list me plase all rv32e registers aliases

The RISC-V RV32E is a 32-bit embedded processor with a reduced register set compared to the RV32I base instruction set. It has 16 registers in total, which are labeled as x0 to x15. Here are the registers and their aliases:

1. x0: zero (Hardwired to zero)
2. x1: ra (Return address)
3. x2: sp (Stack pointer)
4. x3: gp (Global pointer)
5. x4: tp (Thread pointer)
6. x5: t0 (Temporary register 0)
7. x6: t1 (Temporary register 1)
8. x7: t2 (Temporary register 2)
9. x8: fp/s0 (Frame pointer or Saved register 0)
10. x9: s1 (Saved register 1)
11. x10: a0 (Function argument 0 or return value 0)
12. x11: a1 (Function argument 1 or return value 1)
13. x12: a2 (Function argument 2)
14. x13: a3 (Function argument 3)
15. x14: a4 (Function argument 4)
16. x15: a5 (Function argument 5)

These are the register names and their common aliases for the RISC-V RV32E architecture.

decipher please this command:

```
docker  run -v $PWD:/build -w /build -u $USER xora/qdos-gcc qdos-gcc -o hello hello.c
```

This command is running a Docker container to compile a C program for the QDOS operating system using the QDOS-GCC cross-compiler. Let's break down the command and its options:

1. `docker run`: This command is used to run a Docker container.
2. `-v $PWD:/build`: The `-v` flag maps a local directory to a directory inside the container. Here, `$PWD` is an environment variable representing the current working directory on your local machine. This is being mapped to the `/build` directory inside the container.
3. `-w /build`: The `-w` flag sets the working directory inside the container. In this case, the working directory is set to `/build`, which is the same directory that has been mapped to your local working directory.
4. `-u $USER`: The `-u` flag specifies the user that will run the commands inside the container. `$USER` is an environment variable representing the current user on your local machine.
5. `xora/qdos-gcc`: This is the name of the Docker image being used. It contains the QDOS-GCC cross-compiler.
6. `qdos-gcc -o hello hello.c`: This is the command that will be executed inside the container. `qdos-gcc` is the QDOS-GCC cross-compiler. The `-o hello` option specifies the output file name for the compiled binary (in this case, "hello"). `hello.c` is the source file to be compiled.

In summary, this command runs a Docker container with the `xora/qdos-gcc` image, maps the current working directory to the `/build` directory inside the container, sets the working directory to `/build`, runs the commands as the current local user, and compiles the `hello.c` source file into a binary called "hello" using the QDOS-GCC cross-compiler.

---

can you please transform for windows the command when I have hello.c located here: w:\~tmp\testdocker\ and my user is "petra" ?

To adapt the Docker command for Windows, you'll need to make a few changes to the syntax, including using the correct path format and specifying the user differently. Here's the transformed command for Windows, assuming you have the **`hello.c`** file located in **`W:\~tmp\testdocker\`** and your user is "petra":

powershell Copy code

```powershell
docker run -v W:\~tmp\testdocker:/build -w /build -u "$(id -u petra):$(id -g petra)" xora/qdos-gcc qdos-gcc -o hello hello.c
```

This command assumes you are running it in a PowerShell terminal. Please note that the **`id -u petra`** and **`id -g petra`** commands are used to get the user ID and group ID of the user "petra", respectively. If these commands do not work on your Windows system, you may need to replace them with the appropriate user ID and group ID values.

You may also need to adjust the drive letter (W) and the directory path depending on your system's configuration.

I tried to run this in plain cmd shell, using user "petra" this way: but got message about missing user in passwd ...

docker run -v W:/~tmp/testdocker -w /~tmp/testdocker -u petra xora/qdos-devel qdos-gcc -o hello hello.c

---

I apologize for the confusion. It seems that the **`-u`** flag in the Docker command is causing an issue in the Windows CMD shell. The **`-u`** flag is used to specify the user and group by their IDs, not by their names. In the previous example, I used the **`id`** command, which is not available in the Windows CMD shell.

