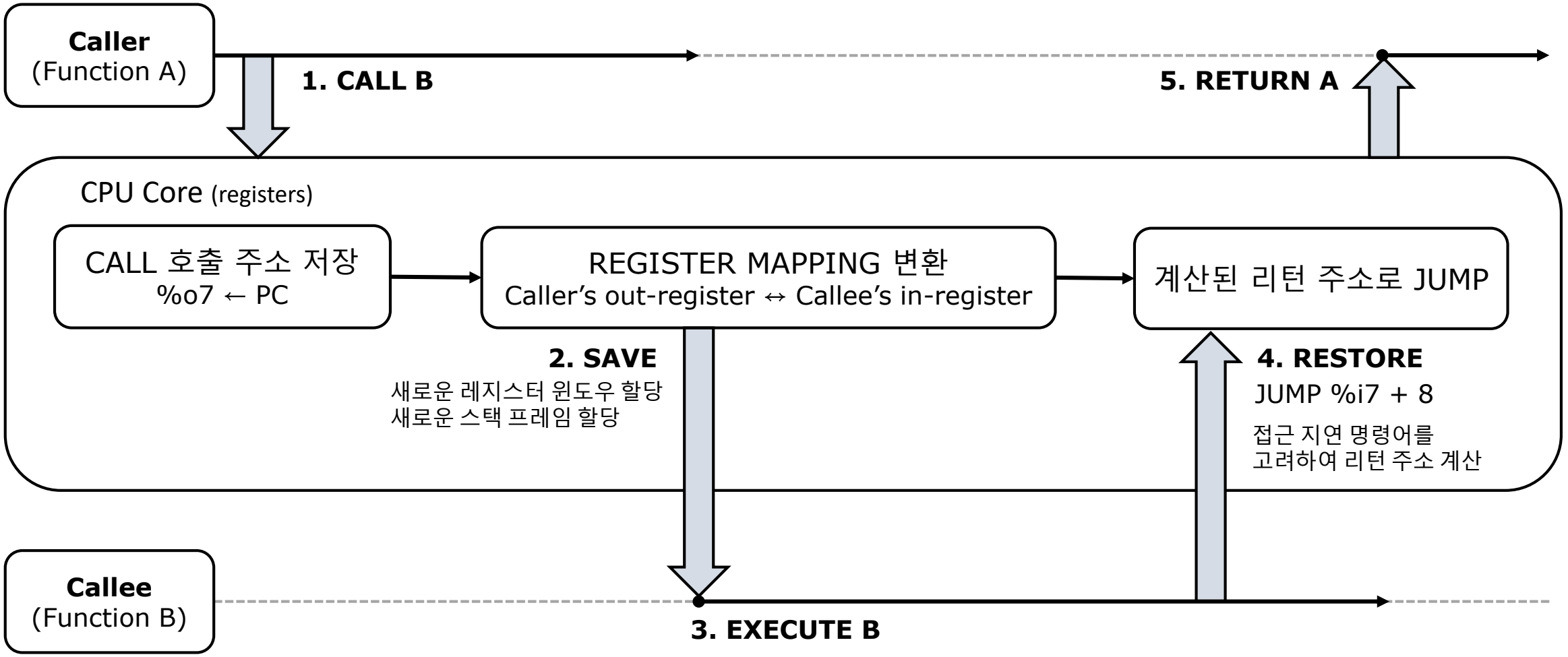
PROCEDURE CALL

- **함수 호출** **Procedure Call**
  - 현재 실행 흐름을 다른 코드 블록으로 잠시 이동시켰다가
    작업이 끝나면 다시 원래 위치로 복귀하는 메커니즘

- **Caller calls Callee**
  - Caller: 다른 함수를 호출하여 제어 흐름을 넘김
  - Callee: 호출되어 실행하는 함수. 전달받은 인자로 작업 수행 후 원래 주소로 복귀

CALLING CONVENTION

- **Function Call & Return Mechanism**



CALLING CONVENTION

- **호출 규약** Calling Convention
  - 이전 그림에서와 같이 함수 A(Caller)가 함수 B(Callee)를 호출
  - 함수 B로 진입하기 전, **Caller의 %o7** 레지스터에 **CALL 명령어 수행 시점의 주소**를 저장
  - Caller가 자신의 out 레지스터에 인자, 리턴 주소를 기록

    → Callee는 자신의 in 레지스터를 통해 메모리 접근 없이 즉시 읽음

  - Callee 전부 실행 후 이전에 저장해 두었던 %i7값을 이용하여 리턴 주소 계산

    return address
    %i7 + 8

    → 8을 더하는 이유: 지연 명령어 수행한 이후의 시점을 리턴 주소로 정해야 하기 때문

    → 지연 명령어(delay slot): 분기 전에 무조건 실행하는 명령어

EXAMPLE: SAVING RETURN ADDRESS

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  final_res = Sum(10);

  TEST_END();
  exit(0);
}
```

```
int Sum(int x)
{
  volatile int res = 0;
  res = Mult(x, 2, 3);
  return res + 5;
}
```

