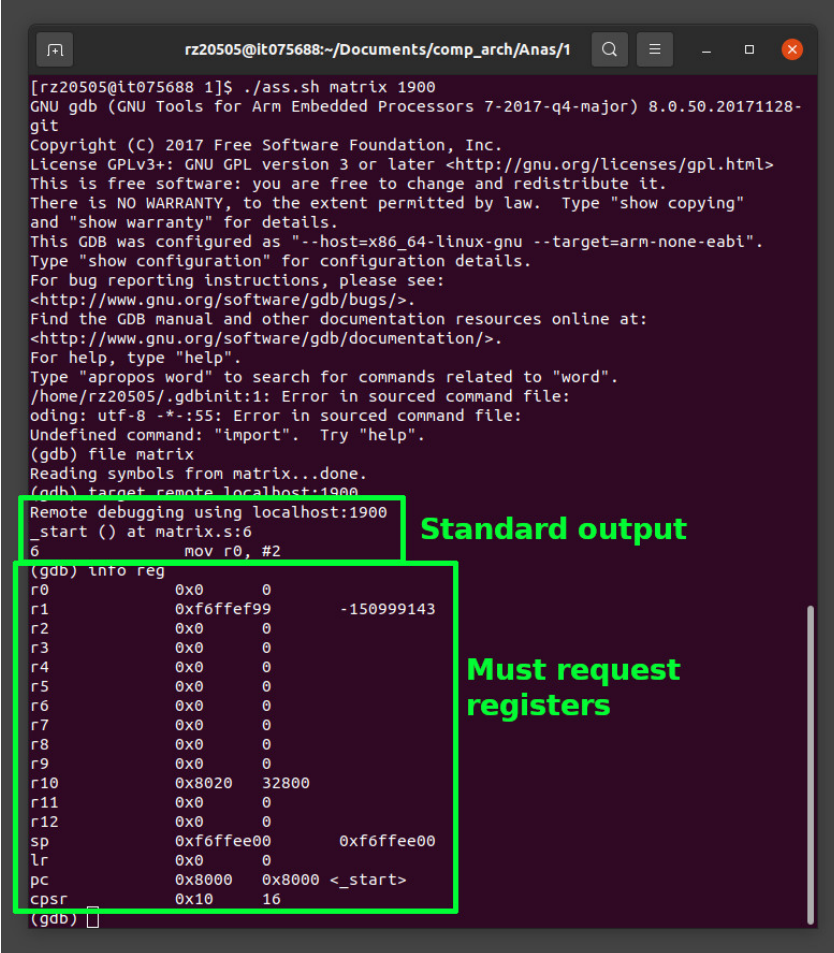


ARM Assembly - getting started

Before you get into programming in assembly, it will be useful to configure your environment. If you do not have an ARM processor on your machine, then you can either connect to a lab machine or install the emulator and debugger on your own machine. I recommend using your own machine because you can install features not available on the lab machines to help you. The instructions in this sheet will work for Ubuntu and WSL.

GDB with/without GEF



```
[rz20505@it075688 1]$ ./ass.sh matrix 1900
GNU gdb (GNU Tools For Arm Embedded Processors 7-2017-q4-major) 8.0.50.20171128-
git
Copyright (C) 2017 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 "--host=x86_64-linux-gnu --target=arm-none-eabi".
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".
/home/rz20505/.gdbinit:1: Error in sourced command file:
oding: utf-8 -*.:55: Error in sourced command file:
Undefined command: "import". Try "help".
(gdb) file matrix
Reading symbols from matrix...done.
(gdb) target remote localhost:1900
Remote debugging using localhost:1900
_start () at matrix.s:6
6      mov r0, #2

(gdb) info reg
r0             0x0          0
r1             0xf6ffef99   -150999143
r2             0x0          0
r3             0x0          0
r4             0x0          0
r5             0x0          0
r6             0x0          0
r7             0x0          0
r8             0x0          0
r9             0x0          0
r10            0x8020       32800
r11            0x0          0
r12            0x0          0
sp             0xf6ffee00   0xf6ffee00
lr             0x0          0
pc             0x8000       0x8000 <_start>
cpsr           0x10        16
(gdb)
```

Standard output

Must request registers

Figure 1: Standard GDB on lab machine

The screenshot shows the GDB interface with GEF installed. The top bar indicates the user is 'harry@Ubuntu' in the directory '~/Documents/Teaching/computer architectu...'. The main window is divided into several sections:

- Legend:** Modified register | Code | Heap | Stack | String
- Registers:** A list of registers from \$r0 to \$pc, each showing its current value and the value it would have if it were modified. For example, \$r0 is 0x00000000, and \$pc is 0x00000000.
- Stack:** A list of stack frames, showing the address, the function name, and the arguments. The current frame is at address 0x00000000.
- Code:** A list of assembly instructions, showing the address, the instruction, and the source code. The current instruction is at address 0x00000000.
- Command Prompt:** A green prompt 'gef>' is visible at the bottom left.

Red boxes highlight the 'registers' section, the 'stack' section, the 'code:arm:ARM' section, and the 'source:gcd.s+15' section. A green box highlights the 'gef>' prompt.

Figure 2: GDB with GEF in Ubuntu/WSL

As you can see from the interface installed locally, the registers, stack, cpsr flags and code are all visible as we step through the program. This gives us a more comprehensive view that we can use to our advantage while debugging programs.

