Computer Architecture

Network Computing Lab.

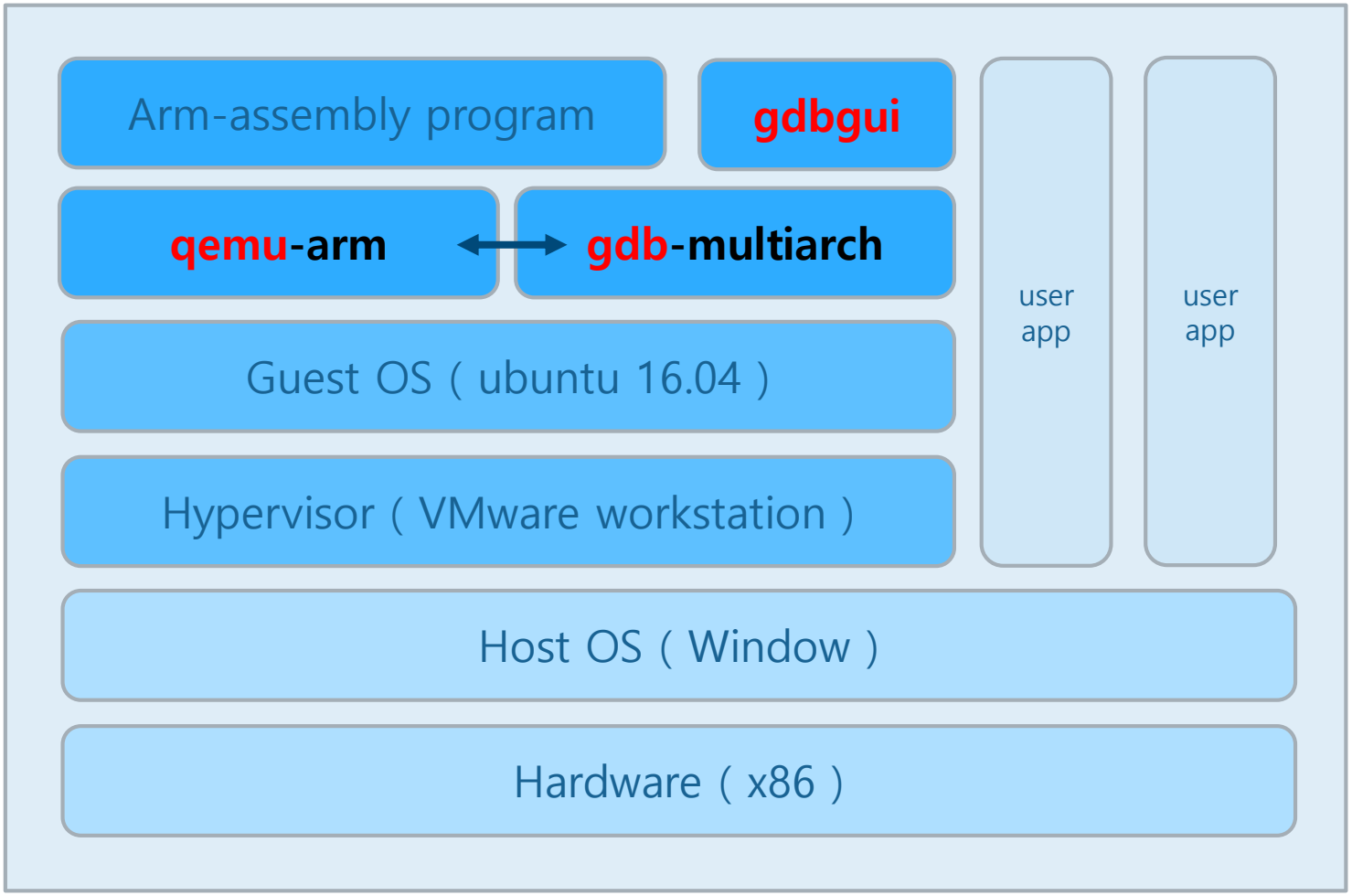
Arm assembly Lab 0

목차

실습목표

- Linux 환경에서 gdbgui 프로그램을 통해 GDB 사용법을 배운다
- ARM assembly에 대한 실습을 통해 이론으로 배웠던 ARM instruction들을 실제로 다루어 본다.
- Assembly로 코드를 작성하면서 메모리에 관점에서 생각해보는 시각을 키우고 메모리에 대한 이해도를 높인다.

ARM 어셈블리 프로그램 개발환경구축



VMware

- 가상 머신에서 “가상“은 “의사“,”유사” 혹은 " 가짜 " 라는 의미로, 가상 하드웨어, 즉 물리적 하드웨어로 구성된 실제 컴퓨터와 달리 소프트웨어적으로 모방한 가상 CPU, 가상 RAM, 가상 하드디스크 등으로 구성된 컴퓨터다.

- VMware는 가상 머신 유틸리티로 가상 머신을 생성할 수 있는 환경을 제공하는 소프트웨어로, 응용 프로그램과 동일한 계층에서 실행된다. 가상 머신 유틸리티를 이용하면 다수의 가상 머신을 생성할 수 있다.

- 예를 들면 윈도우 운영체제를 사용하고 있는 PC에 VMware를 설치하면 그 위에 다른 종류의 운영체제를 설치할 수 있다. 따라서 멀티 부팅과는 달리 특정 시점에 서로 다른 운영체제를 가진 다수의 컴퓨터 시스템을 동시에 사용할 수 있다.

GDB

- GNU 소프트웨어 시스템을 위한 기본 디버거이다. GDB는 다양한 유닉스 기반의 시스템에서 동작하는 이식성 있는 디버거로, 에이다, C, C++, 포트란 등의 여러 프로그래밍 언어를 지원한다.
- 예를 들면 Visual Studio 에서 F5키를 누르면 보이는 화면이 디버깅 모드이고. breakpoint, step into, step over, step out 등의 기능으로 프로그램의 실행을 추적할 수 있다.

```
221             array[y * col + x] = 1;
222         for (y = 0; y < row; y++)
223             for (x = dx + dw; x < dx + 2 * dw; x++)
224                 array[y * col + x] = 1;
225         for (y = dy; y < row; y++)
226             for (x = dw; x < dw + dx; x++)
227                 array[y * col + x] = 1;
228         for (y = 0; y < dy; y++)
229             for (x = dw; x < dw + dx; x++)
230                 array[y * col + x] = 0;
231
232         return array;
233     }
234
```

메모리 1

주소: 0x00B06A00

```
0x00B06A00  55 8b ec 81 ec 08 01 00 00 53
0x00B06A1B  cc f3 ab 8b 45 08 03 45 10 89
0x00B06A36  0f af 45 d4 33 c9 ba 04 00 00
0x00B06A51  c4 04 89 85 fc fe ff ff 8b 85
0x00B06A6C  45 f8 83 c0 01 89 45 f8 8b 45
0x00B06A87  ec 83 c0 01 89 45 ec 8b 45 ec
0x00B06AA2  c8 c7 04 81 01 00 00 00 eb d9
```

Qemu

- Qemu는 가상 머신 에뮬레이터(예: 실제 하드웨어는 intel CPU를 사용하고 있는데 마치 ARM CPU를 사용하는 것과 같은 환경을 제공), 가상화 솔루션이다. 가장 큰 장점은 host PC와 target board가 다른 환경에서 다양한 machine(ARM, SPARC, MIPS, PowerPC 등)을 테스트 할 수 있도록 도와준다.
- 현 수업에서 ARM architecture를 학습하기 때문에 ARM CPU 대상으로 테스트 환경을 구축할 것이다.

설치과정 요약

1. Vmware 설치(Vmware 16 workstation player)
2. Ubuntu 설치(Ubuntu 16.04)
3. Compiler 설치(gcc ARM 용 complier)
4. 가상 머신 에뮬레이터 설치(Qemu - 실제 하드웨어는 intel CPU를 사용하고 있는데 마치 ARM CPU를 사용하는 것과 같은 환경을 제공)
5. Debugger 설치(GNU debugger: GDB-multiarch, ARM 바이너리를 디버깅 할 수 있도록 해준다)
6. GUI에서의 Debugger 동작환경 설치(GDBGUI 0.13.0.0)

설치 가이드

- **통합 설치**
  - **1&2**. Vmware 및 ubuntu를 다운로드후 설치한다 (P11-P18)
  - **3~6 설치**과정을 하나로 묶은 shell 파일(ca_install.sh)을 이용하여 설치한다.
    - 우분투로 login 한 뒤 NCLab 홈페이지 강의자료 코너, lab 밑에 tool 설치용 shell file 제목에서 ca_install.sh 파일을 다운로드한다.
    - sh명령어를 수행한다.

```
$ sudo sh ~/Downloads/ca_install.sh
```

  - (**3~6 설치에 대한 다른 옵션**) 위 방법대신, shell file 내용이 다음 페이지에 나오는데, 그 명령어들을 하나씩 수동으로 수행하여 설치해도 된다.