```
int Mult(int x, int y, int z)
{
  return x*y*z;
}
```

procedure_call_example-1/main.c

```
(gdb) break Init
(gdb) Continue
(gdb) disassemble Init
Dump of assembler code for function Init:
   :
   0x000012e8 <+28>:    mov  0xa, %o0                  // %o0에 인자 10 저장
   0x000012ec <+32>:    call  0x1258 <sum>             // sum 함수 호출
   0x000012f0 <+36>:    nop                            // delay slot: 점프 전 수행
   0x000012f4 <+40>:    mov  %o0, %g2                  // sum 리턴 후 수행할 명령
```

```
(gdb) next
39        final_res = sum(10);
(gdb) step                                             // sum 함수로 진입
sum (x=10) at main.c:22
```

```
(gdb) info register i0 i7
i0             0xa             10
i7             0x12ec          4844

// 레지스터 윈도우가 할당되며 이전 윈도우의 %o0값이 현재 윈도우의 %i0에 나타남
// call 함수 호출 시점의 주소가 caller의 %o7에 저장되고, 그 값이 callee의 %i7로 매핑 됨
```

EXAMPLE: SAVING RETURN ADDRESS

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  final_res = Sum(10);

  TEST_END();
  exit(0);
}
```

```
int Sum(int x)
{
  volatile int res = 0;
  res = Mult(x, 2, 3);
  return res + 5;
}
```

```
int Mult(int x, int y, int z)
{
  return x*y*z;
}
```

procedure_call_example-1/main.c

```
(gdb) disassemble Sum
Dump of assembler code for function sum:
   ⋮
   0x00001264 <+12>:    mov  3, %o2                    // %o2에 인자 3저장
   0x00001268 <+16>:    mov  2, %o1                    // %o1에 인자 2저장
   0x0000126c <+20>:    ld  [ %fp + 0x44 ], %o0
   0x00001270 <+24>:    call  0x1298 <mult>            // mult 함수 호출
   0x00001274 <+28>:    nop                            // delay slot
   0x00001278 <+32>:    mov  %o0, %g1                  // mult 리턴 후 수행할 명령
```

```
(gdb) next
23        res = mult(x, 2, 3);
(gdb) step                                             // mult 함수로 진입
mult (x=10, y=2, z=3) at main.c:29
```

```
(gdb) info register i0 i1 i2 i7
i0              0xa             10
i1              0x2             2
i2              0x3             3                      // 인자로 넘긴 값들 저장
i7              0x1270          4720                   // call 시점의 주소 저장
```

EXAMPLE: SAVING RETURN ADDRESS

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  final_res = Sum(10);

  TEST_END();
  exit(0);
}
```

```
int Sum(int x)
{
  volatile int res = 0;
  res = Mult(x, 2, 3);
  return res + 5;
}
```

```
int Mult(int x, int y, int z)
{
  return x*y*z;
}
```

procedure_call_example-1/main.c

```
(gdb) next instruction
0x00001278 in sum (x=10) at main.c:23                    // sum함수로 복귀
```

```
(gdb) info register pc o7
pc             0x1278          0x1278 <sum+32>
o7             0x1270          4720

// (mult 함수에서의 %i7값 + 8)이 현재 pc값과 동일함
```

```
(gdb) disassemble
Dump of assembler code for function sum:
   0x00001270 <+24>:    call  0x1298 <mult>
   0x00001274 <+28>:    nop
=> 0x00001278 <+32>:    mov   %o0, %g1
   :

