



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 4

함수 호출

1. 소개

함수 호출은 모듈화 및 코드 재사용을 허용하고, 코드 작성 및 디버깅을 더 쉽게 할 수 있어 모든 프로그램에서 중요한 부분입니다. C 프로그래밍 언어에는 난수 생성기 및 일반 수학 함수와 같이 일반적으로 사용되는 C 함수의 프로세서/보드 특정 라이브러리뿐만 아니라 표준 라이브러리도 포함됩니다. 고급 함수는 *호출 규칙*에 따라 어셈블리로 변환됩니다. 이 LAB에서는 C 프로그램에서 함수를 작성하고 사용하는 방법을 보여줍니다. 두 함수 모두 프로그래머가 작성한 함수와 C 라이브러리에 포함된 함수입니다. 또한 함수가 어셈블리 언어로 구현되는 방법을 보여줍니다. LAB이 끝나면 함수와 라이브러리 호출을 사용하는 프로그램 작성에 대한 연습을 제공합니다.

2. 함수를 사용하는 C 프로그램 작성

함수 (서브 루틴[subroutine] 또는 프로시저 [procedure] 라고도 함)는 정의된 작업 및 인터페이스 (입력 및 출력)가 있는 코드 블록으로 패키징된 코드입니다. 이러한 모듈성은 복잡성을 줄이고 코드 재사용을 지원하여 효율성을 높입니다. 함수가 종료되면 함수 호출 직후 프로그램 실행이 재개되는 방식으로 프로그램의 어느 지점에서나 함수를 호출할 수 있습니다. 함수는 다른 함수 (중첩[nested] 함수라고 함) 또는 동일한 함수 (재귀[recursive] 호출이라고 함)에서 호출될 수 있습니다.

함수가 있는 RISC-V 프로그램을 작성하려면 LAB 2 및 3에 설명된 것과 동일한 일반 단계를 따릅니다.

1. RVfpga 프로젝트 만들기
2. C 프로그램 작성
3. Nexys A7 FPGA 보드에 RVfpgaNexys 다운로드 (Verilator 와 Whisper 를 사용하여 시뮬레이션 환경에서 이러한 프로그램을 수행할 수 있습니다)
4. 프로그램 컴파일, 다운로드 및 실행/디버깅

이러한 단계에 대한 자세한 지침은 LAB 2를 참조하십시오. 다음은 각 단계에 대한 간략한 설명입니다.

1 단계. RVfpga 프로젝트 만들기

다음 폴더에 project1이라는 프로젝트를 만듭니다.

```
[RVfpgaPath]/RVfpga/Labs/Lab4
```

2 단계. C 프로그램 작성

이제 프로젝트에 C 프로그램을 추가합니다. 새 파일을 만들고 프로젝트에 다음 C 프로그램을 입력하거나 복사/붙여넣기 합니다. 이 프로그램은 다음 파일에서도 사용할 수 있습니다.

```
[RVfpgaPath]/RVfpga/Labs/Lab4/LedsSwitches_functions.c
```

```
// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }
```

```
void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main ( void )
{
    unsigned int switches_val;

    IOsetup();
    while (1) {
        switches_val = getSwitchVal();
        writeValtoLEDs(switches_val);
    }

    return(0);
}

void IOsetup()
{
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal()
{
    unsigned int val;

    val = READ_GPIO(GPIO_SWs);    // read value on switches
    val = val >> 16;    // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val)
{
    WRITE_GPIO(GPIO_LEDs, val);    // display val on LEDs
}
```


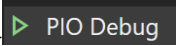


프로젝트의 src 디렉토리에 파일을 저장하고 파일 이름을 LedsSwitches_Functions.c 로 지정합니다.

3 단계. Nexys A7 FPGA 보드에 RVfpgaNexys 다운로드

RVfpga Labs 2 및 3 에서했던 것처럼 Nexys A7 보드에 RVfpgaNexys 를 다운로드 합니다.

4 단계. 프로그램 컴파일, 다운로드 및 실행

이제 RVfpgaNexys 에서 프로그램을 컴파일, 다운로드 및 실행/디버그할 준비가 되었습니다.

실행  및 디버깅 시작 버튼  을 누른 후 getSwitchVal () 함수를 호출하는 19 행에 도달할 때까지 Step Over 버튼  (상단 도구 모음에 있음)을 클릭하거나 또는 F10 을 두 번 눌러줍니다. 그런 다음 Step Into 버튼  (또는 F11)을 누릅니다. 이후 getSwitchVal () 함수로 들어갑니다. 아직 함수를 볼 수 없는 경우, 왼쪽 도구 모음의 VARIABLES → Local 필드를 확장하여 val 변수를 봅니다. val 변수는 프로그램의 이 시점에서 "최적화 됨"으로 나열될 수 있습니다. Step Over 또는 Step Into 를 한 번 수행하고 그림 1 과 같이 스위치 값에 대한 val 변수 변경을 확인합니다.

