SPARC V8
Control Flow 실습

인하대학교 컴퓨터공학과

12211568 김관

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.

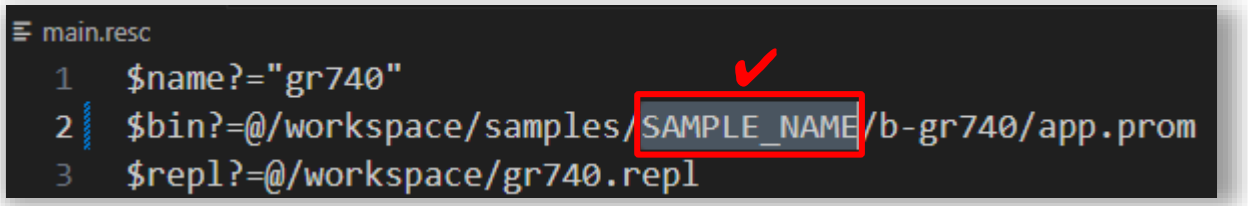
본 자료가 SPARC 기반 시스템의 동작 원리를 이해하고
디버깅 역량을 향상시키는 데 유용한 참고 자료가 되기를 바랍니다.

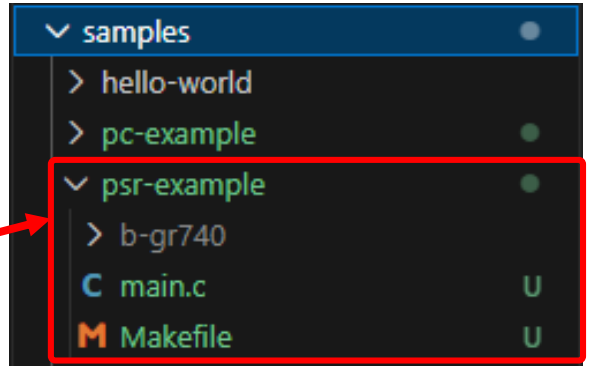
OVERVIEW

- **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

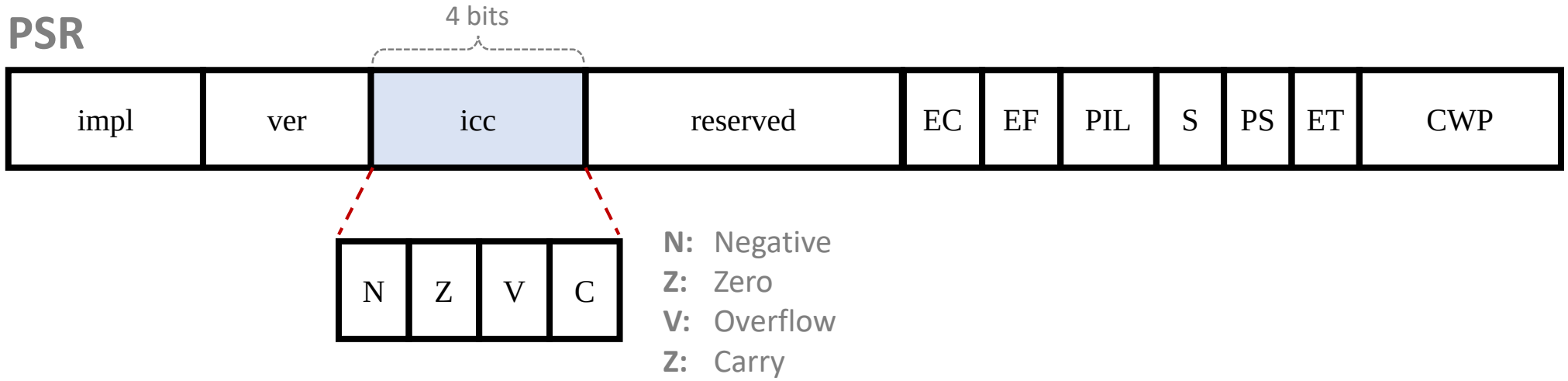
STATUS REGISTER

- **Status Registers**
  - SPARC 아키텍처에는 정수 연산에 대한 상태 레지스터 **PSR** Processor State Register,
    부동소수점 연산에 대한 상태 레지스터 **FSR** Floating-Point State Register 가 존재