설치 가이드 - ca_install.sh

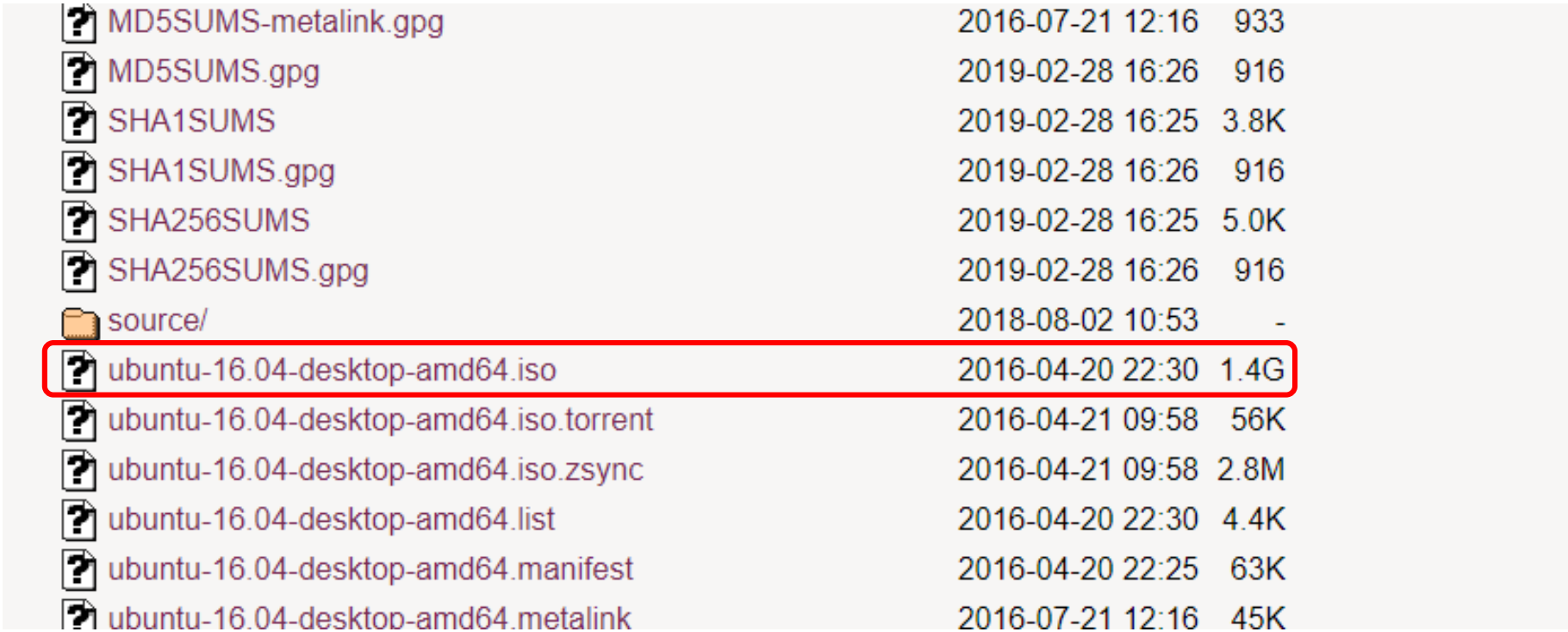
Network Computing Lab.

```
$sudo apt-get install gcc-arm-linux-gnueabi
$sudo apt-get install qemu
$sudo apt-get install gdb-multiarch

$sudo apt install python-pip
$pip install --upgrade
$pip python -m pip uninstall
$pip apt remove python-pip
$whereis pip

$wget https://bootstrap.pypa.io/get-pip.py -O /tmp/get-pip.py
$sudo python3 /tmp/get-pip.py
$pip install --user pipenv
$pip3 install --user pipenv
$echo "PATH=$HOME/.local/bin:$PATH" >> ~/.profile
$source ~/.profile
$whereis pip
$sudo pip install gdbgui==0.13.0.0
```

설치 가이드 - VMware

- 설치도중 필요한 Ububtu 이미지를 미리 다운로드함
- Ubuntu 16.04-desktop-amd64.iso용
  http://old-releases.ubuntu.com/releases/16.04.0/

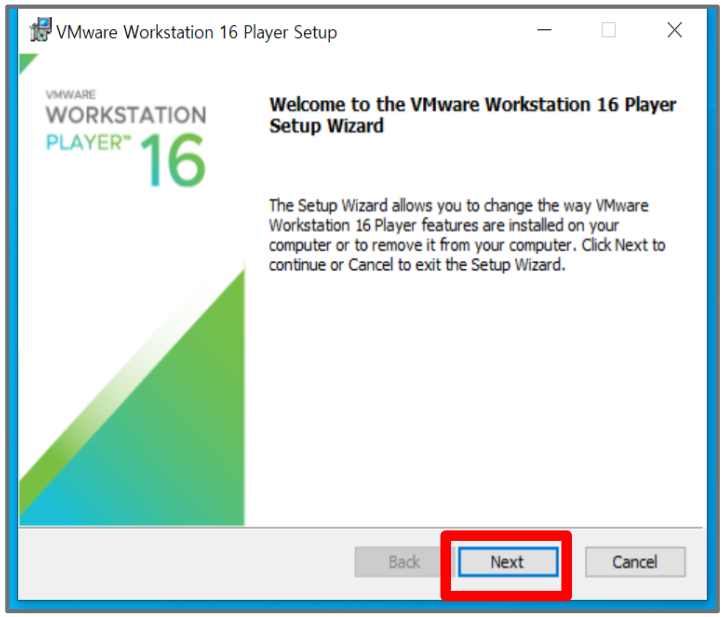
다운로드경로 P16에서 사

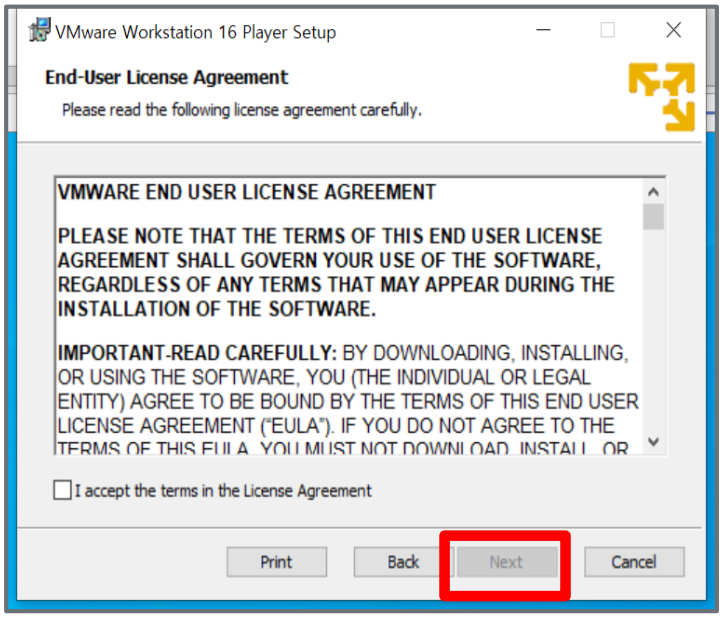


설치 - VMware

- VMware 설치
  https://www.vmware.com/kr/products/workstation-player/workstation-player-evaluation.html
  링크에서 vmware workstation player15.5를 다운로드
- (실제로는 workstation player 16이 다운로드된다.)

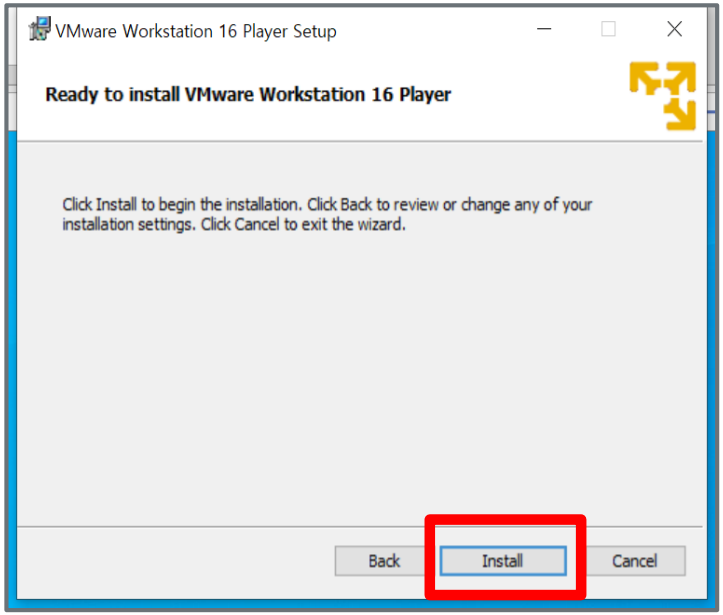
설치 가이드 - VMware

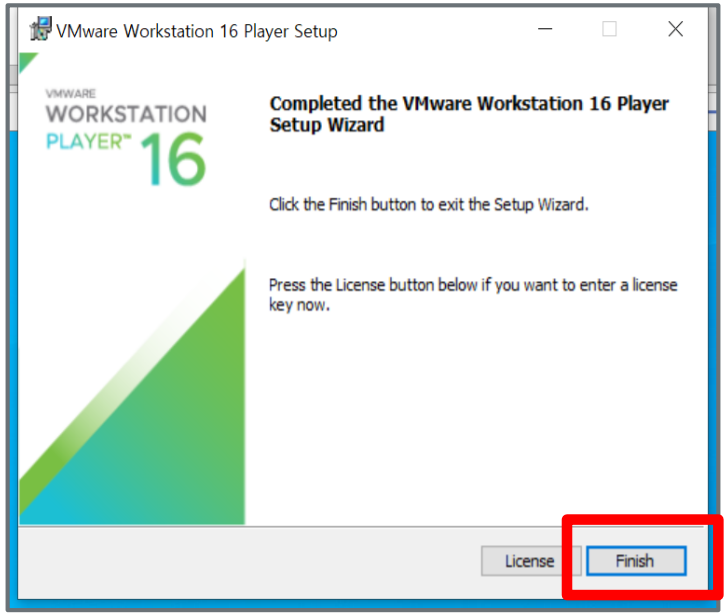




다음과 같은 설치 화면에서, 모든 설정을 건드리지 않고 next를 클릭한다.

설치 가이드 - VMware

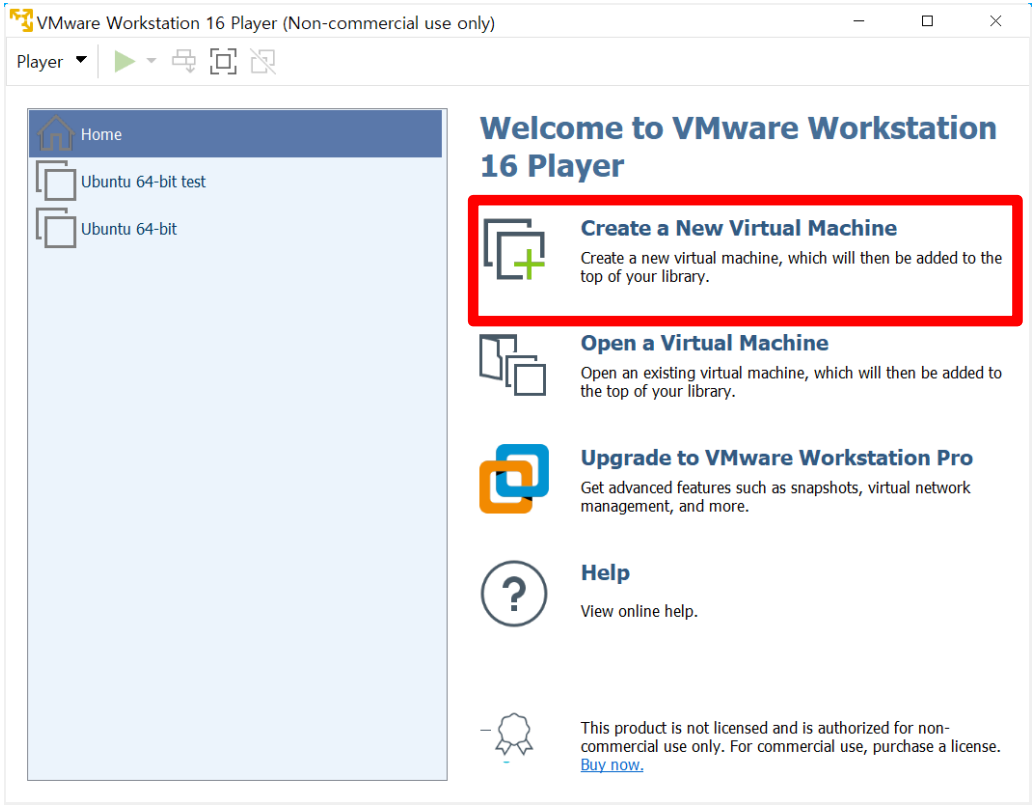




install을 클릭하고 finish를 클릭해서 설치를 마무리한다.