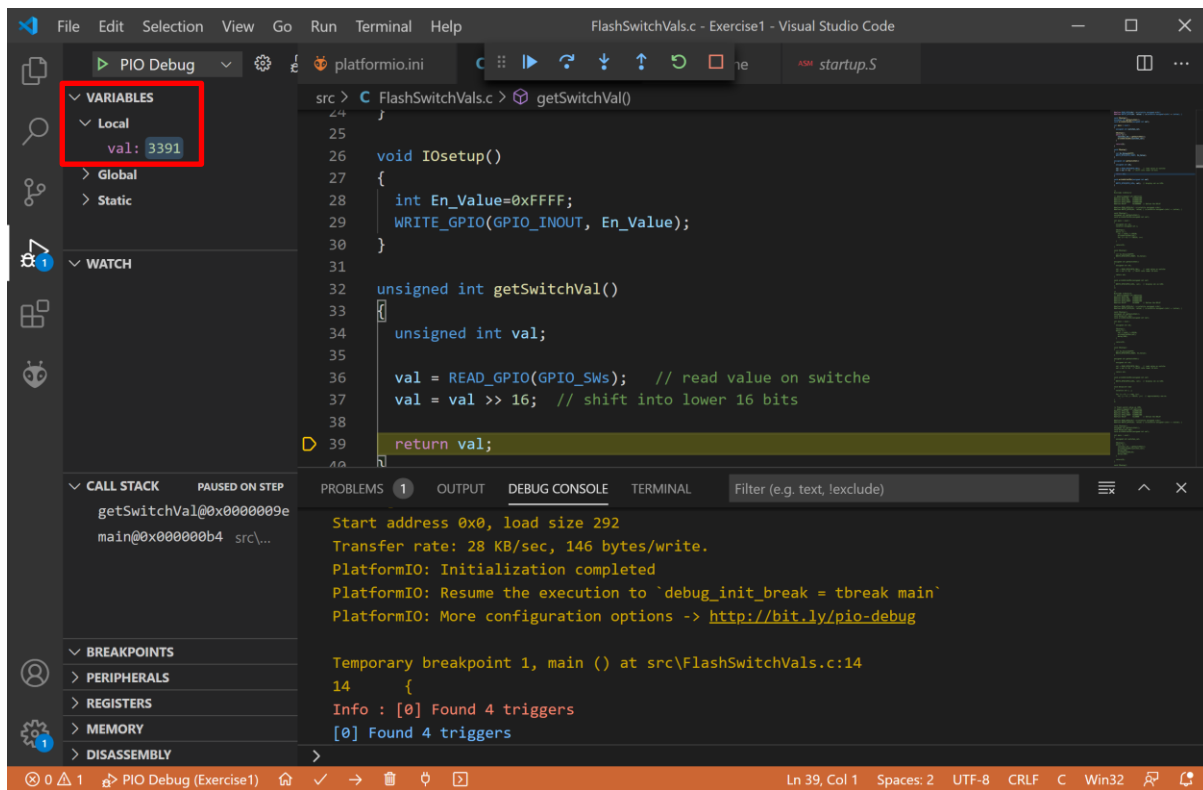


그림 1. getSwitchVal () 함수로 들어가기

이제 줄의 왼쪽을 클릭하여 19 번째 줄에 중단점을 설정합니다. 그림 2 와 같이 이제 중단점 임을 나타내는 빨간색 점이 왼쪽에 나타납니다.

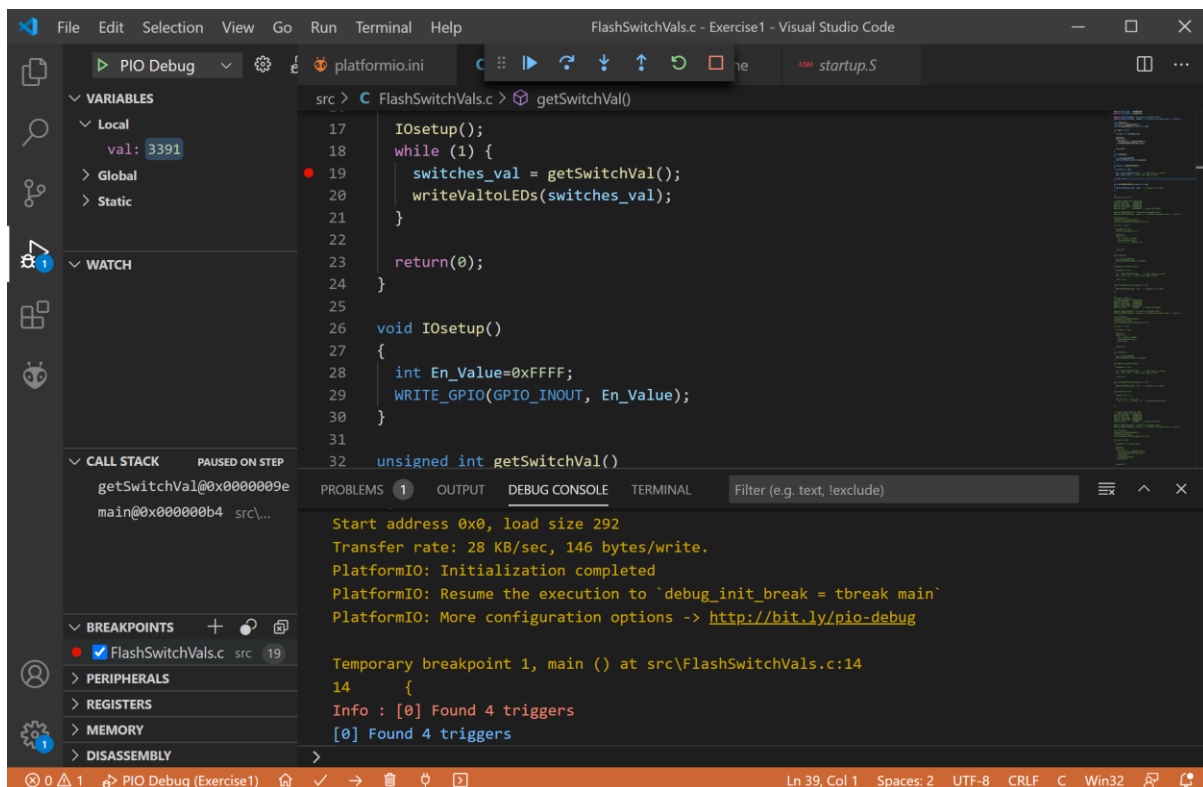




그림 2. 중단점 설정

계속 버튼  (또는 F5)을 누릅니다. 중단점에 도달하면 프로그램이 19 행에서 중지됩니다. 이번에는 Step Over 버튼  (또는 F10)을 누릅니다. 함수는 실행되지만 디버거는 함수에 들어 가지 않습니다. 기능의 효과만 표시됩니다. 특히, switch_val 변수는 그림 3 과 같이 스위치의 값이 됩니다.