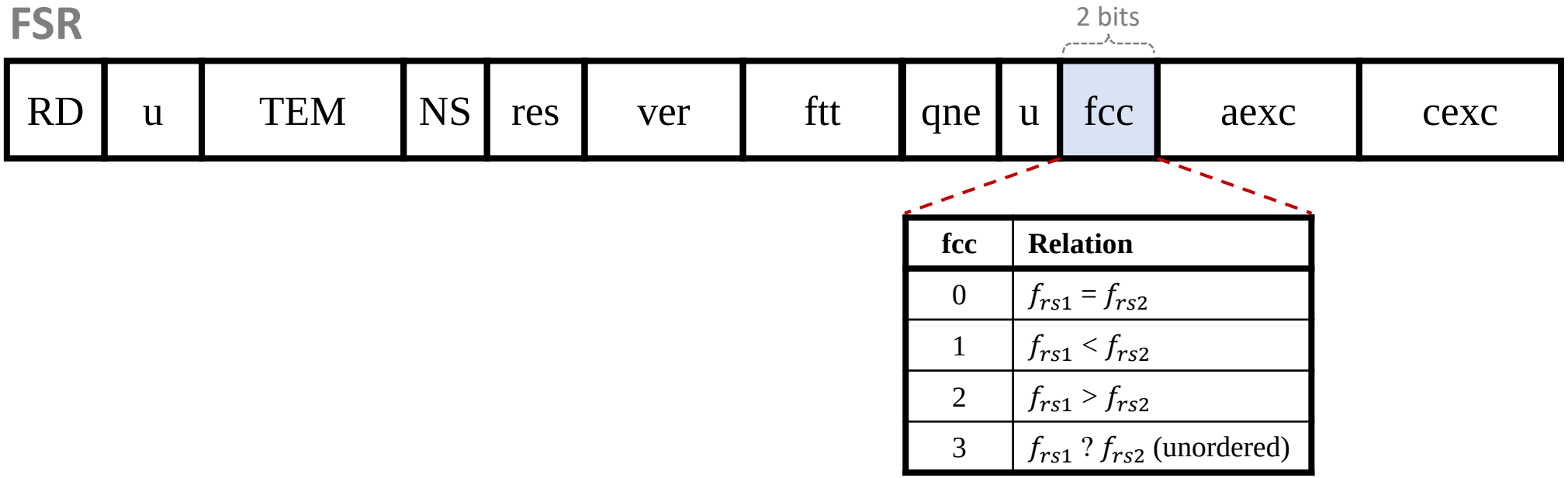
- **Condition Code**
  - 연산 상태를 나타내는 플래그
  - 주로 이름이 **cc**로 끝나는 산술 및 논리 명령어(예: ADDcc. ANDcc)의
    연산 결과에 따라 갱신됨
  - FBfcc와 같은 **분기 Branch 명령어**는
    이 필드에 기반하여 제어 전달을 수행함

STATUS REGISTERS

PSR



FSR



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

CONTROL-TRANSFER INSTRUCTIONS

제어-전달 명령어 Control-Transfer Instruction 는 다음 프로그램 카운터 **nPC** 값을 변경한다.

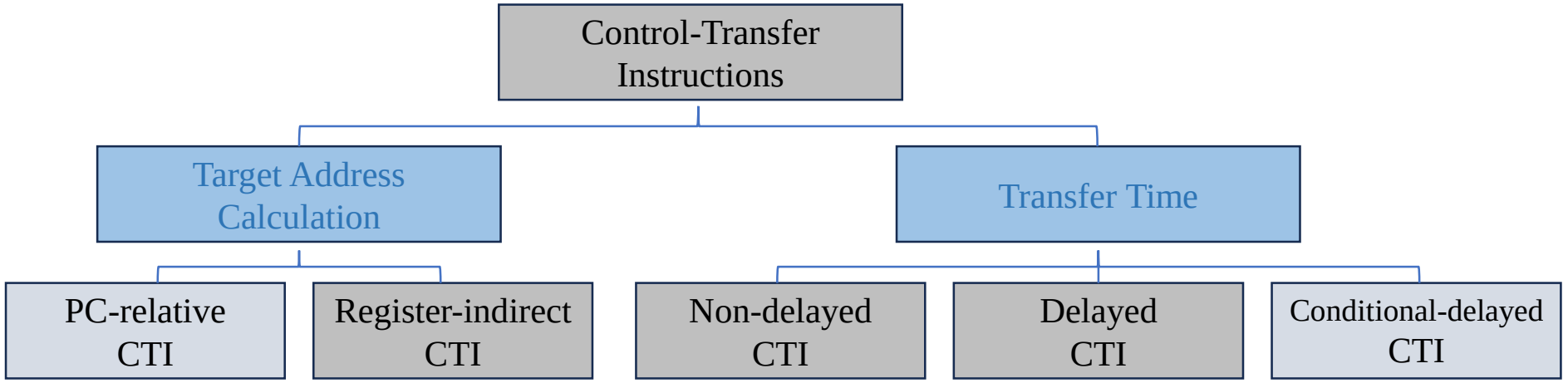
PC Program Counter 레지스터는 현재 실행 중인의 명령어의 주소를 담고,