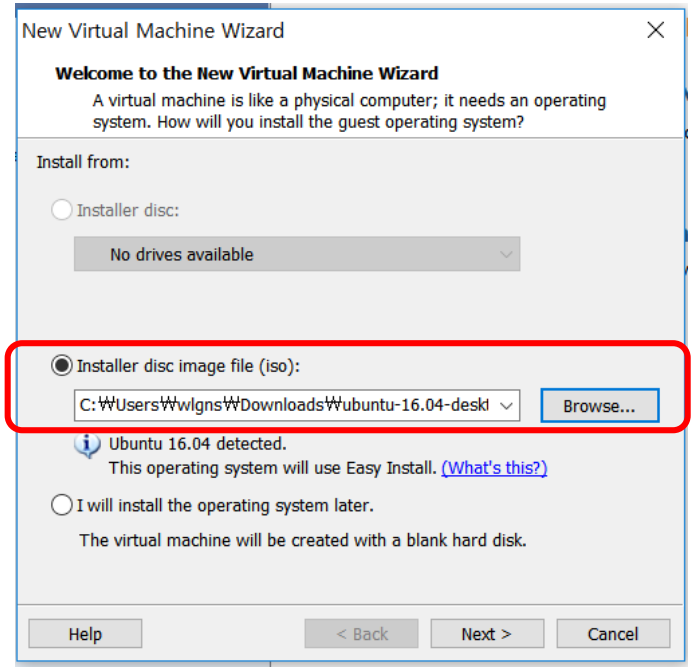
설치 가이드 - VMware

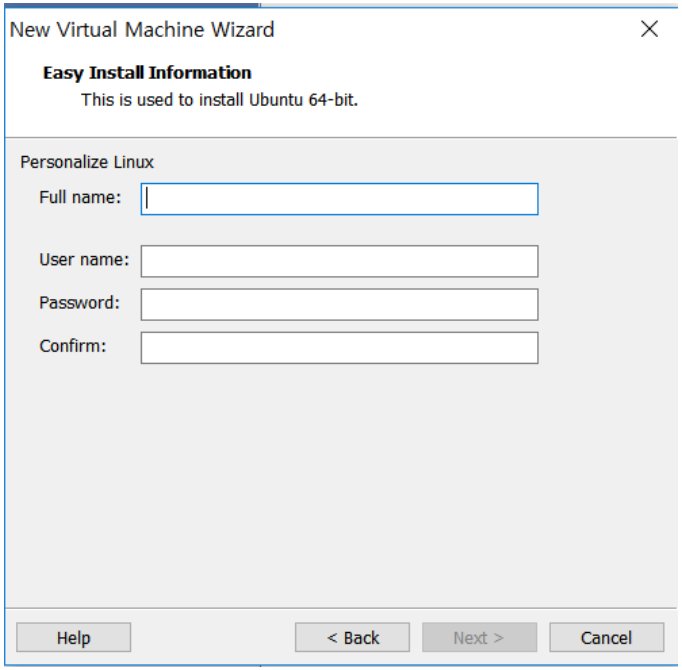
- 다음 절차에 따라 생성
  - Create 버튼 클릭



설치 가이드 - VMware

- 다음 절차에 따라 생성

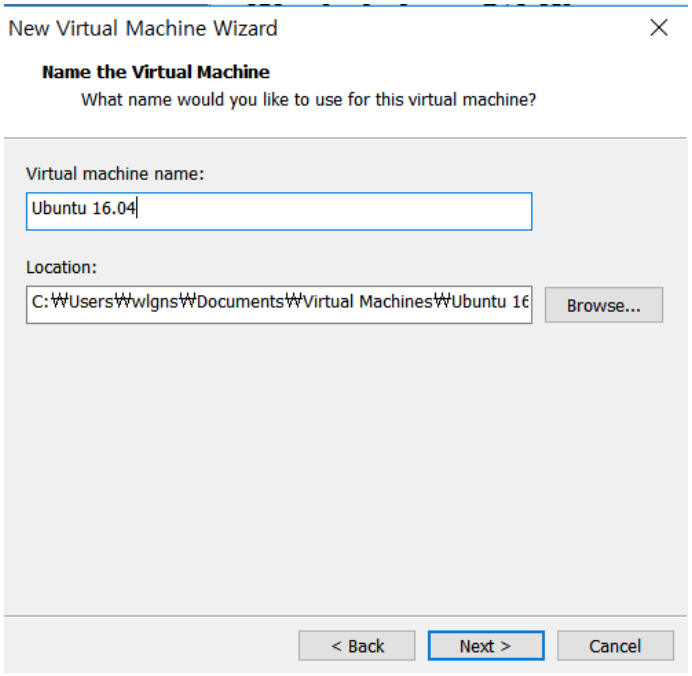


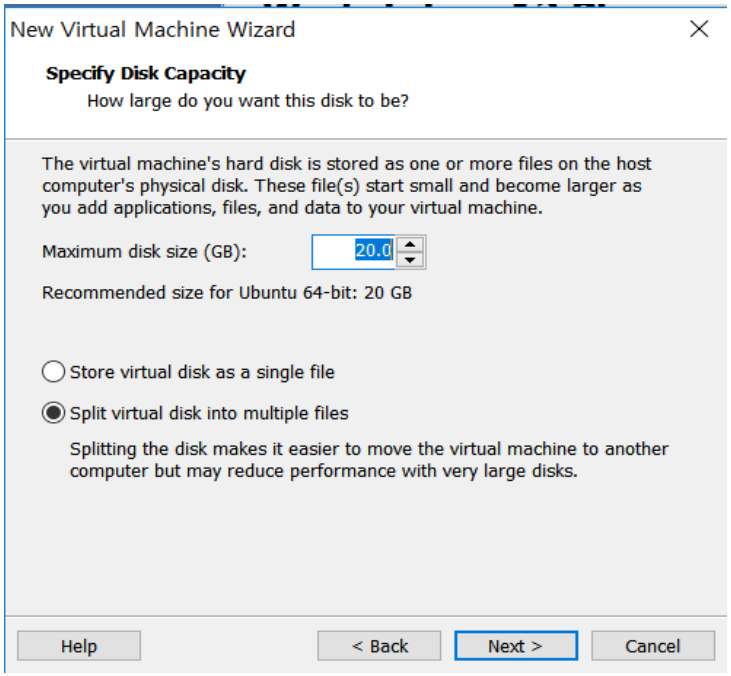


- P11에서 다운받은 Ubuntu image file의 경로를 선택 후 next
- 사용할 ID와 password 입력 후 next

설치 가이드 - VMware

- 다음 절차에 따라 생성

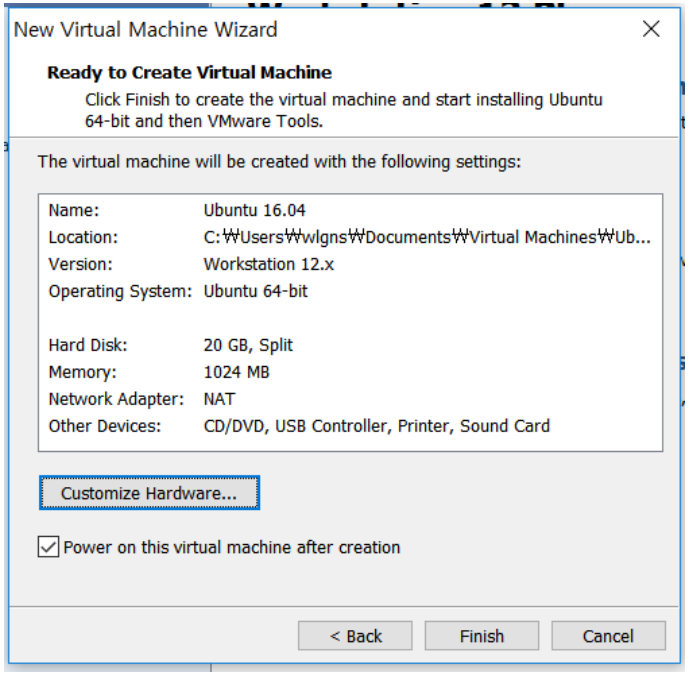


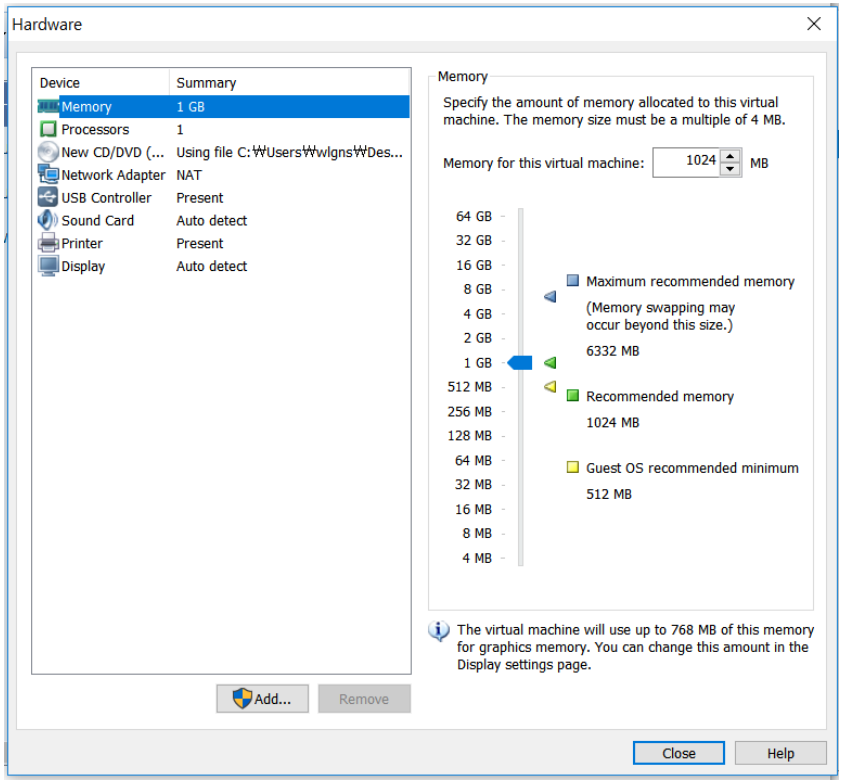


- 생성하는 Virtual machine 가상 storage size 설정 후 next

설치 가이드 - VMware

- 다음 절차에 따라 생성





- Finish를 누르면 ubuntu 설치 시작

설치 가이드 - Compiler

- gcc 베이스 오픈 소스 **ARM용 컴파일러인** arm-linux-gnueabi-gcc를 사용한다.
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install gcc-arm-linux-gnueabi
```

- 컴파일러 설치 확인