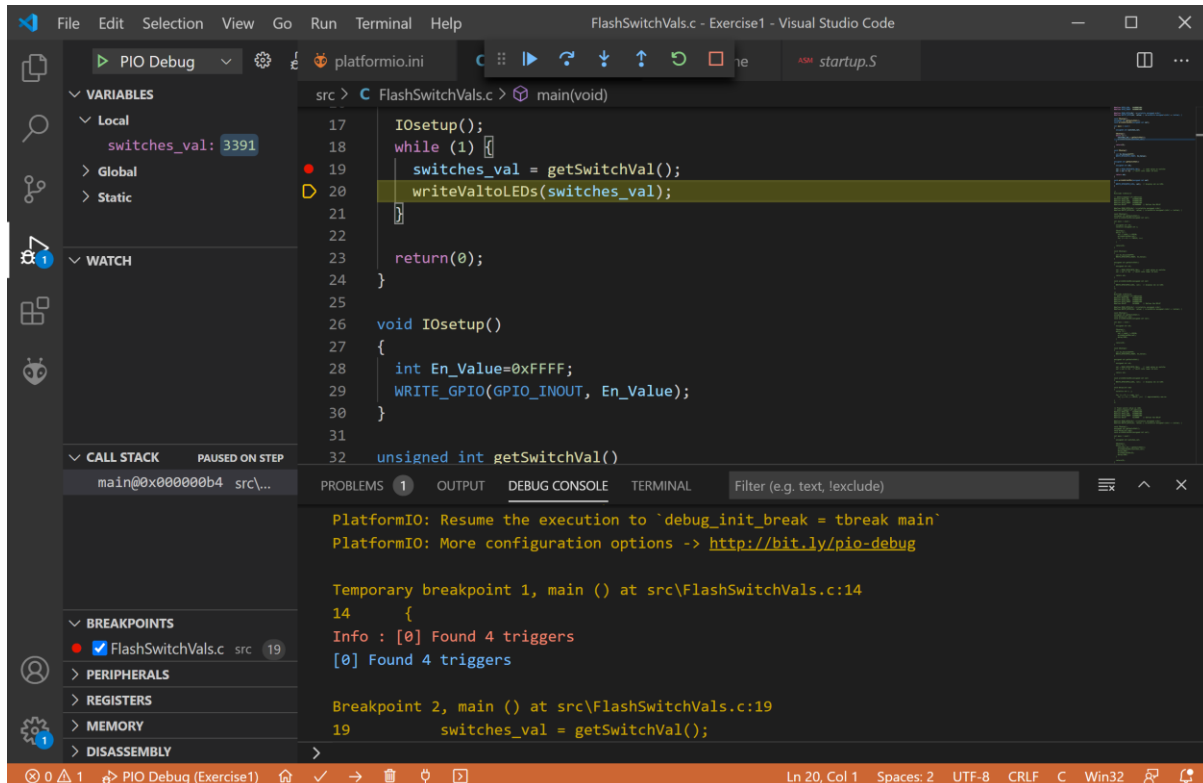


그림 3. 함수 건너 뛰기

3. 라이브러리 함수 호출로 C 프로그램 작성

C 와 같은 고급 프로그래밍 언어에는 프로그래머가 일반적으로 사용하는 함수 라이브러리가 포함됩니다. "C 표준 라이브러리"를 검색하여 일반적으로 사용되는 C 라이브러리 목록을 찾을 수 있습니다. 이러한 함수 라이브러리는 포함된 함수의 선언을 제공하는 헤더 파일을 포함하여 사용할 수 있습니다. 이것은 C 프로그램 파일의 맨 위에 다음 행을 추가하여 수행됩니다.

```
#include <libraryname>
```

"libraryname"은 라이브러리의 이름으로 대체됩니다. 예를 들어, 수학 라이브러리 (math.h)는 부동 소수점 숫자의 절대 값을 계산하는 fabs (), 두 부동 소수점 숫자 중 가장 큰 숫자를 반환하는 fmax () 등과 같은 일반적인 함수를 제공합니다.

또 다른 공통 라이브러리는 C 표준 라이브러리 (stdlib.h)입니다. 이 라이브러리에 포함된 일부 함수는 난수를 생성합니다. 예를 들어, 아래 프로그램은 stdlib.h 헤더 파일 (#include <stdlib.h>)을 포함하고 난수를 반환하는

rand () 함수를 호출하여 LED 에 난수를 표시합니다. 아래 프로그램을 PlatformIO RVfpga 프로젝트에 복사하여 붙여 넣고 Nexys A7 FPGA 보드의 RVfpgaNexys 에서 실행하십시오.

```
#include <stdlib.h>

// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408
#define DELAY         0x1000000 // Define the DELAY

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main(void)
{
    unsigned int val;
    volatile unsigned int i;

    IOsetup();
    while (1) {
        val = rand() % 65536;
        writeValtoLEDs(val);
        for (i = 0; i < DELAY; i++)
            ;
    }
    return(0);
}

void IOsetup() {
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal() {
    unsigned int val;

    val = READ_GPIO(GPIO_SWs); // read value on switches
    val = val >> 16; // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val) {
    WRITE_GPIO(GPIO_LEDs, val); // display val on LEDs
}
```

이 프로그램은 다음 파일에서도 사용할 수 있습니다.

```
[RVfpgaPath]/RVfpga/Labs/Lab4/RandomNumberLEDs.c
```

이러한 C 표준 라이브러리 외에도 Western Digital (WD)이 제공하는 펌웨어 패키지 (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) 내에서 특정 패키지를 SweRV EH1 프로세서는 (PSP, `~/.platformio/packages/framework-wd-riscv-sdk/psp/`)에서 Nexys A7 보드는 (BSP, `~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp/`)에서 찾을 수 있습니다.

시작 안내서 (섹션 6.F - *HelloWorld_C-Lang* 프로그램)에서 설명했듯이 이러한 라이브러리는 `platformio.ini`에 적절한 줄을 추가하고 C 프로그램 시작 부분에 적절한 파일을 포함하여 프로젝트에 포함됩니다.

이러한 라이브러리는 프로그래머가 인터럽트를 사용하고, 문자열을 인쇄하고, 개별 레지스터를 읽고/쓸 수 있도록 하는 함수와 매크로를 제공합니다. RVfpga 시작 안내서와 labs에서는 예제와 실습을 통하여 많은 기능을 사용하게 됩니다.

4. RISC-V 호출 규칙

이 섹션에서는 고급 함수가 RISC-V 어셈블리 언어로 변환되는 방식을 정의하는 RISC-V 호출 규칙에 대해 설명합니다. 이 호출 규칙은 ABI (Application Binary Interface)의 일부입니다. 규칙을 정의하면 다른 프로그래머가 작성하거나 라이브러리에 포함된 함수를 프로그램간에 사용할 수 있습니다. RISC-V에서 점프 및 링크 명령 (`jal`)은 함수 호출을 호출합니다. 예를 들어, 다음 코드는 `func1` 함수를 호출합니다:

```
jal func1
```

이 명령어는 `func1` 레이블로 점프하고 `jal` 이후의 명령어 주소를 반환 주소 레지스터 (`ra = x1`)에 저장합니다. 그런 다음 함수는 반환 (`ret`) 의사 명령어 (또는 점프 레지스터 명령어: `jr ra`)를 사용하여 반환하며, 이는 `ra`에 저장된 주소로 점프합니다.