// delay slot은 call 수행에 걸리는 시간 동안 무조건 수행할 명령어. 즉 이미 수행된 명령어
// return address = call 명령어 수행 시점 + 8
// %o7에 저장한 값을 기반으로 return address를 계산한 후, 해당 위치로 이동한다
```

return address

STACK

- **스택** Stack
  - 함수의 메모리 기반 작업 영역
    - 레지스터 윈도우가 넘칠 때 스택으로 백업된다
  - 높은 주소부터 시작하여 새로 할당될 때마다 낮은 주소로 감소함

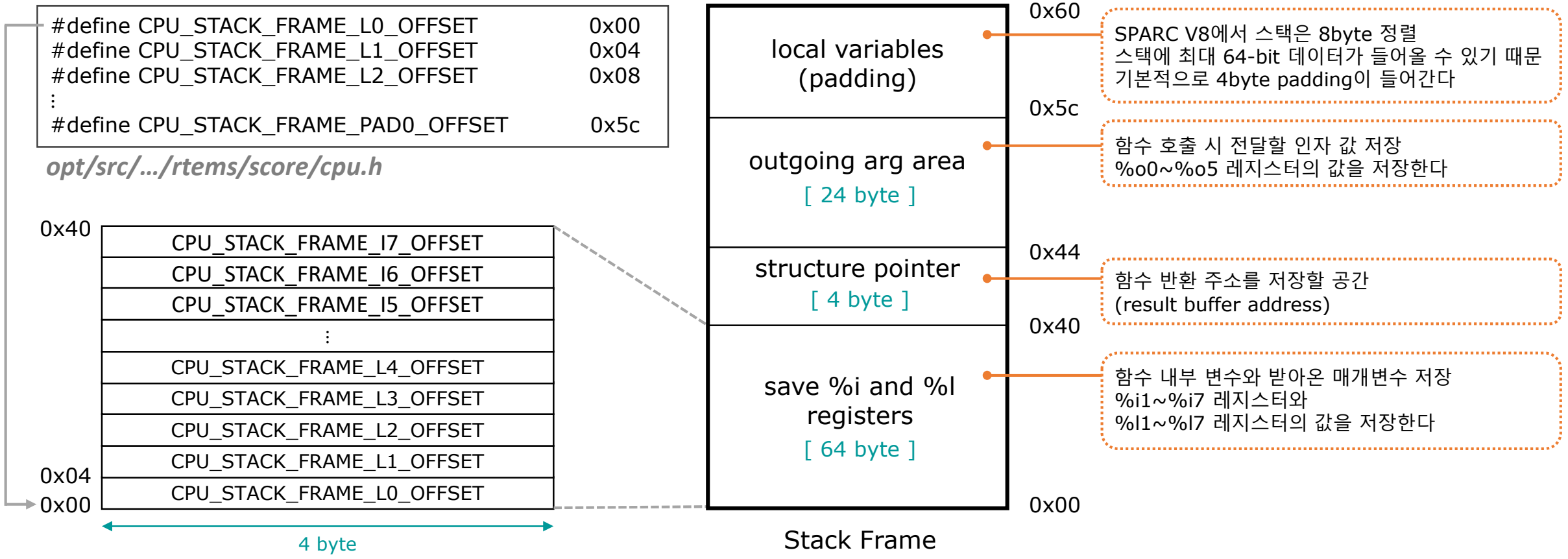
- **스택 프레임** Stack Frame
  - 함수 하나가 실행되는 동안 스택에 잡아두는 전용 작업 공간
  - 스택 프레임의 크기는 반드시 **8의 배수**여야 한다

    → SPARC가 다루는 메모리 공간의 단위 중 제일 큰 단위가 double (8byte)이기 때문

    → 스택 프레임의 최소 크기는 96byte (0x00~0x60)이다

STACK FRAME MEMORY LAYOUT

- 스택 프레임 메모리 구조
  - 레지스터 및 인자 값을 스택 프레임 내 고정된 오프셋에 매핑한다

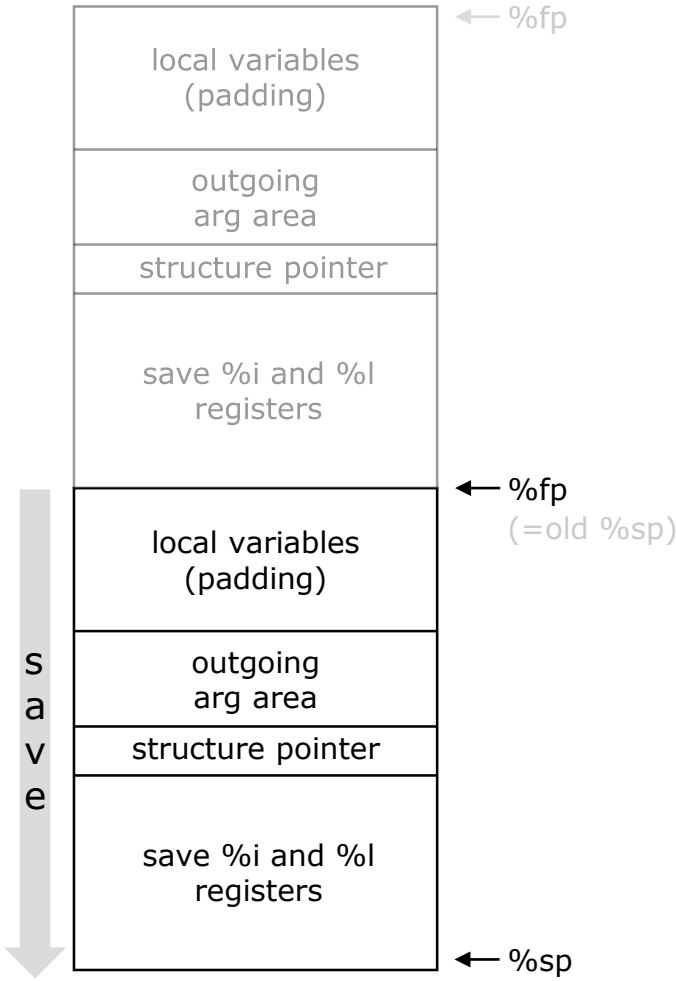


STACK FRAME ALLOCATION AND DEALLOCATION

- **스택 프레임 할당**
  - **%sp(=%o6)**: 현재 스택의 top을 가리키는 레지스터
  - **%fp (=%i6)**: 이전 스택 프레임의 %sp값을 저장함
  - SAVE시 윈도우 레지스터 회전과 동시에 스택 프레임이 할당된다
    - 할당되는 스택 프레임의 크기는 최소 96 byte(0x00~0x60)
    - %sp값을 필요한 크기만큼 새롭게 갱신