nPC Next Program Counter 레지스터는 다음에 인출될 명령어의 주소를 담는다.

SPARC 아키텍처의 제어-전달 명령어들은

대상 주소 계산 방식과 **제어 전달 시점**에 따라 다음과 같이 분류할 수 있다.

Condition Code에 따라 분기를 결정하는 명령어 **Bicc**와 **FBfc**는 **PC-relative / Conditional-delayed** CTI이다.



CONDITION DEPENDENT INSTRUCTION

- **Branch on Integer Condition Codes Instructions (Bicc)**

opcode	*operation*	*icc test*
`ba`	Branch Always	1
`bn`	Branch Never	0
`bne`	Branch on Not Equal	not Z
`be`	Branch on Equal	Z
`bg`	Branch on Greater	not (Z or (N xor V))
`ble`	Branch on Less of Equal	Z or (N xor V)
`bge`	Branch on Greater or Equal	not (N xor V)
`bl`	Branch on Less	N xor V
`bgu`	Branch on Greater Unsigned	not (C or Z)
`bleu`	Branch on Less or Equal Unsigned	(C or Z)
`bcc`	Branch on Carry Clear (Greater than or Equal, Unsigned)	not C
`bcs`	Branch on Carry Set (Less than, Unsigned)	C
`bpos`	Branch on Positive	not N
`bneg`	Branch on Negative	N
`bvc`	Branch on Overflow Clear	not V
`bvs`	Branch on Overflow Set	V

CONDITION DEPENDENT INSTRUCTION

- **Branch on Floating-point Condition Codes Instructions (FBfcc)**

opcode	*operation*	*fcc test*
`fba`	Branch Always	1
`fbn`	Branch Never	0
`fbu`	Branch on Unordered	U
`fbg`	Branch on Greater	G
`fbug`	Branch on Unordered or Geeater	G or U
`fbl`	Branch on Less	L
`fbul`	Branch on Unordered or Less	L or U
`fblg`	Branch on Less or Greater	L or G
`fbne`	Branch on Not Equal	L or G or U
`fbe`	Branch on Equal	E
`fbue`	Branch on Unordered or Equal	E or U
`fbge`	Branch on Greater or Equal	E or G
`fbuge`	Branch on Unordered or Greater or Equal	E or G or U
`fble`	Branch on Less or Equal	E or L
`fbule`	Branch on Unordered or Less or Equal	E or L or U
`fbo`	Branch on Ordered	E or L or G

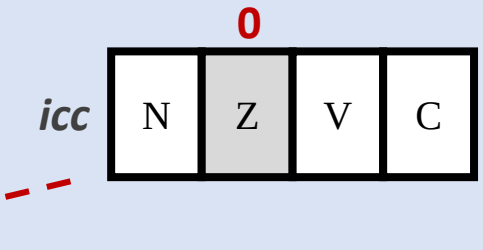
EXAMPLE: BRANCH AFTER INTEGER OPERATION

```
rtems_task Init(rtems_task_argument
ignored)
{
    …
    uint32_t seed = get_runtime_seed();
    int_if_else_demo(seed);
    float_if_else_demo(seed + 1U);
    int_for_while_demo(seed + 2U);
    …
}
```

```
__attribute__((noinline)) static void
int_if_else_demo(uint32_t seed)
{
    int a = (int)(seed & 0x1fU);
    int b = (int)((seed >> 3) & 0x0fU) + 1;
    int sum = a + b;

    if ((sum & 1) == 0) {
        printf(...);
    } else {
        printf(...);
    }
}
```