함수는 입력 인수를 사용하여 호출될 수 있으며 호출 함수에 값을 반환할 수도 있습니다. RISC-V 규칙에 따라 입력 인수는 레지스터 `a0-a7`의 함수에 전달됩니다. 추가 인수가 필요한 경우 스택에 배치됩니다. 다시 규칙에 따라 반환 값은 레지스터 `a0` 및 `a1`에 배치됩니다. 레지스터가 인수를 전달하고 값을 반환하는 데 사용되는 규칙은 RISC-V 호출 규칙에 의해 정의됩니다.

프로그램의 모든 위치에서 함수를 안전하게 호출하려면 함수가 기계의 구조적 상태 (즉, 프로그래머가 볼 수 있는 레지스터의 내용)를 보존하는 것이 필수적입니다. 루프의 인덱스를 저장하기 위해 레지스터 `t0`을 사용하는 루프가 있고 `main` 함수가 있는 프로그램이 있다고 가정합니다. 루프 본문에서 `SortVector`라는 함수가 호출되고 이 함수 `SortVector`는 벡터 `A`의 주소를 저장하기 위해 레지스터 `t0`을 사용합니다 (그림 4 참조). 따라서 레지스터 `t0`은 함수 `SortVector`에서 덮어 쓰여져, 루프의 인덱스를 수정하여 실행이 잘못되는 부작용이 있습니다.

```

main:
    add t2, zero, M
    add t0, zero, zero
    ...
loop1:
    bge t0, t2, endloop1
    ...
    jal SortVector
    ...
    add t0, t0, 1
    j loop1
endloop1:
    ...
    ret

SortVector:
    ...
    la t0, A
    ...
    ret

```

그림 4. 메인 프로그램과 기능 간의 레지스터 사용 충돌 예

main 프로그램의 프로그래머가 루프 인덱스 (예: t1)를 구현하기 위해 다른 레지스터를 선택했다면 이것은 분명히 발생하지 않았을 것입니다. 그러나 프로그래머가 함수를 호출하기 전, 함수 구현의 모든 세부 사항을 안다는 것은 힘든 일입니다. (어떤 경우에도 가능하지 않습니다).

보다 실용적인 해결책은 모든 함수가 수정될 모든 레지스터의 메모리에 임시 복사본을 만들고 호출 프로그램으로 돌아 가기 전에 원래 값을 복원하는 것입니다. 이 솔루션은 LIFO (Last-In-First-Out) 정책을 사용하여 액세스되는 메모리 영역인 **Call Stack** 을 통해 구현됩니다. 이 영역은 프로그램의 라이브 기능 (즉, 시작되었지만 완료되지 않은 기능)과 관련된 모든 정보를 사용 가능한 메모리의 끝 (즉, 상위 주소)에서 시작하며 하위 주소 방향으로 저장하는 데 사용됩니다.

함수는 일반적으로 세 부분으로 구성됩니다.

- ➔ Entry code (**Prologue**)
- ➔ Function **Body**
- ➔ Exit code (**Epilogue**)

*프로로그*는 필요한 경우 함수의 **스택 프레임(stack frame)**을 만들고 레지스터를 스택에 저장해야 합니다. *스택 프레임*은 실행중인 함수가 사용하는 메모리 영역입니다. *에필로그*는 호출자 프로그램의 아키텍처 상태를 복원하고 *스택 프레임*이 차지하는 메모리 공간을 해제하여 *프로로그*를 실행하기 전의 스택을 그대로 둡니다.

스택에 대한 액세스는 스택의 마지막 점유 위치 주소를 저장하는 스택 포인터 (sp = x2)라고 하는 포인터를 통해 관리됩니다. 프로그램이 시작되기 전에 sp 는 스택베이스의 주소 (즉, 스택 영역의 가장 높은 주소)로 초기화되어야 합니다. RVfpga 시스템에서 sp 레지스터는 `~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/startup.S` 파일에 구현된 `_start` 함수에 의해 초기화됩니다. 초기화시 스택은 비어 있습니다. 두 번째 포인터인 프레임 포인터 (fp = x8)는 활성 함수 스택 프레임의 기본 주소 (즉, 가장 높은 주소)를 가리 킵니다.

함수는 **스택 프레임**을 함수 자체에서만 액세스할 수 있는 개인 메모리 영역으로 사용합니다. **스택 프레임**의 일부는 함수에 의해 수정될 아키텍처 레지스터의 사본을 저장하는 데 사용되며, 경우에 따라 메모리 위치를 통해 함수에 매개 변수를 전달하는 방법으로도 사용될 수 있습니다.

표 1은 RISC-V 규칙이 각 정수 레지스터에 할당하는 의도된 역할을 설명합니다. 표 1에도 나와 있듯이 일부 레지스터는 호출된 함수에 의해 보존되어야 하는 반면 다른 레지스터는 함수에 의해 덮어 쓸 수 있습니다 (즉, 보존되지 않음).

- 함수가 보존된 레지스터를 덮어 써야 하는 경우 먼저 **스택 프레임**에서 해당 레지스터의 복사본을 만들고 **호출자**(즉, 호출한 함수)에게 반환하기 전에 값을 복원해야 합니다. 스택 포인터 (sp) 및 반환 주소 레지스터 (ra) 외에도 12 개의 정수 레지스터 s0–s11 은 호출 간에 보존되며 사용하는 경우 **호출 수신자**가 저장해야 합니다.
- 반면에 **호출자**는 일부 레지스터를 **호출 수신자**가 보존할 필요가 없으므로 호출 후, 손실될 수 있음을 알고 있어야 합니다. 인수 및 반환 값 레지스터 (a0–a7) 외에도 7 개의 정수 레지스터 t0–t6 은 호출시 휘발성이며 함수 호출 후 다시 사용되는 경우 **호출자**가 저장해야 하는 임시 레지스터입니다.

표 1. RISC-V 정수 레지스터

Name	Register Number	Use	Preserved
zero	x0	상수 값 0	-
ra	x1	반송 주소	Yes
sp	x2	스택 포인터	Yes
gp	x3	글로벌 포인터	-
tp	x4	스레드 포인터	-
t0–2	x5–7	임시 변수	No
s0/fp	x8	저장된 레지스터/프레임 포인터	Yes
s1	x9	저장된 레지스터	Yes
a0–1	x10–11	함수 인수/반환 값	No
a2–7	x12–17	함수 인수	No
s2–11	x18–27	저장된 레지스터	Yes
t3–6	x28–31	임시 변수	No

그림 4의 예제에서는 이 규칙에 따른 두 가지 솔루션이 있습니다:

- main 프로그램은 t0 대신 SortVector 함수 (예: s0)에 의해 보존되는 루프 인덱스 용 레지스터를 사용할 수 있습니다.
- main 함수는 t0 을 계속 사용할 수 있지만 SortVector 를 호출하기 전에 스택의 내용을 보존하고 SortVector 에서 반환한 후 복원해야 합니다.