```
save  %sp, -104, %sp
 ⋮
```

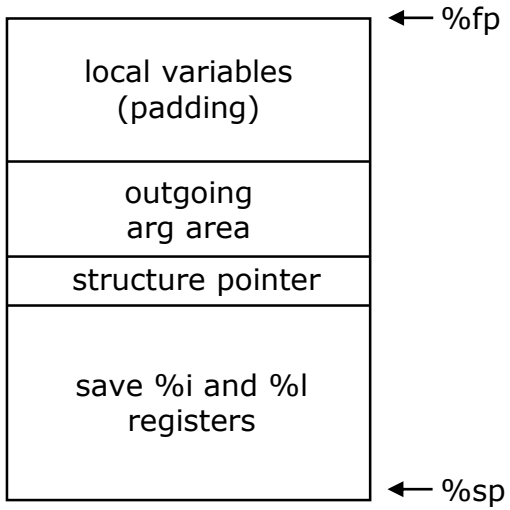
    - %fp의 값은 윈도우 레지스터 회전에 의해 자동으로 갱신됨 (caller %o6은 callee의 %i6과 매핑)



STACK FRAME ALLOCATION AND DEALLOCATION

- **스택 프레임 해제**
  - RESTORE 시 윈도우 레지스터 회전과 동시에 스택 프레임이 해제된다
    - caller의 레지스터로 돌아오면서 %fp, %sp가 원래 값을 되찾음
    - STORE처럼 %sp 값을 따로 조정하지 않음

```
restore
```



- **스택 공간을 미리 할당해두는 이유**
  - 프레임을 미리 잡아두면 [%fp - 4] 같은 **고정 주소**로 안정적으로 접근 가능
  - 프레임마다 최소한의 register-save area(64B)를 예약하여,

  윈도우가 모자란 경우 윈도우 레지스터 값을 쉽게 저장할 수 있도록 함

EXAMPLE: USING STACK(1)

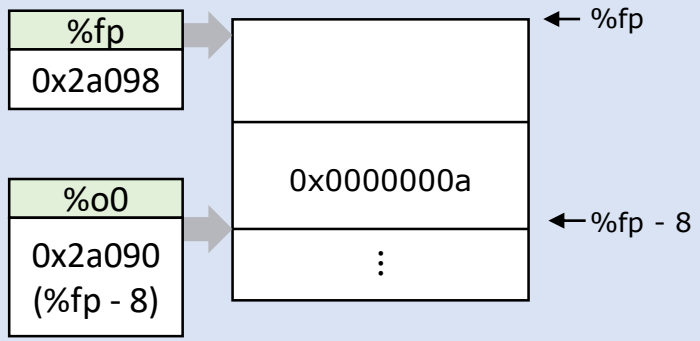
```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  int stack_value = 10;
  int res = Func(&stack_value);
  printf("%d\n", res);

  TEST_END();
  exit(0);
}
```

```
static int Func(int *value)
{
  volatile int local_offset = 7;
  *value += local_offset;
  return *value;
}
```

procedure_call_example-2/main.c

```
(gdb) disassemble Init
Dump of assembler code for function Init:
   ⋮
   0x000012b4 <+28>:    mov  0xa, %g1   ! 0xa <trap_table+10>
   0x000012b8 <+32>:    st  %g1, [ %fp + -8 ]          // 스택(%fp - 8)에 0xa 저장
   0x000012bc <+36>:    add  %fp, -8, %g1              // 스택 주소를 %g1에 저장
   0x000012c0 <+40>:    mov  %g1, %o0                  // %g1 값을 %o0에 복사
=> 0x000012c4 <+44>:    call  0x1258 <Func>
   ⋮
```

```
(gdb) info reg %o0
o0             0x2a090          172176
(gdb) x/wx 0x2a090
0x2a090:       0x0000000a
(gdb) x/wx $fp - 8
0x2a090:       0x0000000a
info reg %fp
fp             0x2a098          0x2a098