```
desktop-yoonjoon@ubuntu:~$ ls /usr/bin/arm*
/usr/bin/arm2hpdl                          /usr/bin/arm-linux-gnueabi-gcov
/usr/bin/arm-linux-gnueabi-addr2line       /usr/bin/arm-linux-gnueabi-gcov-5
/usr/bin/arm-linux-gnueabi-ar              /usr/bin/arm-linux-gnueabi-gcov-tool
/usr/bin/arm-linux-gnueabi-as              /usr/bin/arm-linux-gnueabi-gcov-tool-5
/usr/bin/arm-linux-gnueabi-c++filt         /usr/bin/arm-linux-gnueabi-gprof
/usr/bin/arm-linux-gnueabi-cpp             /usr/bin/arm-linux-gnueabi-ld
/usr/bin/arm-linux-gnueabi-cpp-5           /usr/bin/arm-linux-gnueabi-ld.bfd
/usr/bin/arm-linux-gnueabi-dwp             /usr/bin/arm-linux-gnueabi-ld.gold
/usr/bin/arm-linux-gnueabi-elfedit         /usr/bin/arm-linux-gnueabi-nm
/usr/bin/arm-linux-gnueabi-gcc             /usr/bin/arm-linux-gnueabi-objcopy
/usr/bin/arm-linux-gnueabi-gcc-5           /usr/bin/arm-linux-gnueabi-objdump
/usr/bin/arm-linux-gnueabi-gcc-ar          /usr/bin/arm-linux-gnueabi-ranlib
```

설치 가이드 - Qemu

- Host PC(Intel CPU)에서 ARM 프로그램을 테스트 할 수 있도록 해주는 가상 에뮬레이터인 Qemu를 설치한다.
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install qemu
```

- Qemu 설치 확인

```
desktop-yoonjoon@ubuntu:~$ ls /usr/bin/qemu*
/usr/bin/qemu-aarch64            /usr/bin/qemu-system-alpha
/usr/bin/qemu-alpha              /usr/bin/qemu-system-arm
/usr/bin/qemu-arm                /usr/bin/qemu-system-cris
/usr/bin/qemu-armeb              /usr/bin/qemu-system-i386
/usr/bin/qemu-cris               /usr/bin/qemu-system-lm32
/usr/bin/qemu-i386               /usr/bin/qemu-system-m68k
/usr/bin/qemu-img                /usr/bin/qemu-system-microblaze
/usr/bin/qemu-io                 /usr/bin/qemu-system-microblazeel
/usr/bin/qemu-m68k               /usr/bin/qemu-system-mips
```

설치 가이드 - Debugger

- gdb-multiarch는 여러 가지 architecture를 지원하는 GNU debugger이다(하나의 GDB이다). ARM 바이너리를 디버깅 할 수 있도록 해준다.
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install gdb-multiarch
```

- gdb-multiarch 설치 확인

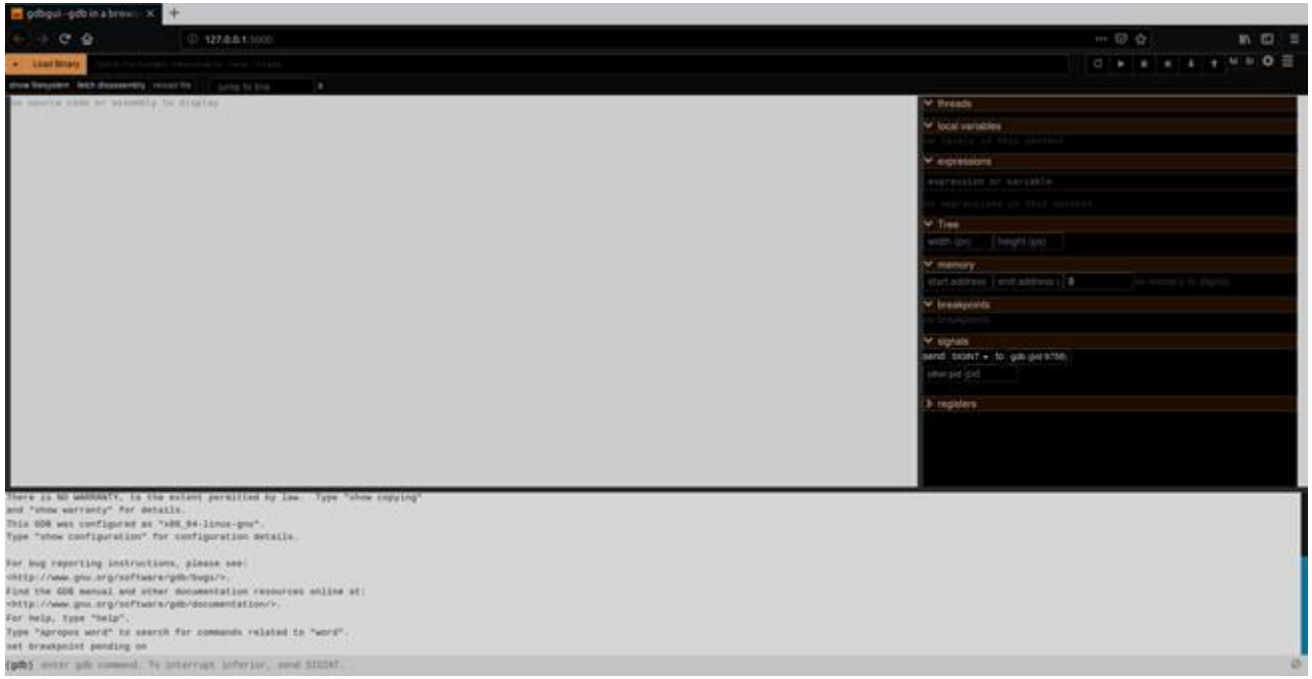
```
desktop-yoonjoon@ubuntu:~$ ls /usr/bin/gdb-multiarch
/usr/bin/gdb-multiarch
```

설치 가이드 - gdbgui

- **gdbgui 설치 확인**

```
$ gdbgui -g gdb-multiarch
```

```
yj@ubuntu:~/lab0$ gdbgui -g gdb-multiarch
Opening gdbgui with default browser at http://127.0.0.1:5000
exit gdbgui by pressing CTRL+C
```



사용법 – Compiler

- 컴파일 형식

```
$ arm-linux-gnueabi-gcc [options] files …
```

  - arm-linux-gnueabi-gcc --help 를 통해 자세한 내용 참고

- 실행 방법(예)

```
$ arm-linux-gnueabi-gcc -g -o lab0 assem.S main.c -static
```

  - main.c와 assem.S file을 compile, assemble 한 뒤 lab0 이름의 실행 가파일이 생성한다.
  - -g 는 디버깅의 편의성을 위한 옵션
  - -static 은 정적 링킹 옵션

사용법 – Qemu

- **Qemu 실행 command format**

```
qemu-arm [options] program [ argments … ]
```

  - qemu-arm --help 를 통해 자세한 내용 참고

- **실행 방법(예)**

```
$ qemu-arm -g 8080 ./lab0
```

  - -g 옵션으로 gdb 연결을 위한 포트 번호를 설정할 수 있다.
  - 위 명령어를 수행하면 연결대기 상태로 들어간다. 다른 터미널창에서 다음 페이지의 gdbgui 명령어를 수행한다.

사용법 – gdbgui

- **gdbgui 실행 command format**

```
$ gdbgui [options] [args]
```

  - gdbgui --help 를 통해 자세한 내용 참고

- **실행 방법(예)**

```
$ gdbgui -g gdb-multiarch
```

gdbgui 0.14.0.0 으로 upgrade 하라는 추천 메시지가 나오는데 upgrade 하지 마세요! 그버전은 아직 불안정합니다.

다음의 예제1 ~ 3 통해, gdbgui 화면에서 레지스터값 확인 및 수정, 메모리 값 확인 및 수정 방법을 설명합니다.

예제1. 실습 목표

- **GDB를 사용하여 다음 페이지 copyarray 함수 분석**
  - **copyarray는 배열 값들을 복사하는 어셈블리어 함수이다.**
  - copyarray 함수를 사용하는 프로그램을 앞서 설치한 크로스 컴파일러를 활용해 컴파일한다.
  - Qemu와 GDB 환경 위에서 프로그램을 수행하고, breakpoint 활용, 레지스터 출력, 메모리 출력 등의 GDB 명령어를 사용해본다.
  - **copyarray 함수를 수행할 때 레지스터와 메모리 변화를 분석한다.**

(예)

```
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :2 12 -1 10 7
Array Copy :2 12 -1 10 7
```

초기조건

copyarray함수 수행후 결과

예제1. 소스 코드

어셈블리코드로 들어올 때 r0는 arrOri 배열시작 주소, r1은 arrCopy 배열시작 주소, r2는 배열내 요소갯수 5 전달

main.c