스택은 함수의 스택 프레임에 더 많은 메모리가 필요하면 확장되고 해당 함수가 완료되면 축소됩니다. 스택은 아래로 (하위 주소로) 성장하고 스택 포인터는 프로시저 입력시 16 바이트 경계에 정렬됩니다. 표준 ABI 에서 스택 포인터는 프로시저 실행하는 동안 정렬된 상태를 유지해야 합니다.

예제

다음 예제에서는 먼저 C (그림 5)에서 다음으로 RISC-V 어셈블리 언어 (그림 6)에서 정렬 알고리즘을 구현합니다. 입력은 N 개 요소의 배열 A 이며, 각각 0 보다 큰 정수 입니다. 출력은 A 의 요소를 내림차순으로 저장하는 또 다른 배열 B 입니다.

C 에서 main 함수는 배열 A 와 B 의 주소와 크기 (N)를 수신하고 A 의 요소를 내림차순으로 B 요소에 저장하는 SortVector 함수를 호출합니다. 이 SortVector 함수는 배열 A 의 주소와 크기를 수신하고 배열 A 의 최대 값을 반환하고 해당 값을 재설정하여 다음 반복에서 더 이상 고려되지 않도록 다른 함수인 MaxVector 를 호출합니다.

```
#define N 8

int MaxVector(int A[], int size)
{
    int max=0, ind=0, j;
    for(j=0; j<size; j++){
        if(A[j]>max){
            max=A[j];
            ind=j;
        }
    }
    A[ind]=0;
    return(max);
}

int SortVector(int A[], int B[], int size)
{
    int max, j;
    for(j=0; j<size; j++){
        max=MaxVector(A, size);
        B[j]=max;
    }
    return(0);
}

int main ( void )
{
    int A[N]={7,3,25,4,75,2,1,1}, B[N];
    SortVector(A, B, N);
    return(0);
}
```

그림 5. C 언어의 정렬 알고리즘

그림 6 은 어셈블리로 작성된 동일한 알고리즘을 보여줍니다. 이전 섹션에서 설명한 개념을 고려하여 프로그램을 분석합니다.

- main 기능

○ 프로로그

- 먼저 함수에 사용되는 보존된 레지스터를 저장하기 위해 스택에 공간이 예약됩니다:
add sp, sp, -16. 규칙에 따라 sp 레지스터는 128 비트 버전의 RISC-V, RV128I 와의 호환성을 유지하기 위해 항상 16 바이트로 정렬되어야 합니다.
- 이 기능에 저장된 레지스터가 사용되지 않으므로 s0-s11 레지스터를 스택에 저장할 필요가 없습니다. 그러나 main 은 ra 에 저장된 값을 업데이트하는 SortVector 함수를 호출하므로 레지스터 ra 를 저장해야 합니다.

○ Function Body

- `SortVector` 함수는 `jal SortVector` 명령어를 사용하여 호출됩니다.
함수를 호출하기 전에 호출 규칙에 따라 3 개의 입력 매개 변수가 레지스터 `a0` (A 주소), `a1` (B 주소) 및 `a2` (A 및 B 배열 크기)에 배치됩니다.

○ 에필로그

- 프롤로그 (`ra`)에서 스택에 저장된 레지스터가 이제 복원됩니다.
- 스택 포인터 (`sp`)도 초기 위치 (`add sp, sp, 16`)로 복원됩니다.

- `SortVector` 함수

○ 프롤로그

- 먼저 함수에서 사용되는 보존된 레지스터를 저장하기 위해 스택에 공간을 예약합니다:
`add sp, sp, -32`
- 그러면 함수 (`s1-s3`)에서 사용하는 저장된 레지스터가 하나씩 스택에 저장됩니다.
- `SortVector` 가 `ra` 에 저장된 값을 덮어 쓰는 `MaxVector` 함수를 호출하기 때문에 레지스터 `ra` 도 저장해야 합니다.

○ 기능 본체

- 먼저 입력 매개 변수 (`a0`, `a1` 및 `a2`)가 보존된 레지스터 (`s1`, `s2` 및 `s3`)로 이동되어 `MaxVector` 함수 실행 후 사용할 수 있습니다.
- 벡터 B 를 계산하는 경우 각 반복에서 A 의 최대 값을 계산하여 B 에 저장하는 루프가 구현 됩니다. A 의 최대 값을 계산하기 위해 `MaxVector` 함수가 루프의 각 반복에서 (`jal MaxVector`) 호출됩니다. 함수를 호출하기 전에 호출 규칙에 따라 이 함수에 대한 입력 매개 변수가 레지스터 `a0` 및 `a1` 로 이동됩니다. 함수 실행이 완료되면 레지스터 `a0` 에 A 의 최대 값을 반환합니다.
- 루프는 대부분 저장된 레지스터를 사용하여 변수를 저장합니다. 이러한 레지스터는 `MaxVector` 실행 후 값을 보존하기 위해 RISC-V 호출 규칙에 의해 보장됩니다 (즉, 함수가 값을 보존해야 함).
- 레지스터 `a0` 및 `a1` 은 함수로 수정할 수 있습니다. 따라서 모든 호출 전에 준비해야 합니다.
- 레지스터 `t1` 은 `MaxVector` 가 반환된 후 재사용 되어야 합니다. 따라서 함수를 호출하기 전에 (`sw t1, 16(sp)`) `SortVector` 의 스택에 보존하고 실행 후 복원해야 합니다 (`lw t1, 16(sp)`).

○ 에필로그

- 프롤로그 중에 스택에 저장된 레지스터가 이제 복원됩니다.
- 스택 포인터 (`sp`)도 초기 위치 (`add sp, sp, 32`)로 복원됩니다.

- - `MaxVector` 기능

○ 프롤로그

- 먼저 함수에 사용되는 보존된 레지스터를 저장하기 위해 스택에 공간이 만들어집니다:
`add sp, sp, -16`
- 그런 다음 함수에서 사용하는 저장된 레지스터 (즉, 레지스터 `s1`)가 스택에 저장됩니다: `sw s1, 0(sp)`. 레지스터가 이 함수에 의해 저장되지 않으면 벡터 A 의 주소를 저장하는 데 이 레지스터를 사용하므로 호출자 함수 (`SortVector`)의 실행이 실패합니다.