// %o0가 저장하고 있는 주소가 가리키는 값은 0xa로, stack_value임
// 0x2a090은 %fp - 8의 주소와 동일함
```



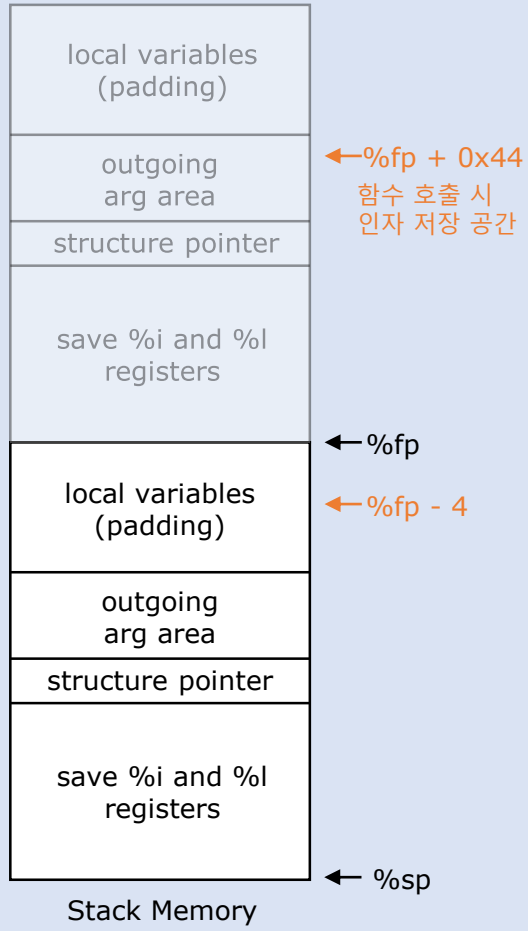
EXAMPLE: USING STACK(1)

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  int stack_value = 10;
  int res = Func(&stack_value);
  printf("%d\n", res);

  TEST_END();
  exit(0);
}
```

```
static int Func(int *value)
{
  volatile int local_offset = 7;
  *value += local_offset;
  return *value;
}
```

procedure_call_example-2/main.c

```
(gdb) disassemble Func
Dump of assembler code for function Func:
   0x00001258 <+0>:     save  %sp, -104, %sp
   0x0000125c <+4>:     st  %i0, [ %fp + 0x44 ]
   0x00001260 <+8>:     mov  7, %g1
   0x00001264 <+12>:    st  %g1, [ %fp + -4 ]
   0x00001268 <+16>:    ld  [ %fp + -4 ], %g2
   0x0000126c <+20>:    ld  [ %fp + 0x44 ], %g1
   0x00001270 <+24>:    ld  [ %g1 ], %g1
   0x00001274 <+28>:    add  %g2, %g1, %g2
   0x00001278 <+32>:    ld  [ %fp + 0x44 ], %g1
   0x0000127c <+36>:    st  %g2, [ %g1 ]
   0x00001280 <+40>:    ld  [ %fp + 0x44 ], %g1
   0x00001284 <+44>:    ld  [ %g1 ], %g1
   0x00001288 <+48>:    mov  %g1, %i0
   0x0000128c <+52>:    restore
   ⋮

// 스택 프레임을 미리 할당 (save %sp, -104, %sp)
// %fp를 통해 caller(Init)의 인자 저장 위치에 쉽게 접근 가능
```



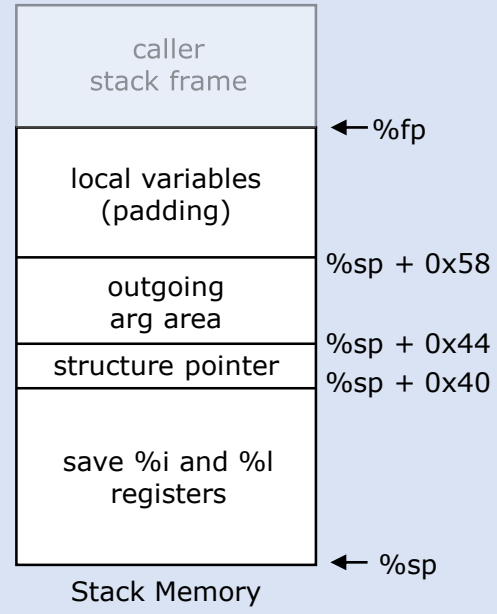
EXAMPLE: USING STACK(2)

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  Func2(1, 2, 3, 4, 5, 6, 7, 8);

  TEST_END();
  exit(0);
}
```

```
static void Func2(int a1, int a2, int a3,
int a4, int a5, int a6, int a7, int a8)
{
  volatile int mix = a1 + a2 + a3 + a4
                   + a5 + a6 + a7 + a8;
  printf("%d\n", mix);
}
```

procedure_call_example-3/main.c

```
(gdb) break Init
(gdb) continue
(gdb) disassemble Init
Dump of assembler code for function Init:
   ⋮
   0x000012e4 <+8>:     mov  8, %g1
   0x000012e8 <+12>:    st  %g1, [ %sp + 0x60 ]
   0x000012ec <+16>:    mov  7, %g1
   0x000012f0 <+20>:    st  %g1, [ %sp + 0x5c ]
   0x000012f4 <+24>:    mov  6, %o5
   0x000012f8 <+28>:    mov  5, %o4
   0x000012fc <+32>:    mov  4, %o3
   0x00001300 <+36>:    mov  3, %o2
   0x00001304 <+40>:    mov  2, %o1
   0x00001308 <+44>:    mov  1, %o0
   0x0000130c <+48>:    call  0x1258 <Func2>
   0x00001310 <+52>:    nop
```



// out register가 담을 수 있는 개수 이상의 인자가 들어오면, stack memory를 이용함
// outgoing arg area(%sp + 0x58) 위쪽에 존재하는 padding 영역에 추가로 저장

EXAMPLE: USING STACK(2)

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  Func2(1, 2, 3, 4, 5, 6, 7, 8);

  TEST_END();
  exit(0);
}
```