control-flow-example/main.c

```
  ⋮
(gdb) disassemble int_if_else_demo
  ⋮
  0x00001268 <+0>:     and  %o0, 0x1f, %o2
  0x0000126c <+4>:     srl  %o0, 3, %o3
  0x00001270 <+8>:     sethi  %hi(0), %g1
  0x00001274 <+12>:    and  %o3, 0xf, %o3
  0x00001278 <+16>:    xor  %g1, -36, %g1
  0x0000127c <+20>:    inc  %o3
  0x00001280 <+24>:    add  %o2, %o3, %o4
  0x00001284 <+28>:    btst  1, %o4
  0x00001288 <+32>:    bne  0x12a4 <int_if_else_demo+60>
  ⋮
  0x000012a4 <+60>: sethi  %hi(0x1d000), %o1          // else
  0x000012a8 <+64>:    or  %o1, 0x398, %o1
  0x000012ac <+68>:    mov  %o7, %g1
  0x000012b0 <+72>:    call  0x19e20 <fiprintf>
```

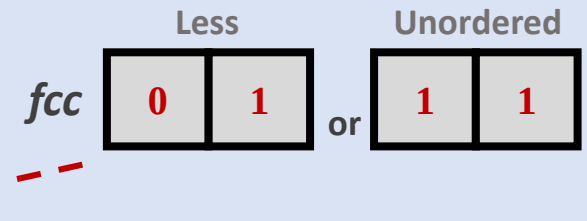


EXAMPLE: BRANCH AFTER INTEGER OPERATION

```
__attribute__((noinline)) static void
float_if_else_demo(uint32_t seed)
{
    double x =
        (double)((seed % 50U) + 1U);
    double y =
        (double)(((seed / 3U) % 9U) + 1U);
    double ratio = x / y;
    double threshold =
        (double)(((seed / 5U) % 4U) + 1U);

    if (ratio >= threshold) {
        printf(...);
    } else {
        printf(...);
    }
}
```

control-flow-example/main.c

```
  ⋮
(gdb) disassemble float_if_else_demo
  ⋮
  0x00001380 <+200>:   fcmped  %f14, %f12
  ⋮
  0x000013a0 <+232>:   fbul  0x13bc <float_if_else_demo+260>
  0x000013a4 <+236>:   ldd  [ %fp + -16 ], %o2
  0x000013a8 <+240>:   sethi  %hi(0x1d000), %o1
  0x000013ac <+244>:   call  0x19e20 <fiprintf>
  0x000013b0 <+248>:   or  %o1, 0x3c0, %o1     ! 0x1d3c0
  0x000013b4 <+252>:   ret
  0x000013b8 <+256>:   restore
  0x000013bc <+260>: sethi  %hi(0x1d000), %o1          // else
  0x000013c0 <+264>:   call  0x19e20 <fiprintf>
  ⋮
```



EXAMPLE: LOOP AFTER INTEGER OPERATION

```
__attribute__((noinline)) static void
int_for_while_demo(uint32_t seed)
{
    int n = (int)(seed % 8U) + 3;
    int for_sum = 0;
    int while_sum = 0;
    int i = 1;

    for (i = 1; i <= n; ++i) {
        for_sum += i;
    }
    printf(...);

    i = 1;
    while (i <= n) {
        while_sum += i;
        ++i;
    }
    printf(...);
}
```

control-flow-example/main.c

```
(gdb) disassemble int_for_while_demo
  ⋮
 /* for */
  0x000013dc <+12>:    clr  %o3
  0x000013e0 <+16>:    add  %i2, 3, %i2
  0x000013e4 <+20>:    add  %o3, %g1, %o3
  0x000013e8 <+24>: inc  %g1
  0x000013ec <+28>:    cmp  %i2, %g1
  0x000013f0 <+32>:    bge,a   0x13e8 <int_for_while_demo+24>
  0x000013f4 <+36>:    add  %o3, %g1, %o3
  ⋮
```

```
 /* while */
  ⋮
  0x00001418 <+72>:    clr  %i3
  0x0000141c <+76>:    add  %i3, %g1, %i3
  0x00001420 <+80>: inc  %g1
  0x00001424 <+84>:    cmp  %i2, %g1
  0x00001428 <+88>:    bge,a   0x1420 <int_for_while_demo+80>
  0x0000142c <+92>:    add  %i3, %g1, %i3
```