- 이 함수는 다른 함수 (리프 함수)를 호출하지 않기 때문에 이 경우 ra 를 저장할 필요가 없습니다.
- 기능 본체
 - 이 함수는 s1 과 일부 임시 레지스터를 사용하여 배열 A 의 최대 값을 계산합니다.
- 에필로그
 - 함수는 호출자에게 반환하기 전에 반환 값 mv a0, t2 를 준비해야 합니다.
 - 프롤로그 (s1) 동안 스택에 저장된 레지스터가 복원됩니다.
 - 스택 포인터 (sp)도 초기 위치 (add sp, sp, 16)로 복원됩니다.

```
.globl main

.equ N, 8

.data
A: .word 7,3,25,4,75,2,1,1

.bss
B: .space 4*N

.text

MaxVector:
    add sp, sp, -16
    sw s1, 0(sp)

    mv s1, zero
    mv t2, zero
loop2:
    beq s1, a1, endloop2
    lw t1, (a0)
    ble t1, t2, else2
    mv t2, t1
    mv t3, a0
else2:
    add a0, a0, 4
    add s1, s1, 1
    j loop2
endloop2:
    sw zero, (t3)

    mv a0, t2
    lw s1, 0(sp)
    add sp, sp, 16
    ret

SortVector:
    add sp, sp, -32
    sw s1, 0(sp)
    sw s2, 4(sp)
    sw s3, 8(sp)
    sw ra, 12(sp)

    mv s1, a0           # Address of vector A
    mv s2, a1           # Address of vector B
    mv s3, a2           # Size of vectors A and B
    mv t1, zero

loop1:
    beq t1, s3, endloop1
    mv a0, s1
    mv a1, s3
```

```

        sw t1, 16(sp)
        jal MaxVector
        lw t1, 16(sp)
        sw a0, (s2)
        add s2, s2, 4
        add t1, t1, 1
        j loop1
    endloop1:

    lw s1, 0(sp)
    lw s2, 4(sp)
    lw s3, 8(sp)
    lw ra, 12(sp)
    add sp, sp, 32
    ret

main:
    add sp, sp, -16
    sw ra, 0(sp)

    la a0, A
    la a1, B
    add a2, zero, N
    jal SortVector

    lw ra, 0(sp)
    add sp, sp, 16
    ret

.end

```

그림 6. 어셈블리 언어의 정렬 알고리즘

그림 7 은 MaxVector 함수의 본문을 실행하는 시점의 스택 상태를 보여줍니다.

- main 함수의 *스택 프레임*은 파란색으로 표시되며 해당 함수의 반환 주소 (ra)를 포함합니다.
- SortVector 함수의 스택 프레임은 녹색으로 표시되며 이 함수에서 사용하는 저장된 레지스터 (s1-s3), 레지스터 t1 및 ra 가 포함됩니다.
- 마지막으로 *활성 스택 프레임* (실행중인 함수의 *스택 프레임*) 인 MaxVector 함수의 스택 프레임은 노란색으로 표시되며 이 함수에서 사용하는 저장된 레지스터 (s1)를 포함합니다.

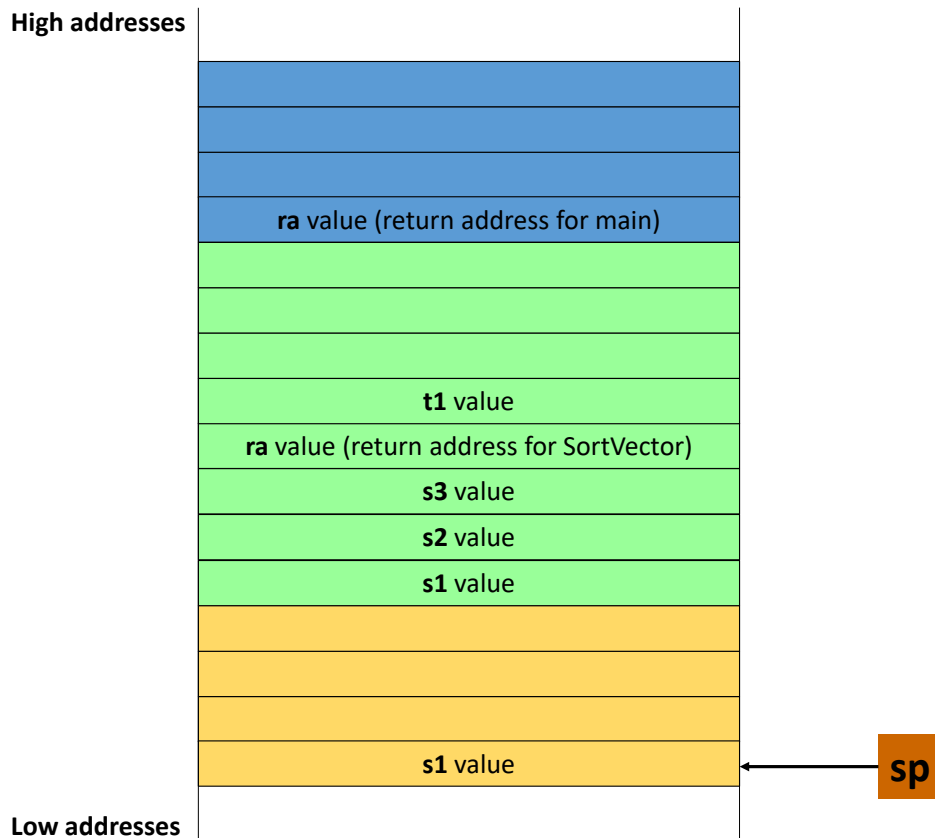


그림 7. 그림 6의 어셈블리 프로그램에 대한 **MaxVector** 함수 본문의 스택 상태

작업: 그림 6의 어셈블리 프로그램은 `[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions`에서 사용할 수 있는 PlatformIO 프로젝트에서 제공됩니다. RISC-V 호출 협약에 따라 다양한 레지스터 (s, ra, a 등)에 저장된 값과 스택에 저장된 값을 분석하기 위한 단계별 디버거 옵션을 사용하여 보드 (또는 ISS 시뮬레이터)에서 프로그램을 실행 합니다.

- 프로그램 컴파일 후 PlatformIO에 의해 생성된 `.pio/build/swervolf_nexys/firmware.dis` 파일은 프로그램의 각 명령어 주소를 알기 위해 유용할 수 있습니다.