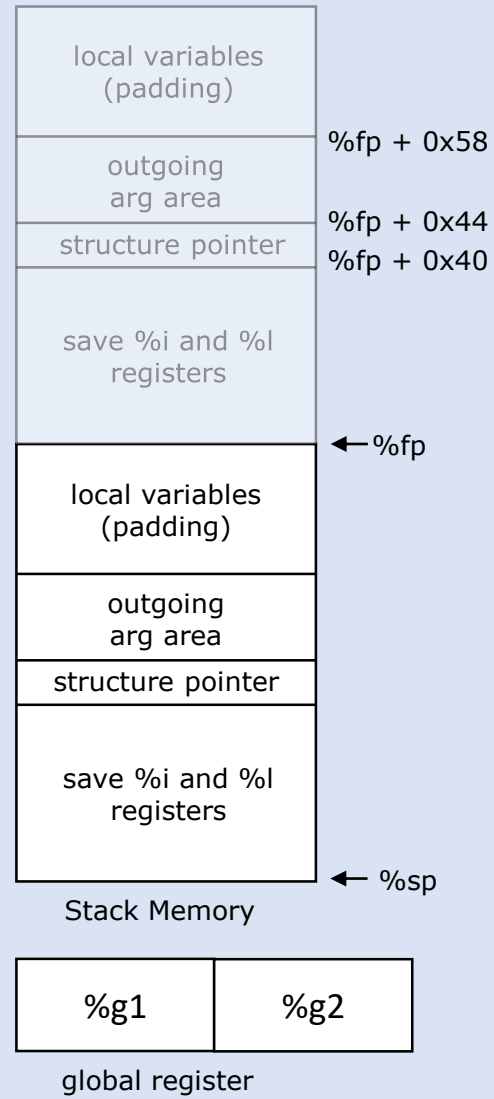
```
static void Func2(int a1, int a2, int a3,
int a4, int a5, int a6, int a7, int a8)
{
  volatile int mix = a1 + a2 + a3 + a4
                   + a5 + a6 + a7 + a8;
  printf("%d\n", mix);
}
```

procedure_call_example-3/main.c

```
(gdb) disassemble Func2
Dump of assembler code for function Func2:
   0x00001258 <+0>:     save  %sp, -104, %sp
   0x0000125c <+4>:     st  %i0, [ %fp + 0x44 ]
   0x00001260 <+8>:     st  %i1, [ %fp + 0x48 ]
   0x00001264 <+12>:    st  %i2, [ %fp + 0x4c ]
   0x00001268 <+16>:    st  %i3, [ %fp + 0x50 ]
   0x0000126c <+20>:    st  %i4, [ %fp + 0x54 ]
   0x00001270 <+24>:    st  %i5, [ %fp + 0x58 ]
   0x00001274 <+28>:    ld  [ %fp + 0x44 ], %g2
   0x00001278 <+32>:    ld  [ %fp + 0x48 ], %g1
   ⋮
(gdb) x/wx $fp+0x48
0x2da38:        0x00000002
(gdb) x/wx $fp+0x44
0x2da34:        0x00000001

(gdb) info register g1 g2
g1             0x2             2
g2             0x1             1
```

stack에 저장되어 있던 값을 global register로 가져온 후 연산을 수행함

a1 + a2 수행 위해 필요한 인자 가져옴



REGISTER SAVING CONVENTION

- **Global Registers**
  - 모든 윈도우가 공유하는 변수
  - 레지스터 윈도우가 회전해도 %g 레지스터는 바뀌지 않음
  - 윈도우 전환과 무관하게 유지되는 값을 담기 위해 사용

- **Volatile**
  - 함수 A가 %g1에 중요한 값을 넣고 함수 B를 호출했는데, 함수 B도 %g1을 사용할 수 있음

    → 함수 A가 저장한 데이터는 유지되지 못한다
  - Global Register는 **연산 도중 잠깐 값을 담아 두는 용도**로만 사용함
  - 스택은 메모리 접근인 반면, %g는 레지스터이므로 빠르게 접근 가능

REGISTER SAVING CONVENTION

- **레지스터 충돌 방지**
  - 모든 레지스터 윈도우는 **동일한 이름의 레지스터 셋을 사용하지만, 물리적으론 다르다**
  - SAVE 명령어가 실행되면 이전 함수의 레지스터 셋은 하드웨어적으로 격리됨
    - Callee에서 local 레지스터를 갱신해도 Caller의 local에는 영향이 가지 않는다
    - 인자 전달을 위해 Caller의 out, Callee의 in 레지스터만 같은 물리적 공간을 사용함

EXAMPLE: PASSING DATA

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  volatile int res = Func3(10);
  printf("%d\n", res);