To run the command in the Windows CMD shell, you can use the **`whoami /user`** command to get the user ID (also known as the Security Identifier or SID) of the current user. However, Docker for Windows does not support passing a Windows SID as a parameter for the **`-u`** flag. Instead, you can run the container without specifying a user, but this will run the container with root privileges. If this is acceptable, you can modify the command as follows:

bash Copy code

```
docker run -v W:\~tmp\testdocker:/build -w /build xora/qdos-devel qdos-gcc -o
hello hello.c
```

If you need to run the container with a specific user, you can create a Dockerfile that sets up the user inside the container and then build and run that container. Here's an example of a Dockerfile that creates a user named "petra":

sql Copy code

```
# Use the xora/qdos-devel image as the base image FROM xora/qdos-devel # Create
a user named "petra" and a group named "petra" with user ID 1000 and group ID
1000 RUN groupadd -g 1000 petra && useradd -u 1000 -g 1000 -m petra # Set the
user for the container USER petra
```

Save this Dockerfile in a directory (e.g., **`W:\~tmp\testdocker\Dockerfile`**). Then, build the Docker image and run the container using the following commands in the Windows CMD shell:

bash Copy code

```
cd W:\~tmp\testdocker docker build -t qdos-devel-petra . docker run -v
W:\~tmp\testdocker:/build -w /build qdos-devel-petra qdos-gcc -o hello hello.c
```

This will create a new Docker image named "qdos-devel-petra" based on the "xora/qdos-devel" image, with a user named "petra" created inside the container. The container will run with the specified user.

I tried it now this way (see that I have -v as W: volume + path then repeated in -w ...) but still after whoami I have problem with user:

w:\~tmp\testdocker>docker run -v W:/~tmp/testdocker -w /~tmp/testdocker -u "falken-nuc/petra" xora/qdos-devel qdos-gcc -o hello hello.c
docker: Error response from daemon: unable to find user falken-nuc/petra: no matching entries in passwd file.

```
w:\~tmp\testdocker>pause
Press any key to continue . . .
```

I apologize for the confusion. It appears that the `-u` flag does not work with Windows usernames directly in the Docker command. Let's try a different approach by creating a Dockerfile that sets up the user inside the container, as I mentioned previously. Please follow the steps below:

1. Create a new text file in the `W:\~tmp\testdocker\` directory and name it `Dockerfile` (without any file extension).
2. Add the following content to the `Dockerfile`:

Dockerfile Copy code

```
# Use the xora/qdos-devel image as the base image FROM xora/qdos-devel # Create a user named "petra" and a group named "petra" with user ID 1000 and group ID 1000 RUN groupadd -g 1000 petra && useradd -u 1000 -g 1000 -m petra # Set the user for the container USER petra
```

3. Open the Windows CMD shell and navigate to the `W:\~tmp\testdocker\` directory:

bash Copy code

```bash
cd W:\~tmp\testdocker
```

4. Build the Docker image using the Dockerfile:

Copy code

```
docker build -t qdos-devel-petra .
```

5. Run the Docker container with the new image:

bash Copy code

```
docker run -v W:\~tmp\testdocker:/build -w /build qdos-devel-petra qdos-gcc -o
hello hello.c
```

This will create a new Docker image named "qdos-devel-petra" based on the "xora/qdos-devel" image, with a user named "petra" created inside the container. The container will run with the specified user.

---

can I use also su as user ?

Yes, you can use the `su` command within the container to switch to another user, but it requires that the user already exists in the container. In your case, since you want to run the container as the "petra" user, you should first create the user "petra" inside the container using a Dockerfile.

Once you have created the Dockerfile, built the image, and started the container as described in my previous response, you can run an interactive shell inside the container using the following command:

bash Copy code

```bash
docker run -it -v W:\~tmp\testdocker:/build -w /build qdos-devel-petra bash
```

This command opens an interactive bash shell inside the container as the "petra" user, because the Dockerfile specifies `USER petra`. If you need to switch to another user (e.g., root), you can use the `su` command in the interactive shell:

bash Copy code

```bash
su root
```

This command will prompt you for the root password. However, since the root password is not set by default in most Docker images, you might need to set the root password in your Dockerfile, like this:

Dockerfile Copy code

```
# Use the xora/qdos-devel image as the base image FROM xora/qdos-devel # Create a user named "petra" and a group named "petra" with user ID 1000 and group ID 1000 RUN groupadd -g 1000 petra && useradd -u 1000 -g 1000 -m petra # Set the root password RUN echo 'root:password' | chpasswd # Set the user for the container USER petra
```

Replace "password" with your desired password for the root user. After building and running the container with this updated Dockerfile, you can use the `su` command to switch to the root user by providing the password you set in the Dockerfile.