- 메모리 콘솔을 사용하여 스택의 진화와 어레이 A 및 B의 내용을 분석할 수 있습니다.

- 이 프로젝트에서는 sp 레지스터가 16 바이트로 정렬되도록 조정된 `link.ld` 스크립트를 사용합니다.

스크립트는 `[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions/ld/link.ld`에서 찾을 수 있습니다.

sp 레지스터의 정렬은 `ALIGN()` 명령을 사용하여 실행됩니다:

```
.stack :
{
    _heap_end = .;
    . = . + __stack_size;
    /* Force 16-B alignment of SP register */
    . = ALIGN(16);
    _sp = .;
} > ram: ram_load
```

5. 연습

이제 다음 연습을 완료하여 함수 호출을 포함하는 고유 한 C/Assembly 프로그램을 만듭니다.

Nexys A7 보드를 컴퓨터에 연결하고 전원을 켜 상태로 두면 다른 프로그램간에 보드에 RVfpgaNexys 를 다시 로드할 필요가 없습니다. 그러나 Nexys A7 보드를 끄면 PlatformIO 를 사용하여 보드에 RVfpgaNexys 를 다시 로드해야 합니다.

Verilator 와 Whisper 를 사용하여 시뮬레이션 환경에서 이러한 프로그램을 수행할 수 있습니다.

연습 1. LED 스위치를 반대로 표시하는 C 프로그램을 작성하십시오. 프로그램 이름을 **DisplayInverse_Functions.c** 로 지정합니다.

예를 들어, 스위치가 (2 진수): 01010101010101 이면 LED 는 다음과 같이 표시되어야 합니다. 10101010101010; 스위치가 1111000011110000 이면 LED 에 0000111100001111 이 표시됩니다. 스위치의 반전된 값을 반환하는 getSwitchesInvert () 함수를 포함합니다. 함수 선언은 다음과 같습니다.

```
unsigned int getSwitchesInvert();
```

연습 2. LED 값을 스위치에 깜박이는 C 프로그램을 작성하십시오. 프로그램 이름을 **FlashSwitchesToLEDs_Functions.c** 로 지정 합니다.

이 값은 약 2 초마다 켜지고 꺼져야 합니다. "num" 밀리 초의 지연을 유발하는 delay ()라는 함수를 포함합니다. 이것은 경험적으로 수행할 수 있으며 정확할 필요는 없습니다. 함수 선언은 다음과 같습니다.

```
void delay(int num);
```

연습 3. 반응 시간을 측정하는 C 프로그램을 작성하십시오. 프로그램은 모든 LED 가 켜진 후 사람이 맨 오른쪽 스위치 (SW [0])를 켜는 데 걸리는 시간을 측정해야 합니다. stdlib.h 라이브러리의 rand () 함수를 사용하여 사용자가 반응 시간을 테스트할 때마다 지연할 임의의 시간을 생성합니다. 프로그램 이름을 **ReactionTime.c** 로 지정합니다.

프로그램은 다음과 같이 작동 합니다.

1. 사용자는 가장 오른쪽에 있는 스위치를 끄고 (아래로) 준비합니다.
2. 프로그램이 모든 LED 를 끈 다음 잠시 동안 기다립니다 (약 3 초 이하). 연습 2 의 delay () 함수를 사용하게 될 것입니다.
3. 그런 다음 모든 LED 가 켜지고 프로그램은 사용자가 맨 오른쪽 스위치를 켤 때까지 밀리 초를 계산하기 시작합니다.
4. 사용자가 맨 오른쪽 스위치 (SW [0])를 켜면 스위치를 켜는 데 걸린 시간 (밀리 초)이 LED 에 2 진수로 표시되고 직렬 콘솔에 10 진수로 표시됩니다.
5. 그런 다음 사용자가 오른쪽 스위치를 아래로 (꺼짐) 토글 하면 반복하게 됩니다.

연습 4. rand () 함수의 한 가지 문제는 예측 가능한 임의의 숫자 시퀀스를 사용한다는 것입니다. 즉, 프로그램을 실행할 때마다 동일한 난수로 시작하고 동일한 난수 순서를 따릅니다. 연습 3 에서 프로그램을 여러 번 실행하여 동일한 난수로 시작하고 동일한 난수 순서를 따르는지 확인합니다.

그러나 srand () 함수를 먼저 사용하면 rand () 함수에 임의의 시작점이 됩니다. 유일한 문제는 srand ()에 그 자체가 무작위이며 부호가 없는 정수인 입력 인수가 주어져야 한다는 것입니다. srand ()에 임의의 숫자, 예를 들면 사용자가 기능을 시작하기 위해 스위치를 끌 때까지의 시간을 (밀리 초) 지정 하십시오.

LED 가 켜지 기 전에 실제로 임의의 시간 순서를 생성하도록 연습 3 을 다시 작성 하십시오. 가능하면 기능을 사용하십시오. 프로그램 이름을 **ReactionTimeTrulyRandom.c** 로 지정합니다.

연습 5. LED 가 반응 시간에 비례하여 증가하는 LED bar 를 표시하도록 연습 4 를 다시 작성하십시오. 이렇게 하면 반응 시간을 보는 사람이 밀리 초 수의 이진 표현을 해석할 필요없이 속도가 빨라지고 있는지 더 쉽게 알 수 있습니다. 켜진 LED 의 각 범위에 해당하는 반응 시간 범위를 선택할 수 있습니다. 예를 들어 빠른 반응 시간을 위해 오른쪽에 있는 몇 개의 LED 만 켜야 합니다. 반응 시간이 늘어남에 따라 왼쪽에 있는 LED 수가 증가합니다. 매우 느린 반응 시간은 모든 LED 에 불이 들어옵니다. 프로그램 이름을 **ReactionTimeBar.c** 로 지정합니다.