Installation for Ubuntu and WSL

Run the following commands:

```
1 sudo apt update && sudo apt upgrade
2 sudo apt install qemu
3 sudo apt install binutils-arm-none-eabi gcc-arm-none-eabi
4 sudo apt install gdb-multiarch
5 sudo apt install qemu-user
6 sudo apt install curl
7 wget -q -O- https://github.com/hugsy/gef/raw/master/scripts/gef.sh |
   sh
8 bash -c "$(curl -fsSL http://gef.blah.cat/sh)"
```

Everything we need is now installed!

Compiling and executing your program

With assembly, we have to run the following 3 commands in order to assemble our code and start the debugger: (Step 3 will produce a warning when using WSL. Give permission to access local/private networks but not public ones.)

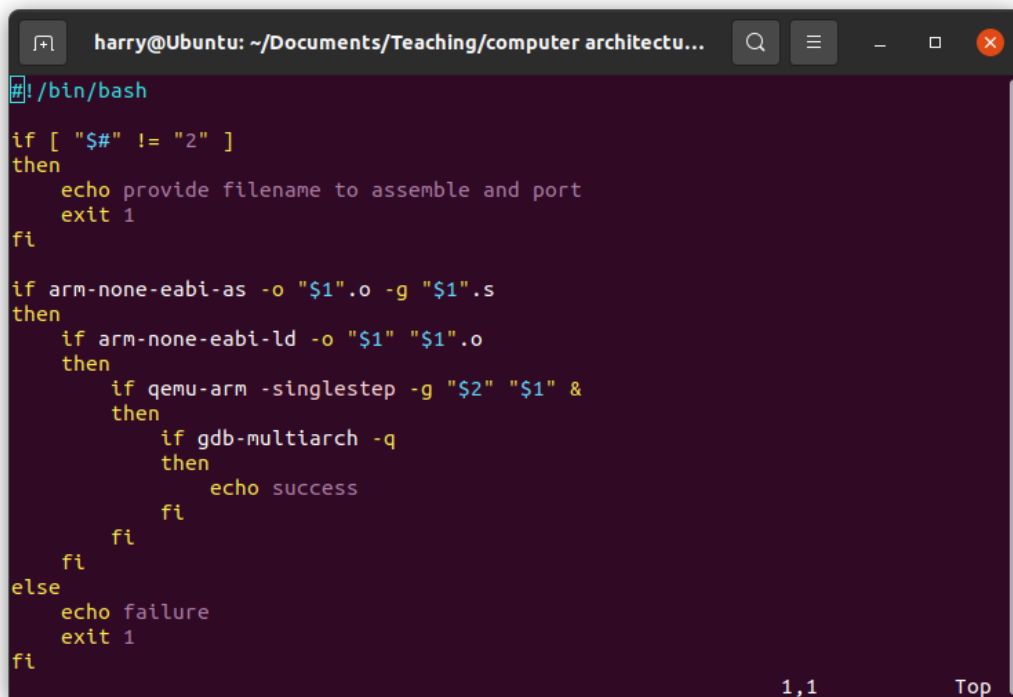
```
1 arm-none-eabi-as -o <FILENAME>.o -g <FILENAME>.s # Create object
2 arm-none-eabi-ld -o <FILENAME> <FILENAME>.o      # Create executable
3 qemu-arm -singlestep -g <PORTNUM> <FILENAME> &   # Start qemu
4 arm-none-eabi-gdb                                # Start gdb
```

2 scripts are provided with this worksheet. `au.sh` is intended for people running an Ubuntu system or VM, and `al.sh` is for people using the lab machines. These files do all of the above steps for you. Think of this like a makefile in C, in that it needs to be in the same directory as the file you want to assemble. With this script in the same directory as your assembly file, the command to run the script is:

```
1 ./au.sh <FILENAME> <PORTNUM>      # For Ubuntu
2 ./al.sh <FILENAME> <PORTNUM>      # For lab machines
```

The port number is included because you often need to change it when you reassemble and restart the debugger. The filename should not have any `.s` or any other appending characters. If your file is called `tst.s`, then the command might be

```
1 ./au.sh tst 1234
```

A screenshot of a terminal window titled "harry@Ubuntu: ~/Documents/Teaching/computer architectu...". The terminal shows the contents of the `au.sh` script. The script starts with a shebang `#!/bin/bash` and a check for the number of arguments. If there are not exactly 2 arguments, it prints "provide filename to assemble and port" and exits with code 1. If there are 2 arguments, it runs `arm-none-eabi-as` to create an object file, then `arm-none-eabi-ld` to create an executable, and finally `qemu-arm` to start the emulator in single-step mode with GDB. It also checks if `gdb-multiarch` is installed. If all steps succeed, it prints "success". Otherwise, it prints "failure" and exits with code 1. The terminal shows the script's execution flow with various `if`, `then`, `fi`, `else`, and `exit` statements. The bottom right of the terminal shows "1,1" and "Top".

```
#!/bin/bash

if [ "$#" != "2" ]
then
    echo provide filename to assemble and port
    exit 1
fi

if arm-none-eabi-as -o "$1".o -g "$1".s
then
    if arm-none-eabi-ld -o "$1" "$1".o
    then
        if qemu-arm -singlestep -g "$2" "$1" &
        then
            if gdb-multiarch -q
            then
                echo success
            fi
        fi
    fi
else
    echo failure
    exit 1
fi
```

Figure 3: Script to automate assembling code and starting debugger