Keep in mind that running processes as the root user can be a security risk, so it's generally recommended to use a non-root user whenever possible.

I am in docker image bash this way, but I cant see any hello.c file:

```
docker run -it -v W:/~tmp/testdocker -w /~tmp/testdocker -u root xora/qdos-devel bash
```

this is my prompt:

```
root@15ea958a7e4f:/~tmp/testdocker#
```

My apologies for the confusion earlier. It seems that the issue is related to the path used in the `-v` flag. The tilde (`~`) in the path might not be interpreted correctly by Docker on Windows. To resolve this, you can use the full path to the directory instead of using the tilde.

Assuming that the `W:\~tmp\testdocker\` directory is the correct location of the `hello.c` file on your system, you should modify the Docker command as follows:

bash Copy code

```
docker run -it -v W:/tmp/testdocker -w /tmp/testdocker -u root xora/qdos-devel
bash
```

I have replaced the `/~tmp/testdocker` with `/tmp/testdocker` in both `-v` and `-w` options.

After running the above command, you should now be able to see the `hello.c` file in the `/tmp/testdocker` directory inside the container:

bash Copy code

```
root@container_id:/tmp/testdocker# ls
```

You should see `hello.c` listed in the output. Now, you can compile the `hello.c` file using the `qdos-gcc` command:

bash Copy code

```
qdos-gcc -o hello hello.c
```

This will create the `hello` binary in the `/tmp/testdocker` directory inside the container, which will also be available in the `W:\~tmp\testdocker\` directory on your host machine.

---

okay, will try

Great! Give it a try, and if you encounter any issues or have further questions, please don't hesitate to ask. I'm here to help. Good luck!

---

when I use /build in path then my working directory is in fact /build in image, and I still cant see my hello.c which is stored on host machine W:/build ...

here I tried /_test :

```
root@96454cc23364:/_test# ls /
W:    bin   build  etc   lib    lib64   media  opt   qdos  run   srv  tmp  var
_test  boot  dev    home  lib32  libx32  mnt    proc  root  sbin  sys  usr
root@96454cc23364:/_test#

root@96454cc23364:/_test# ls /build
0001-include-netinet-tcp.h-fix-bitfield-definitions.patch      gcc-patches        qdos-gcc-utils-v2.tar.bz2
0001-ldold-hack-to-not-use-xtc68-install-inside-gcc-build.patch  gcc-utils-patches   scripts
bison-fseterr.patch                                          libc-4.24.5.tar.bz2  tools.patch
gcc-2.95.3-bufixes.patch.bz2                                    libc.patch         xtc68-gcc-hack.patch
gcc-core-2.95.3.tar.bz2
```

It seems that the issue might be with the volume mapping in the Docker command. The `-v` flag is not mapping the host directory to the desired directory inside the container.

Please make sure that the `hello.c` file is stored in the correct directory on the host machine (i.e., `W:\build` or `W:\_test`, depending on your desired configuration).

Then, adjust the Docker command to map the correct host directory to the desired directory inside the container. Here are two examples based on your previous attempts:

1. If your `hello.c` file is stored in the `W:\build` directory on the host machine, use the following command:

```bash
docker run -it -v W:/build:/build -w /build -u root xora/qdos-devel bash
```

2. If your `hello.c` file is stored in the `W:\_test` directory on the host machine, use the following command:

```bash
docker run -it -v W:/_test:/_test -w /_test -u root xora/qdos-devel bash
```

After running one of the above commands (based on your desired configuration), you should be able to see the `hello.c` file in the mapped directory inside the container:

bash Copy code

```
root@container_id:/build# ls
```

or

bash Copy code

```
root@container_id:/_test# ls
```

Please make sure that you are using the correct host directory and container directory in the `-v` flag of the Docker command, and that the `hello.c` file is actually present in the specified host directory.

---

umm, may be I have tip - I use W: mapped by VisualSubst ... that may be problem

---

Yes, that could be the cause of the issue. Docker might not be able to access the drive that has been mapped using VisualSubst.