연습 6. "Simon says" 게임을 구현하는 C 프로그램을 작성합니다. 다음이 결과가 나타납니다:

1. 프로그램은 가장 오른쪽에 있는 세 개의 LED 에서 패턴을 감박이고 사용자가 가장 오른쪽에 있는 세 개의 스위치를 사용하여 해당 스위치 시퀀스를 누를 때까지 기다립니다. Switches [2: 0]은 LED [2: 0]에 해당하며 LED [0]은 맨 오른쪽 LED 이고 Switches [0]은 맨 오른쪽 스위치입니다.
2. 임의의 패턴은 LED 1 개, LED 2 개, LED 3 개 등으로 시작해야 합니다.
3. 사용자는 가장 오른쪽에 있는 세 개의 스위치를 사용하여 시퀀스를 반복하려고 합니다. 사용자가 스위치를 위로 토글하면 해당 LED 가 켜지고 사용자가 스위치를 다시 아래로 토글하면 꺼집니다.
4. 사용자가 올바른 순서를 입력하면 일시 중지 후 다음 패턴이 표시되고 순서에 LED 가 하나 더 표시됩니다.
5. 사용자가 잘못된 시퀀스를 입력하면 LED 가 계속 켜져 있고 새로운 시퀀스가 재생되지 않습니다.
6. 가장 왼쪽에 있는 스위치 (Switches [15])를 위로 (on) 누른 다음 아래 (off)로 누르면 게임이 재설정됩니다.

선택한 기능을 사용하여 프로그램을 모듈화하고 작성, 디버그 및 이해를 더 쉽게 만듭니다. 프로그램을 작성하려면 원하는대로 표준 C 라이브러리를 사용해야 합니다. 프로그램 이름을 **SimonSays.c** 로 지정합니다.

연습 7. 3*N 요소의 벡터 A 가 주어지면, 우리는 N 요소의 새로운 벡터 B 를 얻고자 합니다. 그래서 B 의 각 요소는 A 의 연속 요소의 삼중(triplet) 항 합의 절대 값이 됩니다. 예를 들면:

$$B[0] = |A[0]+A[1]+A[2]|, \quad B[1] = |A[3]+A[4]+A[5]|, \quad \dots$$

Triplets.S 라는 RISC-V 어셈블리 프로그램을 작성합니다 (프로그램은 RISC-V 호출 규칙을 따라야 함).

- main 프로그램은 다음과 같은 상위 레벨 pseudo-code 에 따라 B 의 계산을 구현합니다.

```
#define N 4

int A[3*N] = {a list of 3*N values};
int B[N];
```



```
int i, j=0;

void main (void)
{
    for (i=0; i<N; i++){
        B[i] = res_triplet(A,j);
        j=j+3;
    }
}
```

- 함수 `res_triplet` 은 위치 `p` 에서 시작하여 벡터 `V` 의 3 개 연속 요소 합 의 절대 값을 반환합니다. 다음과 같은 상위 레벨 pseudo-code 에서 제공하는 사양에 따라 구현됩니다.

```
int res_triplet(int V[ ], int pos)
{
    int i, sum=0;
    for (i=0; i<3; i++)
        sum = sum + V[pos+i];
    sum=abs(sum);
    return sum;
}
```

- 함수 `abs(int x)` 는 입력 인수의 절대 값을 반환합니다.

연습 8. `Filter.S` 라는 RISC-V 어셈블리 프로그램을 작성하십시오 (프로그램은 이전에 연구한 기능 관리 표준을 준수해야 합니다). 아래 pseudo-code 를 사용할 수 있습니다.

```
#define N 6
int i, j=0, A[N]={48,64,56,80,96,48}, B[N];
for (i=0; i<(N-1); i++){
    if( (myFilter(A[i],A[i+1])) == 1){
        B[j]=A[i]+ A[i+1] + 2;
        j++;
    }
}
```

- 메모리 공간을 예약하는 데 필요한 지시문을 포함하여 해당하는 RISC-V 어셈블리 코드를 작성하고 해당 섹션 (.data, .bss 및 .text)을 선언합니다. `myFilter` 함수는 첫 번째 인수가 16 의 배수이고 두 번째 인수가 첫 번째 인수보다 큰 경우 값 1 을 반환합니다. 그렇지 않으면 0 을 반환합니다.
- `myFilter` 함수의 어셈블리 코드를 작성합니다.

연습 9. Coprimes.S 라는 RISC-V 어셈블리 프로그램을 작성하여 정수 쌍 목록 (> 0)이 coprime (또는 상호 소수) 숫자로 구성된 쌍을 찾습니다 (프로그램은 이전에 연구한 기능 관리 표준을 준수해야 함). 두 개의 숫자는 그들이 가진 유일한 공약수가 1 이면 coprime 이라고 이해하면 됩니다.

입력 데이터가 다음 형식의 배열 `D` 에 포함되어 있다고 가정합니다.

$$D = (x_0, y_0, c_0, x_1, y_1, c_1, \dots, x_{N-1}, y_{N-1}, c_{N-1})$$

각 삼중 선 (x_i , y_i , c_i)은 다음과 같이 해석됩니다. x_i 및 y_i 는 한 쌍의 숫자를 나타내며 c_i 는 처음에 0 입니다. 프로그램을 실행 한 후 다음과 같이 c_i 의 값이 수정되어야 합니다. x_i 및 y_i 가 coprime 이면 $c_i = 2$; 그렇지 않으면 $c_i = 1$.

예를 들면:

다음 입력 벡터의 경우: $D = (3,5,0, 6,18,0, 15,45,0, 13,10,0, 24,3,0, 24,35,0)$

최종 결과는 다음과 같아야 합니다. $D = (3,5,2, 6,18,1, 15,45,1, 13,10,2, 24,3,1, 24,35,2)$

- 배열 D 를 순회하고 아래 왼쪽 상자에 주어진 사양에 따라 결과를 생성하는 RISC-V 어셈블리 프로그램을 작성합니다. 프로그램은 함수 `check_coprime (int D [], int i)` 를 호출합니다. 입력 인수는 D 의 시작 주소와 확인하려는 쌍의 번호 (0 에서 $M-1$ 까지)입니다. 이 함수는 배열 D 의 i 번째 쌍의 번호가 coprime 인지 확인하고 결과를 해당 메모리 위치에 저장합니다.
- 아래 오른쪽 상자에 주어진 사양에 따라 `check_coprime` 함수에 대한 코드를 작성합니다. 함수 `gcd(int a, int b)` 는 유클리드(Euclidean) 알고리즘에 따라 Lab 3 에서 구현되었으며 두 입력 인수의 최대 공약수 (gcd)를 반환합니다. gcd 가 1 이면 숫자는 coprime 입니다.

<pre>#define M 6 int D[] = {a list of M*3 int values} void main () { int i; for (i=0; i<M; i++) check_coprime(D,i); }</pre>	<pre>void check_coprime (int A[], int pos) { int res; res = gcd(A[3*pos], A[(3*pos)+1]); if (res == 1) A[(3*pos)+2] = 2; else A[(3*pos)+2] = 1; }</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------