PC-RELATIVE CALCULATION

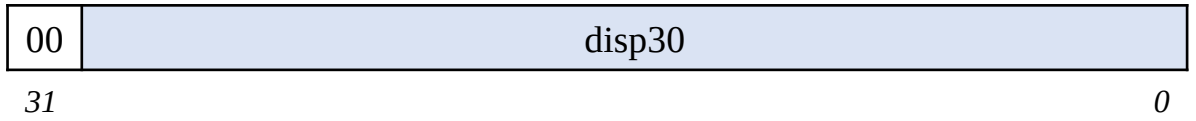
- **PC-상대 주소 지정** **PC-relative Calculation**
  - 해당 명령어들은 PC 레지스터 값을 기준으로 계산
  - 명령어 포멧 자체에 포함된 상태 오프셋을
    현재 PC값(현재 주소)에 더하여 최종 주소를 얻음
  - **Branch**와 **Call**이 이에 해당
  - **명령어 주소**는 항상 **4바이트 정렬**이라
    하위 2비트가 **항상 00**이므로,
    저장할 때는 **비트 절약**을 위해 빼고,
    실제 주소로 쓸 때 다시 붙임

Bicc Format

FBfcc Format

$$\mathbf{nPC \leftarrow PC + sign\_extend(disp22[]00_2)}$$

Call and Link Format

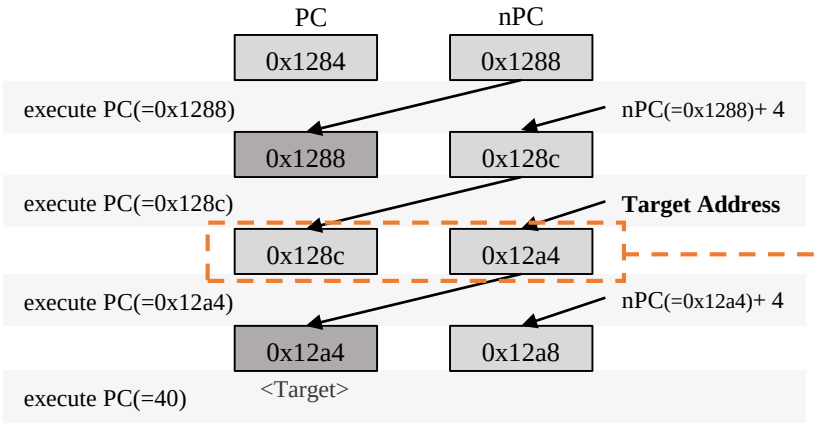


$$\mathbf{nPC \leftarrow PC + disp30[]00_2}$$

EXAMPLE: PC-RELATIVE CALCULATION

```
__attribute__((noinline)) static void
int_if_else_demo(uint32_t seed)
{
    int a = (int)(seed & 0x1fU);
    int b = (int)((seed >> 3) & 0x0fU) + 1;
    int sum = a + b;

    if ((sum & 1) == 0) {
        printf(...);
    } else {
        printf(...);
    }
}
```

control-flow-example/main.c



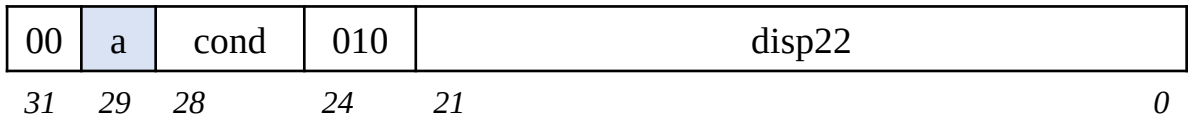

```
(gdb) disassemble int_if_else_demo
   ⋮
   0x00001288 <+32>:    bne  0x12a4 <int_if_else_demo+60>
   ⋮
(gdb) x /wx 0x1288
   0x1288 <int_if_else_demo+32>:   0x12800007

(gdb) p /t *(unsigned int*)0x1288          // 분기 명령어 opcode 읽기
   1001010 0000000000000000000111          // disp22 필드: 7
                 disp22
(gdb) b *0x1288
(gdb) c
(gdb) info reg pc npc
   pc             0x1288          0x1288 <int_if_else_demo+32>
   npc            0x128c          0x128c <int_if_else_demo+36>
(gdb) si
(gdb) info reg pc npc
   pc             0x128c          0x128c <int_if_else_demo+36>
   npc            0x12a4          0x12a4 <int_if_else_demo+60>
```