To resolve this issue, you can use the actual physical path of the directory where `hello.c` is located on your host machine, instead of using the mapped drive. Please follow these steps:

1. Determine the actual physical path of the `hello.c` file on your host machine. For example, if the `hello.c` file is located in `C:\Users\petra\Documents\testdocker`, use this path in the Docker command.
2. Update the Docker command with the correct physical path. Here's an example with the assumed physical path `C:\Users\petra\Documents\testdocker`:

bash Copy code

```
docker run -it -v C:/Users/petra/Documents/testdocker:/build -w /build -u root
xora/qdos-devel bash
```

After running the updated command, you should be able to see the `hello.c` file in the `/build` directory inside the container:

bash Copy code

```
root@container_id:/build# ls
```

Please replace the example path `C:\Users\petra\Documents\testdocker` with the actual physical path of the `hello.c` file on your host machine.

---

now I tried this but still cant see content of d:/_test mapped from windows ... may it be because I used as user the "root" ??

The issue with not being able to see the content of the mapped directory is likely not related to using the "root" user. Instead, it could be due to some other factors, such as file sharing settings or Docker's access permissions to the host's directories.

Here are some steps to try resolving the issue:

1. Ensure that the Docker Desktop application has access to the drive where your files are located. Open Docker Desktop, go to "Settings" > "Resources" > "File Sharing," and make sure the drive containing the folder with `hello.c` is added to the list of shared drives.
2. Double-check the physical path of the `hello.c` file on your host machine, and make sure the path is correct in the Docker command.
3. If you're still experiencing issues, try sharing a different folder to see if it's specific to the current folder. For example, create a new folder named `docker_test` directly on the `D:` drive, and place the `hello.c` file inside this new folder. Update the Docker command to use this new path:

bash Copy code

```bash
docker run -it -v D:/docker_test:/build -w /build -u root xora/qdos-devel bash
```

4. If none of the above steps resolve the issue, you might want to try restarting Docker Desktop, or even reinstalling it, to ensure that there are no issues with the installation.

Please try these steps and let me know if the issue persists.

---

docker run -it -v D:/_test -w /_test -u root xora/qdos-devel bash

---

I see that you're trying to run the Docker command with the `-v` flag to map the host directory `D:/_test` to the `/_test` directory inside the container. However, it seems that you missed the colon (`:`) to separate the host path from the container path in the `-v` flag.

Please update your command as follows:

bash Copy code

```bash
docker run -it -v D:/_test:/_test -w /_test -u root xora/qdos-devel bash
```

This command will map the `D:/_test` directory on the host machine to the `/_test` directory inside the container. After running this command, you should be able to see the `hello.c` file in the `/_test` directory inside the container:

bash Copy code

```bash
root@container_id:/_test# ls
```

If you still cannot see the `hello.c` file, please ensure that the file is actually located in the `D:/_test` directory on your host machine and that Docker Desktop has access to the `D:`

drive, as I mentioned in my previous response.

---

well, that was the problem with accessing files! thanks

You're welcome! I'm glad the issue has been resolved, and you can now access the files in the mapped directory. If you have any further questions or need assistance with anything else, please don't hesitate to ask. I'm here to help. Good luck with your project!

---

not so fast, now I have this error:

```
root@11b2b5035b3a:/_test# qdos-gcc -o hello hello.c
cpp0: hello.c: Value too large for defined data type
root@11b2b5035b3a:/_test#
```

The error message "Value too large for defined data type" indicates that there might be an issue with the `hello.c` source code or with the environment/configuration of the qdos-gcc compiler. To narrow down the cause of the issue, please try the following steps:

1. Verify the content of `hello.c`:
   Make sure that your `hello.c` file has the correct content and is not corrupted or containing any unintended data. You can check the content of the file inside the Docker container by running:

```bash
cat hello.c
```

This will display the content of the `hello.c` file. Ensure that the content matches the expected C source code.