```c
#include <stdio.h>

extern void copy_array(int arr1[], int arr2[], int n);

void print_array(int arr[], int n)
{
  for(int i = 0; i < n ; i++)
    printf("%d ",arr[i]);
  printf("\n");
}

int main()
{
  int arrOri[5] = {2, 12, -1, 10, 7};
  int arrCpy[5] = {0, };
  int num = 5;

  printf("Array Original :");
  print_array(arrOri, num);

  printf("Array Copy :");
  print_array(arrCpy, num);

  copy_array(arrOri, arrCpy, num);

  printf("Array Original :");
  print_array(arrOri, num);

  printf("Array Copy :");
  print_array(arrCpy, num);

  return 0;
}
```

assem.S

```
.text
.global copy_array
.type copy_array,STT_FUNC

copy_array:
  sub sp, sp, #8
  str r3, [sp,#0]
  str r4, [sp,#4]
  mov r4, #0

cwhile:
  cmp r2, r4
  ble cexit
  ldr r3, [r0,r4,lsl#2]
  str r3, [r1,r4,lsl#2]
  add r4, r4, #1
  b cwhile

cexit:
  ldr r4, [sp,#4]
  ldr r3, [sp,#0]
  add sp, sp, #8
  mov pc, lr
.end
```

r4는 배열의 index를 나타냄

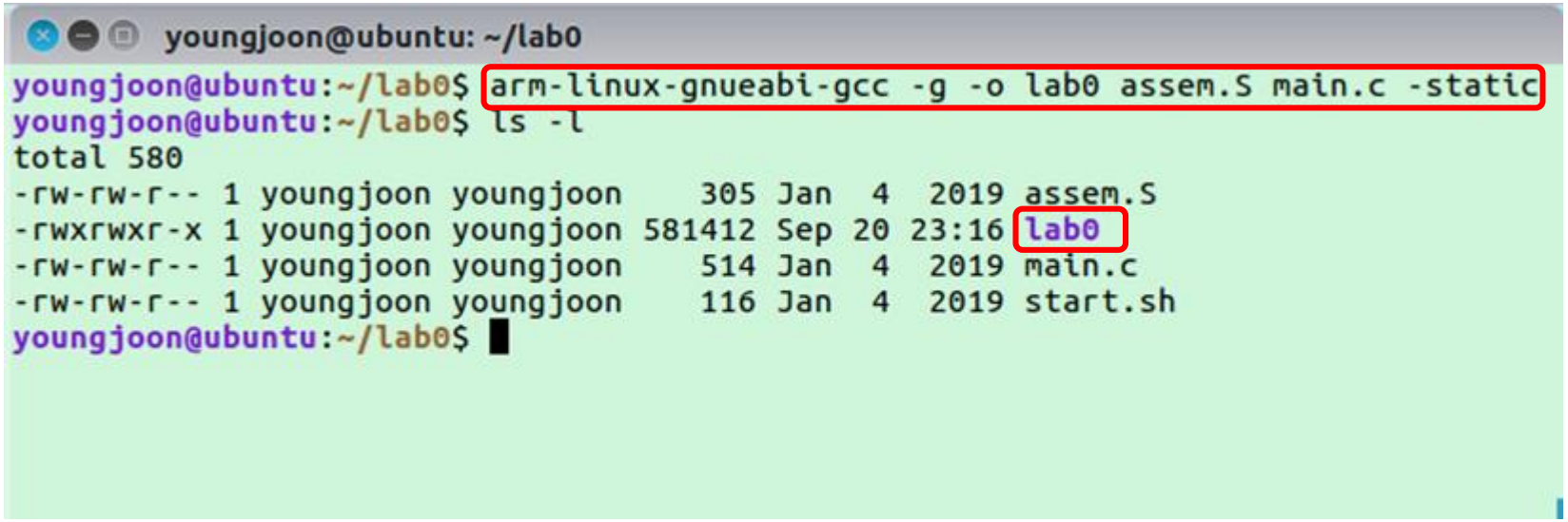
Index값 * 4를 계산

예제1. 실습 과정

- **소스코드 다운로드 및 압축 해제**
  - Ubuntu에서 lab0 디렉토리 생성
  - NCLab 홈페이지 강의자료 코너에서 lab0 제목에서 lab0.zip (소스코드 압축 파일)를 lab0 디렉토리에 다운로드
  - unzip 명령어를 이용하여 lab0.zip 압축해제

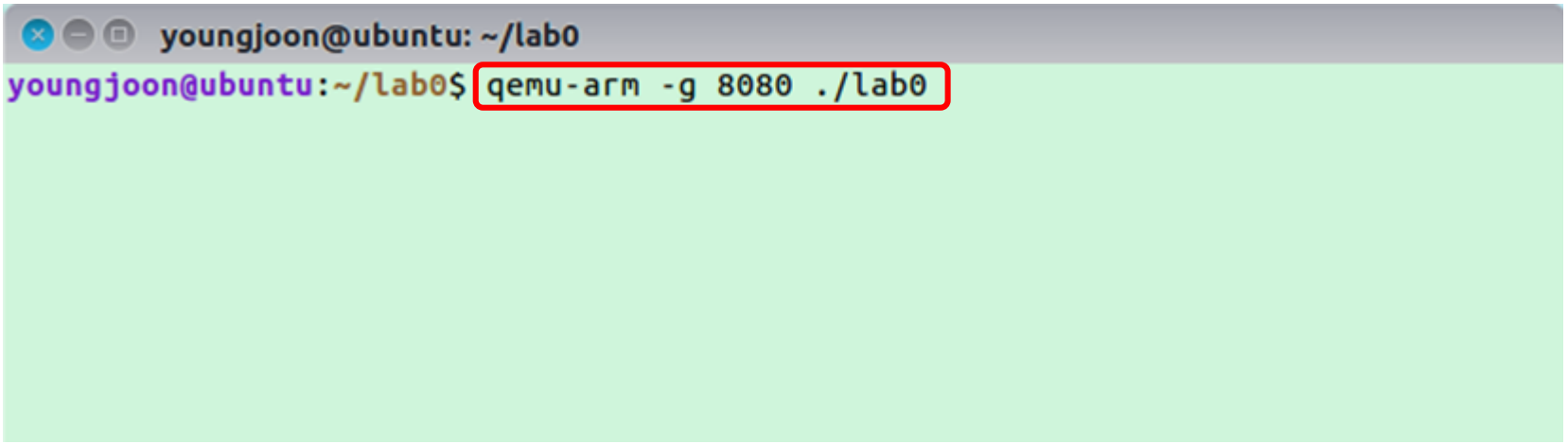
예제1. 실습 과정

- **소스코드를 압축 해제 했다면, 컴파일을 해보자.**
  - 아래와 같이 lab0 이름의 실행 파일이 생성된다.



예제1. 실습 과정

- **실행파일을 qemu로 동작시킨다.**
  - 아래와 같이 커맨드를 입력하면 GDB 연결 대기 상태가 된다.



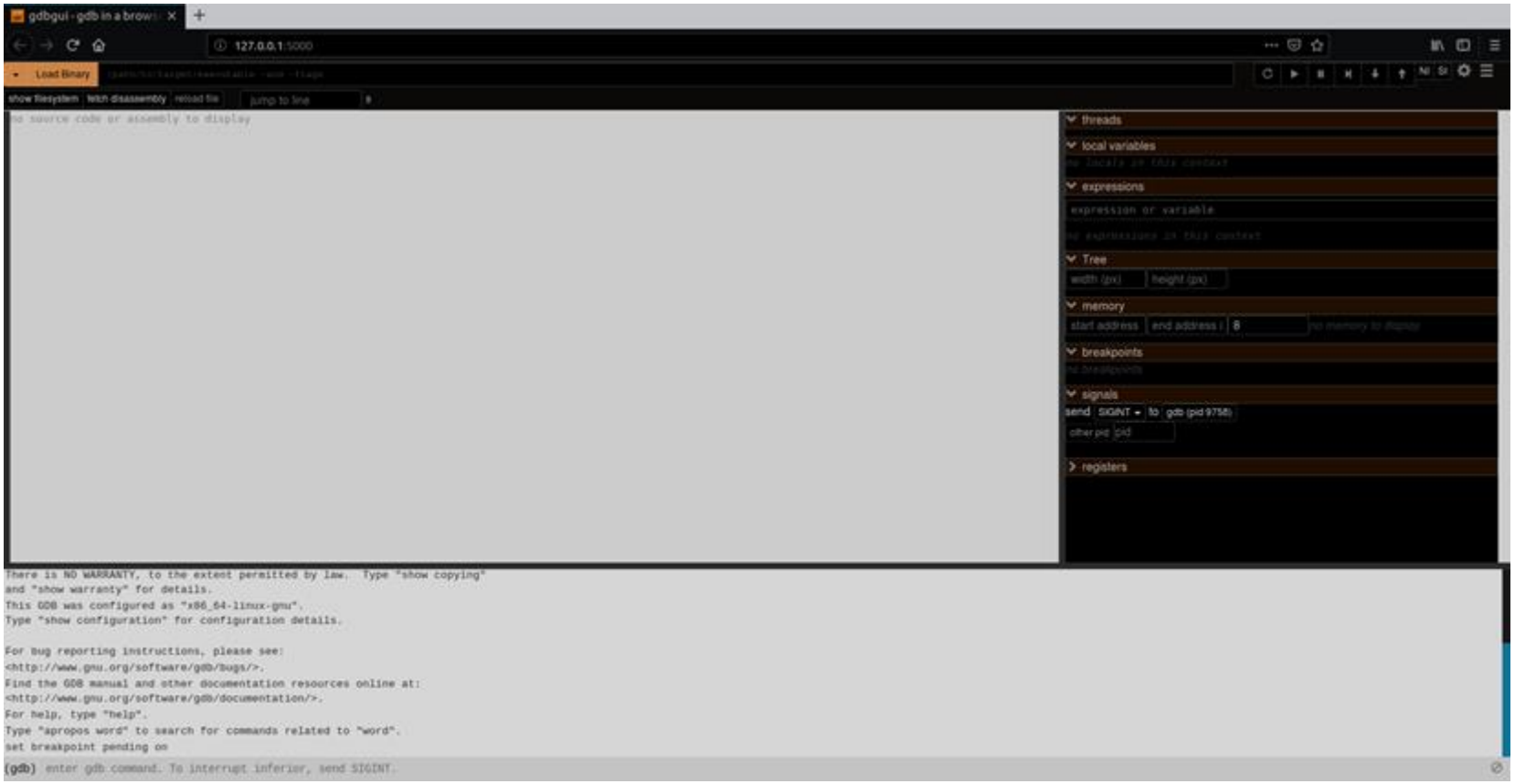
예제1. 실습 과정

- **Qemu는 대기 상태로 두고, 새로운 터미널을 연다.**
- **새로운 터미널에서 gdbgui를 실행시킨다.**
  - 터미널 단축키는 ctrl+alt+T 혹은 기존 터미널에서 ctrl+shift+T
  - $ gdbgui -g gdb-multiarch

```
youngjoon@ubuntu:~$ gdbgui -g gdb-multiarch
Opening gdbgui with default browser at http://127.0.0.1:5000
exit gdbgui by pressing CTRL+C
```

예제1. 실습 과정

- gdbgui 실행 화면



예제1. 실습 과정

- **gdbgui 하단의 터미널에서 동작할 file을 알려준다.**
  - (gdb) file [실행파일 이름] 입력

```
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
set breakpoint pending on
complete file ~/la
file ~/lab0
file ~/lab0/lab0
file ~/lab0/lab0
Reading symbols from ~/lab0/lab0...
done.
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