  TEST_END();
  exit(0);
}
```

```
int Func3(int base)
{
  volatile int first, second;
  first = Mult(base, 2);
  second = Mult(base, 3);

  return first + second;
}
```

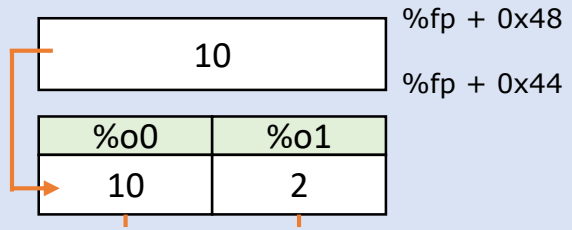
```
int Mult(int x, int y)
{
  return x * y;
}
```

procedure_call_example-4/main.c

(gdb) **disassemble Func3**

Dump of assembler code for function Func3:

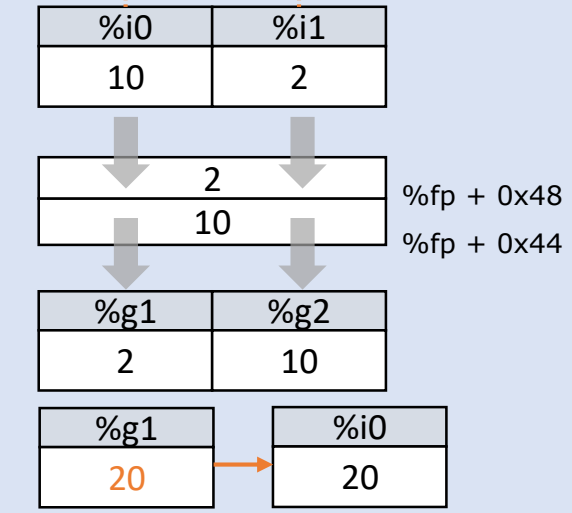
```
   0x00001288 <+8>:    mov  2, %o1
   0x0000128c <+12>:   ld  [ %fp + 0x44 ], %o0
   0x00001290 <+16>:   call  0x1258 <Mul>
   ⋮
```



(gdb) **disassemble Mult**

Dump of assembler code for function Mul:

```
   0x00001258 <+0>:    save  %sp, -96, %sp
   0x0000125c <+4>:    st  %i0, [ %fp + 0x44 ]
   0x00001260 <+8>:    st  %i1, [ %fp + 0x48 ]
   0x00001264 <+12>:   ld  [ %fp + 0x44 ], %g2
   0x00001268 <+16>:   ld  [ %fp + 0x48 ], %g1
   0x0000126c <+20>:   smul  %g2, %g1, %g1
   0x00001270 <+24>:   mov  %g1, %i0
   0x00001274 <+28>:   restore
```



// Func3이 인자를 저장한 %o0, %o1 레지스터는 Mult의 %i0, i1에 대응된다

// 스택에서 %g1, %g2 레지스터로 인자 값을 받아서 연산 수행

EXAMPLE: PASSING DATA

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  volatile int res = Func3(10);
  printf("%d\n", res);

  TEST_END();
  exit(0);
}
```

```
int Func3(int base)
{
  volatile int first, second;
  first = Mult(base, 2);
  second = Mult(base, 3);

  return first + second;
}
```

```
int Mult(int x, int y)
{
  return x * y;
}
```

procedure_call_example-4/main.c

```
(gdb) info reg o0 o1 g1 g2
o0             0x14            20
o1             0x2             2
g1             0x14            20
g2             0xa             10
```

%i0	%i1
20	2

%o0	%o1
20	2

%g1	%g2
20	10

```
(gdb) disassemble Func3
Dump of assembler code for function Func3:
   ⋮
   0x00001298 <+24>:    mov  %o0, %g1                    // 다음 함수 위한 인자 준비
   0x0000129c <+28>:    st  %g1, [ %fp + -4 ]
   0x000012a0 <+32>:    mov  3, %o1
=> 0x000012a4 <+36>:    ld  [ %fp + 0x44 ], %o0
   0x000012a8 <+40>:    call  0x1258 <Mult>
```

```
(gdb) info reg o0 o1 g1 g2                            // %o0, %o1에만 인자 저장
o0             0x14            10                     // %g1, %g2는 임시 저장소
o1             0x3             3
g1             0x14            20
g2             0xa             10
```

%o0	%o1
10	3

%g1	%g2
20	10

EXERCISE 1: FUNCTION CALLS AND REGISTER WINDOW

- **1. 함수 호출에 따른 레지스터 값의 변화 관찰하기**
  - SPARC V8은 함수 호출에 의해 발생하는 오버헤드를 줄이기 위해 레지스터 윈도우를 사용한다. 다음의 코드를 참고하여 in, out 레지스터에 대해 이해한다.

```
/* C 코드 */
int Mult(int x, int y, int z)
{
  return x*y*z;
}
```

```
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();
  final_res = Mult(1,2,3);

  TEST_END();
  exit(0);
}
```

```
/* SPARC 어셈블리 코드 */
(gdb) disassemble Init
Dump of assembler code for function sum:
   ⋮
   0x00001264 <+12>:    mov  3, %o2
   0x00001268 <+16>:    mov  2, %o1
   0x0000126c <+20>:    ld  [ %fp + 0x44 ], %o0
   0x00001270 <+24>:    call  0x1298 <mult>
(gdb) step
mult (x=10, y=2, z=3) at main.c:29

(gdb) info reg i7
i7  ____________________________
```