2. Test with a minimal "Hello, World!" program:
   Create a simple "Hello, World!" program in C and try to compile it. Replace the content of **`hello.c`** with the following code:

c Copy code

```c
#include <stdio.h> int main() { printf("Hello, World!\n"); return 0; }
```

Save the updated **`hello.c`** file, and then try compiling it again inside the Docker container:

bash Copy code

```bash
qdos-gcc -o hello hello.c
```

If the minimal program compiles successfully, the issue might be specific to the original **`hello.c`** code. In that case, you will need to review the code to identify any issues.

3. Check the qdos-gcc configuration:
   If the minimal "Hello, World!" program also fails to compile, there might be an issue with the qdos-gcc environment or configuration. Ensure that you're using the correct version of the qdos-gcc compiler and that the compiler is properly set up.

If none of the above steps help resolve the issue, please provide more information about the **`hello.c`** source code and any additional settings or configurations you might be using. This will help me provide more specific guidance to resolve the error.

---

thi is some problem in gcc 64bit vs filesystem, something known

---

It seems that the issue you're encountering is related to the 64-bit version of the qdos-gcc compiler and the filesystem. Here are a few steps to help you resolve this issue:

1. Check for known issues and workarounds:
   Search for any known issues related to the 64-bit qdos-gcc compiler and the filesystem,

especially for the specific version you are using. You can look for bug reports or forum discussions that might offer a solution or workaround.

2. Use a 32-bit version of the qdos-gcc compiler:
   If the issue is specific to the 64-bit version of the compiler, you might consider using a 32-bit version of the qdos-gcc compiler, if available. You can look for a 32-bit version of the xora/qdos-devel Docker image or build your own Docker image with a 32-bit qdos-gcc compiler.
3. Use an older version of the qdos-gcc compiler:
   If the problem is related to a specific version of the qdos-gcc compiler, you could try using an older version that does not have the issue. You can search for older versions of the xora/qdos-devel Docker image or build your own Docker image with an older qdos-gcc compiler.

If you continue to experience the issue and need more specific assistance, please provide additional information about your setup, such as the exact version of the qdos-gcc compiler and any other relevant details. This will help me provide a more tailored solution to your problem.

---

is somehow possible to map in windows docker different filesystem than ntfs?

Docker for Windows relies on the host's filesystem, which is typically NTFS for Windows systems. However, it is possible to use other filesystems by mounting them inside the Docker container using a volume or a bind mount. While the container's filesystem will still be layered on top of the Windows host's NTFS filesystem, the mounted filesystem can be different.

Here's an example of how you can use a different filesystem with Docker on Windows:

1. Create a virtual disk with your desired filesystem (e.g., ext4, FAT32, etc.):
   You can use a tool like DiskGenius or similar software to create a virtual disk with the desired filesystem. Save the virtual disk as a file (e.g., `my_ext4_disk.vhdx`) on your Windows machine.
2. Mount the virtual disk on your Windows machine:

Mount the virtual disk file (e.g., `my_ext4_disk.vhdx`) using Disk Management or PowerShell commands. Assign a drive letter (e.g., `X:`) to the mounted virtual disk.

3. Share the drive with Docker:
   Make sure Docker Desktop has access to the mounted virtual disk. Open Docker Desktop, go to "Settings" > "Resources" > "File Sharing," and add the drive letter (e.g., `X:`) to the list of shared drives.
4. Use the mounted virtual disk in your Docker container:
   Update your Docker command to map the directory from the mounted virtual disk (e.g., `X:/my_directory`) to a directory inside the container (e.g., `/my_directory`):

   bash Copy code

   ```bash
   docker run -it -v X:/my_directory:/my_directory -w /my_directory
   my_docker_image bash
   ```

By following these steps, you can access the files stored on the virtual disk with a different filesystem inside your Docker container. Note that the performance may not be optimal, and you might encounter some compatibility issues or limitations depending on the specific filesystem used.

---

how to create and mount virtual fat32 ?

To create a virtual FAT32 disk in Windows and mount it, follow these steps:

1. Create a virtual FAT32 disk:
   a. Press **`Win + X`** and select "Windows PowerShell (Admin)" to open PowerShell as an administrator.
   b. In PowerShell, enter the following command to create a virtual FAT32 disk. Replace `X` with the desired size of the virtual disk in MB:

   mathematica Copy code

   ```mathematica
   New-VHD -Path "C:\path\to\your\virtual_disk.vhdx" -SizeBytes XMB -Dynamic
   ```

For example, to create a 1GB virtual disk:

mathematica Copy code

```
New-VHD -Path "C:\path\to\your\virtual_disk.vhdx" -SizeBytes 1GB -Dynamic
```

This command will create a virtual disk with a dynamically expanding size (up to the specified limit) at the specified path.