예제1. 실습 과정

- Qemu에서 대기하고 있는 GDB 연결 포트 번호를 입력한다.
  - (gdb) target remote:[포트 번호]

```
set breakpoint pending on
complete file ~/la
file ~/lab0
file ~/lab0/lab0
file ~/lab0/lab0
Reading symbols from ~/lab0/lab0...
done.
target remote:8080
target remote:8080
Remote debugging using :8080
0x0001044c in _start ()  in _start() 가 나타나면 연결 완료
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

예제1. 실습 과정

- **main 함수에 breakpoint 설정**
  - (gdb) break main 또는 b main

```
break main
break main
Breakpoint 1 at 0x10678: file main.c, line 12.
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

예제1. 실습 과정

- **main 함수까지 실행**
  - (gdb) continue 또는 c 입력

```
break main
break main
Breakpoint 1 at 0x10678: file main.c, line 12.
continue
continue
Continuing.

Breakpoint 1, main () at main.c:12
12      int arrOri[5] = {2,12,-1,10,7};
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

```
11  int main(){
12          int arrOri[5] = {2,12,-1,10,7};
13          int arrCpy[5] = {0, };
14          int num = 5;
```

예제1. 실습 과정

- **copyarray 함수에 breakpoint 설정하고 계속 실행**
  - (gdb) b copyarray
  - (gdb) c

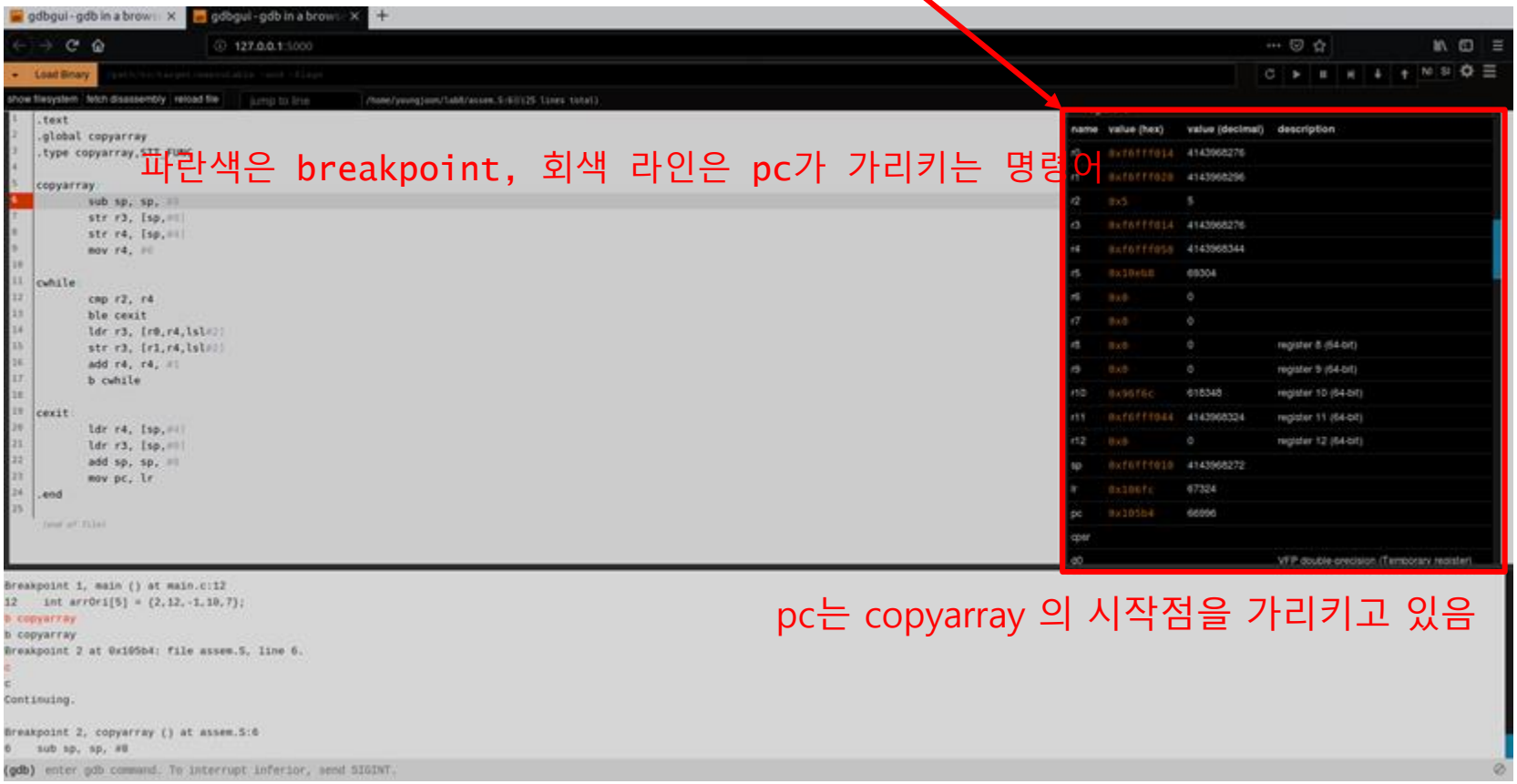
```
Breakpoint 1, main () at main.c:12
12    int arrOri[5] = {2,12,-1,10,7};
b copyarray
b copyarray
Breakpoint 2 at 0x105b4: file assem.S, line 6.
c
c
Continuing.

Breakpoint 2, copyarray () at assem.S:6
6     sub sp, sp, #8
(gdb) enter gdb command. To interrupt inferior,
```

```
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
```

```
1   .text
2   .global copyarray
3   .type copyarray,STT_FUNC
4
5   copyarray:
6           sub sp, sp, #8
7           str r3, [sp,#0]
8           str r4, [sp,#4]
9           mov r4, #0
10
11  cwhile:
12          cmp r2, r4
13          ble cexit
14          ldr r3, [r0,r4,lsl#2]
15          str r3, [r1,r4,lsl#2]
16          add r4, r4, #1
17          b cwhile
```

예제1. 실습 과정

- GDB를 활용하여 copyarray 함수 분석
  - 우측 창 맨 밑에 registers 섹션 클릭



예제1. 실습 과정

- **GDB를 활용하여 copyarray 함수 분석**
  - 아래 registers 섹션에서 $r0 레지스터에는 copyarray 함수의 첫 번째 인자인 arrOri 배열의 시작 주소값이 저장됨

registers

name	value (hex)	value (decimal)	description
r0	0xf6fff014	4143968276	
r1	0xf6fff028	4143968296	

  - 주소(파란색 0xf6~) 클릭하고 메모리 섹션에서 arrOri 배열 값 확인

memory

0xf6fff014 | 0xf6fff033 | 8

address	hex	char
more	{2, 12, -1, 10, 7}	
0xf6fff014	02 00 00 00 0c 00 00 00	....
0xf6fff01c	ff ff ff ff 0a 00 00 00	....
0xf6fff024	07 00 00 00 00 00 00 00	....

Little endian으로 저장된 32비트 data임. 0xf6fff014번지에 02 저장 (byte). 0xf6fff015번지에 00 저장 (byte)....

예제1. 실습 과정

- **GDB를 활용하여 copyarray 함수 분석**
  - 6번 코드 수행하려고 함. 수행전 registers 섹션에서 sp 값 확인
  - (gdb) s 또는 si 를 입력

```
1   .text
2   .global copyarray
3   .type copyarray,STT_FUNC
4
5   copyarray:
6           sub sp, sp, #8
7           str r3, [sp,#0]
8           str r4, [sp,#4]
9           mov r4, #0
10
11  cwhile:
12          cmp r2, r4
13          ble cexit
14          ldr r3, [r0,r4,lsl#2]
15          str r3, [r1,r4,lsl#2]
16          add r4, r4, #1
17          b cwhile
18
19  cexit:
```

disass 명령어를 통해 명령어 수행 확인.

예제1. 실습 과정

- GDB를 활용하여 copyarray 함수 분석
  - 6번 코드 수행후 바뀐 레지스터 값을 확인

r9	0x0	0	register 9 (64-bit)
r10	0x96f6c	618348	register 10 (64-bit)
r11	0xf6fff044	4143968324	register 11 (64-bit)
r12	0x0	0	register 12 (64-bit)
sp	0xf6fff008	4143968264	
lr	0x106fc	67324	
pc	0x105b8	67000	

다음 명령어가 수행 되었으므로 pc는 +4,
sub sp,sp,#8 명령어에 의해 sp 의 값은 8만큼 감소했음을 확인할 수 있음

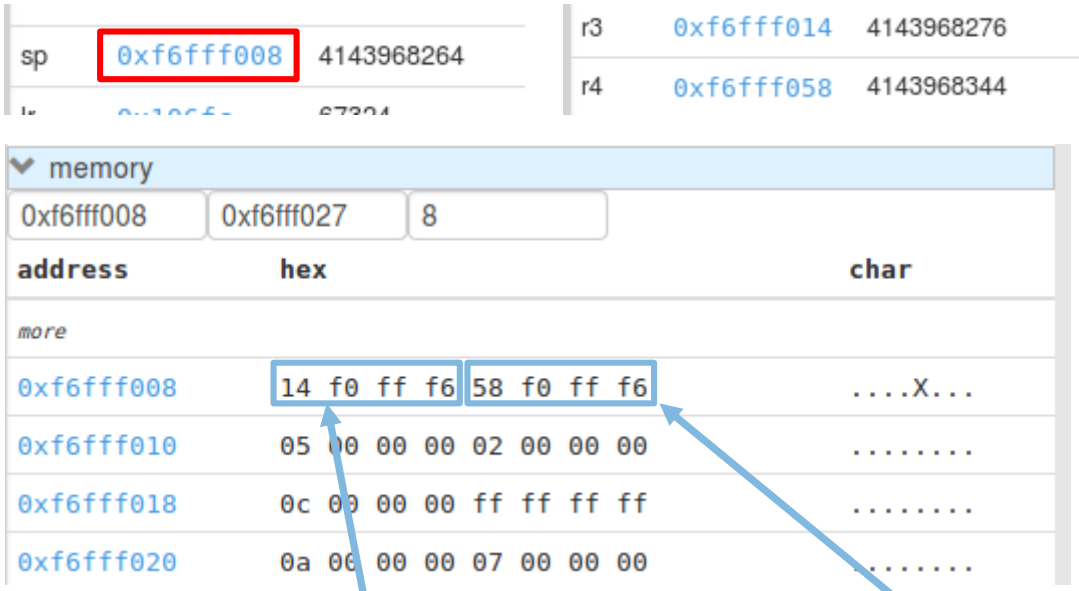
예제1. 실습 과정

- **GDB를 활용하여 copyarray 함수 분석**
  - si 명령어를 두 번 사용하여 2 step 진행