$nPC \leftarrow PC + disp22[]00_2$

CONDITIONAL-DELAYED TRANSFER

- **조건부 지연** **Conditional-delayed Transfer**
  - 조건부 지연 명령어는 **annul 비트(a)**와
    분기 조건의 성립 여부에 따라 제어 흐름을 결정
  - **a 비트**는 **분기 명령어** **Branch** 에서만 지정할 수 있음

Format 2:



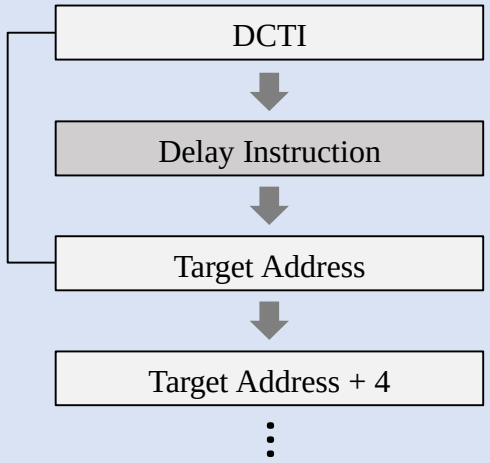
- **지연 명령어 Delay Instruction (DI)**
  - 제어-전달 명령어를 만났을 때 nPC 레지스터가 가리키는 명령어 주소
  - 일반적으로 바로 다음에 오는 순차적인 명령어 (PC+4)

a bit	Type of branch	Delay instruction executed?
a = 0	conditional, taken conditional, not taken	Yes Yes
a = 1	conditional, taken conditional, not taken	Yes No (annulled)

EXAMPLE: A = 1, BRANCH TAKEN

```
int func(int n, int x)
{
   if (n != 0) {
        x = x + 1;
        return x;
    }
   return x;
}
```

```
/* n in %o0, x in %o1, return in %o0 */

  func:
        subcc       %o0, 0
        bne,a       target          // ,a : a = 1
        add         %o1, 1, %o1     // 지연 명령어 Delay Instruction
        retl
        mov         %o1, %o0

  target:
        retl
        mov     %o1, %o0
```



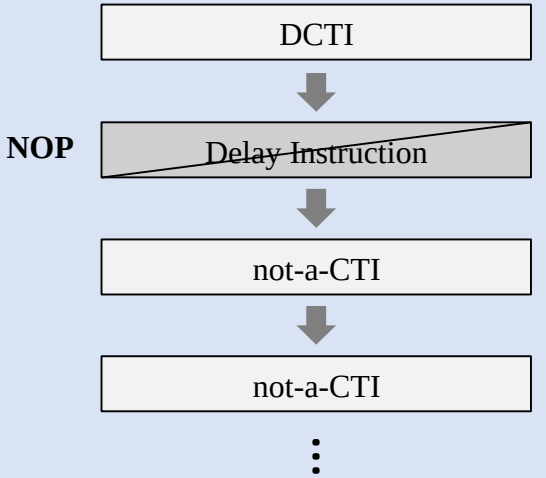
// n이 0이 아닐 때, 즉 **분기 조건을 만족할 때,**
// 지연 명령어가 취소되지 않고 순차적으로 수행된다.

// 이러한 어셈블리 구조는 사용자와 무관하게
// 컴파일러가 아키텍처 특성에 맞춰 성능 최적화를 위해 자동 생성한다.

EXAMPLE: A = 1, BRANCH NOT TAKEN

```
int func2(int n, int x)
{
  if (n > 5) {
      x = x + 1;
  }
   return x;
}
```

```
/* n in %o0, x in %o1, return in %o0 */

 func2:
        subcc       %o0, 5
        bg,a        target          // ,a : a = 1
        add         %o1, 1, %o1     // 지연 명령어 Delay Instruction
        retl
        mov         %o1, %o0

 target:
        retl
        mov     %o1, %o0
```



// n이 5보다 작을 때, 즉 **분기 조건을 만족하지 않을 때,**
// 이미 nPC가 가리키던 지연 명령어는 취소되고, 그 다음 명령어로 넘어간다.

// 이러한 어셈블리 구조는 사용자와 무관하게
// 컴파일러가 아키텍처 특성에 맞춰 성능 최적화를 위해 자동 생성한다.