2. Initialize and format the virtual disk with the FAT32 filesystem:
   a. Press **`Win + X`** and select "Disk Management" to open the Disk Management utility.
   b. In Disk Management, you should see the newly created virtual disk listed as "Unknown" and "Not Initialized." Right-click on the disk (not the volume) and select "Initialize Disk."
   c. In the "Initialize Disk" dialog, select the new virtual disk, choose the partition style (either MBR or GPT, depending on your requirements), and click "OK."
   d. Now, right-click on the "Unallocated" space of the virtual disk and select "New Simple Volume."
   e. Follow the "New Simple Volume Wizard" to create a new volume. When prompted to choose the filesystem, select "FAT32." Assign a drive letter (e.g., **`X:`**) to the new volume and complete the wizard.
3. Mount the virtual FAT32 disk:
   The virtual FAT32 disk should now be mounted automatically and accessible via File Explorer using the assigned drive letter (e.g., **`X:`**). You can use the virtual disk just like any other disk on your system.

Keep in mind that FAT32 has some limitations, such as a maximum file size of 4GB and a maximum volume size of 8TB. Depending on your requirements, you may need to choose a different filesystem or adjust the size of the virtual disk accordingly.

---

is possible to create also plain FAT disk this way?

Yes, you can create a virtual plain FAT (FAT16) disk using a similar process. However, you'll need to use the **`diskpart`** command-line utility to format the virtual disk with the FAT16 filesystem since Disk Management does not provide an option for FAT16. Here's how to create and mount a virtual FAT16 disk in Windows:

1. Create a virtual disk:
   a. Open PowerShell as an administrator by pressing `Win + X` and selecting "Windows PowerShell (Admin)."
   b. In PowerShell, enter the following command to create a virtual disk. Replace `X` with the desired size of the virtual disk in MB:

   mathematica Copy code

   ```
   New-VHD -Path "C:\path\to\your\virtual_disk.vhdx" -SizeBytes XMB -Dynamic
   ```

   For example, to create a 512MB virtual disk:

   mathematica Copy code

   ```
   New-VHD -Path "C:\path\to\your\virtual_disk.vhdx" -SizeBytes 512MB -Dynamic
   ```

2. Initialize the virtual disk:
   a. Press `Win + X` and select "Disk Management" to open the Disk Management utility.
   b. In Disk Management, you should see the newly created virtual disk listed as "Unknown" and "Not Initialized." Right-click on the disk (not the volume) and select "Initialize Disk."
   c. In the "Initialize Disk" dialog, select the new virtual disk, choose the partition style (either MBR or GPT, depending on your requirements), and click "OK."
3. Format the virtual disk with the FAT16 filesystem:
   a. Press `Win + X` and select "Command Prompt (Admin)" to open the Command Prompt as an administrator.
   b. In Command Prompt, run the `diskpart` utility by typing `diskpart` and pressing Enter.
   c. List the available disks by entering the following command:

   Copy code

   ```
   list disk
   ```

   d. Identify the new virtual disk by its size, and select it by replacing `X` with the disk number:

csharp Copy code

```
select disk X
```

e. Create a primary partition on the selected disk:

sql Copy code

```
create partition primary
```

f. Format the partition with the FAT16 filesystem:

lua Copy code

```
format fs=fat
```

g. Assign a drive letter (e.g., `X:`) to the new FAT16 partition:

objectivec Copy code

```
assign letter=X
```

h. Exit the `diskpart` utility by entering:

bash Copy code

```
exit
```

4. Mount the virtual FAT16 disk:
   The virtual FAT16 disk should now be mounted automatically and accessible via File Explorer using the assigned drive letter (e.g., `X:`).

Remember that FAT16 has several limitations, such as a maximum file size of 2GB, a maximum volume size of 2GB, and a maximum of 65,536 files in the root directory. Depending on your requirements, you may need to choose a different filesystem or adjust the size of the virtual disk accordingly.