```
1   .text
2   .global copyarray
3   .type copyarray,STT_FUNC
4
5   copyarray:
6           sub sp, sp, #8
7           str r3, [sp,#0]
8           str r4, [sp,#4]   ← 여기까지 수행된 상태
9           mov r4, #0
10
11  cwhile:
12          cmp r2, r4
13          ble cexit
14          ldr r3, [r0,r4,lsl#2]
15          str r3, [r1,r4,lsl#2]
16          add r4, r4, #1
17          b cwhile
18
19  cexit:
20          ldr r4, [sp,#4]
21          ldr r3, [sp,#0]
22          add sp, sp, #8
23          mov pc, lr
24  .end
```

예제1. 실습 과정

- **GDB를 활용하여 copyarray 함수 분석**
  - $sp 레지스터 값(파란색) 클릭하여 스택에 하위 2 word 확인



str r3, [sp]에 의해 store된 r3값

str r4, [sp,#4]에 의해 store된 r4값

예제1. 실습 과정

- GDB를 활용하여 copyarray 함수 분석
  - 원하는 분석이 끝났다면 c 명령어를 사용하여 프로그램 재개

```
Continuing.
[Inferior 1 (Remote target) exited normally]
Unable to read memory.
Unable to read memory.
Unable to read memory.
```

이후에 Breakpoint가 없기때문에 프로그램 끝까지 실행 후 정상 종료함
빨간색 메시지는 memory section에 입력된 값이 유지되고 있기 때문에 출력됨

  - 프로그램의 수행 결과가 출력됨

```
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :2 12 -1 10 7
Array Copy :2 12 -1 10 7
youngjoon@ubuntu:~/lab0$
```

예제2. 실습 목표

- **레지스터 값 변경 후 수행 결과 확인**
  - copyarray 함수를 호출할 때 $r2 레지스터에는 세 번째 인자인 num 값이 저장된다. (num == 5)
  - copyarray 함수 시작 부분에서 $r2 레지스터 값을 1 감소시키고 계속해서 수행한다.
  - 출력 결과가 어떻게 변하는지 확인한다.

예제2. 실습 과정

- **프로그램 다시 수행하기**
  - 실행파일을 qemu로 재동작시킨다

```
youngjoon@ubuntu: ~/lab0
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
```

예제2. 실습 과정

- **GDB를 Qemu와 다시 연결하기**
  - GDB 터미널에서 Qemu에서 대기하고 있는 GDB 연결 포트 번호를 입력한다.

```
target remote:8080
target remote:8080
Remote debugging using :8080
0x0001044c in _start ()
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

예제2. 실습 과정

- **breakpoint 설정하기**
  - copyarray 함수에 breakpoint를 설정하고 계속 실행한다

```
Breakpoint 1, main () at main.c:12
12    int arrOri[5] = {2,12,-1,10,7};
b copyarray
b copyarray
Breakpoint 2 at 0x105b4: file assem.S, line 6.
c
c
Continuing.

Breakpoint 2, copyarray () at assem.S:6
6     sub sp, sp, #8
(gdb) enter gdb command. To interrupt inferior,
```

```
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
```

```
1   .text
2   .global copyarray
3   .type copyarray,STT_FUNC
4
5   copyarray:
6           sub sp, sp, #8
7           str r3, [sp,#0]
8           str r4, [sp,#4]
9           mov r4, #0
10
11  cwhile:
12          cmp r2, r4
13          ble cexit
14          ldr r3, [r0,r4,lsl#2]
15          str r3, [r1,r4,lsl#2]
16          add r4, r4, #1
17          b cwhile
```

예제2. 실습 과정

- 함수 인자로 넘어온 레지스터 값 확인하기
  - 현재 레지스터 값 확인한다.

r0: arrOri 배열의 시작 주소
r1: arrCpy 배열의 시작 주소
r2: num (5)

registers

name	value (hex)	value (decimal)	des
r0	0xf6fff014	4143968276	
r1	0xf6fff028	4143968296	
r2	0x5	5	
r3	0xf6fff014	4143968276	
r4	0xf6fff058	4143968344	
r5	0x10eb8	69304	
r6	0x0	0	

r0, r1, r2 레지스터에는 copyarray 함수를 호출할 때 입력한 인자들이 저장되어있음.

예제2. 실습 과정

- 레지스터 값 변경 후 프로그램 출력 결과 확인하기
  - num 값이 저장되어 있는 $r2 레지스터의 값을 1감소 시키고 결과를 확인한다. (gdb) set $[register] = [value]
  - (gdb) set $r2 = 4

```
set $r2 = 4
set $r2 = 4
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

registers

name	value (hex)	value (decimal)	description
r0	0xf6fff014	4143968276	
r1	0xf6fff028	4143968296	
r2	0x4	4	
r3	0xf6fff014	4143968276	

$r2 레지스터 값이 5 => 4 로 변경됨

예제2. 실습 과정

- **레지스터 값 변경 후 프로그램 출력 결과 확인하기**
  - 프로그램을 끝까지 수행하고 출력값을 확인한다.

```
set $r2 = 4
set $r2 = 4
c
c
Continuing.
[Inferior 1 (Remote target) exited normally]
Unable to read memory.
Unable to read memory.
```

```
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :2 12 -1 10 7
Array Copy :2 12 -1 10 0
youngjoon@ubuntu:~/lab0$
```

기존 num에 해당하는 5개 값을 복사한 것이 아닌, 1이 감소된 4개 값를 복사했다.

예제3. 실습 목표

- **메모리 값 변경 후 수행 결과 확인**
  - copyarray 함수를 호출할 때 $r0 레지스터에는 첫 번째 인자인 arrOri 배열의 시작 주소값이 저장된다.
  - **프로그램 수행 도중에 $r0 레지스터가 가리키는 메모리 영역에 접근해 배열의 첫 번째 값을 변경하고 계속해서 수행한다.**
  - 출력 결과가 어떻게 변하는지 확인한다.

예제3. 실습 과정

- **프로그램 다시 수행하기**
  - 실행파일을 qemu로 재동작시킨다

```
youngjoon@ubuntu: ~/lab0
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
```

예제3. 실습 과정

- **GDB를 Qemu와 다시 연결하기**
  - GDB 터미널에서 Qemu에서 대기하고 있는 GDB 연결 포트 번호를 입력한다.

```
target remote:8080
target remote:8080
Remote debugging using :8080
0x0001044c in _start ()
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

예제3. 실습 과정

- **breakpoint 설정하기**
  - copyarray 함수에 breakpoint를 설정하고 실행한다

```
Breakpoint 1, main () at main.c:12
12    int arrOri[5] = {2,12,-1,10,7};
b copyarray
b copyarray
Breakpoint 2 at 0x105b4: file assem.S, line 6.
c
c
Continuing.

Breakpoint 2, copyarray () at assem.S:6
6     sub sp, sp, #8
(gdb) enter gdb command. To interrupt inferior,
```

```
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
```

```
1   .text
2   .global copyarray
3   .type copyarray,STT_FUNC
4
5   copyarray:
6           sub sp, sp, #8
7           str r3, [sp,#0]
8           str r4, [sp,#4]
9           mov r4, #0
10
11  cwhile:
12          cmp r2, r4
13          ble cexit
14          ldr r3, [r0,r4,lsl#2]
15          str r3, [r1,r4,lsl#2]
16          add r4, r4, #1
17          b cwhile
```

예제3. 실습 과정

메모리 값 변경 후 프로그램 출력 결과 확인하기

- $r0 레지스터에 저장된 주소는 arrOri 배열의 시작주소를 나타낸다.
- $r0 레지스터 값을 클릭해 arrOri 배열의 값들을 확인한다.

registers

name	value (hex)	value (decimal)	description
r0	0xf6fff014	4143968276	
r1	0xf6fff028	4143968296	
r2	0x5	5	

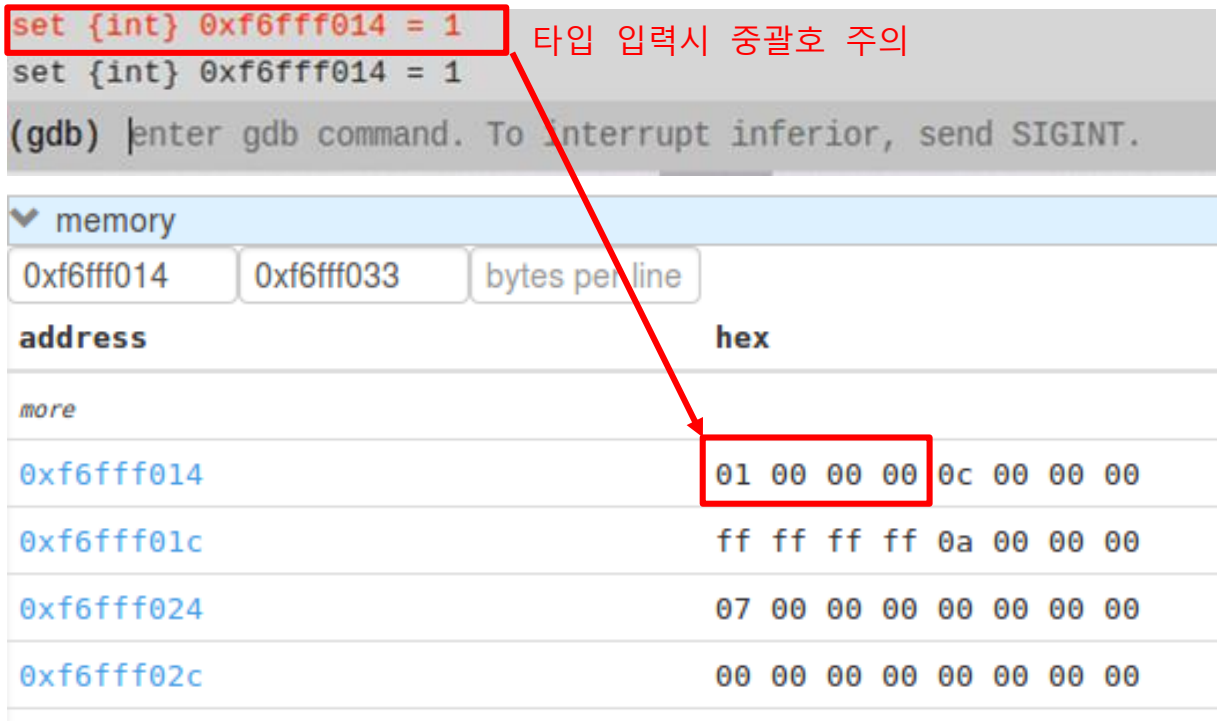
memory

0xf6fff014 | 0xf6fff033 | bytes per line

address	hex
more	{2, 12, -1, 10, 7}
0xf6fff014	02 00 00 00 0c 00 00 00
0xf6fff01c	ff ff ff ff 0a 00 00 00
0xf6fff024	07 00 00 00 00 00 00 00
0xf6fff02c	00 00 00 00 00 00 00 00