EXERCISE 1: FUNCTION CALLS AND REGISTER WINDOW

- 1.1 다음 빈칸에 들어갈 값을 작성하시오. (16진수로 작성)

- 1.2 이후에 다음 명령어를 실행했을 때, 나올 결과에 대해 예측하시오.

```
(gdb) info reg i0 i1 i2
```

EXERCISE 1: FUNCTION CALLS AND REGISTER WINDOW

- **문제 해설(1.1)**
  - 정답: 0x1270
  - Mult 함수에 들어가기 전, Init 함수에서 call을 실행한 위치는 다음과 같다

```
(gdb) disassemble Init
Dump of assembler code for function sum:
   ⋮
 0x00001270 <+24>:    call  0x1298 <mult>
```

  - call 함수 호출 시점의 주소가 caller의 %o7에 저장되고, 그 값이 callee의 %i7로 매핑 된다
  - 따라서 i7 레지스터의 값은 0x1270이 될 것이다

EXERCISE 1: FUNCTION CALLS AND REGISTER WINDOW

- **문제 해설(1.2)**
  - 정답: %i0는 1, %i1은 2, %2는 3의 값을 가지고 있을 것이다
  - Mult 함수에 들어가기 전, 인자들을 저장하는 코드는 다음과 같다.

```
(gdb) disassemble Init
Dump of assembler code for function sum:
   ⋮
   0x00001264 <+12>:    mov  3, %o2
   0x00001268 <+16>:    mov  2, %o1
   0x0000126c <+20>:    ld  [ %fp + 0x44 ], %o0
```

- [%fp + 0x44]는 이전 스택 프레임에서 인자를 저장하는 공간이고, c코드 상에서 1,2,3이라는 인자를 전달하기 때문에 [%fp + 0x44]안에는 1이 들어 있다고 짐작할 수 있다.

  - Mult 함수로 진입하면 레지스터 윈도우가 회전한다.
  - 이전에 %o0~%o2에 저장된 값은 %i0~%i2에 매핑된다.

EXERCISE 2: FUNCTION ARGUMENT PASSING AND THE STACK

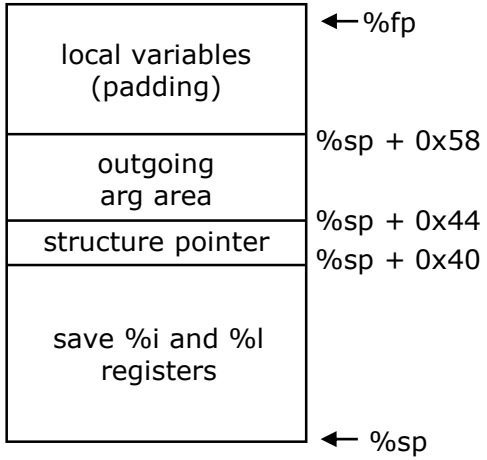
- 2. ①번은 스택에 저장되는 반면, ②번의 경우 %o5 레지스터에 저장된다. 이 이유에 대해 설명하시오.

```
/* C 코드 */
rtems_task Init(
  rtems_task_argument ignored
)
{
  (void) ignored;
  TEST_BEGIN();

  Func2(1, 2, 3, 4, 5, 6, 7, 8);

  TEST_END();
  exit(0);
}
```

```
/* SPARC 어셈블리 코드 */
(gdb) disassemble Init
Dump of assembler code for function Init:
   ⋮
   0x000012e4 <+8>:     mov  8, %g1 ..... ①
   0x000012e8 <+12>:    st  %g1, [ %sp + 0x60 ]
   0x000012ec <+16>:    mov  7, %g1
   0x000012f0 <+20>:    st  %g1, [ %sp + 0x5c ]
   0x000012f4 <+24>:    mov  6, %o5 ..... ②
   ⋮
```

EXERCISE 2: BIT MANIPULATION OF TBR REGISTER

- **문제 해설**
  - 8-out register를 통해 담을 수 있는 인자의 개수는 총 6개이다.
    - %o0~%o5: 함수 호출 시 인자 전달용 (6개)
    - %o6: 스택 포인터(sp)
    - %o7: call 시점의 주소 저장(return address 계산에 사용)
  - 본 예제에서는 총 8개의 인자를 전달했기 때문에, 2개의 인자는 스택에 저장된다.



- 인자로 전달된 1~6까지는 out-register에 저장되고, 7,8은 현재 스택 프레임 내의 추가 저장 공간에 위치할 것이다. (%sp + 0x58이후)
- 4 byte 단위이므로 각각 [%sp + 0x5c], [%sp + 0x60]에 저장된다.