---

how to get New-VHD ?

`New-VHD` is a PowerShell cmdlet available in Windows systems with Hyper-V installed. To use **`New-VHD`**, you need to have Hyper-V enabled on your Windows machine. Here's how to enable Hyper-V:

1. Check system requirements:
   Ensure that your system meets the following requirements to enable Hyper-V:
      A 64-bit processor with Second Level Address Translation (SLAT).
      CPU support for VM Monitor Mode Extension (VT-c on Intel CPUs).
      Minimum of 4GB system RAM.
2. Enable Hyper-V:
   a. Press **`Win + X`** and select "Apps and Features" to open the Apps & Features settings.
   b. Click "Programs and Features" under the "Related settings" section.
   c. In the Programs and Features window, click "Turn Windows features on or off."
   d. In the "Windows Features" dialog, scroll down and find "Hyper-V." Check the box next to "Hyper-V" to enable it. Click "OK."
   e. Restart your computer when prompted.

Once Hyper-V is enabled, you should be able to use the **`New-VHD`** cmdlet in PowerShell.

To create a new virtual disk using the **`New-VHD`** cmdlet, follow these steps:

1. Open PowerShell as an administrator by pressing **`Win + X`** and selecting "Windows PowerShell (Admin)."
2. In PowerShell, enter the following command to create a virtual disk. Replace **`X`** with the desired size of the virtual disk in MB:

   mathematica Copy code

   ```mathematica
   New-VHD -Path "C:\path\to\your\virtual_disk.vhdx" -SizeBytes XMB -Dynamic
   ```

   For example, to create a 1GB virtual disk:

   mathematica Copy code

   ```mathematica
   New-VHD -Path "C:\path\to\your\virtual_disk.vhdx" -SizeBytes 1GB -Dynamic
   ```

This command will create a virtual disk with a dynamically expanding size (up to the specified limit) at the specified path.

---

umm, I have installed Docker using WSL2 probably, it prevented Hyper-V ???

Yes, if you have Docker installed with WSL2 (Windows Subsystem for Linux 2) on Windows 10 or 11, it is likely that you are using the new WSL2 backend for Docker Desktop. In that case, the Hyper-V features are disabled, as WSL2 and Hyper-V cannot be used simultaneously.

If you want to create a virtual disk without using Hyper-V, you can use third-party tools like DiskGenius or other virtualization software that supports creating virtual disks.

Another alternative is to create a virtual disk with WSL2. Follow these steps:

1. Launch your preferred WSL2 distribution (e.g., Ubuntu) from the Start menu.
2. Create a virtual disk file with the desired size. Replace **`X`** with the desired size in MB, and replace **`/mnt/c/path/to/your/virtual_disk.img`** with the path where you want to save the virtual disk file:

   javascript Copy code

   ```javascript
   dd if=/dev/zero of=/mnt/c/path/to/your/virtual_disk.img bs=1M count=X
   ```

   For example, to create a 1GB virtual disk:

   javascript Copy code

   ```javascript
   dd if=/dev/zero of=/mnt/c/path/to/your/virtual_disk.img bs=1M count=1024
   ```

3. Create a loop device for the virtual disk file:

   bash Copy code

   ```bash
   sudo losetup -fP /mnt/c/path/to/your/virtual_disk.img
   ```

4. Check the loop device associated with the virtual disk file:

   css Copy code

   ```
   sudo losetup -a
   ```

   Note the loop device (e.g., **`/dev/loop0`**) associated with the virtual disk file.
5. Create a FAT16 or FAT32 filesystem on the loop device:

   bash Copy code

   ```
   sudo mkfs.vfat -F 16 /dev/loopX
   ```

   Replace **`/dev/loopX`** with the loop device associated with the virtual disk file.

Now, you can mount the virtual disk file in your WSL2 distribution using the following command:

bash Copy code

```
sudo mount /dev/loopX /mnt/my_virtual_disk
```

Replace **`/dev/loopX`** with the loop device associated with the virtual disk file, and **`/mnt/my_virtual_disk`** with the desired mount point.

Keep in mind that this virtual disk will be accessible only within the WSL2 environment. If you need to access the files from Windows, you can use third-party tools like DiskGenius to create and manage virtual disks.