예제3. 실습 과정

- **메모리 값 변경 후 프로그램 출력 결과 확인하기**
  - arrOri 배열의 메모리 영역에 접근해 첫 번째 값을 변경한다.
  - (gdb) set {int} 0xf6fff014 = 1



예제3. 실습 과정

- **메모리 값 변경 후 프로그램 출력 결과 확인하기**
  - 프로그램을 끝까지 수행하고 출력값을 확인한다.

```
set {int} 0xf6fff014 = 1
set {int} 0xf6fff014 = 1
continue
continue
Continuing.
[Inferior 1 (Remote target) exited normally]
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

```
youngjoon@ubuntu:~/lab0$ qemu-arm -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :1 12 -1 10 7
Array Copy :1 12 -1 10 7
youngjoon@ubuntu:~/lab0$
```

arrOri 배열의 첫 번째 요소 값이 2에서 1로 변경되어 복사된 것을 볼 수 있다.

부록 - GDB 명령어

- gdbgui 환경이 아닌 터미널에서 gdb를 사용할 경우 다양한 명령어들의 숙지가 필요하다.
- 어셈블리 프로그램 뿐만 아니라 다른 고급 프로그램 디버깅시 매우 유용하다.

```
desktop-yoonjoon@ubuntu:~/ca_lab/ex1$ gdb-multiarch
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

```
(gdb) disass main
Dump of assembler code for function main:
   0x00010660 <+0>:     push    {r11, lr}
   0x00010664 <+4>:     add     r11, sp, #4
   0x00010668 <+8>:     sub     sp, sp, #48     ; 0x30
   0x0001066c <+12>:    ldr     r3, [pc, #224]  ; 0x10754 <main+244>
   0x00010670 <+16>:    ldr     r3, [r3]
   0x00010674 <+20>:    str     r3, [r11, #-8]
   0x00010678 <+24>:    ldr     r3, [pc, #216]  ; 0x10758 <main+248>
   0x0001067c <+28>:    sub     r12, r11, #48   ; 0x30
   0x00010680 <+32>:    mov     lr, r3
   0x00010684 <+36>:    ldm     lr!, {r0, r1, r2, r3}
   0x00010688 <+40>:    stmia   r12!, {r0, r1, r2, r3}
   0x0001068c <+44>:    ldr     r3, [lr]
```

```
(gdb) info reg
r0             0xf6ffefb4       -150999116
r1             0xf6ffefc8       -150999096
r2             0x5      5
r3             0xf6ffefb4       -150999116
r4             0xf6ffeff8       -150999048
r5             0x10eb8  69304
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x96f6c  618348
r11            0xf6ffefe4       -150999068
r12            0x0      0
sp             0xf6ffefb0       0xf6ffefb0
```

```
(gdb) x/14i 0x00105b4
=> 0x105b4 <copy_array>:        sub     sp, sp, #8
   0x105b8 <copy_array+4>:      str     r3, [sp]
   0x105bc <copy_array+8>:      str     r4, [sp, #4]
   0x105c0 <copy_array+12>:     mov     r4, #0
   0x105c4 <cwhile>:    cmp     r2, r4
   0x105c8 <cwhile+4>:  ble     0x105dc <cexit>
   0x105cc <cwhile+8>:  ldr     r3, [r0, r4, lsl #2]
   0x105d0 <cwhile+12>: str     r3, [r1, r4, lsl #2]
   0x105d4 <cwhile+16>: add     r4, r4, #1
   0x105d8 <cwhile+20>: b       0x105c4 <cwhile>
   0x105dc <cexit>:     ldr     r4, [sp, #4]
   0x105e0 <cexit+4>:   ldr     r3, [sp]
   0x105e4 <cexit+8>:   add     sp, sp, #8
```

부록 - GDB 명령어

기본적인 GDB 명령어

명령어	기능
c (Continue) 	프로그램 수행
k (Kill)	프로그램 수행 종료
s (Step) 	현재 행 수행 후 정지, 함수 호출시 함수 안으로 들어감 (s # : #번 연속 수행)
n (Next) 	현재 행 수행 후 정지, 함수 호출시 함수 수행 다음 행으로 이동 (n # : #번 연속 수행)
si (Step Instruction)	어셈블리 명령어 단위의 수행 (진행은 Step과 같음)
ni (Next Instruction)	어셈블리 명령어 단위의 수행 (진행은 Next와 같음)
disass 함수이름	특정 함수를 disassemble
list	소스코드 출력

부록 - GDB 명령어

Network Computing Lab.

기본적인 GDB 명령어

명령어	기능
b* [function]	특정 함수에 breakpoint 설정
b* [address]	특정 주소에 breakpoint 설정
x/[n]x [$register]	특정 레지스터로부터 n워드 만큼 출력
x/[n]x 주소	특정 주소로부터 n 워드 만큼 출력
info reg	현재 레지스터 정보를 출력
set {type} [address] = [value]	특정 메모리 값 변경
set [$register] = [value]	특정 레지스터 값 변경
cl	브레이크 포인트 지우기
d	모든 브레이크 포인트 지우기

부록 - GDB 명령어

- 명령어 참고 블로그
  https://mintnlatte.tistory.com/581
- GDB의 사용법에 대해 더 알아보고 싶다면, 아래의 링크 (GDB wiki 참고)
  http://www.gnu.org/software/gdb/documentation/
- GDB를 사용한 디버깅 예시를 알고 싶다면 아래의 링크 (GNU KOREA 참조)
  http://korea.gnu.org/manual/release/gdb/gdb.html

부록 - ARM Assembly Directive

- 의사 명령어 혹은 지시어(Directive Language)란 어셈블러에게 지시를 내리는 문장이다. 예를 들어 copyarray 코드에서 .global, .type, .text, .end 등을 말한다. 지시어는 기계어로 변환되지 않고 단지 개발자가 어셈블리어 프로그래밍하는 것을 도와준다.
- 지시어는 컴파일러마다 문법이 다르며 다음에 나오는 예제에서는 gcc를 기준으로 설명한다.

부록 - ARM Assembly Directive

Network Computing Lab.

예제

```
1:    .include "config.inc" @#include
2:    .text @RO-Data
3:    .code 32 @16=Thumb, 32=Arm
4:    .extern _main @extern
5:    .global _start @global label
6:    .equ REG_SYSTEM, 0x20000000 @#define
7:
8:_start: @_start 레이블
9:    MOV r0,REG_SYSTEM @define 상수 이용.
10:    BL _func1 @Branch Command
11:    ....
12:_func1:
13:    ....
14:   MOV pc,lr @return
15:MSG: .ascii "ARM Assembly Guide",0 @ascii 상수
16:VAR1: .byte 0x42,'A' @byte형 상수
17:    .end @end directive
```

부록 - ARM Assembly Directive

- **including (예제 소스 line no. 1)**
  .include는 C언어에서 #include 지시자와 같은 역할입니다. 다른 파일을 load합니다. 주의할 점은 include하는 파일안에 파일의 끝을 뜻하는 .end 지시자가 사용되면 그 이후 모든 내용은 무시되기 때문에, include하는 파일에는 .end를 사용하지 않아야 합니다.
- **프로그램 영역 선언 (예제 소스 line no. 2)**
  GCC에서 영역은 .text 영역과 .data영역으로 구분됩니다. .text영역은 program code나 상수 등 프로그램의 RO-Data가 들어가는 영역이라고 생각하시면 됩니다. .data영역은 변수 데이터, 즉 프로그램의 RW-Data에 들어가는 영역입니다.
- **ARM/THUMB 모드 (예제 소스 line no. 3)**
  .code는 데이터가 16bit인지 32bit인지 설정하는 지시어입니다. 16이면 Thumb모드를 뜻하고, 32이면 Arm 모드를 뜻합니다.

부록 - ARM Assembly Directive

- **extern/global 선언 (예제 소스 line no. 4~5)**
  .extern은 C언어와 마찬가지로 외부에 선언된 객체(label)을 해당 파일에서 사용할 때 사용합니다. .global은 해당 파일에서 선언된 label을 다른 외부 파일에서 사용할 수 있도록 해줍니다.
- **Pre-Procssing 상수 (예제 소스 line no. 6~7)**
  .equ는 C언어에서 전처리기 상수(#define)같은 개념입니다. 임베디드 시스템에서 어셈블리로 코딩할 때 자주 사용되니 꼭 알아 두세요.
- **레이블 선언 (예제 소스 line no. 8,12)**
  어셈블리 코딩은 label 또는 명령으로 구분할 수 있습니다. label은 특정 메모리 주소를 가리키는 지표가 되며, 해당 라인에 레이블명 앞에 빈칸이 없이 [레이블 이름:] 으로 선언할 수 있습니다. 명령은 앞에 빈칸이 있어야합니다. 즉, 해당 라인 앞에 빈칸이 없으면 레이블이고, 빈칸이 있으면 명령으로 해석하시면 되겠습니다. 레이블은 C언어에서 변수, 상수, 함수의 역할을 한다고 보시면 됩니다. C언어를 컴파일하면 각각의 변수, 상수, 함수가 각각의 주소를 갖는 label로 변환됩니다.

부록 - ARM Assembly Directive

- **데이터 쓰기 (예제소스 line no. 15~16)**
  1byte를 Write할때는 .byte, 2byte는 .hword, 4byte는 .word를 사용합니다. (32bit 시스템으로 가정) 문자열일 경우 .ascii를 사용하고, 끝에 Null을 뜻하는 0를 넣어줘야 합니다. .ascii보다 더 편리한 .asciz 지시자는 null을 자동으로 넣어줍니다.
  또한, 콤마(,)를 이용해서 연속으로 데이터를 선언할 수 있습니다.
- **End directive (예제소스 line no. 17)**
  .end는 해당 파일의 끝을 뜻합니다. 이 지시자 뒤에 나오는 모든 내용은 무시 됩니다.

- **참조**
  https://m.blog.naver.com/PostView.nhn?blogId=gangst11&logNo=145839687&proxyReferer=https%3A%2F%2Fwww.google.com%2F